

Datenmodellierung mit UML

Quellen

- "Database Design for Smarties"; R. J. Muller; Morgan Kaufmann; 1999.
- "SQL-99 Complete, Really"; Peter Gultzan, Trudy Pelzer; Miller Freeman; 1999.
- "Objektorientierte Software-Entwicklung"; Bernd Oesterreich; Oldenbourg; 2001.
- www.rational.com
- ANSI 331.1 2000 SQL-J Part 1, ANSI 331.2 2000 SQL-J Part 2

Warum UML für Datenmodellierung

- Die klassische Design-Methode im DB-Bereich, die Entity-Relationship-Methode, hat zunehmende Nachteile:
 - Mangelnde Standardisierung, mangelnde Einheitlichkeit.
 - Fremdkörper in der allgegenwärtigen Objekt-Modellierung.
 - Knappe Semantik.
- UML hat zunehmende Vorteile:
 - OMG steht kurz vor der Verabschiedung eines Datenmodellierungs-Profils.
 - Tools ermöglichen die Erzeugung von DDL aus UML.
 - UML ist semantisch eine Obermenge der ERM.
- Die Verwendung von UML drängt sich immer stärker auf!

Modellierungs-Ebenen

- DB-Level Design
 - Relationale Konstrukte wie Tabellen, Attribute, Views werden so gut wie möglich mit dem "Look and Feel" von UML dargestellt.
 - Vorteil: bekannte grafische Symbole, Ersatz der nicht standardisierten ER-Notation, direkte Implementation in SQL.
 - Nachteil: Klar fokussiert auf die Möglichkeiten einer relationalen Datenbank, Mapping in Objekte für die Applikation nicht gelöst.
- OO-Level Design
 - Datenmodell nach OO-Ansätzen erstellen.
 - Automatische Generierung eines Tabellenmodelles.
 - Vorteil: Fokussierung auf OO-Technik.
 - Nachteil: Automatisches Tabellenmodell kann holprig sein und ineffiziente SQL-Abfragen zur Folge haben.

DB-Level Design

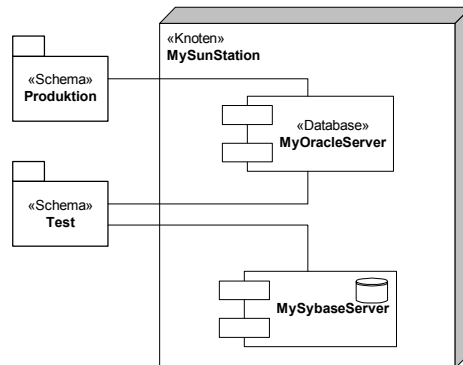
Darstellung der Implementation einer relationalen Datenbank mit der UML-Notation.

Bis zur Verabschiedung durch OMG, hält sich dieses Skript teilweise an die Notation von Rational und teilweise an eigene Erweiterungen.

Alle SQL-Beispiele genügen dem SQL-99 Standard.

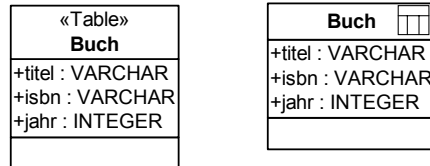
Datenbank und Schema

- Das physisch lauffähige Datenbank-System (Installation) wird als UML-Komponente dargestellt.
- Das logische Datenbank-Schema wird als UML-Paket dargestellt und ist in einem oder mehreren Datenbank-Systemen implementiert.
- Das Datenbank-System läuft auf einem Datenbank-Rechner, dargestellt als UML-Knoten.



Tabelle

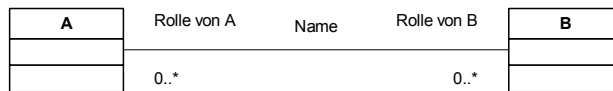
- Eine Tabelle wird als UML-Klasse mit einem entsprechenden Stereotyp dargestellt:



- Attribute sind naturgemäss öffentlich, die Sichtbarkeit kann deshalb unterdrückt werden.
- Datentypen sind gemäss einem Standard (SQL-2, SQL-99) oder dem verwendeten DB-Produkt anzugeben.

Beziehungen

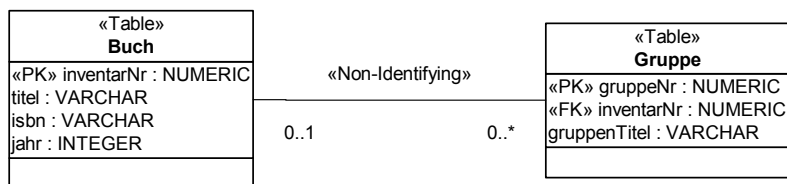
- In der klassischen Datenmodellierung ist die primäre Aussage einer Beziehung, wieviele Datensätze der einen Tabelle mit wievielen Datensätzen der anderen in Beziehung stehen (Kardinalität, Multiplizität).
- Eine der klassischen Datenmodellierung ungefähr entsprechende Beziehung zwischen zwei Tabellen wird in UML als Assoziation dargestellt.



- Beziehungen zwischen Tabellen werden technisch durch Attribute mit Schlüsselwerten realisiert.
- Primärschlüssel identifizieren einen Datensatz eindeutig.
- Fremdschlüssel beziehen sich auf einen Datensatz mit dem korrespondierenden Primärschlüssel.

Primärschlüssel / Fremdschlüssel

- Primärschlüssel und Fremdschlüssel werden mit den Stereotypen «PK» und «FK» dargestellt.
- Der Stereotyp «Non-Identifying» auf der Assoziation macht deutlich, dass Datensätze aus beiden Tabellen unabhängig voneinander existieren können. Datenbanktechnisch gesprochen: Der Primärschlüssel von 'Buch' ist nicht Teil des Primärschlüssels von 'Gruppe'.



Kombinierte Primär-/Fremdschlüssel

- Datensätze aus der einen Tabelle können existentiell von Datensätzen der anderen Tabelle abhängig sein.
- Datenbanktechnisch äussert sich dies häufig so, dass ein Attribut gleichzeitig zum Primärschlüssel der eigenen Tabelle und zum Fremdschlüssel auf die abhängende Tabelle gehört. Dargestellt wird dies mit dem Stereotyp «PFK».
- Der Stereotyp «Identifying» auf der Assoziation verdeutlicht die Abhängigkeit.



Indices

- Indices werden als UML-Operationen mit dem Stereotyp «Index» dargestellt. Die Funktionsparameter können die indizierten Attribute ausdrücken.
- Ein Index wird in der Datenbank-Technik auch benutzt um die Einmaligkeit eines Attribut-Wertes zu erzwingen. Da die meisten Datenbank hierzu direkt den Unique-Constraint kennen ist dieser zu bevorzugen. Ein Index wird durch die Datenbank selbstständig erzeugt.

«Table» Buch
«PK» inventarNr : NUMERIC titel : VARCHAR untertitel : VARCHAR isbn : VARCHAR jahr : INTEGER
«Index» tc_titel(in titel, in subtitel) «Index» tc_jahr() «Unique» tc_isbn()

Das Präfix 'tc_' steht für
'Table Constraint'

Eigenschaften für Attribute

Angabe

- Default-Wert
- Null / Not Null
- Automatische Zähler
- Aufzählung
- Berechnung
- Bedingung

UML Darstellung

attribut = defaultwert
attribut { null }
attribut { auto }
attribut { wert1, wert2, ... }
/attribut { = ausdruck }
attribut { boolscher ausdruck }

- Beispiel

«Table» Entleihung	
status : VARCHAR	{laufend, gemahnt, zurück}
bis : Date = today + 30	{not null}
mahndatum : Date	{null}
laufnummer : NUMERIC	{auto}

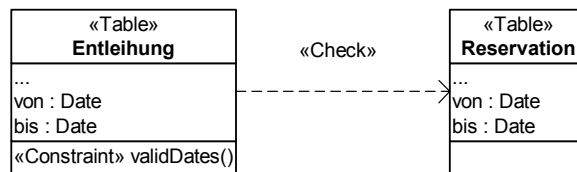
Constraints

- Ein Constraint stellt eine Bedingung an einzelnes Attribut (Column Constraint) oder einen Datensatz (Table Constraint). Beispiele in SQL-99:

```
create table Entleihung(  
    ...  
    von date,  
    bis date,  
    constraint validDates check ( bis > von )  
)  
create table Entleihung(  
    ...  
    von date,  
    bis date,  
    constraint validDates  
    check ( von > (select max(bis) from Reservation )  
)
```

Constraints ff

- Constraints können als UML-Operationen dargestellt werden, unter Kennzeichnung mit dem Stereotyp «Constraint».
- Es ist nützlich, die Abhängigkeiten eines Constraints zu den referenzierten Tabellen grafisch darzustellen.



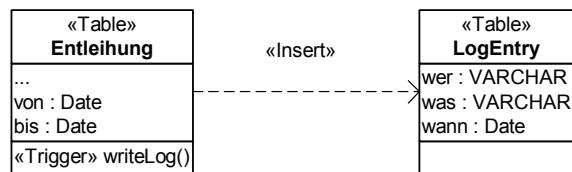
Triggers

- Ein Trigger enthält Aktionen in Form von SQL-Befehlen, die beim Einfügen, Löschen oder Ändern eines Datensatzes automatisch aufgerufen werden.
- Trigger werden verwendet, um Hilfsdaten oder redundante Zusammenzüge von Daten zwangsweise zu aktualisieren.
- Beispiel:

```
create trigger t1 after insert into Entleiherung
insert LogEntry
values ( USER, "Entleiherung", CURRENT_DATE )
```

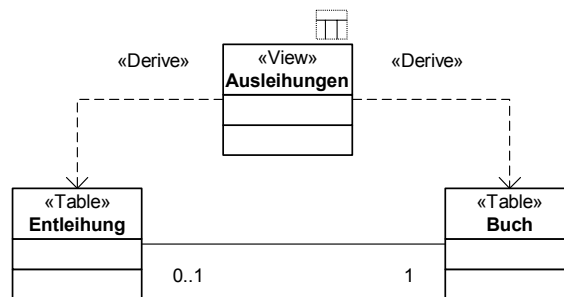
Triggers

- Trigger können als UML-Operationen dargestellt werden, unter Kennzeichnung mit dem Stereotyp «Trigger».
- Es ist nützlich, die Abhängigkeiten eines Triggers zu den modifizierten oder benutzten Tabellen grafisch darzustellen.



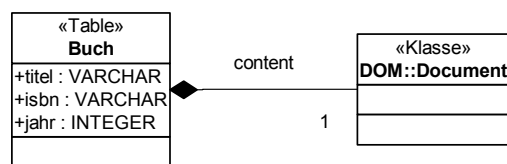
Views

- Views sind virtuelle Tabellen, welche ihre Daten aus einer Select-Abfrage über andere Tabellen beziehen.
- Zur Kennzeichnung wird der Stereotyp «View» verwendet. Die der View zugrundeliegenden Tabellen werden mit einer Abhängigkeitsbeziehung und dem Stereotyp «Derive» gekennzeichnet.



Java-Objekte

- Verschiedene DB-Systeme (z.B. Oracle, Sybase, Cloudscape) können Java-Objekte als Tabellenattribute speichern.
- Methoden und Attribute können in SQL-Befehlen direkt aufgerufen werden.
- Für wichtige Klassen, deren Objekte in Tabellen der DB abgelegt werden, lohnt sich das grafische Hervorheben:

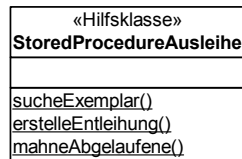


Globale Objekte

- In Datenbanken gibt es oft globale Objekte, beispielsweise Stored Procedures, benutzerdefinierte SQL-Funktionen, Assertions, benutzerdefinierte Datentypen usw.
- Globale Objekte gehören zum Schema. Im Sinne von UML können sie als Funktionen einer *Hilfsklasse* aufgefasst und entsprechend dargestellt werden.
- In der Implementation gehören Stored Procedures nicht zu einer Klasse, sie müssen global eindeutige Namen haben.

Globale Java-Objekte

- Immer mehr DB-Systeme erlauben, dass statische Methoden von Java-Klassen die Stelle der klassischen Stored Procedures einnehmen. Damit kann der Entwurf als UML-Klasse fast 1:1 in die Implementation übernommen werden.

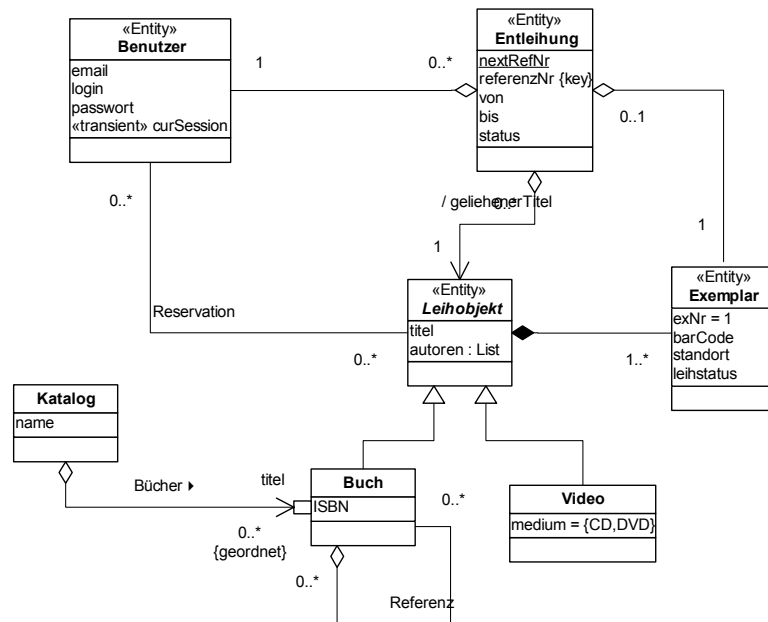


OO-Level Design

Betrachtung einiger typischer OO-Strukturelemente und ihre Abbildung in ein relationales Datenbank-Modell.

Gewisse Abbildungsmechanismen können durch Tools automatisiert werden, andere benötigen Design-Entscheide eines Software-Ingenieurs.

Referenz-Klassendiagramm



In diesem Klassendiagramm sind viele Elemente von UML aufgeführt. Der Zweck dieses Diagramms ist nicht ein perfektes Design, sondern eine zusammenhängende Diskussionsgrundlage. Im weiteren Rahmen dieses Skriptes werden die verschiedenen Syntax-Elemente besprochen, eventuell auch ergänzt oder geändert für die Diskussion.

In der Regel wird man für die Datenmodellierung mehrere Schritte anstreben:

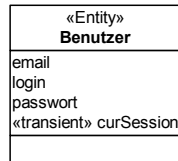
1. Ein UML-Diagramm, das möglichst nur die Anforderungen an die statischen Daten widerspiegelt, frei von technischen Aspekten.
2. Ein UML-Diagramm, das weitere Details, wie Integritätsbedingungen, Ordnungskriterien, Folgeoperation etc. aufzeigt, jedoch immer noch möglichst frei von technischen Aspekten.
3. Ein UML-Diagramm, das technische Aspekte zeigt, wie die Verwendung von Funktionsaufrufen anstelle eines Direktzugriffs auf Daten, Sichtbarkeitsangaben usw.
4. Ein normalisiertes Tabellenmodell, eventuell ohne grafische Darstellung, direkt in Form von SQL-Code.

Aufgabe

- Überlegen Sie sich die Abbildungsmöglichkeiten obiger Konstrukte in eine relationale Datenbank.

Klasse

- Nach UML 1.4 kann man die Klassen-Stereotypen «Entity», «Control» und «Boundary» unterscheiden.
- Klassen, deren Daten in einer Datenbank abzulegen sind, werden mit dem Stereotyp «Entity» deklariert. Aus *einer* «Entity»-Klasse ist als grobe Daumenregel *eine* Tabelle zu erstellen.



```
create table Benutzer (  
    idBenutzer    integer,  
    primary key   (idBenutzer),  
    email         varchar(64),  
    login         varchar(32),  
    passwort      varchar(32)  
)
```

- Alle Tabellen einer Datenbank unterliegen der Normalisierung. Dadurch können sich aus einer Klasse letztlich mehrere Tabellen ergeben.

Siehe auch 'Überführungsschema UML -> Tabellen'

Klasse ff

- Eine Klasse kann aus *technischen* Gründen in mehrere Tabellen mit einer 1:1 Beziehung untereinander aufgespaltet werden.
- Häufig gebrauchte und abgefragte Attribute werden beispielsweise in einer ersten Tabelle gespeichert, weniger häufig gebrauchte in einer zweiten Tabelle.
 - Die häufig gebrauchte Tabellen ist gesamthaft kleiner, SQL-Abfragen werden dadurch performanter. Dies spielt für sehr grosse Tabellen eine wichtige Rolle ($> 10^6$ Datensätze).
 - Auf der ersten Tabellen können andere Benutzerrechte etabliert werden als auf der zweiten.
 - Eine Tabelle kann in einem verteilten System zur Verfügung gestellt werden, die andere nicht.
 - Eventuell bestehen Limitationen in der gesamten Grösse eines Datensatzes, welche die Aufspaltung notwendig machen.

Abstrakte Klasse

- Abstrakte Klassen werden ebenfalls als eine Tabelle implementiert. Die Tabellen der konkreten Unterklassen haben einen Bezug (Fremdschlüssel) auf die Tabelle der abstrakten Klasse. Siehe Folien über Vererbung.
- Klassen mit nur statischen Attributen können als Singleton-Pattern aufgefasst werden. In der Datenbank ergibt dies eine Tabelle mit nur *einem* Datensatz.

Klasse als SQL-J Datentyp

- SQL-J ist ein ANSI-Standard, der die Verwendung von Java-Klassen als SQL-Datentypen beschreibt. SQL-J wird von den grossen DB-Herstellern unterstützt.
- Beispiel: Das Attribut 'adresse' der Klasse Benutzer ist vom Typ 'Adresse'. 'Adresse' ist eine Java-Klasse.

«Entity» Benutzer
email
login
passwort
adresse : Adresse

Adresse
+strasse
#plz
#ort
+getOrtByPlz()
+getOrt()
+setOrt()
+setPlz()
+getPlz()

```
create table Benutzer (  
    idBenutzer    integer,  
    primary key   (idBenutzer),  
    email         varchar(64),  
    login         varchar(32),  
    passwort      varchar(32),  
    adresse       Adresse  
)
```

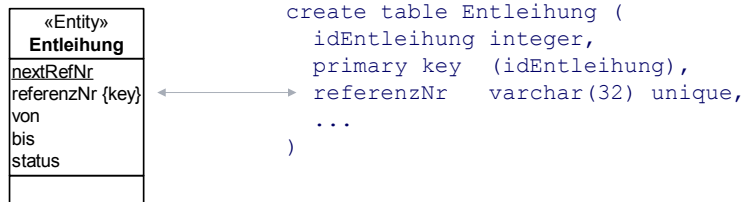
- Mit SQL-J kann eine Datenbank wesentlich kompakter gestaltet werden, weil nicht jedes Objekt zwangsweise in einzelne Felder einer Tabelle zerlegt werden muss.
- Der Datentyp 'Adresse' ist gemäss SQL-J Standard, resp. gemäss den Regeln für ein bestimmtes DB-Produkt zu importieren.

Objekt-Identität

- In der Applikation ist die Identität eines Objektes (OID) durch seine Speicheradresse (Pointer, Referenz) gegeben.
- In der Datenbank benötigt jedes Objekt einen Primärschlüssel. Aufgrund dieses Primärschlüssel können Beziehungen in der Datenbank festgehalten werden.
- Primärschlüssel-Attribute sind heute fast immer künstlich erzeugte Nummerierungen oder ID's. Diese technischen Primärschlüssel treten nur innerhalb der Applikation und der Datenbank in Erscheinung.
- Primärschlüssel brauchen im UML-Datenmodell nicht aufgeführt zu werden.

Objekt-Identität ff

- Neben dem technischen Primärschlüssel werden häufig andere, eindeutige Attribute für die *externe* Referenzierung von Datensätzen bezeichnet (z.B. AHV-Nummer, Vertragsnummer, Name+Vorname+Geburtsdatum). Diese heissen Sekundärschlüssel.
- Durch Angabe des Sekundärschlüssels kann ein bestimmter Datensatz (Objekt) eindeutig aufgefunden werden. In einem UML-Diagramm können Sekundärschlüssel beispielsweise mit der Zusicherung {key} gekennzeichnet sein:

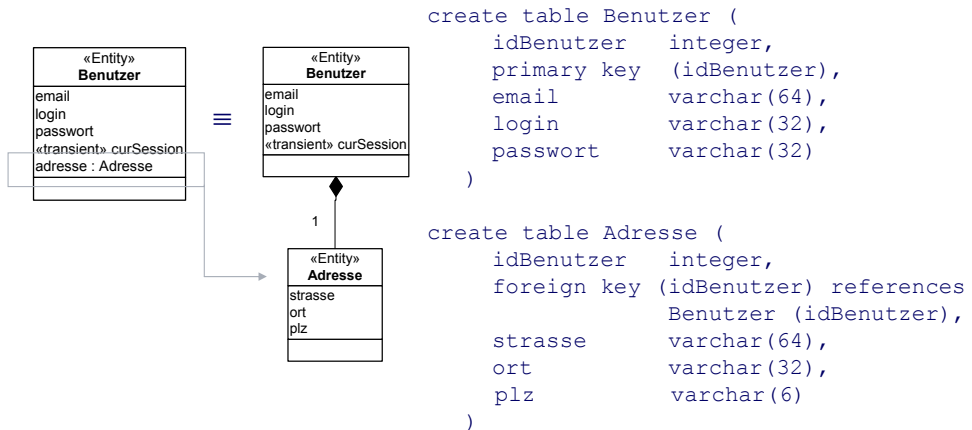


Einfache Attribute

- *Ein einfaches* Attribut mit einem Basisdatentyp der Klasse wird zu *einem* Attribut der Tabelle.
- Gewisse Datentypen sind in OO Sprache technisch als Klasse implementiert, werden jedoch mehr wie Typen verwendet. Beispielsweise String, Datum und Zeit. Da in SQL effektiv Typen für diese drei Beispiele zur Verfügung stehen, können daraus direkt Attribute einer Tabelle erstellt werden.

Komplexe Attribute

- Hat ein Attribut eine komplexe Struktur (eigene Klasse) so sind eine oder allenfalls mehrere weitere Tabellen notwendig.



Mehrwertige Attribute

- Pro mehrwertiges Attribut ist eine zusätzliche Tabelle erforderlich. Beispiel:



```
create table Leihobjekt (  
    idLeihobjekt integer,  
    titel varchar( 64 ),  
    primary key ( idLeihobjekt )  
)
```

```
create table Autor (  
    position smallint, -- Position in der Liste  
    autor varchar( 128 ),  
    idLeihobjekt integer,  
    foreign key idLeihobjekt  
    references Leihobjekt ( idLeihobjekt )  
)
```

Statische und transiente Attribute

- Design prüfen
 - Sind statische und transiente Attribute wirklich sinnvoll in dieser Klasse? Können sie an andere Klassen delegiert werden
- Transiente Attribute
 - In Tabellendefinition weglassen oder mit null-Wert belegen.
- Statische Attribute
 - Singleton Pattern d.h. *eine* Tabelle mit *einem* Datensatz.
 - Tabelle mit gesammelten statischen Attributen aller Klassen:

```
create table statischeAttribute (  
    parName varchar(32), // parName zum Beispiel nextRefNr  
    tabName varchar(32), // tabName zum Beispiel Entleihung  
    wert      varchar(64) // wert zum Beispiel 10000  
)
```

Sichtbarkeitsangaben

- In relationalen Datenbank existiert kein Sichtbarkeits-Konzept wie bei Klassen. Alle Attribute sind public.
- public ist die bevorzugte Angabe in einem konzeptionellen Datenmodell.
- private oder protected könnte verwendet werden, um auszudrücken, dass ein Attribut immer nur indirekt über eine SQL-Funktion gelesen oder geändert wird.
- Häufig ist die Situation, dass gewisse Tabellenattribute automatisch gesetzt oder berechnet werden, beispielsweise über Trigger. Solche Attribute sollten dann als Read-only markiert werden, beispielsweise mit «readonly» oder «ro» .

In einem konzeptionellen Datenmodell wird ja dargestellt, welche Information wichtig und notwendig ist. Ob der Zugriff letztlich direkt oder über get()/set()-Methoden stattfindet ist wenig relevant. Folglich wird man hauptsächlich mit öffentlichen Attributen arbeiten.

In SQL ist ein Feld einer Tabelle immer "public". Bei der Abbildung von Attributen einer Klasse in eine Tabelle muss also auf die Einschränkung der Sichtbarkeit verzichtet werden.

Innerhalb einer View ist ein Feld zwangsläufig Read-Only, wenn es mit einer Funktion aus einer Basistabelle berechnet wurde. Ein solches Feld kann durch einen entsprechenden Stereotyp gekennzeichnet werden, beispielsweise «RO» oder kann im Operationenteil beschrieben werden. Als Operationsname wäre dann der Alias-Name des Attributes in der View zu verwenden.

Innerhalb einer Java-Klasse, welche als SQL-Datentyp im Rahmen von SQL-J verwendet wird, kann ein Attribut auch "protected oder "private" sein. Der Zugriff erfolgt dann durch Aufruf der entsprechenden öffentlichen Instanz- oder Klassenmethode.

Konversion von Basistypen in Java

- Basis-Datentypen
 - Konversion gemäss SQL-99 möglich. Maximale Stringgrösse bei SQL!
 - Definieren, ob Null-Wert erlaubt oder nicht.
 - Constraints definieren (Aufzählung, Defaultwert, UML-Zusicherung)
- Objekt-Datentypen
 - in binäres Tabellenattribut serialisieren oder zugehörige Klasse ebenfalls in Tabellen abbilden und mit Fremdschlüssel referenzieren.
 - In SQL-J ist die Klasse des Objektes direkt als SQL-Datentyp verwendbar.

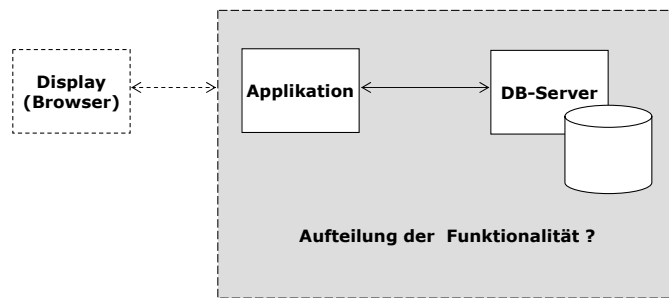
Konversion von Basistypen in Java ff

Am Beispiel Java und SQL-99

Java	SQL-99	Probleme
char	CHAR(1), NCHAR(1)	
byte	BIT(8)	
short	SMALLINT	
int	INTEGER	
long	NUMERIC(19)	
float	FLOAT	
double	DOUBLE PRECISION	
boolean	BOOLEAN	existiert nicht je nach Produkt
BigDecimal	NUMERIC(N,M)	N,M festlegen. Rundungsfehler-Probleme
Date	DATE, TIME, TIMESTAMP	
String	VARCHAR(N), NVARCHAR(N), CLOB	N festlegen, Entscheid für CLOB oder nicht
byte[N]	BIT[N*8]	
byte[]	BLOB	

Methoden

- Grundsatzfrage: Wie weit soll die Applikation und wie weit die Datenbank für die Durchführung von Methoden zuständig sein?



Notizen

Welche Art Methoden bietet die DB an?

- **Stored Procedures** sind aus OO-Sicht globale Methoden, sie werden meist in folgender Art aufgerufen:

```
exec myprocedure transient_parameters
```

- **Benutzerdefinierte SQL-Funktionen** werden aus OO-Sicht häufig im Sinne von Instanz-Methoden, beispielsweise wie folgt, verwendet:

```
select myfunction( table_attributes )...  
from ... where ....
```

- **SQL-Trigger** stellen aus OO-Sicht eine Instanz-Methode mit Seiteneffekt dar. Der schreibende Zugriff auf ein Attribut bewirkt gleichzeitig gewisse Folgeoperationen. *Rein* oo-mässig würde dies als set()-Methode auf ein geschütztes Attribut realisiert.

Welche Art Methoden bietet die DB an?

- **Datenbankseitige Java-Klassen** im Rahmen von SQL-J:

```
select data_attr, class_name.classmethod(param), ...
```

Die Klasse `class_name` stellt hier die Klassenmethode `classmethod()` zur Verfügung.

- **Datenbankseitige Java-Objekte** im Rahmen von SQL-J:

```
select data_attr, object_attr.instanceMethod(param), ....
```

Für jedes konkrete Objekt `object_attr` wird bei der Abarbeitung des `select`-Befehles die Methode `instanceMethod()` aufgerufen.

`param` kann ein gewöhnlicher SQL-Wert oder ein Objekt sein.

- Die Syntax weicht bei den verschiedenen DB-Produkten ab

Aufteilungskriterien

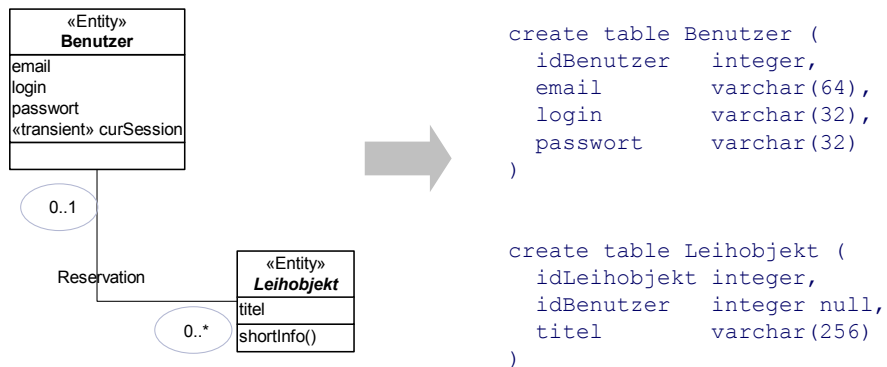
- Argumente für die Implementation einer Methode auf dem Datenbank-Server:
 - wenn sie dem Applikations-Entwickler einen Dienst zur Verfügung stellt, der unabhängig von der clientseitig verwendeten Programmiersprache sein soll, beispielsweise eine rekursive SQL-Abfrage, die mit prozeduralen Elementen gemischt ist.
 - wenn sie den direkten Zugriff auf die Daten im Sinn von *Zugriffsrechten* verhindern soll.
 - wenn sie für die Einhaltung *harder* Constraints zuständig ist.
 - wenn sie sehr viele einzelne SQL-Befehlen ausführen muss (Performance).
 - wenn der Code zentral verwaltet werden soll.

Aufteilungskriterien

- Argumente für die Implementation einer Methode in der Applikation:
 - wenn sie irgendeine Art Benutzer-Interaktion erfordert (auch Fehlermeldungen!).
 - Wenn sie für die Transaktionskontrolle zuständig ist (commit, rollback).
 - wenn sie rechenintensiv ist, d.h. viel CPU-Zeit benötigen.
 - wenn sie mit unvorhersagbarem Zeitbedarf abläuft und damit den Abschluss einer Transaktion behindern können.
 - wenn sie auf transiente oder auf dem Client lokalisierte Objekte zugreifen muss, beispielsweise für File-IO, WAN-Dienste.
- "Datenbank-Unabhängigkeit" ist ein höchst problematisches Argument!

Assoziation

- Die Abbildung einer Assoziationen aus einem Klassendiagramm in eine relationale Datenbank geschieht durch Einführung einer Primär-/Fremdschlüssel-Beziehung:



Assoziation ff

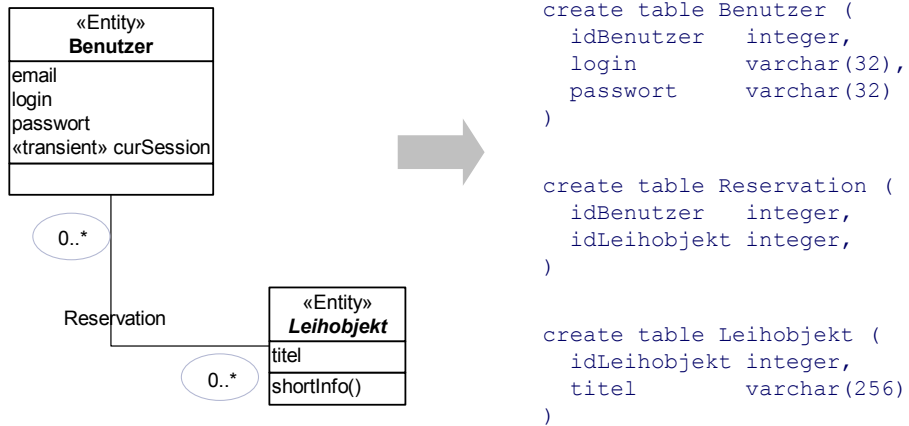
- Die Primär-/Fremdschlüsselbeziehung wird durch entsprechende Constraints in SQL abgesichert:

```
create table Benutzer (  
    idBenutzer    integer,  
    login         varchar(32),  
    passwort      varchar(32),  
  
    primary key (idBenutzer)  
)  
  
create table Leihobjekt (  
    idLeihobjekt integer,  
    idBenutzer   integer null,  
    titel        varchar(256),  
  
    primary key (idLeihobjekt),  
  
    foreign key (idBenutzer)  
    references Benutzer(idBenutzer)  
    on delete set null  
)
```

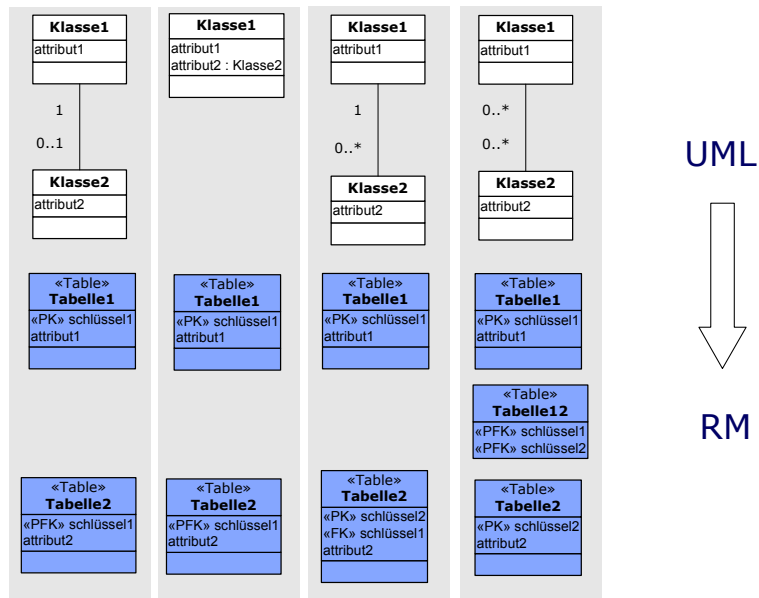
A diagram consisting of a single arrow pointing from the 'idBenutzer' column in the 'Leihobjekt' table to the 'idBenutzer' column in the 'Benutzer' table, indicating a foreign key relationship.

Assoziation ff

- Liegt auf beiden Seiten der Assoziation eine 0..* Multiplizität vor, ist eine zusätzliche Verknüpfungstabelle zu erzeugen.



Überführungsschema UML -> Tabellen



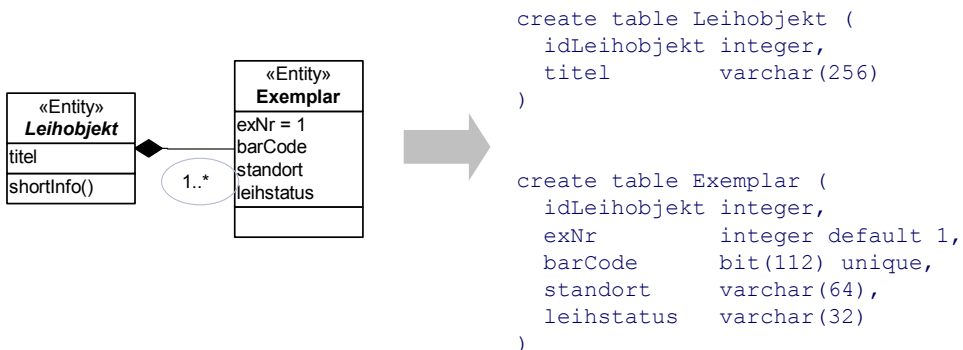
Dieses Überführungsschema zeigt kochbuchartig, wie man aus einem UML-Modell ein Relationenmodell (Tabellenmodell) erstellt. Das Schema gilt für Assoziationen, sinngemäss aber auch für Aggregationen und Kompositionen.

Die Überführung vom UML-Modell in das Relationenmodell RM geschieht unter den Gesichtspunkten:

- Möglichst wenig Relationen (Tabellen) erzeugen.
- Alle Informationen und Zusammenhänge im UML müssen im RM erhalten bleiben.
- Im Relationenmodell sind nur atomare Domänen verfügbar.
- Redundanzfreiheit durch Normalisierung.

Komposition

- Eine Komposition drückt eine existentielle Abhängigkeit aus. Wenn das übergeordnete Objekt gelöscht wird, haben auch seine einzelnen Komponenten-Objekte keinen Sinn mehr.
- Der Abbildung in Tabellen liegen dieselben Überlegungen wie für eine gewöhnliche Assoziation zugrunde.



Siehe auch Folie 'Vergleich Aggregation / Komposition'

Komposition ff

- Meist ist ein Teil des Primärschlüssels des abhängigen Teils gleichzeitig der Fremdschlüssel auf das übergeordnete Teil.
- Die existentielle Abhängigkeit wird durch einen Constraint implementiert:

```
create table Leihobjekt (  
    idLeihobjekt integer,  
    titel          varchar(256),  
    primary key (idLeihobjekt),  
    ...  
)  
  
create table Exemplar (  
    idLeihobjekt integer,  
    ...  
    primary key (idLeihobjekt, exNr),  
    foreign key (idLeihobjekt) references Leihobjekt(idLeihobjekt)  
    on delete cascade initially deferred,  
)
```

Komposition ff

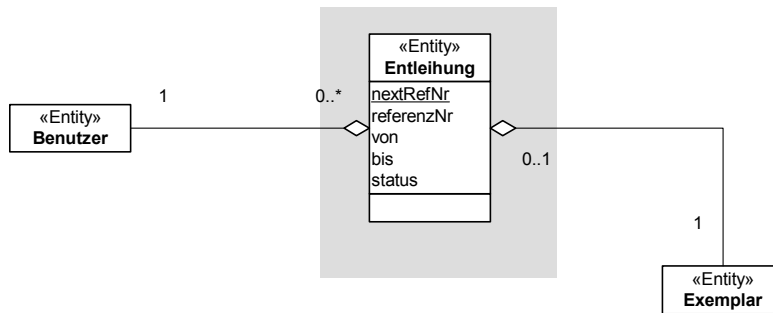
- Die Multiplizität **1..*** wird durch Constraints implementiert:

```
create table Leihobjekt (  
    ...,  
    check ( idLeihobjekt in (select idLeihobjekt from Exemplar) )  
    initially deferred  
)
```

- Die Klausel `initially deferred` bedeutet, dass so bezeichnete Constraints erst zum commit-Zeitpunkt geprüft werden und nicht unmittelbar nach dem Ändern/Löschen/Einfügen eines Datensatzes. Damit wird verhindert, dass sich Constraints gegenseitig ausschliessen *während* dem Bearbeiten der Daten.

Aggregation

- Eine Aggregation drückt aus, dass die aggregierten Objekte eine gewisse konzeptionelle Einheit bilden. Beispiel: Entleihung.
- Gegenüber der Komposition entfällt die existentielle Abhängigkeit.
- Die aggregierten Teile haben häufig die Multiplizität 1 oder 1..*.



Aggregation ff

```
create table Entleiherung (
  idEntleiherung integer,
  referenzNr      varchar(32) unique,
  von             date,
  bis            date,
  idBenutzer      integer,
  idLeihobjekt    integer,
  exNr           integer,
  primary key (idEntleiherung),
  foreign key (idBenutzer)
    references Benutzer(idBenutzer) on delete restrict,
  foreign key (idLeihobjekt,exNr)
    references Exemplar(idLeihobjekt,exNr) on delete restrict,
)
```

Aggregation ff

- Der Abbildung in Tabellen liegen dieselben Überlegungen wie für eine gewöhnliche Assoziation zugrunde.
- Der Aggregationskonstrukt  hat keine Bedeutung für die Implementation. Er ist identisch mit der Assoziation .
- Die Aggregation wird grundsätzlich mit einer Primär-/Fremdschlüssel-Beziehung implementiert.
- In welcher Tabelle sich der Primär- und der Fremdschlüssel befinden, hängt von den Multiplizitäten der Aggregation ab:
 - besitzt *eine* Seite die Multiplizität 1 oder 0..1, enthält die zugehörige Tabelle den Primärschlüssel, und die Tabelle der andere Seite den Fremdschlüssel.
 - besitzen beide Seiten die Multiplizität 1 oder 0..1 können beide oder nur eine der Tabellen einen Fremdschlüssel auf einen Primärschlüssel der anderen Tabelle enthalten (je nach Navigierbarkeit).
 - besitzen beide Seiten eine Multiplizität der Art 0..*, 1..*, n..m, ist eine zusätzliche Verknüpfungstabelle notwendig.

Siehe auch Folie 'Vergleich Aggregation / Komposition'

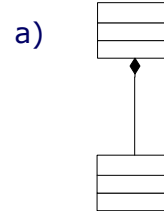
Multiplizitäten

- Die allgemeinen Multiplizitäten der Form **0..*** oder **0..1** finden ihren Niederschlag in entsprechenden Tabellenstrukturen.
- Multiplizitäten der Form **1..*** oder **1..1** finden Ihren Niederschlag in entsprechenden Tabellenstrukturen und zusätzlichen Constraints. Das gilt ebenfalls für spezielle Multiplizitäten, wie beispielsweise **3..3** oder **1,3,5** oder **1..3,10..20**
- Anstelle von Constraints können *globale* Assertions verwendet werden. Beispiel für die Sicherstellung, dass zu jedem Leihobjekt mindestens ein Exemplar gehört.

```
create assertion a1
check (( select count(*) from Leihobjekt l
         where not exists ( select * from Exemplar e
                           where e.idExemplar = l.idExemplar )
       ) = 0
      )
```

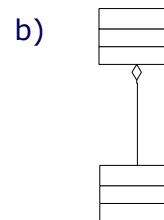
Vergleich Aggregation / Komposition

- Nebestehende Beispiele a) und b) weisen aufgrund der Multiplizitäten exakt diesselben existentiellen Abhängigkeiten auf. Wofür also die Unterscheidung Aggregation/Komposition?



- Komposition: Die Lizenz gehört *immer zum gleichen* Kunden.

- Aggregation: Die Lizenz kann auf einen anderen Kunden *übertragen* werden.

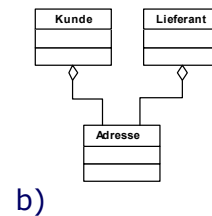
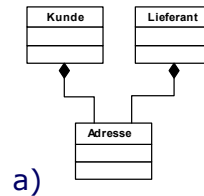


In beiden Beispielen kann der Kunde nur gelöscht werden, wenn auch die Lizenz gelöscht wird. Die Multiplizität von Lizenz zu Kunde verlangt das. Soll das nicht gelten müsste die Multiplizität von 1 auf 0..1 gewechselt werden. Lizenzen sind auch nicht "shareable", sonst wäre die Multiplizität von Lizenz zu Kunde 0..* oder 1..*.

Die Aussagen eines Klassendiagramms müssen zu jedem Zeitpunkt gelten. Allerdings nur zu jedem Zeitpunkt, wo das System in Ruhe ist, also gerade keine Änderungen stattfinden. In Übergangszuständen (innerhalb einer Transaktion) gibt es selbstverständlich Situationen, wo das Klassendiagramm nicht gilt. Es müssen ja beispielsweise Objekte in Aggregaten der Reihe nach gelöscht oder eingefügt werden. Rein aufgrund der Multiplizitäts-Angaben ist es möglich, ein Lizenzobjekt von einem Kunden auf einen anderen zu übertragen. Vor und nach der Übertragung gehört die Lizenz genau *einem* Kunden. Die Kennzeichnung als Komposition kann nun also so interpretiert werden, dass das Lizenzobjekt am Kundenobjekt für immer festgeklebt ist. Das entspricht im Übrigen auch der Situation auf einer rein technischen Ebene, beispielsweise in C++: Eingebettete Objekte können nicht übertragen werden.

Vergleich Aggregation / Komposition ff

- Welcher Unterschied besteht zwischen den Beispielen a) und b)?
- Komposition: Eine Kunde und ein Lieferant können nie dieselbe Adresse besitzen.
- Aggregation: Eine Kunde und ein Lieferant können u.U. dieselbe Adresse teilen.



In beiden Beispielen kann der Kunde nur gelöscht werden, wenn auch die Lizenz gelöscht wird. Die Multiplizität von Lizenz zu Kunde verlangt das. Soll das nicht gelten müsste die Multiplizität von 1 auf 0..1 gewechselt werden. Lizenzen sind auch nicht "shareable", sonst wäre die Multiplizität von Lizenz zu Kunde 0..* oder 1..*.

Die Aussagen eines Klassendiagramms müssen zu jedem Zeitpunkt gelten. Allerdings nur zu jedem Zeitpunkt, wo das System in Ruhe ist, also gerade keine Änderungen stattfinden. In Übergangszuständen (innerhalb einer Transaktion) gibt es selbstverständlich Situationen, wo das Klassendiagramm nicht gilt. Es müssen ja beispielsweise Objekte in Aggregaten der Reihe nach gelöscht oder eingefügt werden. Rein aufgrund der Multiplizitäts-Angaben ist es möglich, ein Lizenzobjekt von einem Kunden auf einen anderen zu übertragen. Vor und nach der Übertragung gehört die Lizenz genau *einem* Kunden. Die Kennzeichnung als Komposition kann nun also so interpretiert werden, dass das Lizenzobjekt am Kundenobjekt für immer festgeklebt ist. Das entspricht im Übrigen auch der Situation auf einer rein technischen Ebene, beispielsweise in C++: Eingebettete Objekte können nicht übertragen werden.

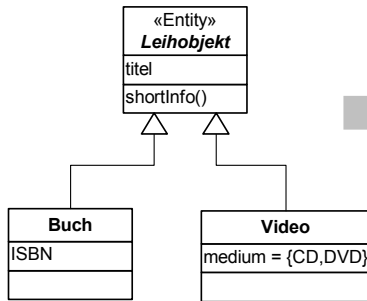
Vererbung

- Relationale Datenbanken kennen keine Vererbungsmechanismen.
- Bei der Abbildung von Vererbungshierarchien in Tabellen können verschiedene Ziele im Vordergrund stehen:
 1. Ersichtlichkeit des Vererbungsgedanken
 2. Vermeidung redundanter Strukturen
 3. Flexibilität für neue Zweige in der Vererbung
 4. Hohe Performance, Platzbedarf unerheblich
 5. Einfach gestaltete Queries für das Abfragen, Einfügen, Ändern, Löschen von Objekten.
- Die Punkte 1,2 und 3 einerseits und die Punkte 4 und 5 andererseits wirken gegeneinander beim Aufbau einer Tabellenstruktur.

Notizen

Vererbung, Ziele 4 und 5

- Die ganze Vererbungshierarchie wird in eine einzige Tabelle abgebildet:



```
create table Leihobjekt (
    idLeihobjekt integer,
    typ          varchar(32),
    titel        varchar(256),
    isbn         varchar(13) null,
    medium       varchar(16) null,
    primary key(idLeihobjekt),
    check (typ in('Buch', 'Video')),
    check (medium in 'CD', 'DVD'),
    check (isbn like '_-____-__-'),
    unique isbn
)
```

Vererbung, Ziele 4 und 5 ff

- Das Abfragen und Bearbeiten dieser einzigen Tabelle ist sehr einfach.
- Es besteht die Gefahr, dass wegen Applikations- oder Benutzerfehlern falsche Daten erfasst werden. Beispiel: Ein Buch ohne isbn-Nummer, dafür mit Angabe eines Mediums.
- Mit Constraints kann sichergestellt werden, dass für jeden Typ nur die gültigen Attribute in der Tabelle erfasst werden:

```
create table Leihobjekt (  
    ...  
    check ( (typ = 'Buch' and medium is null)  
    or (typ = 'Video' and isbn is null )  
)
```

Vererbung, Ziele 4 und 5 ff

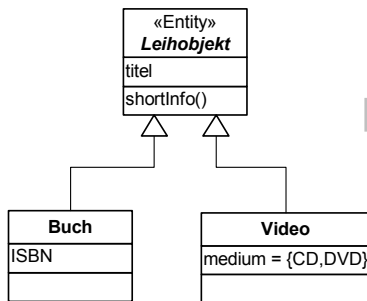
- Mit Hilfe von Views kann dem Applikationsprogrammierer *eine* virtuelle Tabelle pro "Unterklasse" zur Verfügung gestellt werden:

```
create view Buch as
  select idLeihobjekt, typ, titel, isbn
  from Leihobjekt where typ = 'Buch'
  with check option
```

```
create view Video as
  select idLeihobjekt, typ, titel, isbn
  from Leihobjekt where typ = 'Video'
  with check option
```

- Hauptproblem: Die Struktur der Tabelle und ihrer Constraints ist sehr wartungsintensiv, wenn neue "Unterklassen" eingeführt werden.

Vererbung, Ziele 1-3



```
create table Leihobjekt (
    idLeihobjekt integer,
    titel          varchar(256),
    ...
)
```

```
create table Buch (
    idLeihobjekt integer,
    isbn          varchar(13),
    ...
)
```

```
create table Video (
    idLeihobjekt integer,
    medium        varchar(8),
    ...
)
```

Vererbung, Ziele 1-3 ff

```
create table Leihobjekt (  
    idLeihobjekt integer,  
    primary key(idLeihobjekt),  
    check ( idLeihobjekt in (select idLeihobjekt from Buch)  
        or idLeihobjekt in (select idLeihobjekt from Video))  
    initially deferred  
    ...  
)  
  
create table Buch (  
    idLeihobjekt integer,  
    foreign key(idLeihobjekt)  
        references Leihobjekt(idLeihobjekt)  
    on delete cascade initially deferred  
    ...  
)  
  
create table Video (  
    idLeihobjekt integer,  
    foreign key(idLeihobjekt)  
        references Leihobjekt(idLeihobjekt)  
    on delete cascade initially deferred  
    ...  
)
```

Vererbung, Ziele 1-3 ff

- Mit einer Tabelle pro Klasse entsteht eine klare Struktur.
- Änderungen in der Struktur (neue Attribute, neue Unterklassen) haben nur noch lokale Auswirkungen auf die betroffenen Tabellen.
- Attribute mit Null-Werten (isbn, medium) können vermieden werden und damit auch die entsprechenden Constraints.
- SQL-Abfragen über alle "Unterklassen", zum Beispiel Bücher oder Videos, können mit Views unterstützt werden:

```
create view Buch as
  select idLeihobjekt, titel, isbn
  from Leihobjekt l, Buch b
  where l.idLeihobjekt = b.idLeihobjekt
```

- Alle Attribute von Büchern liefert die Abfrage

```
select * from Buch
```

Vererbung, Ziele 1-3 ff

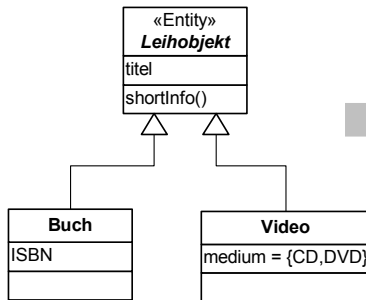
- Das Löschen beliebiger Leihobjekte geschieht durch Löschen des Eintrages in der Tabelle `Leihobjekt`. Die korrespondierenden Einträge in `Buch` oder `Video` werden automatisch gelöscht aufgrund des `on delete Constraints`.
- Das Einfügen neuer Bücher oder Videos kann durch direkte Anwendung zweier SQL-Befehle bewerkstelligt werden, beispielsweise:

```
insert Buch (idLeihobjekt, isbn) values( 1, '0-12345-123-1' )  
insert Leihobjekt (idLeihobjekt, titel) values( 1, 'XML' )
```
- Die Verwendung einer Stored Procedure wäre ebenfalls denkbar, beispielsweise in der Art:

```
addBuch 1, 'XML', '0-12345-123-1'
```
- Ein Stored Procedure versteckt letztlich die Tabellenstruktur vor der eigentlichen Applikation.

Vererbung, Kompromiss

- Ein Kompromiss, der vorallem zweckmässig ist, wenn die Applikation immer nur Daten der einen *oder* anderen Klasse benötigt, ist folgender:

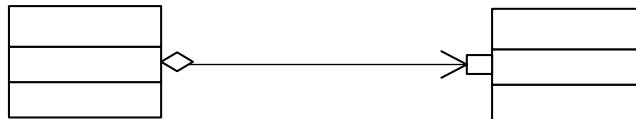


```
create table Buch (
    idLeihobjekt integer,
    titel          varchar(256),
    isbn           varchar(13),
    ...
)

create table Video (
    idLeihobjekt integer,
    titel          varchar(256),
    medium         varchar(8),
    ...
)
```

Geordnete Assoziation

- Eine geordnete Assoziation von A nach B bedeutet, dass die assoziierten Elemente von B in einer bestimmten Reihenfolge vorliegen. Beispielsweise sollen in einem Katalog die Bücher alphabetisch geordnet vorkommen:



(Das Ordnungs - Attribut 'titel' wird von der Klasse Leihbojekt geerbt)

Geordnete Assoziation ff

- Das Ordnungskriterium kann als Anweisung an die Applikation verstanden werden, die Daten aus der DB in einer bestimmten Reihenfolge zu entnehmen (ORDER BY Klausel in SQL).

```
select isbn, titel
from Leihobjekt join Buch join Katalog
where Katalog.name = 'Lehrbücher'
order by Buch.titel
```

- Die Ordnung muss eventuell explizit implementiert werden, wenn sie nicht aus den Nutzdaten (titel) herleitbar ist:

```
create table Buch (
    ....
    position integer null
)
```

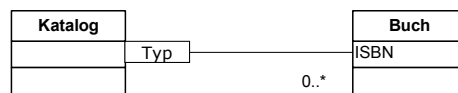
(Vorausgesetzt jedes Buch gehört maximal zu einem Katalog. Ansonsten müsste eine zusätzliche Beziehungstabelle geschaffen werden)

Das Attribut 'position' ist notwendig, wenn die Ordnung sich nicht auf ein bestimmtes effektiv vorhandenes Attribut bezieht. Beispiele hierzu sind:

Vornamen einer Person sind geordnet, aber nicht aus der alphabetischen Reihenfolge ableitbar. Die Autoren eines Buches sind geordnet (Hauptautor zuerst), die Reihenfolge ist aber nicht aus dem Namen ableitbar, sie muss einfach zusätzlich bekannt sein.

Qualifizierte Assoziation

- Eine qualifizierte Assoziation von A nach B ordnet einem a mehrere Pakete von b's zu. Jedes Paket ist gekennzeichnet durch den Wert des qualifizierenden Attributes.
- Aus Sicht des Applikationsentwicklers entspricht eine qualifizierte Assoziation häufig der Verwaltung von Objekten in einer Hashtabelle oder einem Dictionary.
- Im folgenden Beispiel sind Bücher einem Katalog in Paketen bestimmter Buchtypen zugeordnet, beispielsweise Typ 'Lehrbuch', Typ 'Manual' usw.



- Zu beachten: die Multiplizität ist pro unterschiedlichen Wert des qualifizierenden Attributes zu verstehen.

Qualifizierte Assoziation ff

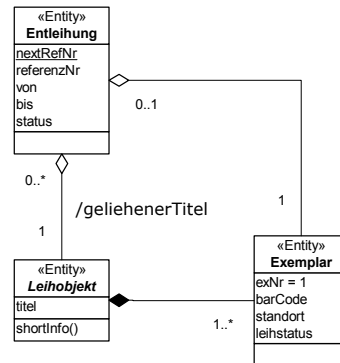
- Das qualifizierende Attribut findet sich auch als Attribut einer Tabelle in der Datenbank wieder. Für obiges Beispiel:

```
create table Katalog (  
    idKatalog integer,  
    primary key(idKatalog)  
)  
  
create table Buch (  
    idLeihobjekt integer,  
    isbn          varchar(13),  
    idKatalog     integer,  
    typ           varchar(32),  
    primary key(idLeihobjekt),  
    foreign key(idKatalog) references Katalog(idKatalog),  
    ...  
)
```

- Falls ein Buch mehrere Typisierungen haben kann oder zu mehreren Katalogen gehören kann, ist eine Zwischentabelle zu erzeugen mit den Attributen `idLeihobjekt`, `idKatalog` und `typ`.

Abgeleitete Assoziation

- Eine abgeleitete Assoziation stellt eine Beziehung dar, die an sich wichtig ist, jedoch aus anderen Assoziationen abgeleitet werden kann.
- Beispiel: Die Assoziation 'geliehenerTitel' zeigt, von welchem Leihobjekt ein Exemplar ausgeliehen wurde. Diese Assoziation ist an sich über die Beziehung von 'Entleihung' zu 'Exemplar' und die Beziehung von 'Leihobjekt' zu 'Exemplar' berechenbar.



Abgeleitete Assoziation ff

- Die Berechnung abgeleiteter Assoziationen ist im Rahmen einer Datenbank meist einfach. Beispiel: Auffinden des entliehenen Titels aufgrund der Angabe von Referenznummer der Entleihung:

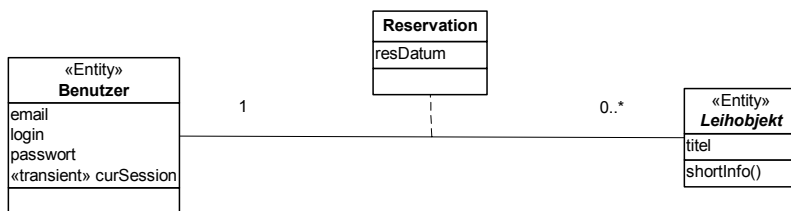
```
select o.idLeihobjekt, o.titel
from Leihobjekt o join Exemplar x join Entleihung e
where e.referenzNr = 10000
```

- Aus Performance-Gründen ist es jedoch durchaus möglich, die abgeleitete Assoziation auch tatsächlich zu implementieren. Beispielsweise mit einem zusätzlichen Fremdschlüssel in der referenzierenden Tabelle:

```
create table Entleihung (
    ...
    foreign key (idLeihobjekt)
        references Leihobjekt(idLeihobjekt) on delete restrict,
)
```

Assoziationsklasse

- Eine Assoziationsklasse stattet eine Assoziation zwischen zwei Klassen mit Attributen aus.
- Eine Assoziationsklasse entspricht exakt der 'Attributierten Relation' in der klassischen ER-Modellierung.
- Eine Assoziationsklasse hat keine eigene Identität (d.h. im Tabellenmodell keinen Primärschlüssel).
- Beispiel 'Reservation' als Assoziationsklasse:



Assoziationsklasse ff

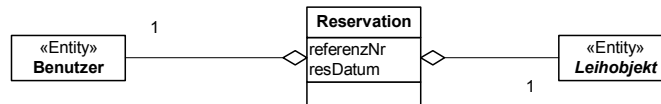
- Realisierung als Tabelle:

```
create table Reservation (  
    idBenutzer    integer,  
    idLeihobjekt integer,  
    resDatum      date,  
    foreign key (idBenutzer)  
        references Benutzer(idBenutzer) on delete cascade,  
    foreign key (idLeihobjekt)  
        references Leihobjekt(idLeihobjekt) on delete cascade,  
)
```

- in Rahmen einer Assoziationsklasse kann ein Objekt der einen Klasse nur *eine* Beziehung mit einem Objekt der anderen Klasse haben. *Mehrere* Reservationen desselben Buches für *einen bestimmten* desselben Benutzers sind nicht erlaubt.

Assoziationsklasse ff

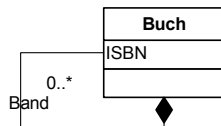
- In der Regel werden Assoziationsklassen vermieden: Sobald eine Assoziation Attribute hat, ist eine eigenständige Klasse fast immer gerechtfertigt.
- Im Bibliotheksbeispiel ist eine Reservation durchaus ein eigenständiges Objekt mit einer eigenen Identität (analog zur Entleihung).



```
create table Reservation (
    referenzNr integer,
    primary key(referenzNr),
    idBenutzer integer,
    idLeihobjekt integer,
    resDatum date,
    ...
)
```

Rekursive Assoziation

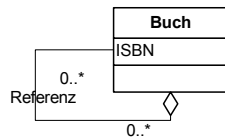
- Assoziationen können rekursiv sein. Die Abbildung in relationale Tabellen bereitet keine besonderen Schwierigkeiten.
- Im folgenden Beispiel liegt eine Assoziation mit einseitiger 0..* Multiplizität vor. Daher kann die Assoziation in der Tabelle Buch dargestellt werden:



```
create table Buch (
    idLeihobjekt integer,
    idLeihobjektParent integer null,
    isbn varchar(13),
    primary key(idLeihobjekt),
    foreign key(idLeihobjektParent)
    references Buch(idLeihobjekt),
    ...
)
```

Rekursive Assoziation ff

- Im folgenden Beispiel liegt eine Assoziation mit beidseitiger 0..* Multiplizität vor. Daher wird die Assoziation in einer speziellen Tabelle dargestellt:



```
create table Buch (
    idLeihobjekt integer,
    isbn          varchar(13),
    primary key   (idLeihobjekt)
    ...
)
```

```
create table Referenz (
    idLeihobjektParent integer,
    idLeihobjektChild  integer,
    foreign key(idLeihobjektParent) references Buch(idLeihobjekt),
    foreign key(idLeihobjektChild)  references Buch(idLeihobjekt),
    ...
)
```

Navigierbarkeit

- In UML-Diagrammen wird oft mit Navigierbarkeits-Angaben gearbeitet. Beispiel:



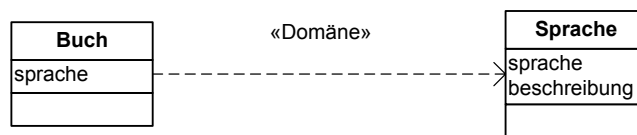
- In der Applikation bedeutet dies, dass aus Sicht des Buches der referenzierende Katalog nicht bekannt ist, während im folgenden Beispiel sowohl vom Katalog auf seine Bücher, wie von einem Buch auf seinen Katalog navigiert werden kann.



- Für die Abbildung in Tabellen ist die Navigierbarkeit nicht relevant. Primär-/Fremdschlüsselbeziehung implizieren nicht irgendeine Navigierbarkeit beim Arbeiten mit SQL-Abfragen.

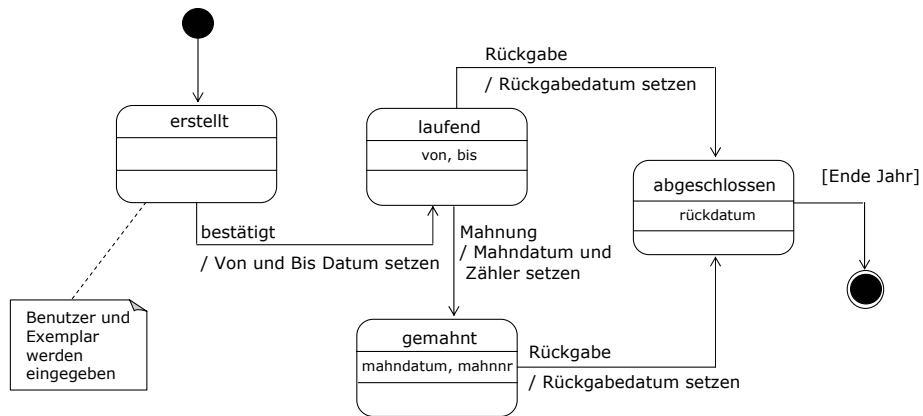
Abhängigkeit

- Abhängigkeiten wurden bereits benutzt um zu zeigen, dass ein Trigger oder eine Constraint-Prüfung von Inhalten einer anderen Tabelle abhängig sind.
- Mit einer Abhängigkeit könnte auch ausgedrückt werden, dass ein Attribut der einen Tabelle (Hier 'sprache' in 'Buch') seine Werte aus einer anderen Tabelle (Hier 'sprache' in 'Sprache') bezieht, *ohne dass eine eigentliche Fremdschlüssel/Primärschlüsselbeziehung vorliegen soll.*



Zustandsdiagramm

- In UML kann einer Klasse ein Zustandsdiagramm zugeordnet werden. Beispiel für die Klasse 'Entleihung':



Zustandsdiagramm, Trigger

- Die Abbildung dieses Zustandsdiagrammes in die Datenbank ist wünschenswert:
 - um das Setzen von Zustandsvariablen zu automatisieren
 - um verbotene Übergänge zu verhindern
- Das Mittel der Wahl sind Trigger. Diese treten in Aktion, wenn das Objekt erzeugt, geändert oder gelöscht wird. Trigger-auslösende SQL-Befehle sind also beispielsweise:

```
insert Entleiher values ( ... )
```

```
update Entleiher set status = ...
```

- Im folgenden wird die Klasse 'Entleiher', erweitert um die Attribute 'mahndatum', 'mahnr' und 'rückdatum' betrachtet.

Zustandsdiagramm, Trigger ff

- Um beim Erstellen einer neuen Entleihung automatisch den Zustand 'erstellt' zu setzen, wird folgender Trigger verwendet:

```
create trigger t1 after insert on Entleihung
referencing new row as new
for each row
update Entleihung set status = 'erstellt'
where Entleihung.idLeihobjekt = new.idLeihobjekt
```

Zustandsdiagramm, Trigger ff

- Um beim Ändern des Zustandes automatisch die relevanten Zustandsvariablen zu setzen werden z.B. folgende Trigger verwendet:

```
/* Ereignis 'bestätigt' der Klasse Entleiherung,  
Übergang zu 'laufend' */
```

```
create trigger t2 after update on Entleiherung  
referencing new row as new  
for each row  
when( old.status = 'erstellt' and new.status = 'laufend' )  
update Entleiherung set von = current_date,  
                        bis = curren_date + interval '30' day  
where Entleiherung.idEntleiherung = new.idEntleiherung
```

Zustandsdiagramm, Trigger ff

```
/* Ereignis 'Mahnung' der Klasse Entleiherung,  
   Übergang von 'laufend' zu 'gemahnt' */  
  
create trigger t3 after update on Entleiherung  
referencing old row as old, new row as new  
for each row  
when( old.status = 'laufend' and new.status = 'gemahnt' )  
update Entleiherung set mahndatum = current_date,  
                    mahn timer = mahn timer + 1  
where Entleiherung.idEntleiherung = new.idEntleiherung
```

Zustandsdiagramm, Trigger ff

```
/* Ereignis 'Rückgabe' der Klasse Entleiherung,  
   Übergang von 'laufend' oder 'gemahnt' zu 'abgeschlossen' */  
  
create trigger t4 after update on Entleiherung  
referencing old row as old, new row as new  
for each row  
when( (old.status = 'laufend' or old.status = 'gemahnt') and  
       new.status = 'abgeschlossen' )  
update Entleiherung set rückdatum = current_date  
where Entleiherung.idEntleiherung = new.idEntleiherung
```

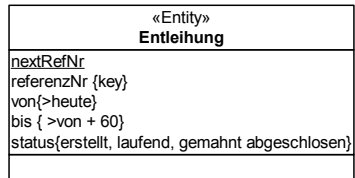
Zustandsdiagramm, Trigger ff

- Um beim Ändern des Zustandes zu prüfen, ob der Übergang erlaubt ist, wird beispielsweise folgender Trigger verwendet:

```
/* Ereignis 'Rückgabe' der Klasse Entleihung,  
   Übergang von 'laufend' oder 'gemahnt' zu 'abgeschlossen' */  
  
create trigger t5 after update on Entleihung  
referencing old row as old, new row as new  
for each row  
when( not((old.status = 'erstellt' and new.status = 'laufend')  
          or ( ... ) or ( ... )) )  
rollback transaction /* nicht SQL-99 Standard, aber häufig */
```

Constraints

- UML erlaubt es, Zusicherungen oder Constraints zu definieren.
- Eine Zusicherung beschreibt eine Bedingung an Attribute, Objekte, Objektkollektionen usw.
- Einfache Zusicherungen können mit geschweiften Klammern bei den entsprechenden Modell-Elementen aufgeführt werden.



- Komplexere Constraints können mit OCL , booleschen Ausdrücken, Pseudocode etc. ausgedrückt werden.

Constraints und Assertions in SQL

- SQL Constraints und SQL-Assertions eignen sich sehr gut für die Implementation von UML-Constraints von Entity-Klassen.
- für die Implementation des UML-Constraints {key} eignet sich beispielsweise der SQL Table-Constraint primary key oder unique
- Für die boolsche Bedingungen {bis > von + 60} eignet sich beispielsweise der check Table Constraint, beispielsweise:

```
create table Entleiherung (  
    date von,  
    date bis check ( bis > (von + interval 60 days) )  
    ...  
)
```

SQL-99 verlangt von Constraints, dass zum gegebenen Überprüfungs-Zeitpunkt (unmittelbar nach Ausführung eines SQL-Befehles auf der entsprechenden Tabelle oder am Schluss der Transaktion) folgende Bedingung erfüllt ist:

```
NOT EXISTS ( SELECT * FROM table WHERE NOT ( check-condition ) )
```

table ist die Tabelle in der der Constraint definiert ist. Eine *leere* Tabelle kann als keine Constraints verletzen und *gelöschte* Datensätze können ebenfalls nicht dazu führen, dass der Constraint verletzt ist. Hingegen ist es Sache der Datenbank, zu merken, dass die Prüfung eines Constraints ausgelöst werden muss. Dies ist eventuell nicht nur bei Änderungen der "eigenen" Tabelle notwendig, sondern auch bei Änderungen in Tabellen, auf die sich die Constraint-Bedingung bezieht.

Bei den marktbeherrschenden Produkten dürfen check-Klauseln (noch) keine Unterabfragen enthalten, sondern lediglich einfache Bedingungen, welche innerhalb des Datensatzes geprüft werden können, beispielsweise
`check( char_length( name ) > 2 )`

Constraints und Assertions in SQL

- Für die Aufzählung status {'erstellt', 'laufend', 'gemahnt', 'abgeschlossen'} eignet sich beispielsweise die Definition einer SQL-Domäne:

```
create domain Status as varchar(32)
    check ( status in( 'erstellt', 'laufend',
                      'gemahnt', 'abgeschlossen') )

create table Entleiher (
    ...
    status Status
)
```

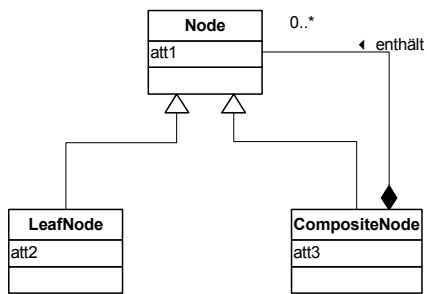
- Assertions sind *globale* Constraints (nicht an eine Tabelle oder Domäne gebunden).

```
create assertion ass_name check ( search_condition )
```

Design Patterns

- Einige Design Patterns sind in UML Datenstrukturen oft anzutreffen. Zwei sollen hier betrachtet werden:
- Composite
 - Eine Baumstruktur von Objekten. Es werden Blattknoten und zusammengesetzte Knoten unterschieden. Beispiele:
Textdokument, Teile einer Maschine, Dateisystem, XML-Datei.
- Dynamic Attribute
 - Die Eigenschaften von Objekten einer Klasse unterscheiden sich und können zeitlich variieren. Beispiel: Jedes Objekt der Klasse Gerät hat unterschiedliche Konfigurationseigenschaften.
Personen verschiedene Fähigkeiten in unterschiedlicher Tiefe.

Composite Design Pattern

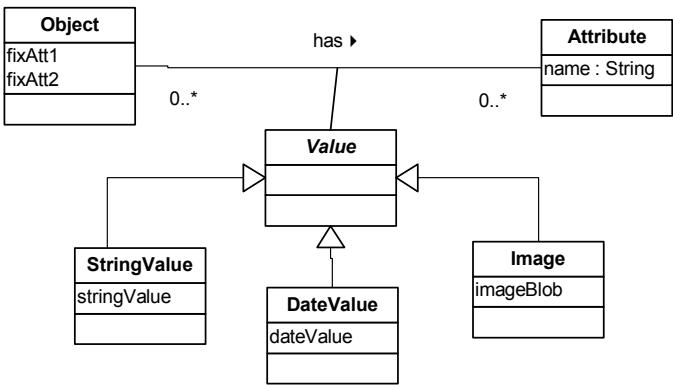


```
create table Node (
    idNode integer,
    idParentNode integer,
    att1 ...
)

create table LeafNode (
    idNode integer,
    att2 ...
)

create table CompositeNode (
    idNode integer,
    att3 ...
)
```

Dynamic Attribute Design Pattern



Dynamic Attribute Design Pattern - Aufgaben

1. Dynamic Attribute Pattern: Abbildung in Tabellen ohne den Vererbungsteil?
2. Dynamic Attribute Pattern: Abbildung in Tabellen mit dem Vererbungsteil?

Normalisierung

- Das aus einem Klassendiagramm hergeleitete Datenmodell ist in jedem Fall auf Normalisierung zu prüfen. Das heisst für *jede* Tabelle:
 - Nur atomare Domänen für Attribute (ausser bei OORDBMS)
 - Keine partiellen Schlüsselabhängigkeiten
 - Keine transitiven Abhängigkeiten von Attributen