Kapitel 6: Objektorientierte Datenbanksprachen und Erweiterungen von SQL

6.1 Schwachpunkte relationaler Datenbanksysteme

Relationale Datenbanksysteme haben sich in vielen Anwendungen bewährt und repräsentieren in vielerlei Hinsicht den Stand der Kunst in der Datenbanktechnologie. Es gibt jedoch auch Anwendungsklassen, die von relationalen Datenbanksystemen nur sehr schlecht unterstützt werden. Eine bessere Unterstützung solcher Anwendungsklassen ist eine Hauptmotivation für objektorientierte Datenbanksysteme (bzw. allgemeiner jede Art von "postrelationalen" Datenbanksystemen). Zu diesen besonders anspruchsvollen Anwendungsklassen zählen vor allem:

- *CAD (Computer−Aided Design):*
  Im mechanischen CAD ist z.B. die Geometrie von Werkstücken zu modellieren und in einer Datenbank abzulegen. Ein Standardvorgehen dafür ist die Beschreibung eines 3−dimensionalen Körpers durch Anwenden von Zusammensetzungsoperationen auf eine Reihe von (ggf. transformierten) Standardvolumina wie Quader, Kegel und Zylinder.
  Im elektronischen CAD (VLSI−Entwurf) ist der Aufbau von Schaltungen zu modellieren. Dies sind z.T. sehr tiefe Hierarchien, die aus einer enormen Anzahl von Bausteinen bestehen können. Darüber hinaus ist u.a. auch das Layout eines Chips zu beschreiben, was wiederum auf geometrische Strukturen führt.
- *Geowissenschaften:*
  Kartographische Daten sind in praktisch allen Geowissenschaften zu verwalten (Geographie, Geologie, Seismologie, Ozeanographie, ...). Diese Anwendungen benötigen beispielsweise Daten über Ländergrenzen, Stadtgrenzen, Grundstückskataster, Straßenverläufe, Leitungspläne, Gewässer, usw. Häufig kommen zu den eigentlichen geographischen Daten noch weitergehende, durch Meßstationen oder auf sonstige Weise erhobene ortsbezogene Daten wie z.B. Emissionswerte, Gewässerverschmutzung, aktuelle Autoverkehrsdichte, Flächennutzungspläne, usw., die die Grundlage für ökologische Analysen bilden. Mit diesen Daten lassen sich auch Dienste für den Alltagsgebrauch realisieren, beispielsweise ein Autofahrernavigationssystem.
- *Desktop Publishing:*
  Auf dem Rechner erstellte Textdokumente haben eine inhaltliche Struktur (Kapitel, Abschnitte, Absätze, Sätze, Referenzen) und eine Layout−Struktur (Seitenumbruch usw.). Beide Aspekte sind zu modellieren und in einer Datenbank zu hinterlegen. Darüber hinaus werden sogenannte "Hypertext"−Strukturen populär, bei denen man vom "linearen" Lesen eines Dokuments abweicht und stattdessen Referenzen (sog. "Hyperlinks") verfolgt. Beispielsweise könnte man beim Nachschlagen in einem Lexikon oder auch beim Durcharbeiten eines Lehrbuches über solche Referenzen je nach Vorkenntnissen oder spezifischen Informationsbedürfnissen geeignet geführt werden. Ferner entwickeln sich Textdokumente durch Hinzunahme von Bildern und Grafiken immer mehr zu multimedialen Dokumenten bis hin zur Integration von Video− und Audioaufzeichnungen. Nachrichtenarchive − beispielsweise in Zeitungs− und Fernsehredaktionen, aber auch für öffentliche Bildungs− und Informationsangebote − werden auf multimedialem Desktop Publishing basieren.

Forderungen an objektorientierte bzw. “objektrelationale” Datenbanksysteme

Forderung nach Unterstützung komplexer Objektstrukturen:

Jedes Objekt der Anwendung soll als ein − komplex strukturiertes − Objekt der Datenbank repräsentierbar sein.

Beispiel:
Repräsentation einer Straße mit ihrem Verlauf als Polylinie, so daß einerseits einfach auf das Objekt “Straße” als Ganzes zugegriffen werden kann, andererseits aber auch einzelne Stützpunkte des Straßenverlaufs einfach verarbeitet werden können.

Forderung nach Erweiterbarkeit des DBS:

Das DBS soll erweiterbar sein, indem anwendungsspezifische Funktionen in der Datenbank registriert werden, die dann in beliebigen Anfragen und für integritätserhaltende Änderungen verwendbar sind.
Die konkrete Implementierung einer solchen Funktion soll austauschbar bleiben, sofern ihre Schnittstelle und ihre Semantik erhalten bleiben.

Beispiel:
Definition einer Längenberechnungsfunktion für die Polylinie eines Straßenverlaufs, so daß die Funktion innerhalb von Queries frei verwendet werden kann.

Forderung nach Wiederverwendbarkeit:

Datenbankteilschemata und Anwendungsfunktionen sollten − sofern sie generischen Charakter haben − direkt wiederverwendbar sein.

Beispiel:
Wiederverwendung des Schemas von Polylinien und der entsprechenden Funktionen für Straßen, Bahnlinien, Stromleitungen, usw.

6.2 Grundkonzepte objektorientierter Datenmodelle

6.2.1 Komplexe Objekte (Strukturelle Objektorientierung)

Jedes *Objekt* hat eine Menge von Attributen (engl.: Attribute, Property). Für jedes Attribut besitzt das Objekt einen Wert; die Gesamtheit der Attributwerte eines Objekts bilden den Zustand des Objekts.

Jedes *Attribut* hat einen Namen und einen Wertebereich (engl.: Domain).
Zulässige Wertebereiche sind
- elementare Wertebereiche (Integer, Real, String, Boolean)
- *Referenzen* (*Relationships*) auf andere Objekte
- Verbunde (Structures, Records), Mengen (Sets), Multimengen (Bags), Listen (Lists), Felder (Arrays) mit zulässigen Wertebereichen für die jeweiligen Komponenten bzw. Elemente, insbesondere auch mit Objektreferenzen als Komponenten bzw. Elemente (Aggregation von Objekten)

Der *Typ* eines Objekts ist die Menge der für das Objekt definierten Attribute.

Die Menge der Objekte gleichen Typs werden zu einer *Klasse* zusammengefaßt (Klassifikation von Objekten).
Klassen sind selbst wieder Objekte (vom Typ Set<Elementobjekttyp>) und können zusätzliche (Meta−) Attribute haben wie z.B. die Kardinalität der Klasse, Schlüsseleigenschaft von Attributen.

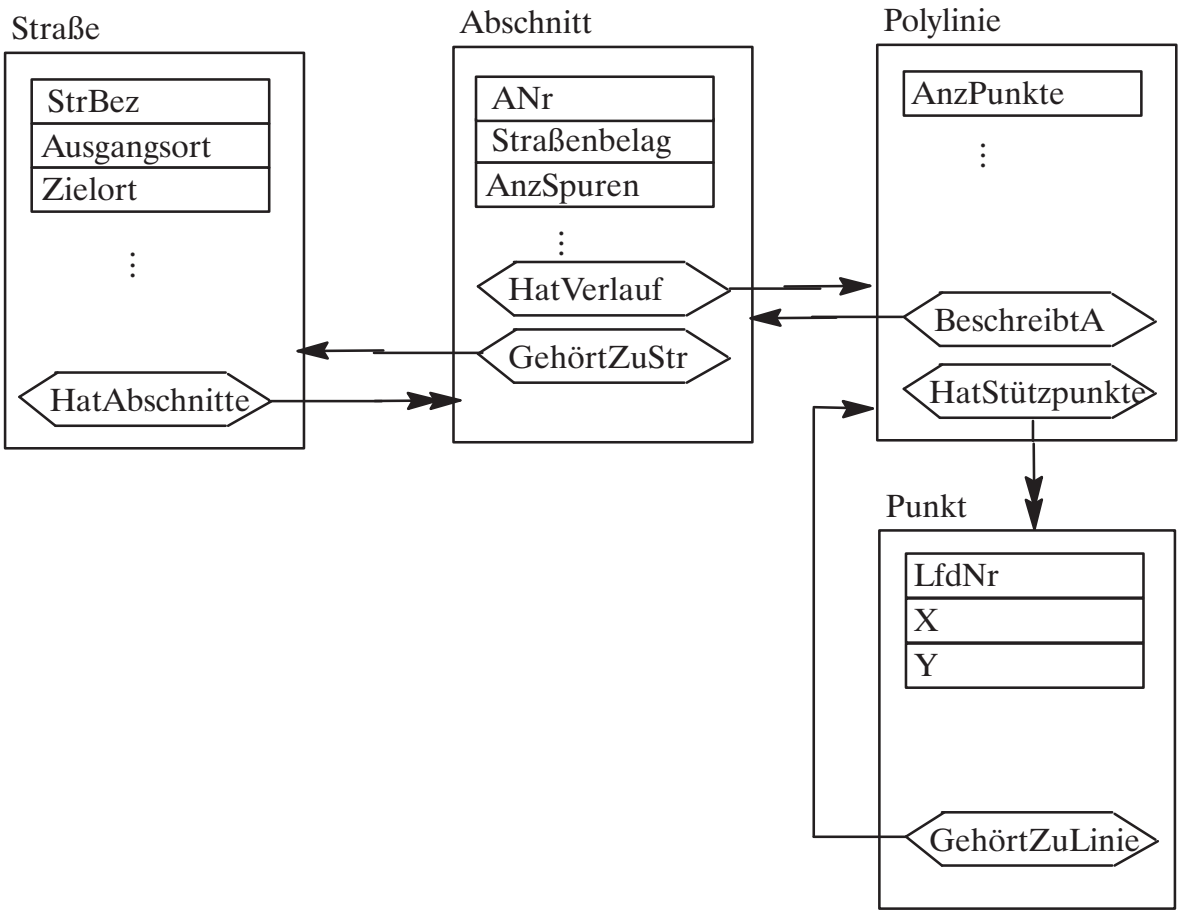
Eine objektorientierte Datenbank ist eine Menge von Objektklassen.

Bemerkung:
In objektorientierten Programmiersprachen ist eine Klasse eigentlich ein Programmodul, d.h. eine Menge von Operationen und ggf. Datenstrukturen, und es gibt Konstruktoren zur Erzeugung von Objekten, auf die dieser Programmodul anwendbar ist (→ Abschnitt 2.2.2). Im Gegensatz zur Situation bei DBS ist man in Programmiersprachen traditionell nicht an deskriptiven Anfragen über Mengen von Objekten interessiert.

Beispiel:

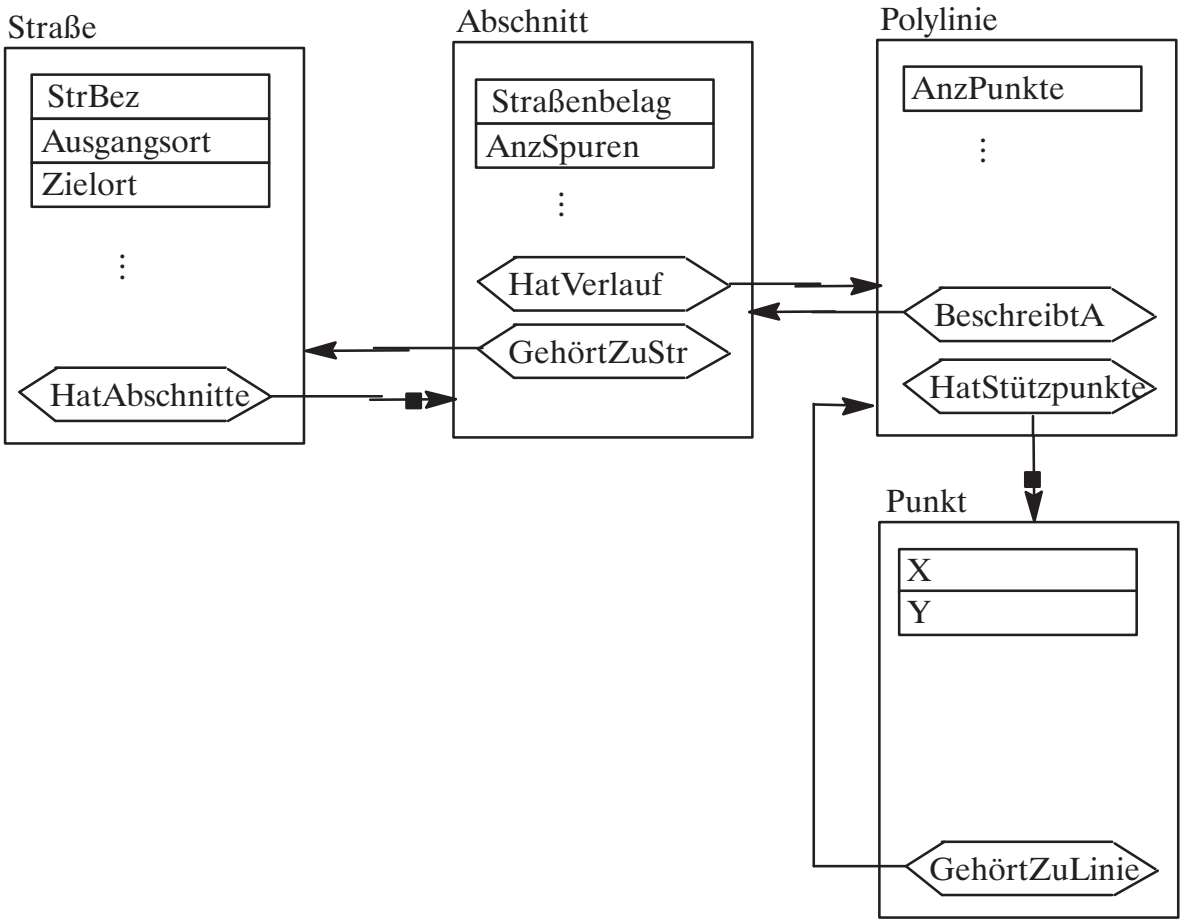
In einer Datenbank sollen Informationen über Straßen verwaltet werden. Eine Straße besteht aus mehreren Abschnitten, die sich im Straßenbelag, Spurenausbau usw. unterscheiden können. Der Verlauf eines Straßenabschnitts soll durch eine Polylinie modelliert werden; zwischen je zwei Stützpunkten wird mittels eines Geradenstücks interpoliert.

Beispiel – Alternative 1:



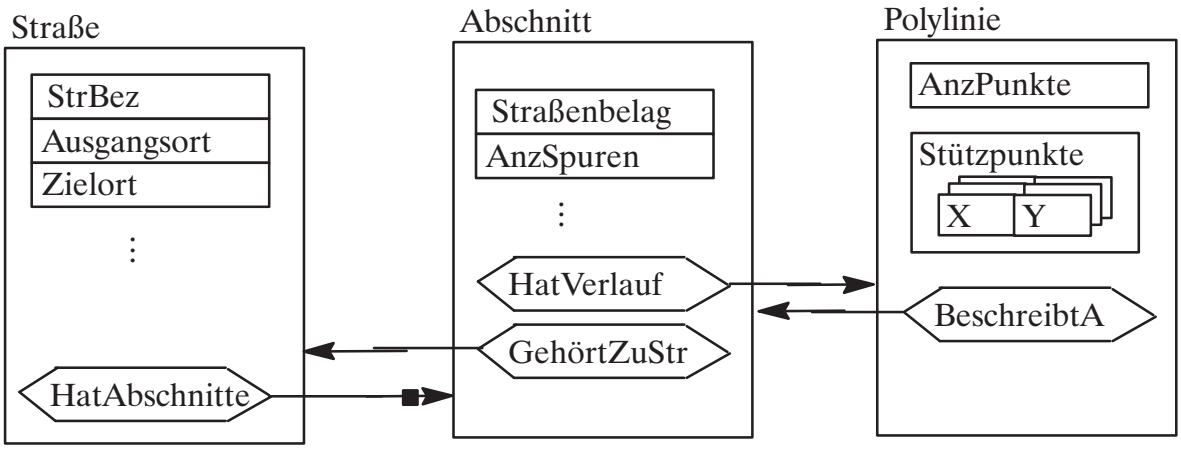

```
interface Straße {
        extent Straßen;
        key StrBez;

        attribute String StrBez;
        attribute String Ausgangsort;
        attribute String Zielort;
        ...
        relationship Set<Abschnitt> HatAbschnitte;
        }
interface Abschnitt {
        extent Abschnitte;
        key (GehörtZuStr.StrBez, ANr);

        attribute Integer ANr;
        attribute String Straßenbelag;
        attribute Integer AnzSpuren;
        attribute Integer Tempolimit;
        ...
        relationship Polylinie HatVerlauf;
        relationship Straße GehörtZuStr;
        }
...
```

Beispiel – Alternative 2:




```
interface Straße {
        ...
        relationship List<Abschnitt> HatAbschnitte;
        }
...
interface Polylinie {
        ...
        relationship List<Punkt> HatStützpunkte;
        }
...
```

Beispiel – Alternative 3:

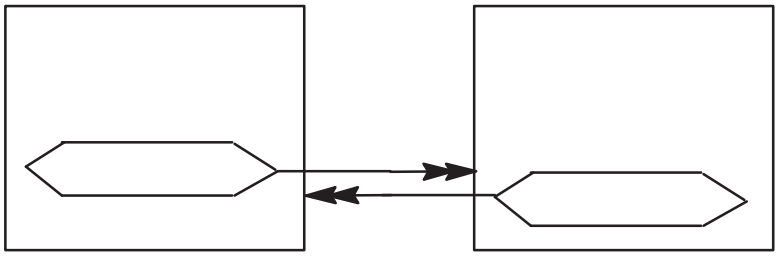


...

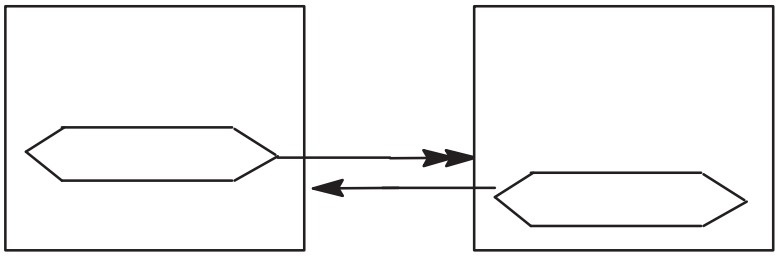
```
interface Polylinie {
        ...
        attribute Integer AnzPunkte;
        attribute List<Struct<X: Real; Y: Real>> Stützpunkte;
        }
```

Verschiedene Arten von Relationships

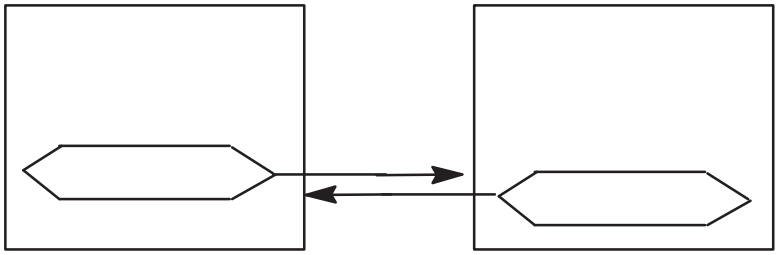
M:N−Relationship (Viele−zu−viele−Beziehung, komplexe Beziehung)



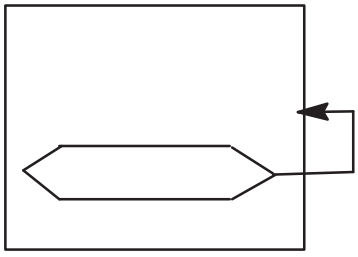
1:N−Relationship (Eins−zu−viele−Beziehung, hierarchische Beziehung) bzw.
N:1−Relationship (Viele−zu−eins−Beziehung, hierarchische Beziehung)



1:1−Relationship (Eins−zu−eins−Beziehung, injektive Beziehung)



Relationship zwischen Objekten derselben Klasse



Beispiel: Relationship “Ehepartner” der Klasse “Person”

Integritätsbedingungen für Relationships

In vielen Fällen soll eine Referenz in einem referenzierten Objekt wieder auf das referenzierende Objekt verweisen, also eine Art Rückwärtszeiger darstellen. Wenn man Referenzen bzw. Relationships als Abbildungen (bzw. Relationen) zwischen zwei Klassen versteht, dann bedeutet dies, daß in vielen Fällen eine Abbildung von Klasse A nach Klasse B und eine Abbildung von B nach A invers zueinander sein sollen.

Beispiele:

1) Gegeben seien zwei Klassen
   Männer mit “relationship Frauen Ehefrau” und
   Frauen mit “relationship Männer Ehemann”.
   Dann soll immer gelten: $\forall$ m $\in$ Männer : m.Ehefrau.Ehemann = m und
   $\forall$ f $\in$ Frauen: f.Ehemann.Ehefrau = f.

2) $\forall$ s $\in$ Straße $\forall$ a $\in$ s.HatAbschnitte : a.GehörtZuStr = s
   und
   $\forall$ a $\in$ Abschnitte $\forall$ s $\in$ a.GehörtZuStr : a $\in$ s.HatAbschnitte

Diese Art von Integritätsbedingungen kann wie folgt spezifiziert werden:

1)
```
interface Männer {
    ...
    relationship Frauen Ehefrau inverse Frauen::Ehemann;
    }
interface Frauen {
    ...
    relationship Männer Ehemann inverse Männer::Ehefrau;
    }
```

2)
```
interface Straße {
    ...
    relationship Set<Abschnitt> HatAbschnitte inverse Abschnitt::GehörtZuStr;
    }
interface Abschnitt {
    ...
    relationship Straße GehörtZuStr inverse Straße::HatAbschnitte;
    ...
    }
```

Die Spezifikation von inversen Relationships erlaubt die Gewährleistung der referentiellen Integrität durch das DBS. Wenn z.B. eine Straße gelöscht wird, müssen die Relationships “GehörtZuStr“ ihrer Abschnitte in Nullzeiger geändert werden (oder diese Abschnitte müssen sogar ebenfalls gelöscht werden).

Objekt−Sharing

Ein Objekt kann von mehreren Objekten referenziert werden. Ein solches Objekt wird als “shared object” bezeichnet.

Beispiel:
Nehmen wir an, verschiedene Straßen können gemeinsame Abschnitte haben (z.B. teilen sich die Mainzer Straße in Saarbrücken und die Bundesstraße B51 einige Abschnitte). Um dies zu erlauben ist lediglich die Definition der Klasse “Abschnitt” wie folgt zu ändern:

```
interface Abschnitt {
    ...
    relationship Set<Straße> GehörtZuStr inverse Straße::HatAbschnitte;
    ... }
```

Änderungen, die auf einem “shared object” über einen bestimmten Referenzpfad durchgeführt werden, sind für alle referenzierenden Objekte gleichzeitig sichtbar.
Beispiel:
Nehmen wir stark vereinfachend an, daß der erste Abschnitt der Mainzer Str. gleichzeitig der letzte Abschnitt der Bundesstraße B51 sei. Dieser Abschnitt soll von 2 auf 4 Spuren verbreitert werden.
Die Wirkung der Änderungsoperation

```
Straße [StrBez=”Mainzer Str.”].HatAbschnitte.first().AnzSpuren = 4
```

wird auch für die B51 unmittelbar wirksam, z.B. in der darauffolgenden Query

```
Straße [StrBez=”B51”].HatAbschnitte.last().AnzSpuren,
```

die den Wert 4 zurückliefert.

Objekt−Identität

Es kann verschiedene Objekte geben, die in allen Attributwerten übereinstimmen. Dennoch sind diese Objekte über ihre *Objekt−ID* (OID) eindeutig unterscheidbar. Zwei Referenzen auf je eines der beiden Objekte sind auf Gleichheit testbar; der Test liefert in diesem Fall “false” als Resultat.

Beispiel:
Es könnte zwei Objekte in der Klasse “Punkt” geben, die in ihren Koordinaten übereinstimmen. Wenn sie aber zu verschiedenen Polylinien gehören und eine der beiden Linien wird verschoben, wird dabei nur das eine Punkt−Objekt geändert.

6.2.2 Objektmethoden und Kapselung (Verhaltensmäßige Objektorientierung)

Für jede Klasse kann eine Menge von *Methoden* definiert werden. Dies sind Funktionen und Änderungsoperationen, die auf jedes Objekt der Klasse anwendbar sind. Zusätzlich zu dem Objekt, auf das eine Methode angewandt wird, kann eine Methode weitere Parameter (sowohl Werte als auch Objekte) haben und auch ein Funktionsresultat zurückliefern.

Die Signatur einer Methode m ist eine Liste von Typen $<T_1, ..., T_n, T_{n+1}>$ derart, daß die Methode n Parameter der Typen $T_1, ..., T_n$ und ein Resultat des Typs T_{n+1} hat. In Anlehnung an algebraische Strukturen wird auch wie folgt geschrieben: $m: T_1 \times ... \times T_n \rightarrow T_{n+1}$.

Der Aufruf einer Methode für ein Objekt x wird wie der Zugriff auf ein Attribut von x geschrieben, nämlich in der Form x.m oder x−>m (bzw. x.m(...) oder x−>m(...)). Damit kann auf gespeicherte Daten und (mittels Methoden) berechnete Daten einheitlich zugegriffen werden.

Zur Schnittstellendefinition einer Klasse gehört nur die Angabe der Namen und Signaturen von Methoden. Die Implementierung einer Methode ist nach außen hin verborgen. Eine Klasse bildet somit einen *Abstrakten Datentyp (ADT)*, also ein Modul, das Basisoperationen auf einer Menge von Objekten (gleichen Typs) anbietet und das die Implementierung der Operationen sowie z.T. auch die zugrundeliegenden Datenstrukturen für die Objekte verbirgt. Durch dieses "Information−Hiding"−Prinzip wird eine hohe Modularität errreicht; große Systeme werden leichter wartbar. Man spricht in diesem Kontext auch von "*gekapselten Objekten*": die Implementierung der Methoden ist austauschbar, ohne die Schnittstelle zu ändern.

Zu den für eine Klasse angebotenen Methoden gehört insbesondere immer eine Methode zur Erzeugung und evtl. Initialisierung neuer Objekte der Klasse (sog. "Konstruktoren"). Diese Methode trägt üblicherweise denselben Namen wie die Klasse selbst und wird häufig gar nicht explizit angegeben.

Zur Implementierung der Methoden können die Objekte einer Klasse weitere Attribute haben, die nach außen nicht sichtbar sind. Diese Attribute heißen "private" (engl.: private, hidden) Attribute, die an der Schnittstellen sichtbaren Attribute heißen "öffentliche" (engl.: public) Attribute.

Beispiele:

1) ADT "Elektronisches Telefonbuch":
   Ein elektronisches Telefonbuch bietet Methoden wie das Nachschlagen der Telefonnummer zu einem gegebenen Namen und einer Adresse an. Die zugrundeliegende Datenstruktur könnte ein Array mit sortierten Einträgen, eine Hashtabelle oder ein Suchbaum sein. Da die Datenstruktur und die Implementierung der Methoden auf der jeweiligen Datenstruktur gekapselt − also nach außen nicht sichtbar − sind, läßt sich die Implementierung jederzeit austauschen, ohne daß sonstiger Programmcode, der die Methoden des ADTs Telefonbuch benutzt, geändert werden muß.

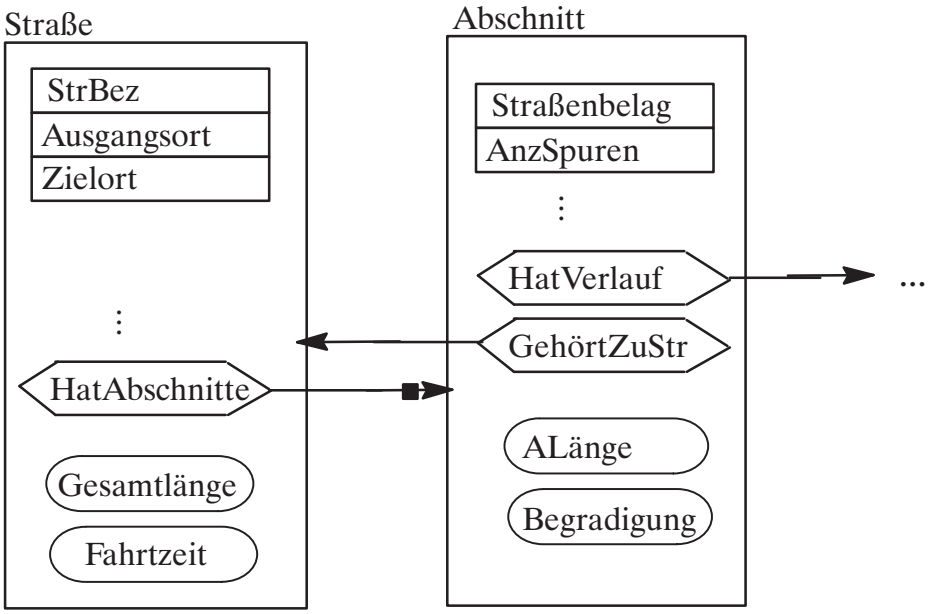
2) Straßen und Straßenabschnitte:

```
interface Straße {
        ...
        private attribute Integer Durchschnittstempo;
        Real Gesamtlänge ( );
        Integer Fahrtzeit ( );
        }
interface Abschnitt {
        ...
        Real ALänge ( );
        Boolean Begradigung (in Punkt, in Punkt);
        }
```

mit der folgenden Implementierung für Gesamtlänge und ALänge:

```
Straße:: Gesamtlänge () {
        float l = 0.0;
        Set<Ref<Abschnitt>> MeineAbschnitte = this->HatAbschnitte;
        Iterator<Abschnitt> it = MeineAbschnitte->create_iterator();
        Ref<Abschnitt> a;
        while (a=it.next())
                { l += a->ALänge(); }
        return l };
Abschnitt:: ALänge () {
        float l = 0.0;
        List<Ref<Punkt>> MeinePunkte = this->HatVerlauf->HatStützpunkte;
        Iterator<Punkt> it = MeinePunkte->create_iterator();
        Ref<Punkt> p, q;
        p = it->next();
        while (q=it->next()) {
                l += sqrt( (p->X - q->X) * (p->X - q->X) +
                        (p->Y - q->Y) * (p->Y - q->Y));
                p = q; };
        return l };
```

Graphische Illustration:



Vorteil der Kapselung:

Nehmen wir an, die Implementierung der Längenberechnung soll wie folgt geändert werden. Zwischen je zwei aufeinanderfolgenden Stützpunkten soll der Verlauf durch einen Kreisbogen interpoliert werden. Dazu soll zusätzlich zu den Stützpunkten die Steigung einer durch den ersten Stützpunkt gehenden Tangente an das erste Kreisbogenstück gespeichert werden. Damit sind alle Kreisbogenstücke eindeutig bestimmt.

Um diese Änderung zu realisieren, muß die Klasse Polylinie um ein zusätzliches Attribut "Tangentensteigung" erweitert werden, und die Implementierung der Methode "ALänge" ist zu ändern. Die Schnittstellen der Klassen "Abschnitt" und "Straße" aber bleiben unverändert.

Analoge Überlegungen gelten beispielsweise auch für eine Umstellung der Punktkoordinaten von kartesischen Koordinaten auf Polarkoordinaten usw.

6.2.3 Vererbung

Eine Klasse B heißt *Subklasse* einer Klasse A und A heißt *Superklasse* von B, wenn

- die Attribute und Methoden von B eine (evtl. unechte) Obermenge der Attribute und Methoden von A sind und
- die Objektmenge von B eine (evtl. unechte) Teilmenge der Objektmenge von A ist.

B wird als *Spezialisierung* von A und A als *Generalisierung* von B bezeichnet. Man sagt auch, daß B die Attribute und Methoden von A "erbt". Dadurch werden Schemadefinition und Methoden der Klasse A in der Klasse B wiederverwendet.
Zusätzlich können für B weitere Attribute und Methoden definiert werden.

Die Generalisierungs−/Spezialisierungsbeziehung zwischen Klassen bildet eine partielle Ordnung, und man spricht auch von einer Generalisierungshierarchie oder einer Vererbungshierarchie.

Ein Objekt einer Subklasse kann überall dort verwendet werden, wo ein Objekt der entsprechenden Superklasse verwendet werden darf (Substitutionsprinzip).

Beispiele:

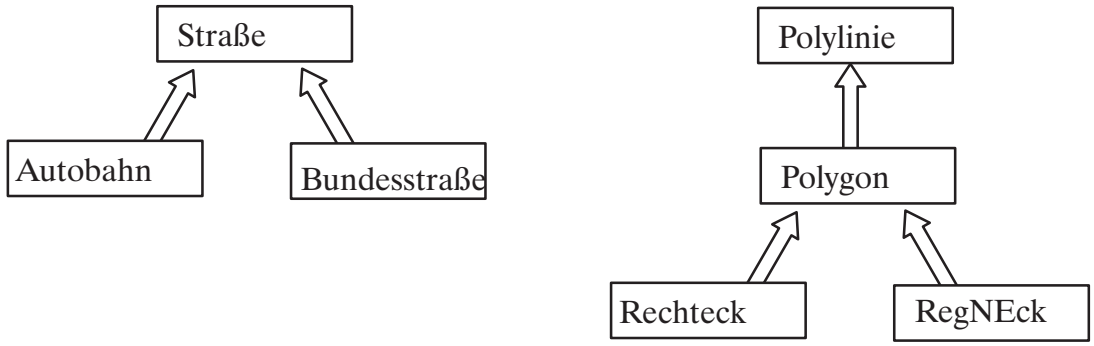
1) Spezialisierung von Straßen in Bundesstraßen und Autobahnen:

```
interface Bundesstraße: Straße {
        extent Bundesstraßen;
        attribute Integer Verkehrsdichte;
        }

interface Autobahn: Straße {
        extent Autobahnen;
        attribute Integer Mindesttempo;
        attribute Array<Integer> VerkehrsdichteProTag;
        relationship Set<Punkt> Auffahrten;
        }
```

2) Spezialisierung von Polylinien in Polygone und weitergehende Spezialisierungen (Rechtecke, regelmäßige N−Ecke):

```
interface Polygon: Polylinie {
        extent Polygone;
        relationship Punkt Zentrum;
        relationship Stadt BeschreibtS inverse Stadt::Stadtgrenze;
        relationship Land BeschreibtL inverse Land::Landesgrenze;
        Real Fläche ( );
        }

interface Rechteck: Polygon {
        extent Rechtecke;
        attribute Real Diagonallänge;
        }

interface RegNEck: Polygon {
        extente RegNEcke;
        attribute Integer AnzEcken;
        Real Inkreisradius ();
        Real Umkreisradius ();
        }
```

Graphische Darstellung:



Die Subklassen einer Superklasse müssen weder disjunkt sein noch müssen sie zusammen eine Überdeckung der Superklasse bilden.

Beispiele:

1) Es gibt Straßen, die weder Bundesstraßen noch Autobahnen sind.
2) Quadrate sind sowohl Rechtecke als auch regelmäßige N−Ecke.

Überschreiben von Methoden (Overriding)

Die von einer Superklasse geerbten Methoden können in einer Subklasse anders implementiert werden. Dies kann aufgrund der besonderen "Semantik" der Subklassenobjekte oder auch aus Effizienzgründen sinnvoll sein.

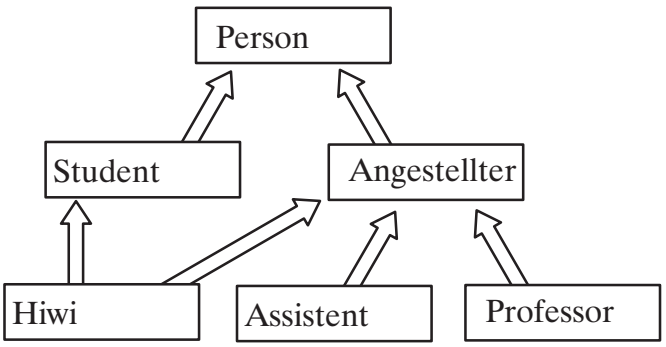
Beispiele:

1) Spezialisierung von Polygonen in Polygone mit Löchern (z.B. Seen mit Inseln):
   Die Methode "Fläche" sollte überschrieben werden, indem z.B. die Fläche der Löcher abgezogen wird.
2) Spezialisierung von Polygonen in Rechtecke und RegNEcke:
   Die Methode "Fläche" könnte überschrieben werden, um die Flächenberechnung effizienter durchzuführen.
3) Spezialisierung von Polylinien in Polygone:
   Die Methode "SchneidetBox (in Rechteck)", die testet, ob eine Polylinie ein gegebenes Rechteck schneidet, sollte überschrieben werden. Ein Polygon schneidet ein Rechteck auch, wenn kein einziger Punkt seines Randes im Rechteck liegt, nämlich wenn das Rechteck vollständig im Polygon liegt.

Mehrfachvererbung

Eine Klasse kann mehrere Superklassen haben. Bei der Vererbung müssen dann ggf. Namenskonflikte bzgl. der geerbten Attribute oder Methoden durch Umbenennung gelöst werden.

Beispiel:



wobei sowohl die Klasse “Student” als auch die Klasse “Angestellter” jeweils ein Attribut “Wochenpensum” haben.

Eines der beiden von Hiwi−Objekten geerbten Attribute “Student.Wochenpensum” und “Angestellter.Wochenpensum” muß umbenannt werden, z.B. das von Student geerbte Attribut in “Hiwi.Vorlesungsstunden”.

Verarbeitung polymorpher Objektmengen

Die Objekte einer Klasse mit einer oder mehreren Subklassen haben eine heterogene Struktur. Dennoch können sie bzgl. der gemeinsamen Attribute und Methoden einheitlich verarbeitet werden. Wenn jedoch Methoden in den Subklassen überschrieben worden sind, sind mit einem Methodennamen eigentlich verschiedene Implementierung der Methodenausführung verbunden.
Die Methode heißt dann auch “überladen” (engl.: overloaded).

Beispiel:

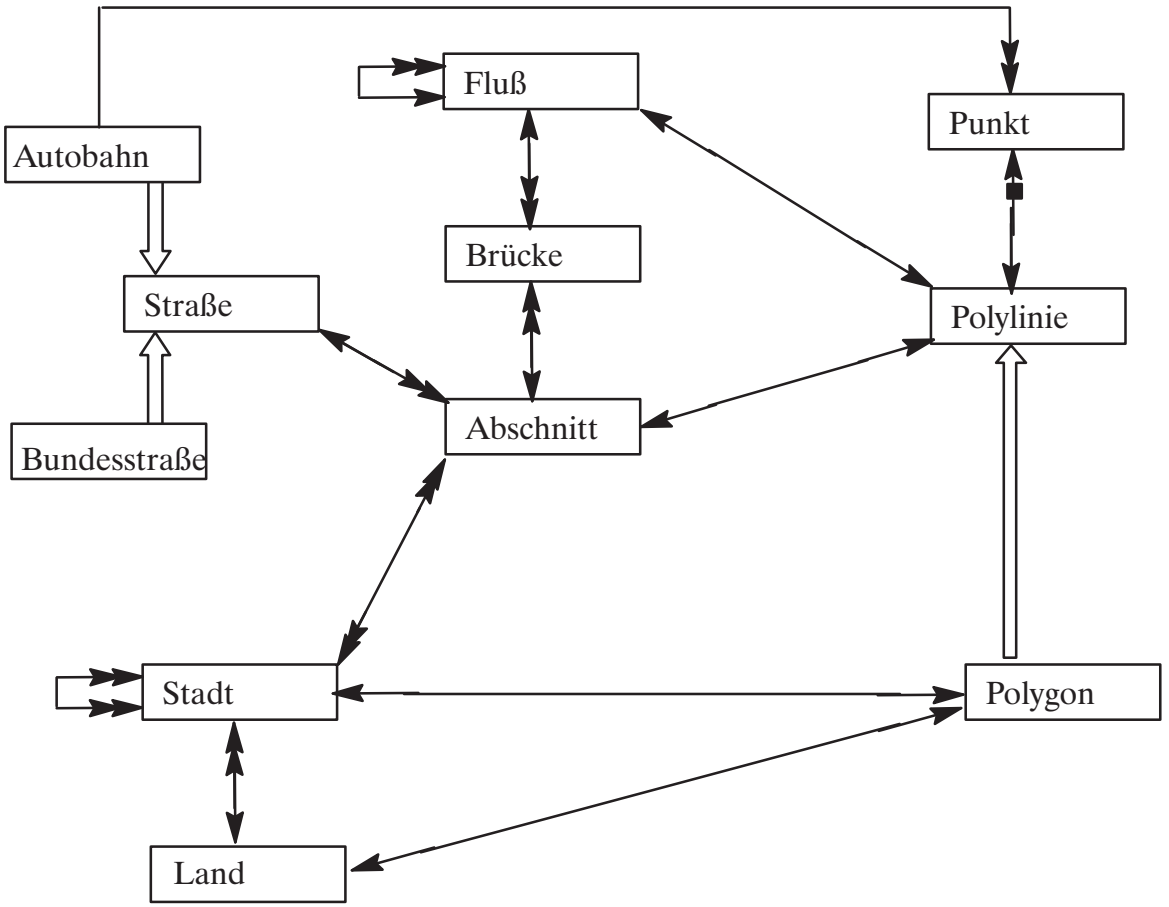
Für eine Menge von Objekten der Klasse “Polylinie” soll mittels der Methode “SchneidetBox” getestet werden, welche Polylinien ein gegebenes Rechteck schneiden.
Da die Klasse “Polylinie” auch Polygone enthält, für die die Methode “SchneidetBox” überschrieben wurde, müssen zwei verschiedene Methodenimplementierungen aufgerufen werden, je nachdem, ob es sich um ein Polygon oder eine offene Polylinie handelt.
Damit bei der Verarbeitung solcher polymorpher Objektmengen jeweils die für jede Subklasse passende Methodenimplementierung verwendet wird, darf der Code für überschriebene Methoden nicht statisch gebunden werden. Vielmehr entscheidet es sich erst zur Laufzeit von Objekt zu Objekt, welche Implementierung je nach Subklassenzugehörigkeit verwendet werden muß. Diese dynamische Festlegung der passenden Methodenimplementierung heißt “*dynamisches Binden*” oder “*spätes Binden*” (engl.: late binding).

6.3 Schemabeschreibung nach dem ODMG−Standard

Die Object Database Management Group (ODMG), ein Herstellerkonsortium, hat einen Standard zur objektorientierten Schemabeschreibung entwickelt, der eine Schemabeschreibungssprache ODL (Object Definition Language) beinhaltet, sowie einen Standard für eine objektorientierte Anfragesprache OQL (Object Query Language) und einen Vorschlag für deren Einbettung in C++ und Smalltalk. Dieser Standard wird u.a. von dem objektorientierten Datenbanksystem O2 weitestgehend unterstützt.

ODL bietet im wesentlichen die in Abschnitt 14.2 vorgestellten Grundkonzepte in syntaktisch ausgefeilter Form an. ODL ist eine Erweiterung der Interface Definition Language (IDL) der Object Management Group (OMG).

Beispiel:



Bemerkungen:
Dieses Beispiel bringt auch einige Schwachstellen des ODMG−Modells zu Tage:

- Es fehlen Möglichkeiten, explizite Integritätsbedingungen zu spezifizieren. Beispielsweise müßte man eigentlich fordern, daß alle Partnerstädte einer Stadt in einem anderen Land liegen, und zwar in paarweise verschiedenen Ländern. Ein weiteres Beispiel wäre, daß man nicht wirklich erzwingen kann, daß bei Polygonen der erste und der letzte Stützpunkt übereinstimmen.
- Ternäre (oder noch höherstellige) Beziehungen sind nicht ausdrückbar. Beispielsweise kann man eine Beziehung zwischen zwei Städten und einer Straße, die die jeweils schnellste Verbindung zwischen den Städten angibt, nicht sauber modellieren.

Ausführliche Modellierung des Beispiels in ODL:

```
interface Straße {
        extent Straßen;
        key StrBez;
        attribute String StrBez;
        attribute String Ausgangsort;
        attribute String Zielort;
        relationship Set<Abschnitt> HatAbschnitte inverse Abschnitt::GehörtZuStr;
        Real Gesamtlänge ( );
        Integer Fahrtzeit ( );
        }
interface Bundesstraße: Straße {
        extent Bundesstraßen;
        attribute Integer Verkehrsdichte;
        }
interface Autobahn: Straße {
        extent Autobahnen;
        attribute Integer Mindesttempo;
        attribute Array<Integer> VerkehrsdichteProTag;
        relationship Set<Punkt> Auffahrten;
        }
interface Abschnitt {
        extent Abschnitte;
        key (GehörtZuStr.StrBez, ANr);
        attribute Integer ANr;
        attribute String Straßenbelag;
        attribute Integer AnzSpuren;
        attribute Integer Tempolimit;
        relationship Polylinie HatVerlauf inverse Polylinie::BeschreibtA;
        relationship Straße GehörtZuStr inverse Straße::HatAbschnitte;
        relationship Set<Brücke> HatBrücken inverse Brücke::GehörtZuA;
        relationship Set<Stadt> FührtDurch inverse Stadt::LiegtAn;
        Real ALänge ( );
        }
interface Polylinie {
        extent Polylinien;
        attribute Integer AnzPunkte;
        relationship List<Punkt> HatStützpunkte inverse Punkt::GehörtZuLinie;
        relationship BeschreibtA inverse Abschnitt::HatVerlauf;
        relationship BeschreibtF inverse Fluß::HatVerlauf;
        Real Länge ( );
        void Begradigung (in Punkt, in Punkt) Raises (Zu_große_Anschlußkrümmung);
        Boolean Schneidet (in Polylinie);
        Boolean SchneidetBox (in Rechteck);
        Boolean LiegtInBox (in Rechteck);
        }
interface Punkt {
        extent Punkte;
        attribute Real X;
        attribute Real Y;
        relationship Polylinie GehörtZuLinie inverse Polylinie::HatStützpunkte
        Boolean LiegtInBox (in Rechteck);
        Boolean LiegtInRegion (in Polygon);
        }
```

```
interface Polygon: Polylinie {
        extent Polygone;
        relationship Punkt Zentrum;
        relationship Stadt BeschreibtS inverse Stadt::Stadtgrenze;
        relationship Land BeschreibtL inverse Land::Landesgrenze;
        Real Fläche ( );
        }
interface Fluß {
        extent Flüsse;
        key Flußbez;
        attribute String Flußbez;
        attribute Array<Real> Verschmutzungsgrad;
        relationship Punkt Quelle;
        relationship Punkt Mündung;
        relationship Fluß MündetIn inverse Fluß::HatZuflüsse;
        relationship Set<Fluß> HatZuflüsse inverse Fluß::MündetIn;
        relationship Polylinie HatVerlauf inverse Polylinie::BeschreibtF;
        relationship Set<Brücke> FließtUnter inverse Brücke::ÜberquertF;
        }
interface Brücke {
        extent Brücken;
        attribute Real Höhe;
        attribute Real Spannweite;
        relationship Punkt Anfang;
        relationship Punkt Ende;
        relationship Fluß ÜberquertF inverse Fluß::FließtUnter;
        relationship Abschnitt GehörtZuA inverse Abschnitt::HatBrücken;
        }
interface Stadt {
        extent Städte;
        attribute String Stadtname;
        attribute Integer Einwohnerzahl;
        relationship Person Bürgermeister;
        relationship Set<Stadt> Partnerstädte inverse Stadt::Partnerstädte;
        relationship Land GehörtZuL inverse Land::HatStädte;
        relationship Polygon Stadtgrenze inverse Polygon::BeschreibtS;
        relationship Set<Abschnitt> LiegtAn inverse Abschnitt::FührtDurch;
        Real Einwohnerdichte ( );
        void Eingemeindung (Inout Stadt);
        }
interface Land {
        extent Länder;
        key Landesbez;
        attribute String Landesbez;
        relationship Person Staatsoberhaupt;
        relationship Set<Stadt> HatStädte inverse Stadt::GehörtZuL;
        relationship Polygon Landesgrenze inverse Polygon::BeschreibtL;
        Integer Bevölkerung ( );
        }
```

Das ODMG−Modell sieht darüber hinaus eine Reihe von vordefinierten, parameterisierten Klassen (Typgeneratoren, sog. “Templates”) vor, von denen neu definierte Klassen geeignet erben können. Dies sind vor allem die Klasse “Collection<T>” mit einem Klassenparameter T sowie deren Subklassen Set<T>, Bag<T>, List<T> und Array<T>. Durch Einsetzen einer konkreten Klasse für den Parameter T entsteht eine Klasse, die alle Methoden der entsprechenden Kollektionsklasse erbt.

Die wichtigsten Attribute und Methoden der parametrisierten Kollektionsklassen sind:

für Collection<T>:

```
attribute Integer cardinality;
attribute Boolean empty?;
Collection<T> create ();
void delete ();
void insert_element (in T);
void remove_element (in T);
Collection<T> select (in String);  // mit einem Prädikat als Parameter vom Typ String
Boolean exists? (in String);
Boolean contains_element? (in T);
Iterator create_iterator ();
...
```

für Set<T>:

```
Set<T> union (in Set<T>);
Set<T> intersection (in Set<T>);
Set<T> difference (in Set<T>);
Boolean is_subset? (in Set<T>);
Boolean is_proper_subset? (in Set<T>);
Boolean is_superset? (in Set<T>);
Boolean is_proper_superset? (in Set<T>);
...
```

für List<T>:

```
T retrieve_first_element ();
T retrieve_last_element ();
T retrieve_element_at (in Integer);  // mit einer Listenposition als Parameter
void insert_first_element (in T);
void insert_last_element (in T);
void insert_element_after (in T, in Integer);
void insert_element_before (in T, in Integer);
...
```

6.4 Objektorientierte Anfragesprachen

Grundprinzipien der Sprache OQL (Object Query Language)

OQL ist die Anfragesprache des ODMG−Standards. Sie wird in ihren wesentlichen Elementen u.a. vom objektorientierten Datenbanksystem O2 unterstützt.
OQL folgt der syntaktischen Grundidee von SQL, indem es einen SELECT−FROM−WHERE−Block unterstützt. Darüber hinaus aber trägt OQL den Besonderheiten des ODMG−Modells Rechnung, indem es vor allem die Traversierung von Relationships (Objektreferenzen) auf einfache Weise ermöglicht und Methodenaufrufe in Queries erlaubt.

Traversierung von Relationships

Ein Ausdruck der Form C.R mit einem Klassennamen C (bzw. einer entsprechenden Objektvariablen, dem Pendant zu einer Tupelvariablen) und einer Relationship R von der Klasse C zur Klasse Z kann überall verwendet werden, wo der Ausdruck Z verwendet werden darf. Die Bedeutung des Ausdrucks C.R ist die Menge der Objekte der Klasse Z, die von der − ggf. durch ein Prädikat eingeschränkten − Klasse C aus über die Relationship R erreichbar sind.
Einschränkungen einer Klasse C, also die Selektion von Teilmengen, erfolgen über die WHERE−Klausel oder in der Form C[Prädikat].

Damit können ganze *Pfadausdrücke* der Form $C_0.R_1.R_2. \ldots .R_n.A$ gebildet werden, wobei
- R_1 eine Relationship der Klasse C_0 ist,
- R_i eine Relationship von der Klasse C_{i-1} zur Klasse C_i ist und
- A ein Attribut der Klasse C_n ist.

Pfadausdrücke erlauben es, “implizite Joins” auf einfache Weise auszudrücken.

Außerdem ist es möglich, SELECT−FROM−WHERE−Blöcke auch in der SELECT−Klausel oder der FROM−Klausel zu schachteln (wohingegen in SQL diese Schachtelung nur in der WHERE−Klausel erlaubt ist, und dies auch nur in eingeschränkter Form).

Beispiele:

1) Welche Flüsse überquert die B51?

```
SELECT S.HatAbschnitte.HatBrücken.ÜberquertF
FROM Straßen S
WHERE S.StrBez=”B51”
```

oder:

```
SELECT B.ÜberquertF
FROM (   SELECT A.HatBrücken
         FROM (   SELECT S.HatAbschnitte
                  FROM S IN Straßen
                  WHERE S.StrBez=”B51”
               ) AS A
      ) AS B
```

2) Welche Straßen überqueren die Mosel?

```
SELECT S
FROM Straßen S
WHERE S.HatAbschnitte.HatBrücken.ÜberquertF.Flußbez=”Mosel”
```

3) Welche Partnerstädte haben die Städte der Elfenbeinküste?

```
SELECT L.HatStädte.Partnerstädte
FROM Länder L
WHERE L.Landesbez=”Elfenbeinküste”
```

4) In welchen Ländern haben Städte der Elfenbeinküste Partnerstädte?

```
SELECT DISTINCT ( L.HatStädte.Partnerstädte.GehörtZuL )
FROM Länder L
WHERE L.Landesbez=”Elfenbeinküste”
```

5) Ausgabe des Verlaufs der B51:

```
SELECT S.HatAbschnitte.HatVerlauf.HatStützpunkte
FROM Straßen S
WHERE S.StrBez=”B51”
```

bzw. vollständig:

```
SELECT FLATTEN ( A.HatVerlauf.HatStützpunkte )
FROM (   SELECT S.HatAbschnitte
         FROM Straßen S
         WHERE S.StrBez=”B51”
         SORT A IN S.HatAbschnitte BY A.ANr ) AS A
```

Einbezug von Methodenaufrufen

Ein Ausdruck der Form C.M(...) mit einem Klassennamen C und einer Methode M der Klasse C kann überall verwendet werden, wo − unter Berücksichtigung des Resultattyps von M − Attribute von C verwendet werden dürfen.

Beispiele:

1) Wieviele Kilometer hat das gesamte Straßennetz?

```
SELECT SUM ( S.Gesamtlänge() )
FROM Straßen S
```

2) Welche Städte haben eine Fläche von mehr als 30 Quadratkilometern?

```
SELECT S.Stadtname
FROM Städte S
WHERE S.Stadtgrenze.Fläche() > 30
```

3) In welchen Städten mit einer Fläche von mehr als 30 Quadratkilometern haben alle durch die Stadt führenden Straßenabschnitte ein Tempolimit von nicht mehr als 80 km/h?

```
SELECT S.Stadtname
FROM Städte S
WHERE S.Stadtgrenze.Fläche() > 30
AND ( FOR ALL A IN S.LiegtAn : A.Tempolimit <= 80 )
```

4) In welchen Ländern mit mehr als 100 Millionen Einwohnern haben Städte der Elfenbeinküste Partnerstädte?

```
SELECT DISTINCT PL
FROM (   SELECT L.HatStädte.Partnerstädte.GehörtZuL
         FROM L IN Länder
         WHERE L.Landesbez=”Elfenbeinküste” ) AS PL
WHERE PL.Bevölkerung() > 100000000
```

5) Bestimmung aller Straßen mit einem Stützpunkt innerhalb des Rechtecks mit der linken unteren Ecke (1,2) und der rechten oberen Ecke (5,8)

```
SELECT S.StrBez
FROM Straßen S
WHERE   S.HatAbschnitte.Verlauf.HatStützpunkte.LiegtInBox (
         STRUCT(lu:STRUCT(x:1,y:2), ro:STRUCT(x:5,y:8)) )
```

6) Bestimmung aller Straßen, die das Rechteck mit der linken unteren Ecke (1,2) und der rechten oberen Ecke (5,8) schneiden

```
SELECT S.StrBez
FROM Straßen S
WHERE   S.HatAbschnitte.Verlauf.SchneidetBox (
         STRUCT(lu:STRUCT(x:1,y:2), ro:STRUCT(x:5,y:8)) )
```

Berücksichtigung der Generalisierungshierarchie

Eine Query über einer Klasse C wird über C und allen Subklassen von C ausgewertet, sofern darin nur Attribute und Methoden von C verwendet werden.
Durch explizite “Klassenindikatoren” kann eine Anfrageauswertung auch auf bestimmte Subklassen beschränkt werden.

Beispiele:

1) Welche Polylinien, unter Einbeziehung von Polygonen, schneiden das Rechteck mit der linken unteren Ecke (1,2) und der rechten oberen Ecke (5,8)?

```
SELECT P
FROM Polylinien P
WHERE P.SchneidetBox (
      STRUCT(lu:STRUCT(x:1,y:2), ro:STRUCT(x:5,y:8)) )
```

2) Welches ist das niedrigste Mindesttempo auf einer Straße von mehr als 1000 Kilometern Gesamtlänge?
(Annahme: Nur Autobahnen haben eine solche Länge.)

```
SELECT MIN ( ((Autobahn) S).Mindesttempo )
FROM Straßen S
WHERE S.Gesamtlänge() > 1000
```

3) Auf welchen Autobahnen, die die Mosel überqueren, gibt es einen Abschnitt mit einem Tempolimit von 60 km/h ?

```
SELECT A.GehörtZuStr.StrBez
FROM (   SELECT F.FließtUnter.GehörtZuA
         FROM Flüsse F
         WHERE F.Flußbez = “Mosel” ) AS A
WHERE ( EXISTS B IN Autobahnen: B.StrBez = A.GehörtZuStr.StrBez )
AND A.Tempolimit = 60
```

oder:

```
SELECT ((Autobahn) A.GehörtZuStr).StrBez
FROM (   SELECT F.FließtUnter.GehörtZuA
         FROM Flüsse F
         WHERE F.Flußbez = “Mosel” ) AS A
WHERE A.Tempolimit = 60
```

Gruppierung und Aggregation

Gruppierungen führen auf geschachtelte Objekte (mit mengenwertigen Attributen), die eleganter als in SQL weiterverarbeitet werden können.
Das Resultat von

```
... GROUP BY G1: Gruppierungsattribut1, ...
```

hat den Typ

```
set< struct(G1: type_of(Gruppierungsattribut1), ...),
     partition: bag<type_of(Nichtgruppierungsattribute)>>.
```

Aggregationsfunktionen können auch benutzerdefinierte Methoden sein.

Beispiele:

1) Ausgabe der Gesamteinwohnerzahl aller Städte, die an der E5 liegen, gruppiert nach Ländern.

```
SELECT   Land,
         Gesamteinwohner: SUM (SELECT R.Einwohnerzahl FROM partition),
         Stadteinwohner: (SELECT R.Stadtname, R. Einwohnerzahl FROM partition)
FROM (   SELECT S
         FROM Städte S
         WHERE S.LiegtAn.GehörtZu.StrBez = "E5" )
      AS R
GROUP BY Land: R.GehörtZuL
```

2) Sei Bestellungen eine Objektklasse mit Attributen BestNr, KNr, PNr, Summe, ...
Sei ferner Zahlen eine folgendermaßen definierte Klasse (ohne Extent) mit Funktionen für die Berechnung des Mittelwerts, des Medians und der Standardabweichung einer Multimenge von reellen Zahlen:

```
interface Zahlen {
     attribute bag<Zahl: real> Zahlenmultimenge;
     Real mean();
     Real median();
     Real stddev();
     }
```

a) Ausgabe von Mittelwert und Standardabweichung der Bestellungssummen.

```
SELECT Z.mean(), Z.stddev()
FROM (   SELECT B.Summe FROM Bestellungen B ) AS Z
```

b) Ausgabe von Mittelwert und Standardabweichung der Bestellungssummen für jedes einzelne Produkt.

```
SELECT   G.Prod,
         G.Best.mean(), G.Best.stddev()
FROM (   SELECT Prod, (SELECT B.Summe FROM partition) AS Best
         FROM Bestellungen B
         GROUP BY Prod: B.PNr ) AS G
```

6.5 Objektorientierte Erweiterungen von SQL und "Objektrelationale" Datenbanksysteme

Wesentliche Konzepte objektorientierter Datenmodelle und Anfragesprachen werden zunehmend auch in relationale DBS integriert und führen zu sogenannten objektrelationalen Datenbanksystemen. Insbesondere wird die Sprache SQL in diese Richtung weiterentwickelt; der künftige Standard SQL3 wird diese Entwicklung reflektieren.

Komplexe Datentypen mit Typhierarchie

Die Typkonstruktoren ROW, SET, LIST und MULTISET sind orthogonal kombinierbar zur Konstruktion komplexer Datentypen. Mit CREATE TABLE wird eine Ausprägung eines Typs "MULTISET OF ..." angelegt.
Mit der UNDER−Klausel können Typspezialisierungen vorgenommen werden. Bei Anlegung entsprechender Tabellen bezieht die allgemeinere Tabelle die speziellere mit ein, bildet also eine Obermenge (bzw. eigentlich Ober−Multiset).

Beispiele:

```
CREATE ROW TYPE Punkt (X FLOAT, Y FLOAT);
CREATE ROW TYPE Polylinie
   (AnzPunkte INTEGER,
   HatStützpunkte LIST OF Punkt);
CREATE ROW TYPE Straße
   (StrBez VARCHAR(10),
   Ausgangsort VARCHAR(30), Zielort VARCHAR(30),
   HatVerlauf Polylinie);
CREATE TABLE Straßen OF TYPE Straße;

CREATE TYPE Autobahn UNDER Straße
  (Mindesttempo FLOAT);
CREATE TABLE Autobahnen OF Autobahn;

SELECT * FROM Autobahnen
   liefert alle Autobahnen
SELECT * FROM Straßen
   liefert alle Straßen inklusive der Autobahnen
```

In Oracle sind ab Version 8 speziellere Konstruktoren zum Aufbau komplexer Datentypen unterstützt, nämlich Arrays und Nested Tables.

Beispiele:

- ```
  CREATE TYPE Kenntnissetyp AS VARRAY(5) OF VARCHAR(30);
  CREATE TABLE Angestellte
    (AngNr NUMBER,
    Name VARCHAR(50),
    Kenntnisse Kenntnissetyp);
  INSERT INTO Angestellte VALUES (0815, 'Heinz Becker',
    Kenntnissetyp ('Oracle', 'Java', 'Urpils') );
  ```
- ```
  CREATE TYPE Pruefungstyp AS
    (Fach VARCHAR(30), Datum DATE, Note NUMBER);
  CREATE TYPE Pruefungstabellentyp AS TABLE OF Pruefungstyp;
  CREATE TABLE Studenten
    (MatrNr NUMBER,
    Name VARCHAR(30),
    AbgelegtePruefungen Pruefungstabellentyp)
    NESTED TABLE AbgelegtePruefungen STORE AS Pruefungstabelle;

  INSERT INTO Studenten VALUES (4711, 'Jacques Bistro',
     Pruefungstabellentyp (Pruefungstyp ('Praktische Informatik', 11.11.1998, 1.3),
                           Pruefungstyp ('Nebenfach', 24.12.1998, 1.7) ) );
  SELECT P.Note
  FROM THE (   SELECT AbgelegtePruefungen
               FROM Studenten
               WHERE Name = 'Jacques Bistro' ) P
  WHERE P.Fach = 'Praktische Informatik';
  ```

Benutzerdefinierte Funktionen und ADTs

Die Definition von Funktionen ist ähnlich der von Stored Procedures, wobei der SQL3−Standard 4GL−artige Erweiterungen um Kontrollflußkonstrukte u.ä. vorsieht.
Bei Abstrakten Datentypen werden private und öffentliche Attribute und Funktionen unterschieden, also eine echte Kapselung unterstützt.

Beispiele:

```
CREATE TYPE Rechteck (lu Punkt, ro Punkt);
CREATE FUNCTION  SchneidetBox (p Polylinie, r Rechteck) RETURNS BOOLEAN
BEGIN ... END;
CREATE FUNCTION Gesamtlänge (p Polylinie) RETURNS FLOAT
BEGIN ... END;

SELECT S.StrBez, Gesamtlänge(S.HatVerlauf)
FROM Straßen S
WHERE S.Ausgangsort='Saarbrücken'
AND SchneidetBox (S.HatVerlauf, <lu: <X:5.71, Y:6.24>, ro: <X:17.5, Y:22>>)

CREATE TYPE Straße
  (StrBez VARCHAR(10),
  Ausgangsort VARCHAR(30), Zielort VARCHAR(30),
  PRIVATE
    HatVerlauf Polylinie,
  PUBLIC
    FUNCTION Gesamtlänge (s Straße) RETURNS FLOAT
      ...
    END FUNCTION );
CREATE TABLE Straßen OF TYPE Straße;
```

In Oracle sind ab Version 8 Abstrakte Datentypen unterstützt, wobei es allerdings offensichtlich keine privaten Attribute gibt.

Beispiel:

```
CREATE TYPE Kontotyp
   (Kontonr NUMBER, Inhaber VARCHAR(30), Kontostand MONEY,
   MEMBER FUNCTION Tageszinsen (Zinssatz IN NUMBER) RETURN MONEY,
   MEMBER PROCEDURE Einzahlung (Betrag IN MONEY),
   MEMBER PROCEDURE Auszahlung (Betrag IN MONEY) RETURN Fehlertyp );
CREATE OR REPLACE TYPE BODY Kontotyp AS
   MEMBER FUNCTION Tageszinsen (Zinssatz IN NUMBER) RETURN MONEY
   IS BEGIN ... RETURN Kontostand*Zinssatz/100; END;
   MEMBER PROCEDURE Einzahlung (...) IS BEGIN ... END; ...
   END;
```

Objekte und Referenzen

werden in verschiedenen Nuancen diskutiert (z.B. durch Unterscheidung VALUE TYPE vs. OBJECT TYPE) und z.T. in Produkten angeboten (u.a. auch in Oracle Version 8)

Ergänzende Literatur zu Kapitel 6

R.G.G. Cattell, Object Data Management, Addison−Wesley, 2nd Edition, 1994

R.G.G. Cattell (Editor), The Object Database Standard: ODMG−93 Release 1.2, Morgan Kaufmann, 1996

A. Kemper, G. Moerkotte, Object−Oriented Database Management: Applications in Engineering and Computer Science, Prentice−Hall, 1994

G. Lausen, G. Vossen, Objekt−orientierte Datenbanken: Modelle und Sprachen, Oldenbourg−Verlag, 1996

G. Saake, I. Schmitt, C. Türker, Objektdatenbanken: Konzepte, Sprachen, Architekturen, International Thomson Publishing, 1997

M. Stonebraker, Object−relational DBMSs − The Next Great Wave, Morgan Kaufmann, 1996

C.J. Date, H. Darwen, Foundation for Object/Relational Databases: The Third Manifesto, Addison−Wesley, 1998

Oracle 8: The Complete Reference, Osborne, 1998