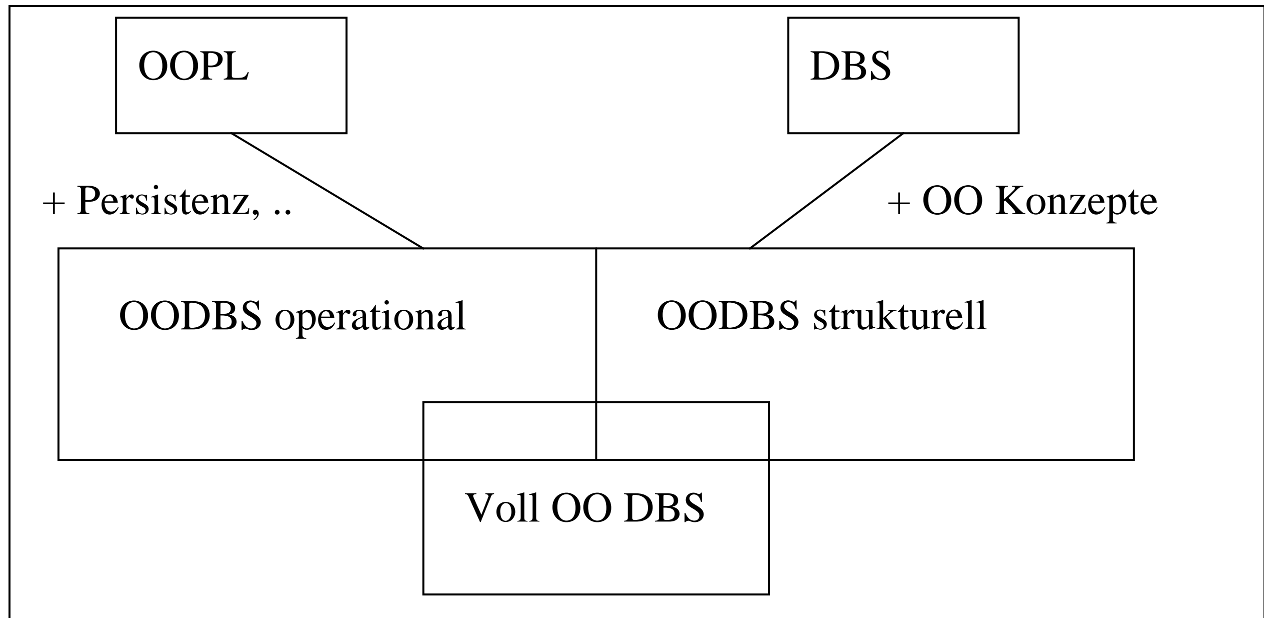
# 2. Objektorientierte Datenbanken

In komplexeren Anwendungen wie CAD, Bürokommunikation und Expertensystemen reichen die Fähigkeiten klassischer Datenbankmodelle und auf ihnen basierender Systeme nicht aus. Objektorientierte Modelle und Datenbanksysteme sind in den letzten Jahren in der Datenbankforschung als adäquate Unterstützung für diese Anwendungen in den Mittelpunkt gerückt.




Während das Relationenmodell quasi "mit einem Schlag" definiert wurde (Arbeiten von Codd), entwickelten sich die Datenbankmodelle für die auf objektorientierten Konzepten beruhenden Systeme weit auseinander. Zum Einen entwickelte sich der *evolutionäre* Ansatz, in dem das relationale Datenmodell in mehreren Schritten über NF2-Modelle zu Datenmodellen, die auf sogenannten komplexen Objekten beruhen. Der andere Ansatz (*revolutionär* genannt) beruht auf Anleihen der Datenbankwelt aus dem Bereich der Programmiersprachen, insbesondere aus den objektorientierten Programmiersprachen. Besondere Probleme werfen vor allem die Optimierung und die Hintergrundspeicherverwaltung auf. Es wurden daher in Anlehnung an objektorientierte Programmiersprachen objektorientierte Datenbankprogrammiersprachen entwickelt, die bessere Voraussetzungen zur Entwicklung von Datenbankanwendungen bieten.
Zwei Richtungen und drei Arten von Systemen können zunächst grob unterschieden werden:

**Persistenz in objektorientierten Programmiersprachen**:
Ausgehend von einer objektorientierten Programmiersprache (SMALLTALK, C++, ...) wird dies schrittweise um Datenbankkomponenten wie die persistente Speicherung von Objekten, Speicherstrukturen für Mengen von Objekten und einigen speziell für Datenbankanwendungen wichtigen Operatoren (z.B. das Durchlaufen von Mengen) erweitert. Hierbei werden die Programmiersprachen-Fähigkeiten zur Beschreibung des Verhaltens der Datenbankobjekte zwar ausgenutzt, die komplexe Struktur der Anwendungsobjekte jedoch wird mit herkömmlichen Mitteln beschrieben (operational oder verhaltensmäßig objektorientierte Datenbanken).
**Erweiterungen von Datenbanksystemen um objektorientierte Konzepte:**

Herkömmliche Datenbankarchitekturen werden schrittweise mit Typkonstruktoren, Objektidentität und anderen objektorientierten Konzepten ausgestattet. In vielen Fällen wird hierbei die Beschreibung des Verhaltens der Objekte vernachlässigt. (*Strukturell objektorientiert*)

# 2.1 KONZEPTE OBJEKTORIENTIERTER SYSTEME

## 2.1.1 Objekte, Klassen, Typen, Methoden und ihre Einkapselung

In der Softwareentwicklung werden Objekte der realen Welt durch ihre abstrakte Beschreibung mittels Methoden und Variablen abgebildet. Jedes Objekt hat ein modellierbares Verhalten und enthält Informationen über seinen eigenen Zustand, der durch die Daten und möglichen Aktionen beschrieben wird. So gehören zu einem Objekt Lastkraftwagen beispielsweise Daten wie Gewicht, Länge, Anzahl_Türen, Farbe, etc. (Variablen) und mögliche Aktionen wie Fahren, Beladen und Entladen (Methoden).
Während Objekttypen in den bisher betrachteten Datenbankmodellen im wesentlichen nur Datenstrukturen definierten, werden in objektorientierten Programmiersprachen neben den Datenstrukturen auch noch alle Funktionen (Methoden) definiert, die mit diesen Daten durchgeführt werden können.
**Kapselung:** Die inneren Daten eines Objektes sind nur über seine Methoden zugänglich. Die Zusammenfassung von Methoden und dazugehörigen Daten nennt man Kapselung. Diese Technik dient dazu, internen Informationen vor anderen Methoden zu verbergen, um sie vor "Störungen" von außen zu schützen.

**Klassen:** Objekte können, wenn sie gemeinsame Muster bezüglich ihrer Beschreibung aufweisen, in Klassen zusammengefaßt werden. Dadurch müssen bestimmte Methoden und dazugehörige Variablen nur einmal in der Klasse beschrieben werden. Objekte, die zu einer Klasse gehören und denen bestimmte Werte für die einzelnen Variablen zugewiesen wurden, nennt man Instanzen oder auch Exemplare.

**Typisierung:** Attribute und Methoden können typisiert oder untypisiert sein. In SMALLTALK sind die Attribute und Methoden ohne Typ, C++, Eiffel und Delphi sind dagegen streng typisierte Sprachen. Da Datenbanksysteme von der Typisierung "leben" - unter anderem auch aus Optimierungs- und Effizienzgesichtspunkten - werden in SMALLTALK-basierten Datenbankmodellen Typen wieder eingeführt.

**Vererbung, Typ- und Klassenhierarchien**
Wie in semantischen Datenbankmodellen durch IS-A-Beziehungen können in objektorientierten Programmiersprachen Objekttypen oder Klassen in Hierarchien angeordnet werden. Durch den Mechanismus der Vererbung kann man eine Klasse von Objekten als Spezialfall einer allgemeinen Klasse definieren. Dabei werden automatisch alle Methoden und Variablendefinitionen von der Oberklasse an die Unterklasse vererbt. Zusätzlich zu den Eigenschaften, die von Unterklassen ererbt wurden, können sie durch eigene Methoden und Variablen eine Spezialisierung erhalten. So kann beispielsweise die Klasse der Lastkraftwagen eine Unterklasse von Fahrzeugen sein. Neben Lastkraftwagen gibt es dann weitere Unterklassen z.B. Fahrrad, PKW und BUS.

**Polymorphismus:** Die Technik der Vererbung in Klassenhierarchien eröffnet die Möglichkeit zum Polymorphismus, bei dem sich ein Programmelement zur Laufzeit auf Exemplare verschiedener Klassen beziehen kann (dynamisches Binden).

## 2.2 KONZEPTE OBJEKTORIENTIERTER DATENBANKMODELLE

### 2.2.1 Der Strukturteil

Im Strukturteil eines objektorientierten Datenbankmodells werden - ähnlich semantischen Datenbankmodellen und typisierten OOPLs - die statischen Aspekte der Objekte aus der zu modellierenden Anwendung und ihre Beziehungen untereinander beschrieben.

## Typkonstruktoren und komplexe Objekte

**Tupelkonstruktor**: ein Tupelkonstrukteur faßt mehrere Komponenten unterschiedlicher Typen zu einem neuen Typ zusammen. Instanz dieses neuen Typs ist dann ein Tupel bestehend aus Instanzen der zugrundeliegenden Typen.
**Mengenkonstruktor**: Ein Mengenkonstruktor erzeugt aus mehreren Elementen eines zugrundeliegenden Typs einen neuen Typ. Instanz dieses neuen Typs ist dann eine Menge bestehend aus mehreren Instanzen des zugrundeliegenden Typs.
**Listenkonstruktor**: Ein Listenkonstruktor erzeugt ebenfalls aus mehreren Elementen eines zugrundeliegenden Typs einen neuen Typ. Instanz dieses neuen Typs ist jedoch eine Liste aus Instanzen des zugrundeliegenden Typs. Eine Liste kann im Gegensatz zur Menge Elemente mehrfach beinhalten und ist geordnet.

**Objektidentität**

Im Relationenmodell können Datenbankobjekte nur durch Schlüssel identifiziert werden. Nimmt man die Schlüsselattribute in andere Relationenschemata auf, so könne andere Objekte auf diejenigen verweisen, die durch diese Schlüsselwerte repräsentiert werden.
Eine der Hauptforderungen für objektorientierte Datenbankmodelle ist die Trennung des in der Datenbank dargestellten Objektes von seinen Werten: jedes Objekt hat seine (unveränderbare) Identität unabhängig von allen Attributswerten, die es beschreiben.

**Klassen und Typen**

Im Datenbankbereich werden Objekte mit ähnlichen Eigenschaften in Mengen zusammengefaßt. Im Relationenmodell gehören Tupel mit der gleichen Struktur zu einer Relation. Die Struktur wird durch das Relationenschema beschrieben.
Im folgenden werden drei Möglichkeiten zur Festlegung eines Datenbankschemas zur Spezifikation von Objekttypen vorgestellt:
**Typ-basiertes Schema**: ein komplexer Typ legt die möglichen Instanzen fest. Dieser Typ ist meistens eine Menge von Tupeln.
**Klassen-typ-basiertes Schema**: Die zusammengehörigen Objekte werden in einer Objektmenge gesammelt und ihnen jeweils ein Wert (die Instanz eines Typs) zugeordnet. Die Definition eines Schemas für den Objekttyp besteht dann aus zwei Teilen: die Angabe einer "Objektfabrik", aus dem man Objekte einer bestimmten Art erzeugen und in einem "Sammelbehälter" zusammenfassen kann, und die Angabe eines Typs für die Zustände dieser Objekte.
**Klassen-basiertes Schema**: Es wird auf die Angabe eines Typs verzichtet, und es wird nur die Definition der Klasse mit dem zugehörigen Sammelbehälter zur Verfügung gestellt. Die Werte oder Objekte, die allen Elementen dieses Sammelbehälters als Beschreibung zugeordnet werden können, sind dann beliebig.

**Beziehungen zwischen Klassen**

Es gibt zwei grundlegende Arten von Beziehungen zwischen Klassen: die **Klasse-Komponenten-Beziehung**, mit der Objekte und ihre Komponentenobjekte verbunden

werden, sowie die **Klasse-Unterklasse-Beziehung**, durch die Objekte in unterschiedlichen Rollen betrachtet werden können.
Ein Komponentenobjekt kann Komponente mehrerer anderer Objekte sein, sogar Komponente von Objekten unterschiedlicher Klassen. Solche Komponentenobjekte nennen wir **gemeinsame Komponentenobjekte**. Komponentenobjekte, die nur in einem Objekt auftauchen dürfen, nennen wir **privat**. Ein Komponentenobjekt kann von der Existenz des ihn umgebenden, zusammengesetzten Objektes abhängen: es wird erst mit ihm erzeugt und auch wieder gelöscht, wenn das zusammengesetzte Objekt gelöscht wird (**abhängige Komponentenobjekte**).
Ist eine Komponente nur von ihrem umgebenden Objekt aus sichtbar, kann man also nur von dem zusammengesetzten Objekt aus auf das Komponentenobjekt zugreifen, so nennen wir das Komponentenobjekt **eingekapselt**.

**Strukturvererbung**

Mit den bisher besprochenen Konzepten können wir die meisten Objekttypen (die bisher bei den betrachteten Datenbankanwendungen auftraten) adäquat modellieren. Die Strukturhierarchie kann man zunächst grob einteilen, indem man die zugrundeliegenden Integritätsbedingungen beziehungsweise die zu vererbenden Konzepte betrachtet. Die Strukturhierarchie kann eine Integritätsbedingung zwischen Objektmengen ausdrücken. Die Strukturhierarchie kann die Vererbung von Interface-Information, wie die Menge der Attribute, oder Implementierungs-Informationen, wie den Typ der Attribute, oder Instanz-Infomationen, wie die zugeordneten Attributwerte steuern.

| Begriffe | Bedeutung in OOPLs | Bedeutung in OODMs |
|---|---|---|
| Klassenhierarchie | Vererbung der Implementierung | Integritätsbedingung an Objektmengen |
| Typhierarchie | Gleiches Verhalten, mehr anwendbare Attribute | Gleiches Verhalten, mehr anwendbare Attribute |
| IS-A-Hierarchie | Integritätsbedingung | Integritätsbedingung und gleiches Verhalten |
| Spezialisierung | wie IS-A-Hierarchie | Festlegung der Domäne von Unterklassen |
| Generalisierung | invers zu Spezialisierung | Festlegung der Domäne von Oberklassen |
| allgemein: | ohne Wertvererbung | mit Wertvererbung |

Die Generalisierung ist hierbei eine Abstraktion, bei der eine Menge von Objekten mit ähnlichen Eigenschaften durch ein generisches Objekt dargestellt wird. Die Generalisierung erlaubt ausgehend von der Untersuchung der Eigenschaften spezifischer Objekte, die Erstellung eines Modells, das diese Objekte durch generische Klassen darstellt.

**Integritätsbedingungen**

Ein optionales Konzept des Strukturteils von OODMs ist die zusätzliche Angabe von Integritätsbedingungen. Integritätsbedingungen sind in klassischen Datenbankmodellen unverzichtbar, auch wenn sie in kommerziellen Datenbanksystem kaum berücksichtigt sind.
Schlüssel sind in Datenbankmodellen nicht nur als Integritätsbedingung, sondern auch als Zugriffshilfe nötig. Der Zugriff auf Objekte in relationalen Datenbanken geschieht am einfachsten über die Angabe eines Schlüsselwertes. In OODMs werden Objekte über einen systeminternen, abstrakten Wert immer eindeutig identifiziert. Dieser Wert ist als

Zugriffshilfe für den Benutzer nicht brauchbar, da er für ihn nicht sichtbar ist. Die Angabe eines Schlüssels, d.h. einer Kombination von Attributen, deren Attributwerte die Objekte eindeutig identifizieren, ist somit in einigen Anwendungen nicht nur als Integritätsbedingung, sondern auch als Zugriffshilfe auf die Objekte erforderlich. Dabei ersetzt ein Schlüssel keinesfalls die Objektidentität: ein Schlüsselwert kann sich ändern, ist aber jetzt ein lokaler Wert des Objekts und wird nicht als Referenz auf das Objekt verwendet (wie im Relationenmodell), die Objektidentität bleibt unveränderlich.
Kardinalitäten: Die Anwendung von Funktionen, Tupel- und Mengenkonstruktoren kann in OODMs eingeschränkt werden. Beziehungen zwischen verschiedenen Klassen können n:m, 1:n oder 1:1 sein (Kardinalitäten von Beziehungen). Beziehungen können durch ein eigenes Konstrukt, durch die Aufnahme allgemeiner Relationen in das OODM oder durch Klassen selbst modelliert werden.

Zwei im Relationenmodell noch wichtige Arten von Integritätsbedingungen sind im OODM überflüssig geworden: Fremdschlüssel und Einschränkungen von Domänen.

### 2.2.2 Der Operationenteil

**Generische und objektspezifische Operationen**
In vielen objektorientierten Datenbankmodellen ist ein Operationenteil nicht enthalten, weil er scheinbar durch ein wesentlich allgemeineres Konzept objektorientierter Systeme ersetzt werden kann: die Methoden. Methoden können für jede Klasse definiert werden und stellen in ihrem Implementierungsteil die Mächtigkeit einer vollen (objektorientierten) Programmiersprache dar.
Im Gegensatz zu den Methoden, die wir objektspezifische Operationen nennen wollen, sollen Anfrageoperationen generische Operationen heißen, da sie fest zum Modell gehören, "generisch" auf alle Arten von Objektmengen angewandt werden können und nicht explizit erzeugt oder definiert werden müssen.
Generische Operationen können Werte aus Zuständen von Objekten extrahieren und in einer Ergebnisinstanz sammeln. Sie können aber auch dynamisch neue Zustandstypen erzeugen, neue Objektmengen zu bestehenden Klassen ermitteln, beziehungsweise dynamisch neue Klassen generieren.
Welche Arten von generischen Operationen gibt es?

- Projektion
- Selektion

Betrachtet man den Strukturteil des OODMs, so können wir die Operationen in zwei grobe Klassen einteilen; Operationen, die die Lage der Klasse innerhalb der Klassenhierarchie verändern und Operationen, die die Lage des Zustandstyps innerhalb der Typhierarchie verändern.

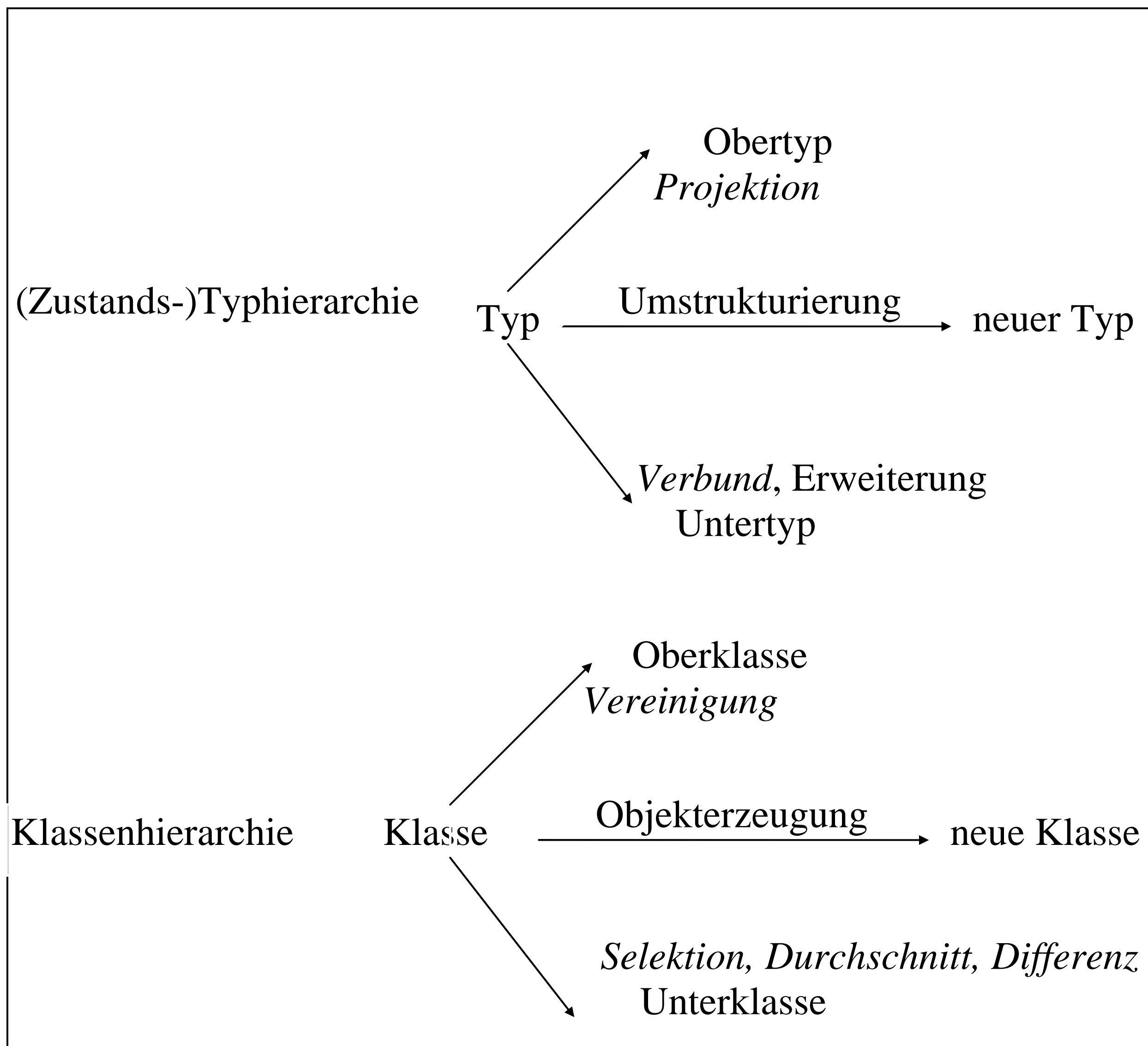



Abb. 2.3: Klassifizierung generischer Operationen für OODMs
(aus: A. Heuer: Objektorientierte Datenbanken)

| **Kriterium** | **Erklärung** |
|---|---|
| Deskriptive Sprache | Die Sprache soll nicht navigierend sein, einen Zugriff auf eine Menge von Objekten ermöglichen |
| Optimierbarkeit | Die Sprache soll nach (etwa algebraischen) Regelsystemen konzeptuell optimierbar sein |
| Effizienz | Die wenigen Grundoperationen sollen mit einer geringen Komplexität implementierbar sein |
| Orthogonalität | Die wenigen Grundoperationen sollen beliebig miteinander kombinierbar sein |
| Erweiterbarkeit | Bei Erweiterung des OODMs soll auch die Sprache leicht erweiterbar sein |
| Abgeschlossen-heit | Das Ergebnis jeder Anfrageoperation soll wieder konsistent im Strukturteil des Datenbankmodells darstellbar sein |
| Adäquatheit | Alle Konstrukte des Datenbankmodells sollen ausgenutzt werden, für alle Strukturen muß es Anfrageoperationen geben |
| Sicherheit | Jede Anfrage soll ein endliches Ergebnis liefern |

| | |
|---|---|
| Vollständigkeit | Es soll zumindest die Mächtigkeit relationaler Anfragesprachen erreicht werden |
| Formale Semantik | Die Operationen der Sprache sollen formal definiert sein |

Tabelle 2.1: Kriterien für generische Operationen und Anfragesprachen
(aus: A.Heuer, Objektorientierte Datenbanken)

**Abgeschlossenheit** bedeutet, daß die Anfrageergebnisse wieder konsistent im Datenbankmodell darstellbar sein müssen. Wird aus einer Objektmenge mit ihren Zuständen etwa eine Teilmenge selektiert, so ist das Ergebnis wieder eine Objektmenge mit Zuständen. Eine konsistente Darstellung der Ergebnisklasse liegt aber nur dann vor, wenn die Klasse auch an einer der Definition der Klassenhierarchie entsprechenden Stelle in der Hierarchie einsortiert wird. Die Schwierigkeit liegt darin, daß wir bereits in der Datenbank vorkommende Objekte zusätzlich in einer Ergebnisklasse unterbringen müssen. Die Objekt-identität bleibt als nach Anfragen erhalten (*objekterhaltende Semantik).* Die auftretenden Schwierigkeiten können umgangen werden, wenn

- die Klasseneigenschaft im Ergebnis nicht erhalten bleiben, sondern zu Relationen über Objekten und Werten ohne eigene Objektidentität übergehen, oder
- neue Objekte erzeugen, deren Objektidentitäten disjunkt sind zu allen weiteren in der Datenbank vorkommenden Objekten.

**Adäquatheit** bedeutet, daß alle Datenmodellkonstrukte in der Anfrage und zu Darstellung des Ergebnisses ausgenutzt werden. Besteht unser Datenmodell also aus komplexen Typen und Klassen, die in einer Hierarchie stehen, so sollen auch die Ergebnisse komplexe Typen und Klassen besitzen können, die mit den anderen Klassen zusammen eine konsistente Hierarchie ergeben. Es reicht also nicht aus, nur Relationen als Ergebnistyp zuzulassen, da dann die anderen Konzepte des OODMs nicht ausgenutzt werden.

### 2.2.3 Assoziativer Objektzugriff

Im folgenden werden zwei komfortable Methoden des assoziativen Zugriffs auf Objekte vorgestellt:

- innerhalb der objektorientierten Programmiersprache mit Hilfe polymorpher Selektionsoperationen und
- interaktiv über eine deklarative Anfragesprache.

Eine Selektionsoperation ermittelt aus einer Kollektion von Objekten genau diejenigen, die ein bestimmtes Selektionsprädikat erfüllen. In einigen DBMS sind *select*-Operationen polymorph vordefiniert, wobei das Selektionsprädikat, das als boolesche Funktion zu implementieren ist, als Parameter übergeben wird.
Im einfachsten Fall wird der *select*-Operation nur eine boolesche Funktion als Parameter übergeben. Die Signatur der (vordefinierten ) Operation sieht wie folgt aus:

**poly** select ($\tau_1 <= \{\tau_2\}$ : $\tau_1 \,||\, (\tau_2 \,||\, \rightarrow$ bool ) $\rightarrow \tau_1$
  **code** select

In dieser Definition sind $\tau_1$, $\tau_2$ Typvariablen, für die beim Aufruf der Operation *select* die aktuellen Typen der Argumente eingesetzt (substituiert) werden. Hierbei sind zwei Einschränkungen zu berücksichtigen:

1. für das erste Argument, den Empfänger, der durch $\tau_1$ vor dem Auftreten von || spezifiziert wird:

a) $\tau_1 <= \{\tau_2\}$ besagt, daß für $\tau_1$ nur ein Typ eingesetzt werden kann, der mengenstrukturiert ist und als Elementtyp $\tau_2$ hat;
b) für das zweite Argument, das unmittelbar auf das erste Auftreten von || folgt:
($\tau_2$ || → bool ) besagt, daß für das zweite Argument nur eine dem Typ $\tau_2$ zugeordnete parameterlose boolesche Funktion eingesetzt werden darf; also eine Operation mit Ergebnistyp *bool* auf beliebigem Empfängertyp $\tau_2$, der aber mit dem Elementtyp des mengenstrukturierten Empfängers der *select*-Operation identisch sein muß, wie aus der gleichen Benennung hervorgeht.

Der Ergebnistyp der *select*-Operation ist dann identisch mit dem des ersten Argumentes, dies ist durch die Wiederverwendung der Typvariablen $\tau_1$ spezifiziert.
Die ebenfalls vordefinierte Implementierung dieser *select*-Operation sieht wie folgt aus:

```
select Code (SelPred) is
  var result: τ1 ;
        candidate: τ2 ;
  begin
        result.create;            |leere Ergebnismenge wird erzeugt
        foreach(candidate  in self )
        if candidate.SelPred
        then result.insert (candidate);
        return result;
  end define selectCode;
```

In dieser Implementierung wird zunächst die Ergebnismenge, auf die *result* verweist, als leere Menge initialisiert. Der Typ der *result*-Menge entspricht dem jeweiligen Mengen-Typ, auf den die *select*-Operation angewendet wird. Danach "läuft" die Variable *candidate* mittels des *Iterators* **foreach** durch die Empfänger-Menge - also **self** - und überprüft, ob das Selektionsprädikat *SelPred* erfüllt ist. Wenn ja, wird *candidate* in *result* eingefügt. Die Menge *result* wird letzendlich übergeben.

Zum Beispiel könnte man - unter Voraussetzung der entsprechenden Typdefinitionen - folgende Selektion durchführen:

```
var meineÄpfel, roteÄpfel: ApfelSet;
        ....
roteÄpfel := meineÄpfel.select (istRot);
```

Hierbei wird vorausgesetzt, daß es auf dem Typ *apfel* - dem Elementtyp von *ApfelSet* - eine entsprechende boolsche Funktion (*istRot :; Apfel* || → *bool*) gibt.

**Assoziativer Zugriff über eine deklarative Anfragesprache**
Insbesondere für die Formulierung interaktiver ad_hoc-Anfragen ist eine deklarative Anfragesprache sinnvoll. Im folgenden wird die Sprache GOMql (eine relationale Anfragesprache basierend auf QUEL als Abfragesprache für INGRES) verwendet.
Eine Anfrage in GOMql hat folgende syntaktische Struktur:

**range** $r_1 : S_1, \ldots, r_m : S_m$
**retrieve** $r_i$
**where** $P(r_1, \ldots, r_m)$

Die $r_j$ sind Bereichsvariablen, die in der **range**-Klausel an mengenwertige Ausdrücke gebunden werden. Dabei ist $S_j$ entweder ein Typname, wodurch $r_j$ an die Extension dieses Typs gebunden wird, oder eine benutzerdefinierte (persistente) Variable, die auf ein Mengen-

objekt verweist. In beiden Fällen wird die Bereichsvariable implizit auf den jeweiligen Elementtyp der Menge eingeschränkt.
Die **where**-Klausel enthält das Selektionsprädikat P, das für jede mögliche Belegung der Bereichsvariablen $(r_1, \ldots, r_m) \in S_1 \times \ldots \times S_m$) überprüft wird. Falls das Selektionsprädikat erfüllt ist, wird die aktuelle Belegung von $r_i$ der Ergebnismenge zugefügt.

## 2.3 KONZEPTE OBJEKTORIENTIERTER DATENBANKSYSTEME

Nachdem wir nun objektorientierte Datenbankmodelle ausführlich besprochen haben, wenden wir uns den weiteren Konzepten eines objektorientierten Datenbanksystems zu. Neben dem Strukturteil, Operationenteil und höheren Konstrukten des OODMs sind in einem vollwertigen Datenbanksystem noch weitere Konzepte zu berücksichtigen. Im manifesto [1] werden folgende spezielle Datenbankmerkmale von einem OODBS gefordert:

- Erweiterbarkeit: Die Möglichkeit, das System auf allen Ebenen um Typen, Funktionen und Speicherstrukturen erweitern zu können.
- Persistenz: Die Fähigkeit des OODBSs, Daten dauerhaft auf externen Speichermedien unterzubringen, ohne explizite Kommandos zur Übertragung von Daten aus dem Hauptspeicher in den externen Speicher.
- Zugriffspfade, Speicherungsstrukturen: Möglichkeit zur Definition von verschiedenen Speicherungsstrukturen für Objekte und von unterschiedlichen Zugriffspfaden über Teile der Zustände von Objekten.
- Transaktionen, Concurrency Control: Die Möglichkeit, Transaktionen aus elementaren Operationen auf der Objektbank zu bilden und die Fähigkeit des Systems, parallel ablaufende Transaktionen zu synchronisieren.
- Recovery: Die Fähigkeit des OODBSs, nach Systemfehlern wieder einen letzten konsistenten Datenbankzustand zu rekonstruieren.
- Datenbank-Programmiersprache: Die Möglichkeit, über die generischen Operationen hinausgehend allgemeine Programme mit Datenbankzugriff schreiben zu können. Die Einbettung der Datenbankoperationen in die Programmierpsrache sollte möglichst die Probleme des "impedance mismatch" vermeiden.

## 2.4 OBJEKTORIENTIERTE DATENBANKSYSTEME

Objektorientierte Datenbanksysteme sind noch relativ neu auf dem Markt. Somit könne viele der bisher vorgestellten Konzepte und Komponenten nur in einigen Prototypen, aber noch nicht in kommerziellen Systemen gefunden werden.

### 2.4.1 GemStone

Das System GemStone ist ein Vertreter der Linie objektorienterte Datenbank-Programmiersprachen und wurde auf SMALLTALK-Basis entwickelt und ist seit Ende 1987 kommerziell verfügbar.
Die ursprüngliche GemStone-Architektur besteht aus der Aufteilung in einen

- Stone-Prozeß (Server): Verwaltung der Objektidentitäten
- Gem-Prozess(e) (Server): Seiten- und Objektpuffer-Verwaltung sowie die Anfragen und Updates

[1] [M.Atkinson, F.Bancilhon, D.DeWitt, K.Dittrich, D.Maier und S.Zdonik. the object-oriented database systsem Manifesto. In: W.Kim,J.-M.Nicolas und S.Nishio, Herausgeber. Proc. 1st International Conference on Deductive uand Object-Oriented Databases, Kyoto. Elsvier,Dezember 1989]

- Anwendungs-Prozeß (Client):

GemStone ist in C implementiert. Schemadefinition und Datenmanipulation wie Anfragen und Pdates werden in der SMALLTALK-ähnlichen Sprache OPAL vorgenommen.
Zusammenfassend kann über den Strukturteil von GemStone gesagt werden:

- Typen und Typkonstruktionen werden über **constraints** und die Set-, Array- und Bag-Klassen simuliert.
- Objekte werden durch Surrogate dargestellt, die in einer Objekttabelle den Speicheradressen zugewiesen werden.
- Die Menge der aktuellen Objekte einer Klasse müssen explizit in einer Variablen der entsprechenden SetOf...-Klasse gesammelt werden. Wie in SMALLTALK darf ein Objekt nur zu genau einer Klasse gehören.
- Beziehungen zwischen Klassen können nur über Komponentenklassen ausgedrückt werden. Unterschiedliche Semantiken sind nicht darstellbar.
- Die Klassenhierarchie ist wie in SMALLTALK eine reine Implementierungshierarchie. Mehrfachvererbung wird nicht unterstützt.

Generische Anfrage- und Update-Operationen gibt es in GemStone nicht, es weden jedoch Standardmehtoden für alle *Collection*-Klassen angeboten, so etwa die aus SMALLTALK bekannten *select:-, reject:- und detect:-* Methoden zum Selektieren von Objekten und *remove:* zum Entferen von Objekten aus einer *Collection*.
Analog zu SMALLTALK sind in OPAL Metaklassen, Methoden sowie die Vererbung und das Overriding von Methoden verwirklicht. Klassen und Methoden werden selbst wieder als Objekte angesehen.

**Persistenz:** Alle Objekte, die in Unterklassen von Object in OPAL erzeugt werden , sind automatisch persistent.
**Pufferverwaltung:**GemStone enthält einen Seiten- und Objektpuffer.
**Speicherstrukturen:**GemStone hat grundlegene Speicherstrukturen für einfache Objekte (Integer-Zahlen, boolsche Werte), Byte-Strings (Real-Zahlen und Zeichenketten), Objektidentitäten und alle Unterklassen von *Bag*.
**Zugriffspfade:** In OPAL gibt es zwei Arten von Zugriffspfaden, den *IdentityIndex* (kann auf beliebige Instanzvariablen mit **constraints** angewandt werden) und den *EqualityIndex* (kann nur auf Instanzvariablen einfacher Objekte oder Strings darstellen).
**Transaktionen und Concurrency Control:**GemStone bietet sowohl optimistisches und pessimistisches concurrency Control.
**Schnittstellen zu Programmiersprachen:** SMALLTALK; C, PASCAL, FORTRAN, COBOL (Konvertierung von GemStone-Objekten zu Daten in den prozeduralen Programmiersprachen ist nicht homogen).
**Schnittstellen zu RDBSs:** verfügbar ist eine Schnittstelle zu Sybase, INGRES, ORACLE und Informix. Das Resultat von SQL-Anfragen in diesen Systemen liefert eine Menge von Tupeln zurück, die in GemStone als Objekte dargestellt werden.
**Interaktive Werkzeuge:** Als Datenbankentwurfswerkzeug bietet GemStone eine *Visual Scheme Designer* (bzw. GeODE = *GemStone Object Development Environment* mit Debugger, Masken- und Programmgeneratoren, 4GL), mit dem ein Datenbankschema zunächst graphisch entworfen und später in die zugehörigen OPAL-Klassendefinitionen umgesetzt werden kann.

### 2.4.2 ONTOS (früher VBASE)

VBASE / ONTOS wurde auf C- und C++-Basis entwickelt und war seit Anfang 1989 erhältlich. ONTOS ist eher ein Datenbankkern auf C++-Basis und weniger ein eigenständiges Datenbanksystem wie VBASE. Der Programmierer hat in ONTOS mehr Eingriffsmöglich-

keiten in die Art der Pufferverwaltung, das Anlegen der Cluster-Strukturen und den Objekttransfer, um die "Performance" der Anwendung zu steigern.
**Strukturteil:** Der Strukturteil besteht im wesentlichen aus Erweiterungen der C++-Klassenhierarchie. Die Objekte aller Unterklassen einer speziell eingeführten Klasse *Object*sind persistent. Zur Simulation der über das C++-Typkonzept hinausgehenden Typkonstruktoren werden die Klasse *Aggregate* und ihre Unterklassen *Set, List* und *Association* (Arrays oder Dictionaries) angeboten.
**Operationenteil:** Der Operationenteil von ONTOS besteht aus mehreren C++-Methoden, die zu den mitgelieferten ONTOS-Klassen gehören.
**Höhere Konzepte:** In ONTOS werden Metaklassen unterstützt. Standardmäßig werden für alle Attribute einer Klasse Anfrage- und Update-Methoden angeboten. Vererbung und Overriding von Mehtoden werden con C++ übernommen.
**Persistenz:** Persistenz ist in ONTOS weder orthogonal noch unabhängig. Das Aktivieren und Deaktivieren eines Objekts liegt in Verantwortung des Programmierers.
**Speicherung:** Auch die Strategie der Pufferverwaltung ist von außen zu beeinflussen, so kann z.B. gewählt werden, ob eine Deaktivierung das Objekt vom client sofort zum Server weitersendet, oder ob dies erst bei einer bestimmten Anzahl von deaktivierten objekten geschieht oder ob gewartet wird, bis ein *Commit* die Übertragung explizit fordert.
**Transaktionen:** ONTOS bietet geschachtelte Transaktionen an und es können unterschiedliche Sperren gesetzt werden.
**Schema-Evolution:** Durch die Existenz der Metaklassen, die dynamisch von einem Programm aus manipuliert werden können, können vielfältige Schema-Evolutions-Mechanismen verwirklicht werden.
**Versionen:** ONTOS erlaubt es, eine Hierarchie von Versionen eines Objektes anzulegen.
**Schnittstellen und Werkzeuge:** ONTOS bietet als Schnittstelle C++ an. Die Werkzeuge *DB-Designer* und *Classify* unterstützen den Datenbankentwurf aus unterschiedlichen Richtungen. *DB-Designer* ermöglicht auch noch die navigation und einfache Manipulatione in der Objektbank.

### 2.4.3 ObjectStore

ObjectStore ist ein Vertreter der Linie *objektorientierte Datenbank-Programmiersprachen* und wird seit 1990 von Object Design Inc. kommerziell vertrieben. ObjectStore ist unter UNIX auf diversen Workstations und unter MS-Windows verfügbar.
Die Client-Server Architektur von ObjectStore erlaubt mehrere Server und mehrere Clients. Der Server verwaltet die Speicherstrukturen, die Sperren von Objekten auf Seitenebene und einen Seitenpuffer. Insgesamt besteht eine ObjectStore-Anwendungsumgebung aus vier Prozessen:

- Client-Prozeß
- Server-Prozeß
- Directory Manager zur für UNIX transparenten Verwaltung des ObjectStore eigenen hierarchischen Systems von Datenbank-Datien,
- Cache Manager auf jedem Rechner im Netz, der die Hauptspeicherverwaltung zusammen mit dem virtuellen Speichermanagment des Betriebssystems übernimmt. Er steuert auch die Kommunikation zwischen Server und Client.

ObjectStore kann in drei verschiedenen Modi betrieben werden:

- aus C-Programmen mit Aufrufen von C-Bibliotheksfunktionen,
- aus C++-Programmen mit Aufrufen von C++-Bibliotheksfunktionen, die wahlweise mit oder ohne generische Klassen angeboten wird,

- mit einer eigenen C++-Erweiterung, der DML, die C++ mit generischen Klassen , Anfrageausdrücken und Ausnahmebehandlung anreichert.