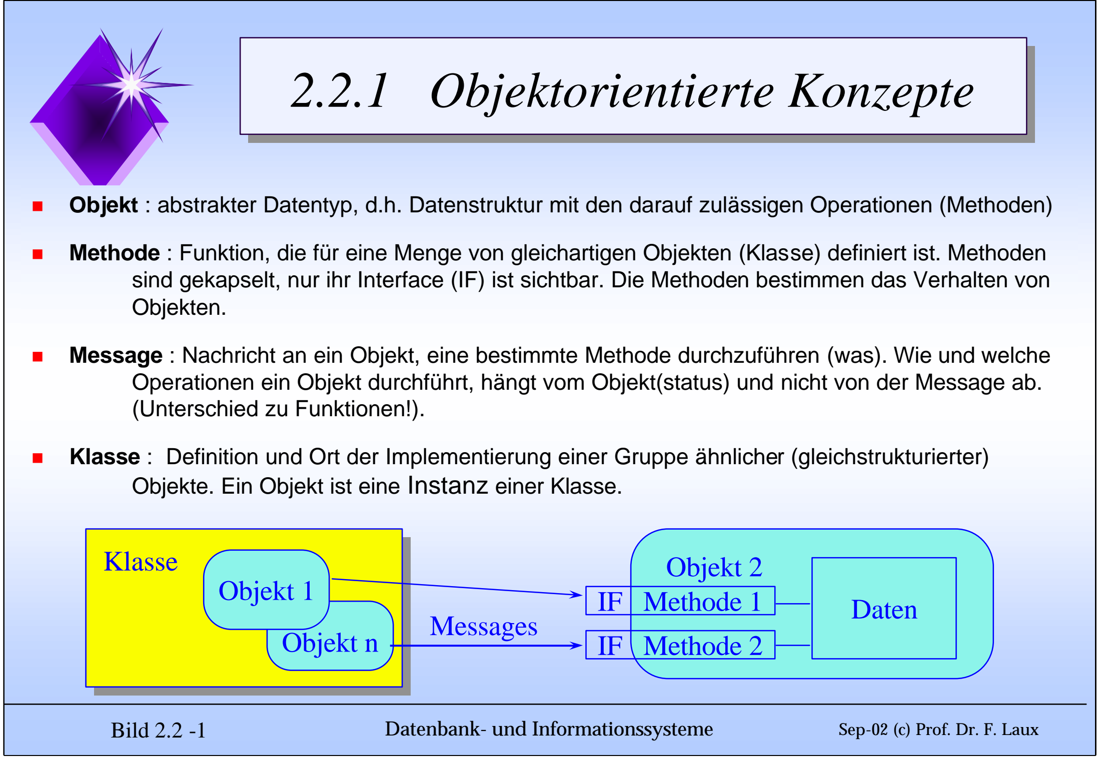


Beim ERM werden die Daten zeitunabhängig modelliert (*statischer Aspekt*). Die Entitäten unterliegen jedoch auch zeitlichen Veränderungen, z.B. kann sich der Wert eines Attributs im laufe der Zeit ändern (*dynamischer Aspekt*).

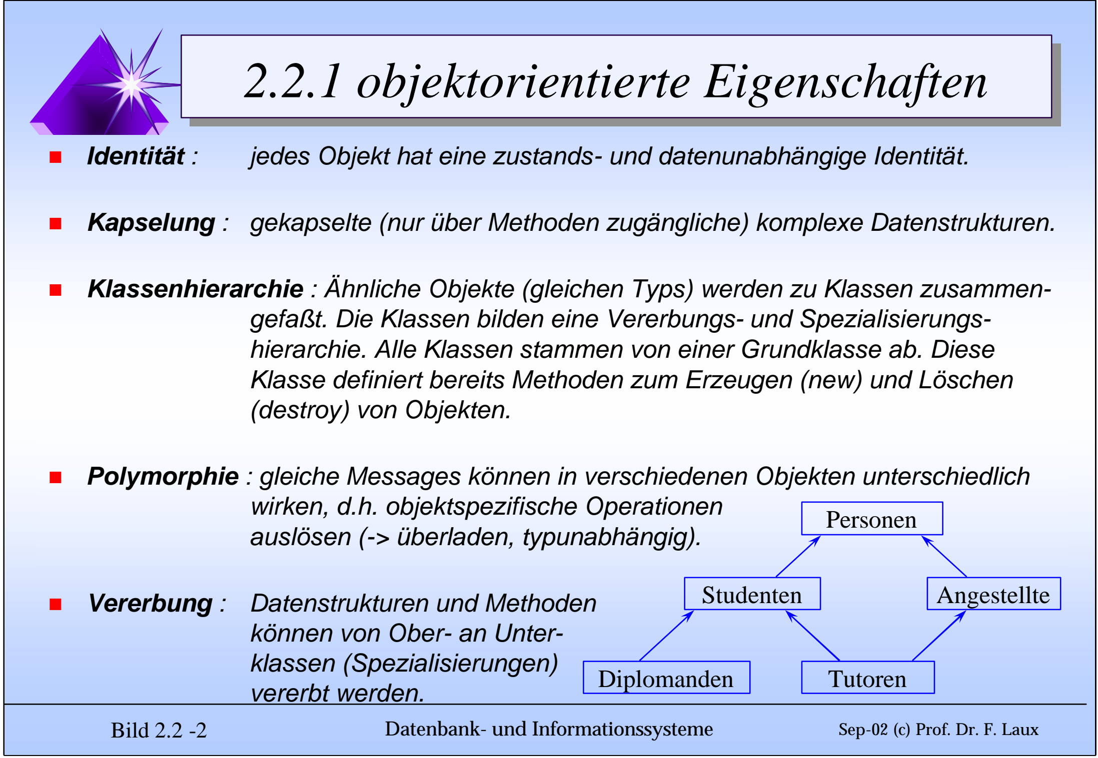
Objektorientierte Modelle betrachten beide Aspekte gleichgewichtig. Demzufolge betrachten wir Objekte als Instanzen von abstrakten Datentypen, d.h. sie besitzen Daten, die durch ihre eigenen Operationen (Methoden) manipuliert werden können. Im Gegensatz zu Entitäten sind Objekte aktiv, d.h. sie können Methoden auf ihren Daten ausführen.

Zur Einführung stellen wir einige objektorientierte Konzepte vor: Im Unterschied zur traditionellen Programmierung (Sprachen der 3. Generation) verwendet man nicht mehr die Funktion als Konstruktionselement eines Programms sondern das Objekt. Man betrachtet ein objektorientiertes (oo) Programm als eine Anzahl kommunizierender Objekte.

Ein Objekt versteckt (*kapselt*) seine Daten vor dem direkten Zugriff und erlaubt einen Zugriff nur über seine eigenen Operationen, die in der oo Welt Methoden genannt werden. Um einen Wert zu ermitteln oder zu verändern, muss eine Methode des Objekts verwendet werden. Diese wird durch eine Nachricht (message) an das Objekt aktiviert. Hier gibt es einen prinzipiellen Unterschied zur Funktion: nicht der Absender einer Nachricht, sondern der Empfänger bestimmt durch seine Methode wie die Operation erfolgt (vgl. Polymorphie).

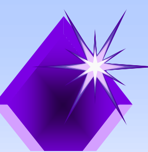
Bei einem korrekt definierten Objekt ist es unmöglich, dieses auf inkonsistente Weise zu verändern, da es die Manipulation ja selbst kontrolliert. Diese Eigenschaft ist für ein DBS zur Konsistenzsicherung sehr nützlich. Das oo Modell integriert sowohl statische Strukturen als auch Methoden zur konsistenzerhaltenden Datenmanipulation (dynamische Eigenschaften, Verhalten). Damit haben wir ein echtes semantisches Modell.

Ähnliche Objekte (d.h. vom gleichen Typ) können zu einer Klasse zusammengefasst werden. Die Klasse ist der Ort der Definition und der Erzeugung von Objekten.



Objekte besitzen fünf wesentliche Eigenschaften:

- Jedes Objekt besitzt eine *Identität*, die von seinen Eigenschaften (Daten) unabhängig ist. Selbst wenn ein Objekt die Klasse wechselt, bleibt seine Identität erhalten.
- Die Daten eines Objekts sind dem direkten Zugriff verborgen (*gekapselt*) und nur über die Methoden seiner Klasse zugänglich.
- Die Klassenhierarchie bietet uns ein neues, übersichtliches Strukturierungsmittel. Die Klassen ordnen sich baumartig in einer Spezialisierungs- und Vererbungshierarchie. Ein Objekt einer Unterklasse ist immer auch ein (spezielles) Objekt einer seiner Oberklassen. Alle Objekte stammen von einer Grundklasse (object) ab, welche jedem Objekt eine Grundfunktionalität verleiht.
- Die Übernahme von Daten und Methoden der Oberklasse durch die Unterklasse (Vererbung) vereinfacht die Klassendefinition und die Modellbildung. Bei der Programmierung spart es Code und fördert die Wiederverwendung.
- Polymorphie vereinfacht ebenfalls die Programmierung (keine Unterscheidung der Zielobjekte beim Sender, das bedeutet, dass keine Case-Statements notwendig sind) und erhöht somit die Lesbarkeit (Wiedererkennung) von Operationen.



2.2.1 Anforderungen an objektorientierte Datenmodelle (ooDM)

- *Anwendung **objektorientierter Konzepte** zur Datenmodellierung mit dem Ziel:*
  - *- **realitätsnäher** (komplexe Strukturen)*
  - *- mit **mehr Semantik** (mehr Objekt- und Beziehungstypen)*
  - *- und mit **Operationen**(Transaktionen)*

  zu modellieren

- *Folgende Anforderungen an ein oo DM sind weitgehend anerkannt (Kriterien des MANIFESTOs (von Atkinson u.a.):*

Notwendige Eigenschaften (golden rules)

1- Objektidentität
2- komplexe Objekte (Erweiterbarkeit)
3- Kapselung und Polymorphie
4- Typen / Klassen-Hierarchie mit Vererbung
5- Datenbankfunktionalität

Optionale Eigenschaften (goodies)

6- Mehrfachvererbung
7- statische Typprüfung
8- lange Transaktionen, Versionen
9- Verteilung

Bild 2.2 -3 Datenbank- und Informationssysteme Sep-02 (c) Prof. Dr. F. Laux

Obwohl die oo Prinzipien unumstritten sind, gibt es kein einheitliches ooM. Das mag daran liegen, dass verschiedene oo Sprachen unterschiedliche Konzepte verwirklichen und diese auch unterschiedlich realisieren. Einige Forschen glauben auch, dass wegen der Erweiterbarkeit des Modells (durch eigen definierte Klassen/Objekttypen) es prinzipiell unmöglich ist, EIN Modell zu haben.

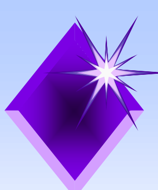
Es gibt also viele mögliche objektorientierte Modelle und sogar einen **Standard der ODMG (Object Data Management Group)**, den wir auf den folgenden Seiten vorstellen werden.

Alle Modelle gehen von Objekten und (abstrakten) Typen bzw. Klassen aus, welche in einer Vererbungshierarchie angeordnet werden. Die Unterschiede liegen in der Art der Vererbung (einfach oder mehrfach). In allen Modellen besteht die Möglichkeit, komplexe Objekte neu zu definieren. Die Objekte besitzen eine Identität und können dauerhaft gespeichert werden (Persistenz). Unterschiede gibt es bezüglich der Typisierung (statisch, dynamisch) und ob auch zur Laufzeit Klassen definiert werden können.

Weitere Eigenschaften werden häufig implementiert: Datenbankfunktionen (Transaktionen, Datensicherung, Multiuserbetrieb), oo Abfragesprache, Versionen.

Spezielle Datenbanken bieten je nach Herkunft: Mehrfachvererbung, statische Typprüfung. Lange Transaktionen werden bei CAD- und CASE-Datenbanken benötigt; und durch die Verteilung und redundante Speicherung (Objektidentität?) der Objekte werden fehlertolerante und hochverfügbare Systeme möglich.

Die auf der Folie genannten Eigenschaften entstammen einem Vorschlag von M. Atkinson u. a., der als 'MANIFESTO' auf der 1. International Conference on Deductive and Object-Oriented Databases in Kyoto (Japan) 1989 veröffentlicht wurde und allgemein als Referenz dient.



Das Objektmodell der ODMG

- Typspezifikation
  - *interface Person { string name();….};*
  - *class Person {attribute string sname;….};*
  - *struct Complex {float re, float im}*
- Extent
  - *Kollektion von Objekten eines Typs*
- Objektidentität
  - *Einmalig innerhalb des ooDBS, durch das ooDBS erzeugt*
  - *Unveränderlich während des gesamten Lebenszyklus eines Objektes*
- Identifikationsschlüssel
  - *Eindeutige (evtl. veränderliche) Eigenschaft eines Objektes*
- Objektname (Alias)
  - *Vom Programm oder Benutzer vergebener Name (Etikett) für ein Objekt*
  - *Häufig als Einstiegspunkt in das ooDBS verwendet*

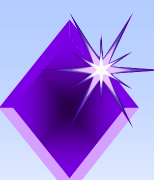
Bild 2.2 -4 Datenbank- und Informationssysteme Sep-02 (c) Prof. Dr. F. Laux

Die Object Data Management Group (ODMG) wurde 1991 durch eine Initiative von Rick Cattell (Sun Microsystems) ins Leben gerufen. Der Verein besteht aus stimmberechtigten Mitgliedern (Firmen, welche die ODMG Spezifikation umsetzen) und Gutachtern (reviewer members, Firmen oder Personen, die ein ernsthaftes Interesse an der Arbeit der ODMG haben). Das erste Release der Spezifikation wurde 1993 herausgegeben. 1994/95 wurde eine Zusammenarbeit mit der Object Management Group (OMG) vereinbart und die Persistenz - und Query Services des Object Request Brokers (ORB) für das Objektmodell übernommen. Inzwischen repräsentieren die Mitglieder nahezu die gesamte Objektdatenbank-Industrie und die Ergebnisse der ODMG sind in den Java Community Process und das INCITS X2H2 (SQL) Komitee eingeflossen. Daher kann man die Spezifikationen der ODMG als Quasi-Standard betrachten.

Das Objektmodell der ODMG unterscheidet zwischen **eingebauten Datentypen** (literal datatype), **Klasse** und **Interface**. Eingebaute Datentypen definieren nur den abstrakten Zustand eines Datentyps, z.B. einer komplexen Zahl in kartesischen Darstellung *struct Complex{float re, float im}* mit Fließkommazahlen für Real- und Imaginärteil. Wird diese Struktur in einer Klasse gekapselt und mit geeigneten Methoden für die Rechenoperationen mit komplexen Zahlen versehen, so sind damit der Zustand als auch das Verhalten implementiert. Wenn nur das Protokoll (die Signaturen der Methoden) festgelegt wird, handelt es sich um eine Schnittstellendefinition für eine Klasse (Interface), welche nur das Verhalten einer Klasse beschreibt.

ÜA: Spezifizieren Sie ein Interface in Java für eine Klasse *Complex*. Implementieren diese zwei mal: 1. mit einem Datentyp *struct Complex1 {float re, float im}* (kartesische Koordinaten), 2. mit einem Datentyp *struct Complex2 {float r, float phi}* (Polarkoordinaten)

Die Objekte eines Typs innerhalb des OODBS werden zu einer getypten, benannten Kollektion (**Extent**) zusammengefasst. Mit Hilfe eines Iterators (prozedural) oder einer Query (deskriptiv) kann das Extent nach Objekten durchsucht werden. Jedes Objekt besitzt eine *einmalige*, *unveränderliche* **Objektidentität (OID)** während seines gesamten Lebenszyklus. Neben der OID können für ein Objekt weitere Identifikationsmöglichkeiten existieren. Diese können aus Attributen des Objektes abgeleitet sein (**Identschlüssel**) oder ein frei wählbarer Name (**Objektname, Alias**) für das Objekt sein. **Beispiel (siehe 2.2-7)** : Einem Objekt des Typs *Buch* kann der Alias „*Mein_Lieblingsbuch*" zugewiesen werden oder es kann durch seine *ISBN* (Attribut des Objekts) identifiziert werden. Zusätzlich besitzt das Buch eine „ewige" *Objektidentität (OID).*



Objekte des ODMG Modells (1/2)

- Objekterstellung
  - interface ObjectFactory { Object new(); }
- Objekt-Interface
  - interface Object {

```
enum      Lock_Type {read, write, upgrade};
void      lock (in Lock_Type mode) raises (LockNotGranted);
boolean   try_lock (in Lock_Type mode);
boolean   same_as(in Object anObject);
Object    copy();
void      delete();   }
```

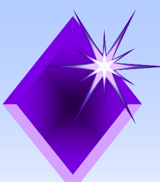
- Lebenszyklus
  - transient; persistent
- Fehlersituationen
  - Exception DatabaseClosed{};
  - Exception TransactionInProgress{}; TransactionNotInProgress{};
  - Exception IntegrityError{}; LockNotGranted{};

Bild 2.2 -5 Datenbank- und Informationssysteme Sep-02 (c) Prof. Dr. F. Laux

Objekte werden dadurch erzeugt, dass eine *Konstruktormethode* ***new()*** eines Object Factory Objekts aufgerufen wird. Ein nicht persistenzfähiges Objekt kann nur **transient** (nicht dauerhaft) sein, d.h. es existiert nur solange wie sein Kontext (Methode, Kollektion, Thread, Prozess). Ein persistenzfähiges Objekt kann transient oder **persistent** (dauerhaft) sein; dies ist von seiner Beziehung zur Datenbank abhängig. Wenn es aus der Datenbank heraus erreichbar ist (persistente Referenz), dann wird es automatisch persistent.

Jedes persistente Objekt (d.h. in der Datenbank gespeicherte Objekt) versteht eine Reihe von Methoden, die es von einem Interface *Object* geerbt hat. Dazu gehören die Einstellung des Sperrmodus, Vergleichsoperationen und Methoden zum Kopieren und Löschen von Objekten. Eine Speichermethode ist nicht erforderlich, wenn ein Objekt **persistenzfähig** ist und es eine persistente Referenz auf das Objekt gibt (*transitive Persistenz*). In diesem Fall wird das Objekt automatisch gespeichert werden. Wenn es keine persistente Referenz gibt, kann innerhalb einer Transaktion die Nachricht ***makePersistent(Object obj)*** an das Datenbankobjekt geschickt werden, um ein persistenzfähiges Objekt in seinem Extent zu speichern.

Alle Datenbankoperationen auf persistenten Objekten müssen innerhalb einer Transaktion ausgeführt werden, sonst tritt eine **TransactionNotInProgress** Ausnahmebedingung auf. Wenn ein neues Transaktionsobjekt erzeugt wird, ist dieses anfänglich „geschlossen“. Durch die ***begin()***-Methode wird die Transaktion gestartet. Ein zweiter Aufruf liefert eine Ausnahme (**TransactionInProgress**). Entsprechendes gilt für ***commit()*** oder ***abort()***, um die Transaktion abzuschließen.



Objekte des ODMG Modells (2/2)

- Typ-Hierarchie
  - 2 nahezu parallele Hierarchien für ‚Literal_type' (Datentyp) und Object_type
  - **Atomare Objekte**
    - Es gibt keine „eingebaute" atomare Objektypen
  - **Atomare Datentypen (Literal_type)**
    - Long, short, float, double, boolean, octet, char, string, enum<>
  - **Kollektions-Objekte/Datentypen**
    - Kollektionen von Objekten des selben Typs
    - Buit-in: Set<t>, Bag<t>, List<t>, Array<t>, Dictionary<t,w>; t = Typ; w=Wert
  - **Strukturierte Objekte/Datentypen**
    - Built-in: Date, Interval, Time, Timestamp

Bild 2.2 -6 Datenbank- und Informationssysteme Sep-02 (c) Prof. Dr. F. Laux

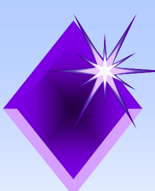
Die ODMG definiert zwei nahezu gleiche Typ-Hierarchien, eine für **Objekte** und eine für **Datentypen**. Aus objektorientierter Sicht ist hier anzumerken, dass damit genau genommen ein hierarchisch strukturiertes, **hybrides** Modell bezüglich der zu speichernden Dinge (Objekte, Daten) definiert wird. Somit ist es möglich, neben Objekten auch Daten zu verwalten. Der einzige Unterschied in der Struktur ist, dass es keine vordefinierten atomare Objekttypen gibt. Hingegen sind die üblichen elementaren Datentypen vordefiniert.

Bei den Kollektions-Typen finden sich Menge (Set), Multimenge (Bag, = Menge mit Dubletten), Liste (List) und Array. Ein besonders wichtiger Typ ist das Dictionary. Es ist eine Menge von **Assoziationen** (Key-Value-Paare), wobei sowohl die **Keys** als auch die **Values** Objekte oder Daten sein können. Die Keys müssen eindeutig sein. Dadurch ist ein Direktzugriff auf jeden Value über seinen Key möglich.

Beispiel eines Dictionaries:

Key<Person>	**Value<string>**
Hans Maier	24 Jahre
W. Müller	22 Jahre
Kurt Bart	24 Jahre
Kim Blake	20 Jahre
Eva Lang	blond
Chris Date	61 Jahre
Joel Vatz	44 Jahre
Pam Pugh	22 Jahre

ÜA: Ist im o.g. Dictionary der **Value** „blond" erlaubt?
Wenn ja, wie müsste der Datentyp für **Value** gewählt werden, damit nur Zeitintervalle zulässig sind?



Vordefinierte Datentypen

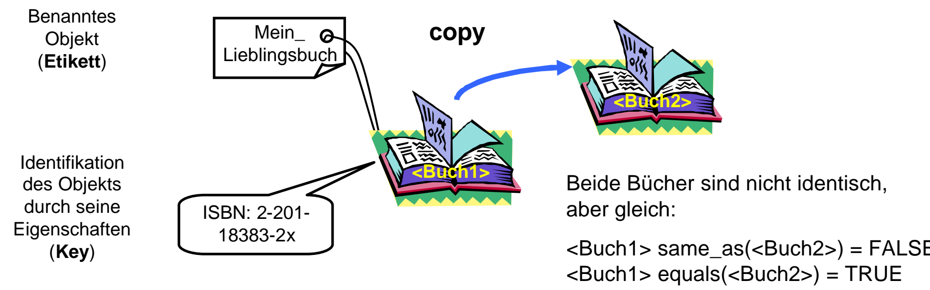
- *Beispiele für vordefinierte Datentypen (Literal_types)*
  - *Einfache Datentypen*
    - *Float: 3.14, String: „hallo“, Complex: (1,i),*
    - *enum gender {male, female}*
  - *Strukturierte Datentypen*
    - *struct Quader {float l, float b, float h}*
  - *Kollektionen eines Datentyps*
    - *set<Primzahlen>*
- *Eigenschaften*
  - *Implizit vorhanden*
  - *Keine Methoden*
  - *Keine Identität*
  - *Können mit ‚copy()‘ kopiert werden*
  - *können mit ‚equals(<literal>)‘ verglichen werden ? same_as*

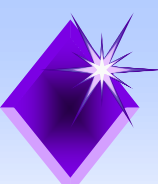
Bild 2.2 -7 Datenbank- und Informationssysteme Sep-02 (c) Prof. Dr. F. Laux

Die vordefinierte Datentypen gliedern sich in **einfache, strukturierte und Kollektions-Datentypen**. Die eingebauten Datentypen werden im Unterschied zu den Objekttypen mit kleinem Anfangsbuchstaben geschrieben. Strukturierte Daten werden mit dem **struct**-Typ erzeugt. Für die Kollektionstypen stehen Konstruktoren **set, bag, list, array** und **dictionary** zur Verfügung.

Zu den Datentypen werden keine Methoden definiert, d.h. es ist durch eine Punkt-Notation möglich, auf die Strukturelemente zuzugreifen. Sie besitzen auch kein Identität, werden nicht zur Laufzeit erzeugt, sondern sie sind durch die Definition bereits vorhanden. Mit der **copy**()-Funktion können Daten kopiert werden. Beim Vergleich zweier Daten ist die Funktion **equals**(**<literal>**) zu verwenden. Zwei gleiche Datenstrukturen, welche die gleichen Werte aufweisen (d.h. den gleichen Zustand haben), sind dann in diesem Sinne gleich. Werden jedoch zwei Objekte im gleichen Zustand mit **same_as**(**<Object>)** verglichen, so erhalten wir das Ergebnis FALSE, da es sich um 2 Objekte handelt. Technisch gesehen vergleicht **same_as** die OID der Objekte, während **equals** die Daten vergleicht. Die OID ist nicht mit einer Referenz auf ein Objekt zu verwechseln. Es können durchaus zwei Referenzen (Namen) für ein und dasselbe Objekt existieren.

Beispiel: Benanntes Objekt, Key und OID





Modellierungssprache

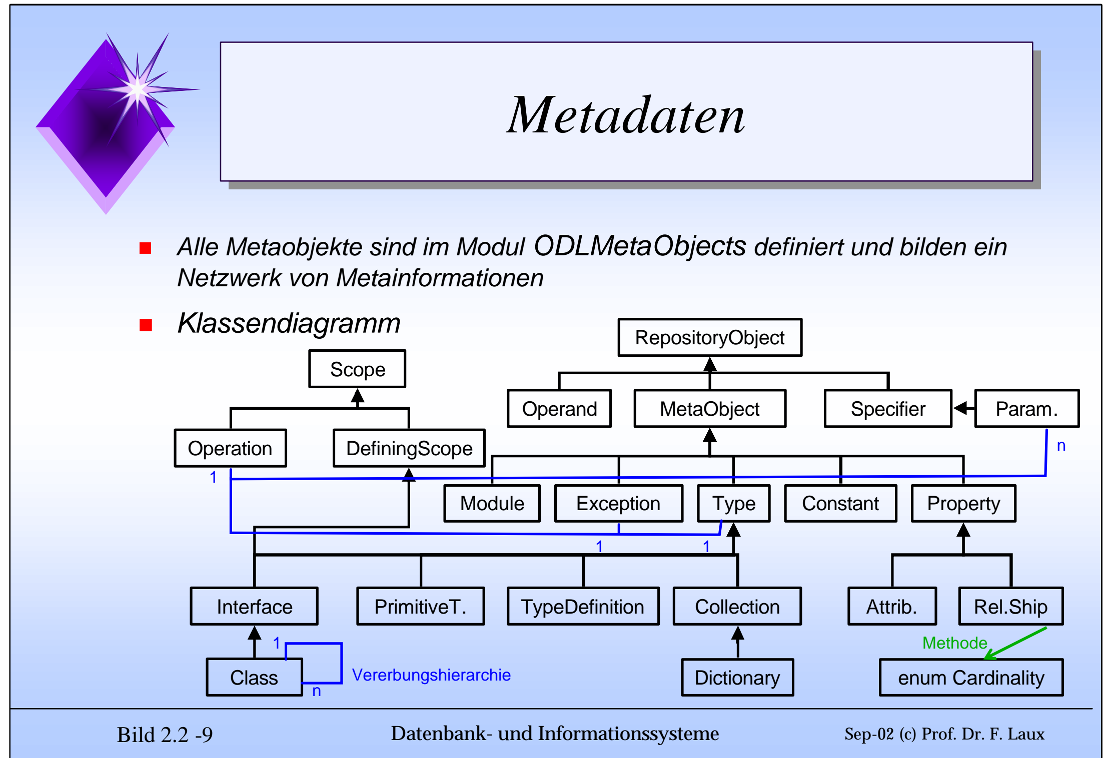
- Object Definition Language (ODL)
- Eigenschaften
  - Modelliert Zustand, Beziehungen, Verhalten (inkl. Ausnahmeverhalten)
  - Programmiersprachenunabhängig
  - kompakt
  - In erweiterter Backus-Naur-Form (EBNF) definiert
  - Anbindungen (Mapping) an die Sprachen Java, C++, SQL:1999, Smalltalk
- Beispiel

```
class Professor extends Employee ( extent professors ) {
        attribute enum Rank (full, associate, assistant} rank;
        relationship Department works_in
            inverse Department::has_professors;
        relationship set<Sections> teaches
            inverse Section::is_taught_by;
        short grant_tenure() raises (IneligibleForTenure);
}
```

Bild 2.2 -8 Datenbank- und Informationssysteme Sep-02 (c) Prof. Dr. F. Laux

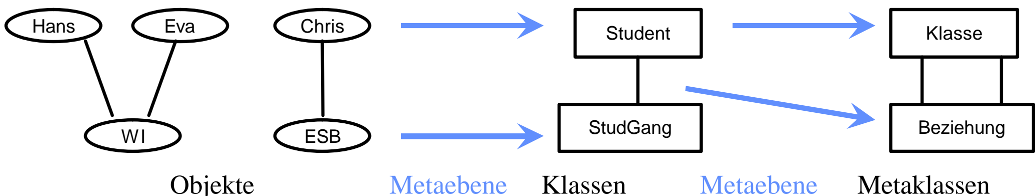
Das Objektmodell der ODMG wird in einer eigenen Definitionssprache, der **Object Definition Language (ODL)** beschrieben. ODL ist unabhängig von einer Programmiersprache, daher sind für die Implementierung eine Anbindungen (**Binding**) zwischen ODL und der jeweiligen Programmiersprache vorzunehmen.

Die Definition erfolgt formal in der erweiterten **Backus-Naur-Form (EBNF).** Sie fällt recht kurz (5 Seiten Definition) und praxisnah aus, so dass sie leicht zu implementieren ist. Zum Beispiel sind nur binäre Beziehungen (**relationship**s) erlaubt. Im Hinblick auf die **Konsistenz** der Datenbank ist vor allem die Spezifikation von Ausnahmebedingungen wichtig. Kann eine Operation aus Konsistenzgründen nicht ausgeführt werden, wird eine Ausnahme (**Exception**) erzeugt, welche die Ausnahmebehandlung auslöst. Im Datenbankschema werden die Methoden für die Ausnahmebehandlung durch das Schlüsselwort „**raises**“ angezeigt.



Um das Metakonzept leichter zu verstehen, erklären wir es an einem einfachen Beispiel:

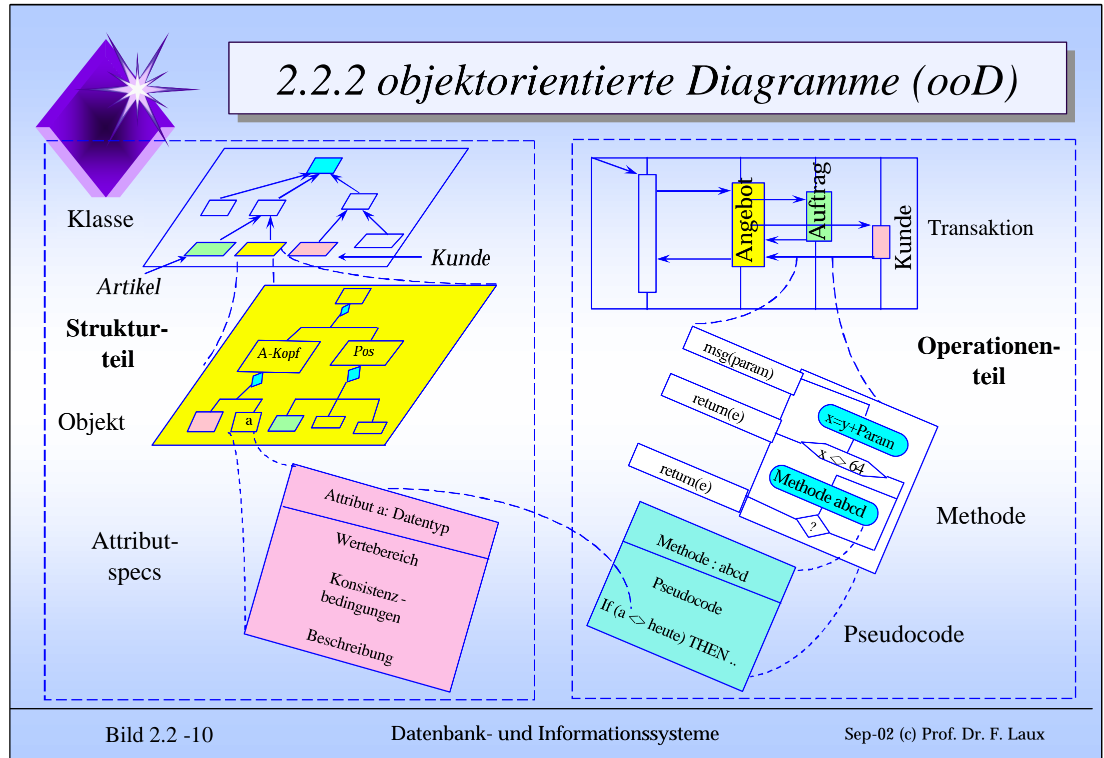
Beginnen wir mit den Objekten Hans, Eva, Chris, WI und ESB. Diese können wir „*klassifizieren*"; Hans, Eva und Chris sind Studenten, WI und ESB sind Studiengänge. Hans und Eva studieren WI, Chris studiert ESB. *Student* und *Studiengang* sind eine Klasseneinteilung für Objekte mit bestimmten gemeinsamen Eigenschaften. Mit diesen Klassen können wir eine Situation allgemein beschreiben, ohne konkrete Studenten oder Studiengänge benennen zu müssen. Diese Beschreibung sind Metadaten für die Objekte. Nun können wir die Klassen *Student*, *Studiengang* und ihre Beziehung zueinander wiederum beschreiben, d.h. heißt wir legen fest, um was es sich bei diesen Dingen (Klassen, Beziehungen) handelt. Wir schaffen also Metaklassen, die festlegen, was wir unter Klasse und Beziehung verstehen und wie sie zusammenhängen, z.B. dass nur binäre Beziehungen erlaubt sind.. Die Metaklassen definieren also mögliche Klassen und ihre mögliche Strukturen.



Objekte Metaebene Klassen Metaebene Metaklassen

Das Klassendiagramm des ODMG Metamodells beschreibt die Objekte in der Datenbank. Die Klassen werden von „**Scope**" oder/und „**RepositoryObject**" abgeleitet. RepositoryObject gibt die allgemeinen Eigenschaften und das Grundverhalten für alle Objekte der Datenbank (Repository) vor. Dazu gehören, die Operanden, Bezeichner, Metaobjekte wie Module, Exceptions, Typen, Konstante usw. Die Klasse „Scope" definiert eine Namenshierarchie für die Metaobjekte im Repository. Scope besitzt ein Methode:
bind(in string <name>, in MetaObject <object>) raises (DuplicateName).

Damit können neue Metaobjekte hinzugefügt werden und mit einem globalen Namen versehen werden, ähnlich wie das Datenbankobjekt mit bind(…) persistente Objekte mit einem Namen etikettieren kann. Besonders wichtig sind die von „Type" abgeleiteten Metaklassen Interface, Class, Primitive-Type, Collection, usw. Die Instanzen dieser Objekte sind die im Schema definierten Klassen des ooDBS.



Da ein ooM sowohl Datenstruktur als auch Verhalten beschreiben muss, haben wir zwei Diagrammtypen. Der Strukturteil umfasst die Klassenhierarchie, die Objektstruktur und die Attributspezifikation (Minispecs); der Operationenteil beschreibt Transaktionen, Methoden und Algorithmen bis zum Pseudocode. Jeder Diagrammtyp kann auf drei Abstraktionsebenen dargestellt werden, dadurch ist ein Top-Down oder Bottom-Up Vorgehen möglich.

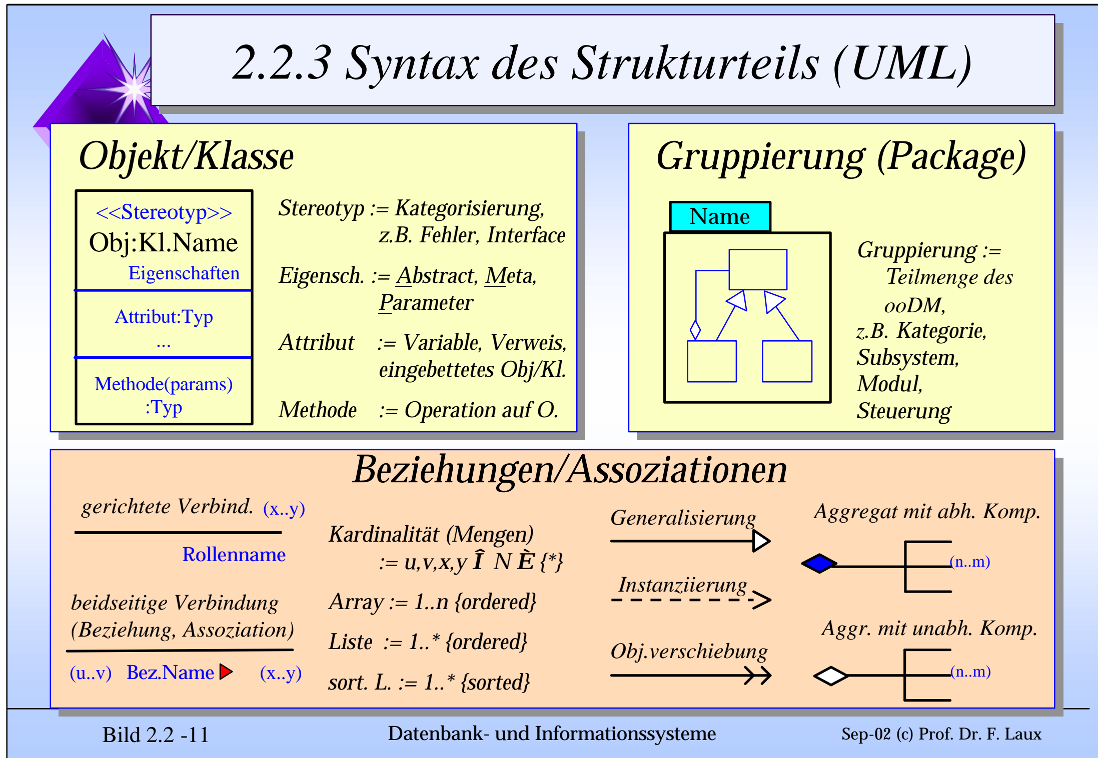
Wie geeignete Strukturen für die Klassen und Objekte gefunden werden können, ist Aufgabe einer Methode (siehe 2.2.5 Methodik der oo Modellierung, Folie 2.2-12). Allerdings kann diese nur Richtlinien und Erfahrungswerte (Daumenregeln) anbieten, da ein oo Modell auch von der fachlichen Sichtweise des Designers abhängt.

Unser Beispiel zeigt Ausschnitte aus einer Auftragsbearbeitung:

Ein Hierarchiepfad könnte z.B. Objekt - jur. Person - Geschäftspartner - *Kunde* sein. Ein Auftrag besteht aus *Auftragskopf (A-Kopf)* und *Auftragspositionen (Pos)*. Ein Attribut besitzt einen Datentyp, der im DBS implementiert sein muß oder es gehört zu einer bereits existierenden Klasse. Als Beispiel einer im Modell implementierten Klasse lässt sich das Datum verwenden. Durch die Hinzunahme benutzerdefinierter Klassen, wird die Erweiterbarkeit des Modells erreicht.

Als Beispiel einer Transaktion greifen wir die Erfassung eines Kundenauftrags heraus. Dabei werden neben dem Auftrag mehrere Klassen benützt (z.B. Angebot, Kunde, Artikel). Die Angebotskonditionen und die Zahlungsbedingungen sind Beispiele für Methoden der Objekte *Angebot* und *Auftrag*.

Die Notation für die Diagramme wird auf den folgenden Seiten vorgestellt.



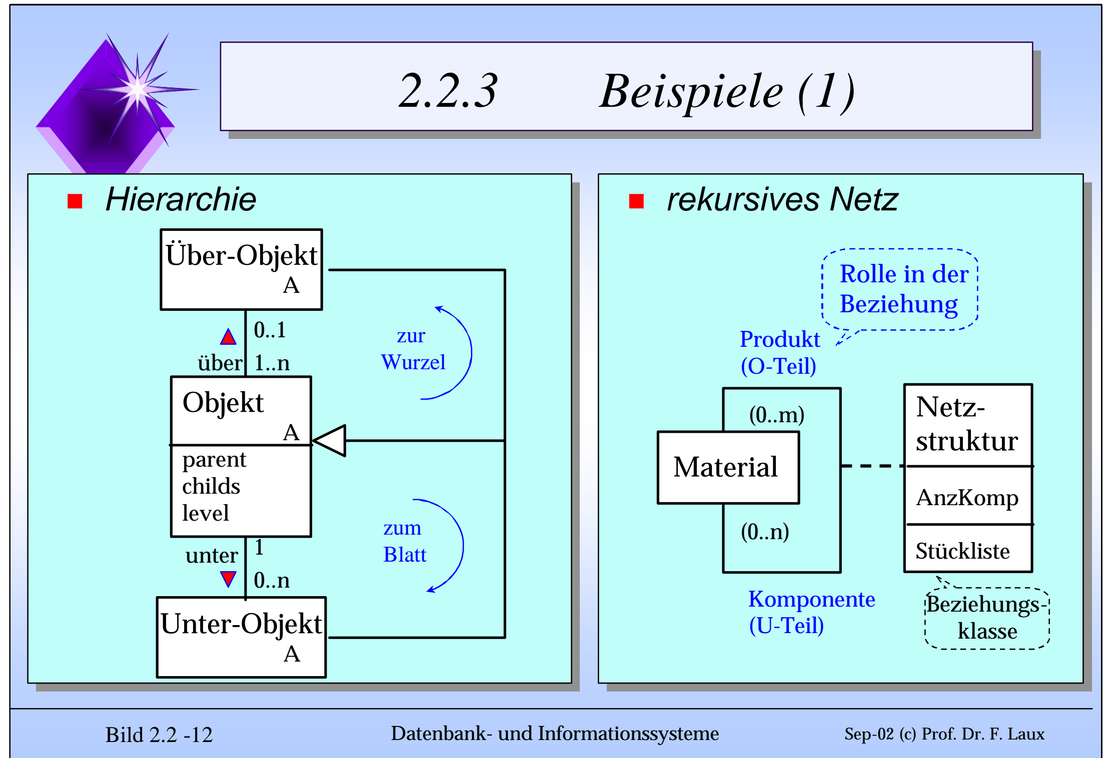
Die Syntax des Strukturteils entspricht der *Unified Modeling Language* (UML). Die UML ist aus den Notationen von Rumbaugh (Object Modeling Technique, OMT), Booch (oo Analysis & Design, OOA & OOD) und Jacobson (oo Software Engineering, OOSE) als gemeinsame Notation hervorgegangen. Sie hat den größten Verbreitungsgrad und wurde von der OMG (Object Management Group) im November 1997 als Industriestandard übernommen.

Objekt/Klasse

Alle Objekte oder Klassen werden als *Rechtecke* gezeichnet. Nur der *Klassenname* bzw. Objekt und Klassenname *muss* angegeben werden. Weitere Angaben, z.B. sprachspezifische *Objekt/Klasseneigenschaften* (abstract := abstrakte Klasse (kann nicht instanziert werden); meta := Klasse, deren Instanzen wiederum Klassen sind; param := Klasse, die durch einen Parameter konkretisiert wird (z.B. Klasse Liste mit Parameter: String = Liste mit Elementen vom Typ String)), *Attribute* oder *Methodennamen* sind *optional*. Über einen Stereotypbegriff können Klassen kategorisiert werden (Fehler-, Interface-, Fach-, GUI-Klasse). Die Kategorien sind erweiterbar. Attribute werden mit Namen und optional mit ihrem Typ angegeben. Als (Daten)Typ sind beliebige Klassen erlaubt. Auch hier sind sprachspezifische "Verzierungen" (- = privat, + = öffentlich, #=geschützt) möglich. Entsprechendes gilt für die Methoden, wobei der Typ sich auf den Rückgabewert bezieht.

Gruppierung

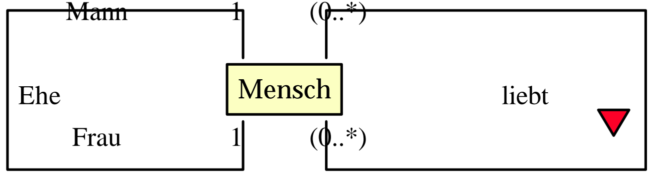
Um die Diagramme nicht zu unübersichtlich werden zu lassen, sind Gruppierungen (*package*) möglich. Eine Gruppe zusammengehöriger Klassen/Objekte wird durch eine *stilisierte Karteikarte* dargestellt, dessen "Reiter" den Gruppennamen enthält. Der o.g. *Stereotyp* kann unabhängig davon in beliebigen Gruppierungen eingesetzt werden (orthogonale Begriffe).



Beziehungen/Assoziationen:

Eine *Beziehung* ist eine allgemeine *semantische Verbindung zwischen Klassen*. Eine *Assoziation* ist eine bidirektionale *Verbindung zwischen Instanzen.*
Die *Kardinalität* der Verbindungen wird durch ein *Min..Max-Paar von natürlichen Zahlen* oder eine bestimmte Zahl angegeben. Eine Beziehung kann einen Namen besitzen, wobei die Leserichtung durch ein kleines Dreieck angegeben wird. Neben dem Beziehungsnamen kann auch ein Rollenname für die beteiligten Objekte der Verbindung angegeben werden.

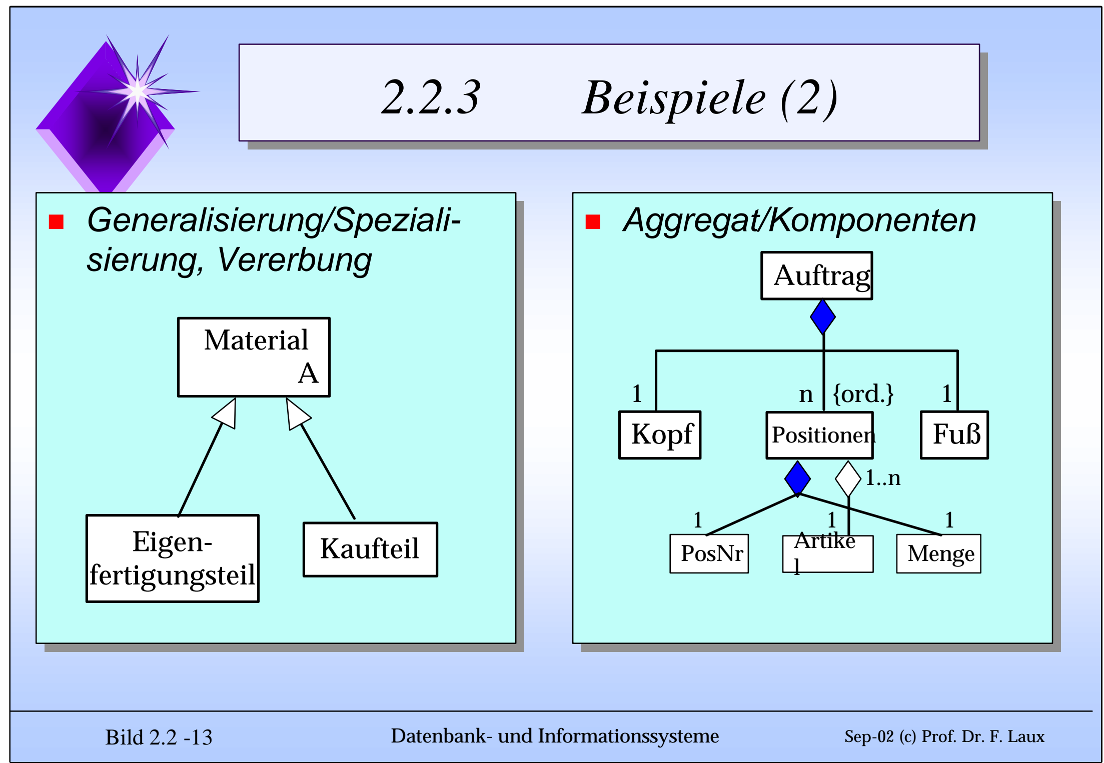
Beispiel:



'Mann' und 'Frau' sind *Rollen* einer Beziehung 'Ehe' zwischen Individuen (Objekten) der Klasse 'Mensch'.

Zusätzlich können Angaben gemacht werden, ob die Verbindungen *ungeordnet* (Menge), *geordnet* (Array: 1..n {ordered}, Liste: 1..* {ordered}, Tupel: n {ordered}) oder *sortiert* sind. (Siehe Bild 2.2-8)

ÜA: Wieviele Root-Objekte gibt es in der oben gezeigten Hierarchie? Entwickeln Sie andere Darstellungen einer Hierarchie, die z.B. nur zum Ober- bzw. zu den Unterobjekten verweisen.



Beziehungen/Assoziationen(Forts.):

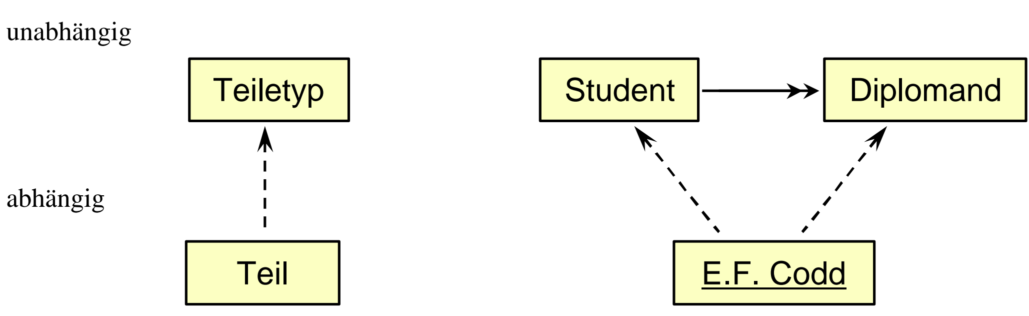
Wichtige Beziehungstypen haben eigene Symbole:

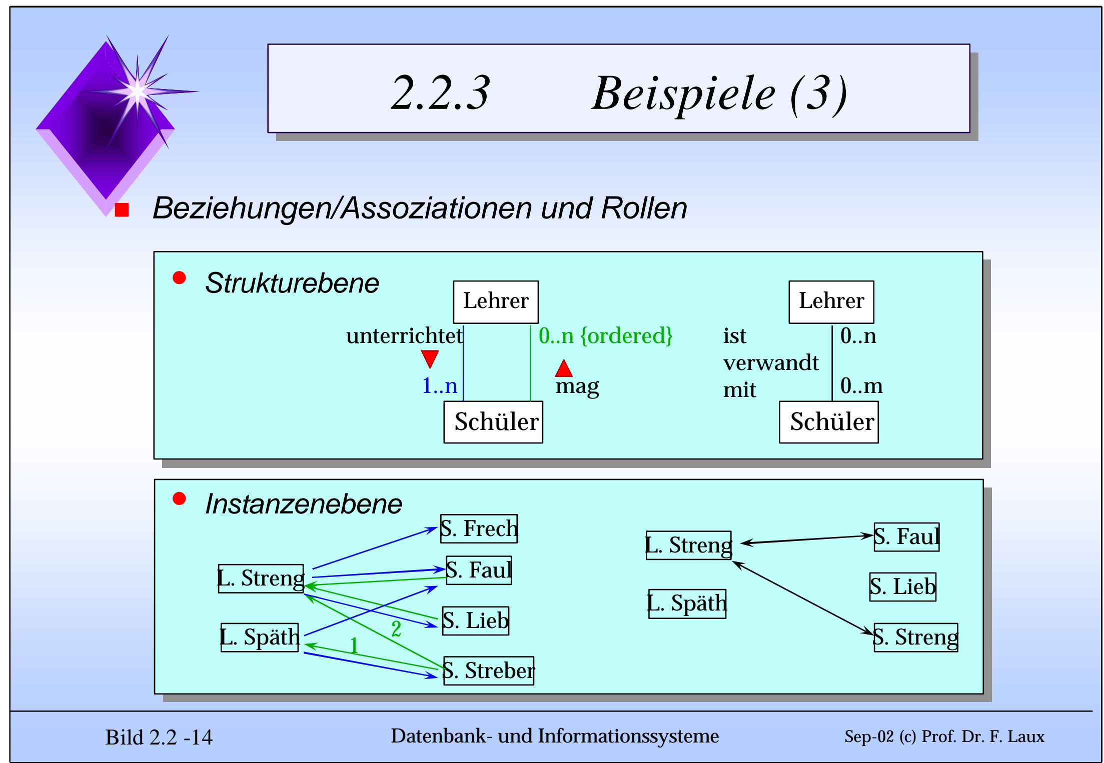
Aggregat/Komponentenbeziehungen werden durch eine Raute auf der Aggregatseite kenntlich gemacht. Die Raute ist ausgefüllt, wenn die Komponenten zu genau einem Aggregat gehören nur zusammen mit diesem existieren (*Komposition*). Sind die Komponenten unabhängig von ihrem Aggregat, ist die Raute nicht ausgefüllt. Die Räder eines Autos sind ein Beispiel für ein Aggregat, und die Zulassung ist ein Kompositum eines öffentl. Verkehrsmittels.

Generalisierungs/Spezialisierungsbeziehungen werden durch eine nicht ausgefüllte Pfeilspitze gekennzeichnet. (Anm.: um zeichentechnische Schwierigkeiten zu vermeiden, sei uns auch eine beliebige Spitze erlaubt)
Klassen/Instanzbeziehungen oder allgemeine Abhängigkeiten wollen wir durch eine gestrichelte Linie mit Pfeil vom Exemplar zur Klasse bzw. vom abhängigen zum unabhängigen Objekt darstellen.

Objekte, welche die Klasse wechseln (*Objektverschiebung*), werden durch eine Linie mit Doppelpfeil angezeigt.

Beispiele:

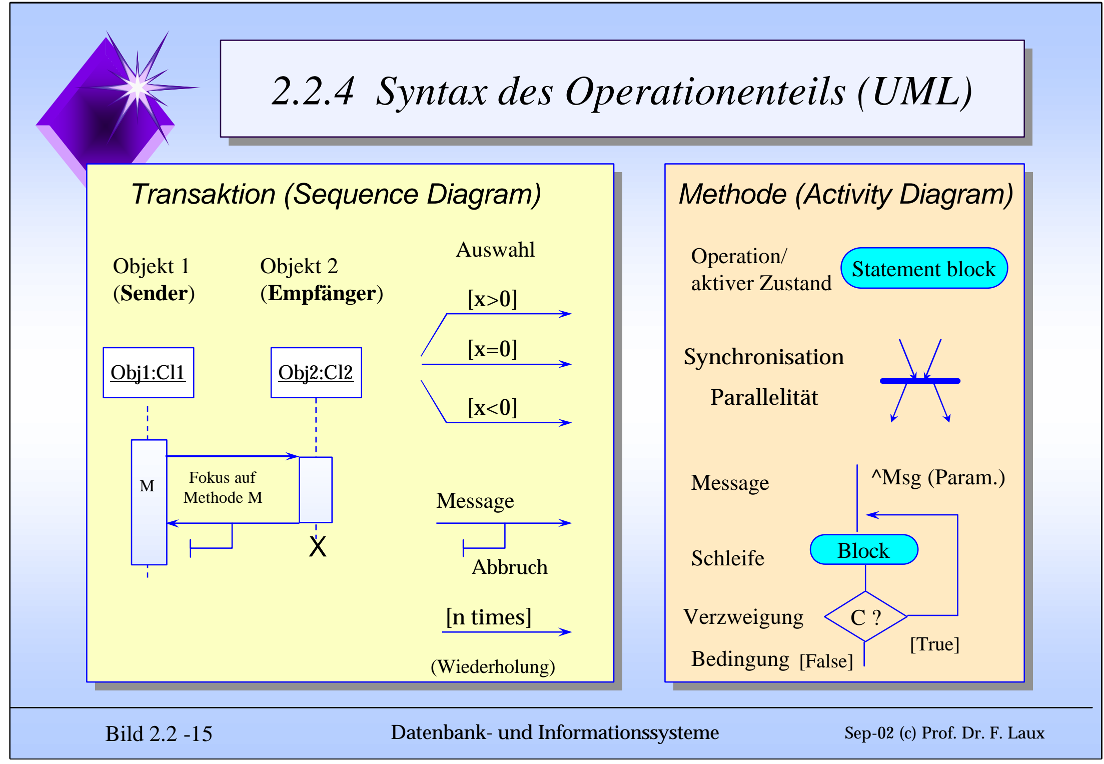




Beidseitige **Beziehungen** (**Assoziationen**) können in beide Richtungen navigiert werden, d.h. durch die Beziehung kann man von einem Objekt zur anderen gelangen und umgekehrt. Bei einseitigen ist dies nur in einer Richtung möglich. Diese Unterscheidung ist eher ein Realisierungsaspekt, da eine gerichtete Beziehung immer auch invers gelesen werden kann. So kann z.B. die Beziehung "unterrichtet" in der Gegenrichtung als "wird unterrichtet von" gelesen werden. Bei vernetzten Beziehungen ist es häufig semantisch klarer, wenn anstelle einer beidseitigen Beziehung zwei gerichtet verwendet werden, um Bedeutungsunterschiede ohne Rollenangabe deutlich zu machen. Die rekursive Beziehung bei Material (vgl. Bild 2.2-6) lässt sich leichter verstehen, wenn man daraus 2 gerichtete Beziehungen (Stückliste und Verwendungsnachweis) macht. Bei Mehrfachbeziehungen (Maximalwert > 1) können zusätzlich Angaben über die Reihenfolge oder Sortierung gemacht werden.

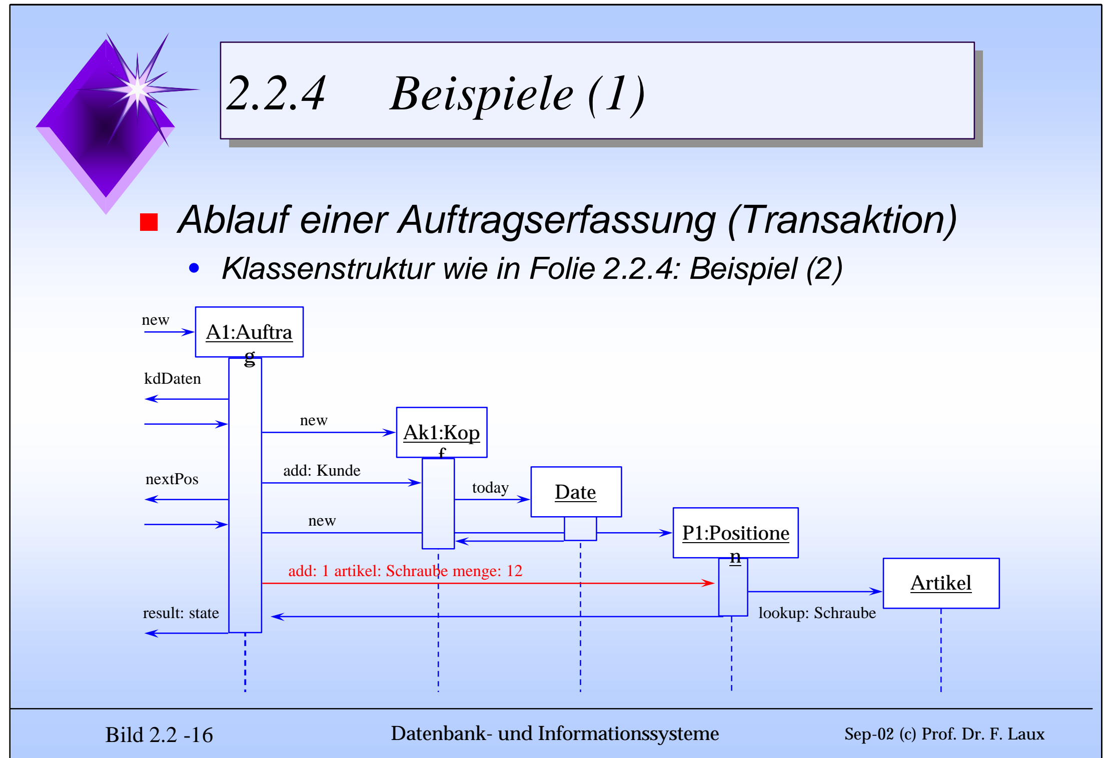
ÜA1: Modellieren Sie o.g. Beispiele unter Verwendung nur einer Klasse neu. Ergänzen Sie das Modell durch Rollenangaben.

ÜA2: Modellieren Sie den Sachverhalt von Bild 2.2-6 (rekursives Netz) durch 2 gerichtete Beziehungen.

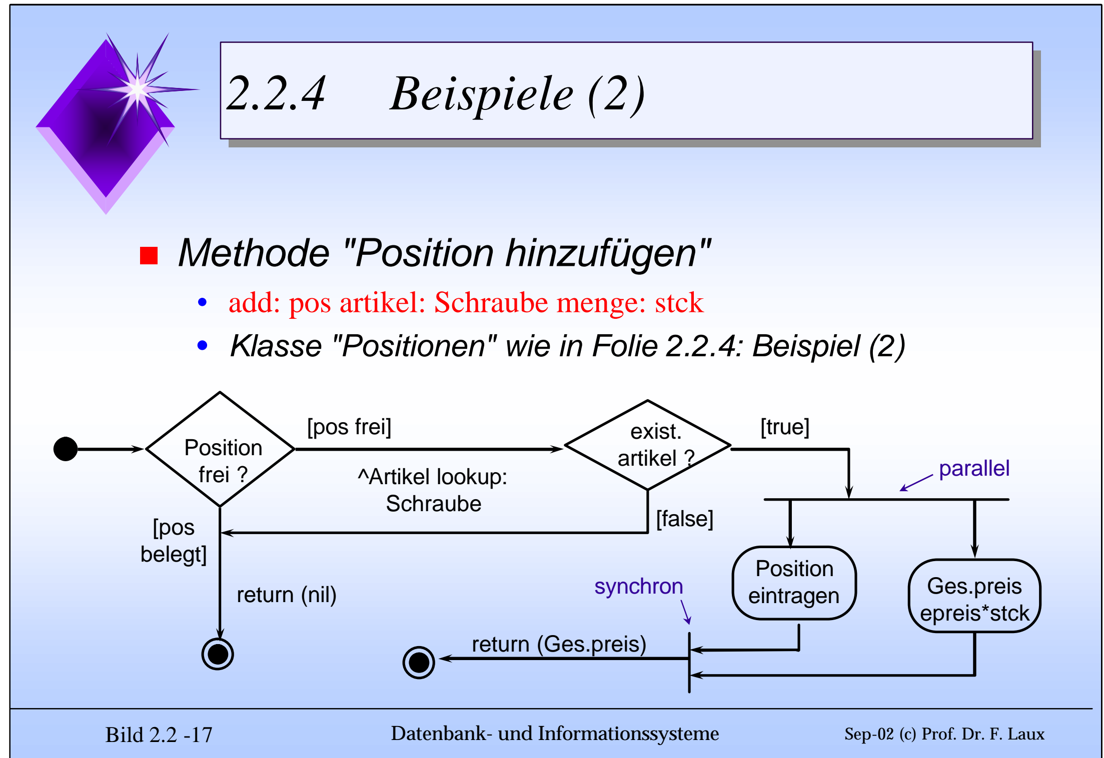


Die Symbole für die Interaktionen im **Transaktionsdiagramm** (in der UML-Notation *Sequence Diagram* genannt) zeigen *Sender* und *Empfänger* einer Nachricht. Ihre Namen werden in ein Rechteck (gleiche Darstellung wie beim Klassendiagramm) geschrieben. Die Zeitdimension verläuft von oben nach unten. Während ein Objekt aktiv ist (den *Fokus* besitzt) wird es als ein Balken gezeichnet, ansonsten als gestrichelte Linie. Wird ein Objekt gelöscht, so wird dies durch ein X angezeigt. Verzweigungen bzw. alternative Nachrichten werden durch Bedingungen realisiert, so daß die Transaktionsdiagramme mit den Sequence Diagrams der UML übereinstimmen. Wiederholungen können ebenfalls durch Bedingungen formuliert werden.

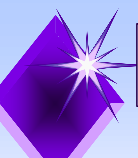
Zur Darstellung der Aktivitäten und des Steuerflusses einer Methode verwenden wir Activity Diagrams (**Aktivitätendiagramme**) der UML. Dieses sind spezielle Zustandsdiagramme bei denen eine Aktivität (Statement Block) als aktiver Zustand betrachtet wird. Ein Zustand wird verlassen, wenn die dazugehörige Aktivität beendet ist. Verzweigungen werden durch eine Raute (ein Stereotyp einer Klasse) visualisiert. Mit Hilfe einer Verzweigung können ebenfalls Schleifen realisiert werden. Synchronisationsbalken dienen dazu, parallele Vorgänge einzuleiten bzw. wieder zu synchronisieren.



Die dargestellte Transaktion ***Auftragserfassung*** beginnt mit der Nachricht *new* an die Klasse *Auftrag*. Das Ergebnis ist eine neue Instanz mit dem Namen *A1*. Diese fordert Kundendaten an (aktives Objekt). Mit diesen Daten wird eine Instanz der Klasse *Kopf* gebildet und der *Kunde* eingetragen (*add:*). Das Objekt *Ak1* besorgt sich bei der Klasse *Date* eine Instanz des aktuellen Datums. Inzwischen fordert der Auftrag *A1* eine Auftragsposition an (*nextPos*). Mit dieser Information wird eine neu erzeugte Instanz von Positionen belegt (*add: 1 artikel: Schraube menge: 12*). Das Objekt *P1* überprüft, ob der Artikel Schraube existiert (*lookup: Schraube*). Damit ist ein (minimaler) Auftrag angelegt und der Benutzer kann über das Ergebnis informiert werden (*result: state*).



Auf diesem Bild schauen wir uns die Operationen der Methode *add:artikel:menge:* genauer an. Zuerst wird geprüft, ob die Position *pos* noch frei ist. Ist dies nicht der Fall, dann endet der Versuch eine Position einzutragen sofort (Ergebnis *nil*-Objekt). Wenn die Position jedoch frei ist, wird eine Nachricht *lookup: Schraube* an die Klasse *Artikel* geschickt (in diesem Beispiel wird angenommen, daß die Klasse *Artikel* alle ihre Objekte kennt). Das Ergebnis dieser Nachricht entscheidet, ob der Artikel bestellt werden kann oder nicht. Im positiven Fall wird die Position eingetragen und der Gesamtpreis berechnet. Beide Aufgaben können parallel durchgeführt werden. Wenn beides geschehen ist (beim Synchronisationsbalken wird auf das Ende beider Operationen gewartet), wird als Ergebnis der Gesamtpreis zurückgegeben.



2.2.5 Methodik der oo-Modellierung

Orientierung an den Vorgehensmodellen von J. Rumbaugh (OMT: Object Modeling Technique) , I. Jacobson (OOSE: Object Oriented Software Engineering, Use Case) und der Fa. Rational (Rational Unified Process (RUP))

- Finden der Geschäftsprozesse und Use-Cases (Vorgänge)

- Finden der Klassen und Objekte
- Klassenhierarchie erstellen
- Objektdiagramme zeichnen
- Attributbeschreibungen

- Sequenzdiagramme erstellen
- Aktivitätsdiagramme erstellen
- Pseudocode für Methoden

Bild 2.2 -18 Datenbank- und Informationssysteme Sep-02 (c) Prof. Dr. F. Laux

Wir orientieren unser Vorgehen bei der objektorientierten Modellierung an den Methoden von Rumbaugh und Jacobson.

Wir zerlegen die Aufgabe in zusammengehörige Vorgänge (Use Cases) oder Geschäftsprozesse, um die Komplexität zu reduzieren. Für jeden Vorgang werden die beteiligten Objekte und der Auslöser (Actor) gesucht. Objekte können durch Gruppierung zusammengehöriger Daten oder durch Zusammenfassen von Operationen (Funktionsgruppen) gebildet werden. Bei der Suche nach geeigneten Objekten bzw. Klassen sind folgende "Daumenregeln" hilfreich:

• Ein Objekt sollte einem realen Gegenstand oder Konzept entsprechen.

• Eine Klasse stellt eine sinnvolle Abstraktion des Problems dar.

• Subklassifikation ist einer Klassenerweiterung vorzuziehen.

• Delegation ist der Vererbung vorzuziehen.

• Ein Objekt hat in der Regel einen Zustand (Attribute) und nichttriviale Eigenschaften (Methoden). Eine Klasse ist mehr als nur ein Name.

• Eine Methode soll nur eine einzige Aufgabe erfüllen.

• Alle Daten, die von mehr als einer Methode oder Subklasse benötigt werden, sollten als Instanzvariable gespeichert werden.

Die Objekte entstammen Klassen, die zu einer Klassenhierarchie zusammengefügt werden. Für jede Klasse werden Strukturdiagramme gezeichnet, um den inneren Aufbau sichtbar zu machen. Zur späteren Implementierung ist eine detaillierte Beschreibung der Attribute (Datentypen) notwendig.

Parallel dazu werden ausgehend von den Use Cases die Transaktionen durch Sequenzdiagramme beschrieben. Die Funktionalität jeder Methode wird durch ein Aktivitätsdiagramm graphisch dargestellt. Um eine präzise Vorgabe für die Implementierung zu erhalten, sollte die Darstellung schließlich durch Pseudocode sprachspezifisch verfeinert werden.