Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät
Department Informatik
Lehrstuhl 6 für Datenmanagement

Skript zur Veranstaltung

Konzeptionelle Modellierung

gehalten im Wintersemester 2016/2017
von Prof. Dr. Richard Lenz

Merkblatt zur Veranstaltung

Konzeptionelle Modellierung

Merkblatt: „Einführung/Begrifflichkeiten“

Um sich genauer mit Themen der Konzeptionellen Modellierung auseinandersetzten zu können, bedarf es einiger Grundbegriffe, die im Folgenden erläutert werden.

Definition (Abstraktion)

Als Abstraktion bezeichnen wir die zweckgerichtete Vereinfachung durch Weglassen von Details.

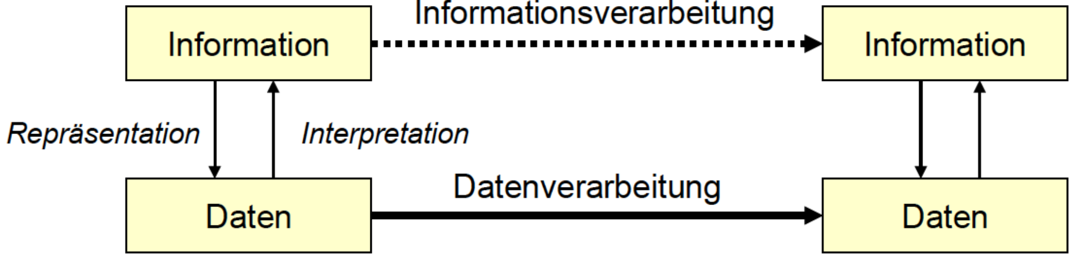
Definition (Modell)

Als ein Modell bezeichnen wir ein zweckgerichtetes, vereinfachtes Abbild der Wirklichkeit. Dabei dient ein Modell verschiedenen Zwecken:

- Spezifizierung (von Software-Systemen)
- Konstruktion (von Software-Systemen)
- Visualisierung (von Software-Systemen)
- Dokumentation (von Software-Systemen)

Wichtig: Daten ↔ Information

Während Daten reine Bitfolgen sind, so braucht es eine Interpretation um zur gewünschten Information zu gelangen.



PROBLEM: BIG DATA

Definition (Datenbank (DB))

Eine Datenbank ist eine Sammlung zusammenhängender Daten. Sie repräsentiert einen Ausschnitt der realen Welt (sogenannte „Miniwelt“), hat einen definierten Zweck und ist eine logisch kohärente Sammlung von Daten. Eine DB besteht aus Nutz- und Metadaten.

Warum Datenbanken?

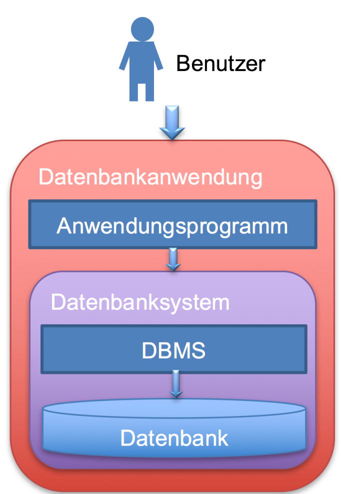
Datenbanken werden immer dann verwendet, wenn es sich zum Beispiel um große Software-Systeme handelt, oder viele Anwendungen/Benutzer mit den gleichen Daten arbeiten. Man verwendet Datenbanken auch, wenn Daten nach Ende eines Programms noch erhalten werden, vor Verlust geschützt werden oder konsistent bleiben sollen.
Im Vergleich zum Dateisystem ist eine Datenbank anwendungsunabhängig!

Vor- und Nachteile von Datenbanken

Vorteile	Nachteile
Vermeidung redundanter Daten Zentrale Kontrolle der Datenintegrität Synchronisation im Mehrbenutzerbetrieb	Hohe initiale Kosten
Fehlertoleranz Performanz Erweiterbarkeit/Flexibilität/Skalierbarkeit	„general-purpose software“ → Konfiguration notwendig
Verkürzte Entwicklungszeiten für Anwendungen Umsetzung von Standards	Signifikanter Overhead → Rechenleistung erforderlich

Definition (Datenbank-Management-System (DBMS))

Als ein Datenbankmanagementsystem (DBMS) bezeichnen wir eine Sammlung von Programmen zur Verwaltung einer Datenbank. Die Aufgaben sind, beschränken sich aber nicht nur auf, die Erzeugung und Wartung einer DB sowie der Konsistente Zugriff auf eine solche.



Definition (Datenbanksystem (DBS))

Als Datenbanksystem verstehen wir die Kombination aus DB und DBMS.

Definition (Datenbankanwendung (DBA))

Als Datenbankanwendung verstehen wir die Kombination aus DBS und Anwendungsprogrammen.

Definition (Datenmodell)

Als das Datenmodell bezeichnen wir eine Strukturierungsvorschrift für Daten. Vereinfacht: Das Datenmodell ist die „Sprache“ zur Beschreibung von Datenstrukturen.

Definition (Datenbankschema)

Als Datenbankschema bezeichnen wir die Beschreibung einer konkreten Datenbank.

Definition (Nutzdaten)

Als Nutzdaten bezeichnen wir die „eigentliche“ Datenbank, also die Daten, welche in der DB gespeichert werden sollen.

Definition (Metadaten)

Als Metadaten definieren wir die Daten „über“ die Daten, welche unter anderem die Struktur der Datenbank resp. die Informationen über deren Speicherungsstrukturen beinhaltet.

Definition (Konzeptionelles Schema)

Als Konzeptionelles Schema bezeichnen wir ein Schema, welches sämtliche Daten auf logischer Ebene beschreibt.

Definition (Externes Schema)

Als Externes Schema bezeichnen wir das Schema, welches den für eine Anwendung relevanten Teil einer Datenbank auf logischer Ebene beschreibt.

Definition (Internes Schema)

Als Internes Schema bezeichnen wir das Schema, welches die internen Speicherungsstrukturen einer Datenbank beschreibt. Dieses Schema ist immer unsichtbar für Anwendungen jeglicher Art!

Definition (Drei-Schema-Architektur nach ANSI/SPARC)

Nach ANSI/SPARC teilen wir die Beschreibungsdaten auf die verschiedenen Schemaebenen auf. Für die externen Schemata gilt, sie bestehen aus anwendungsspezifischen Sichten. Für das konzeptionelle Schema gilt der Grundsatz der Datenunabhängigkeit und Anwendungsneutralität. Die internen Schemata bestehen aus den Zugriffspfaden und den Speicherungsstrukturen.

Definition (Datenunabhängigkeit)

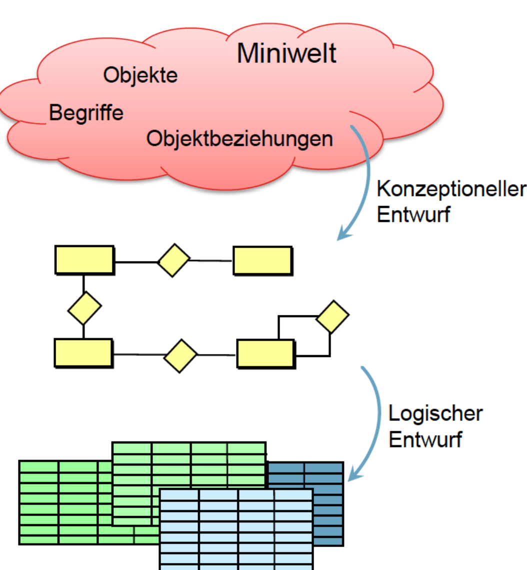
Eine Anwendung heißt datenunabhängig, wenn sie die Daten ohne Bezug auf Details der Speicherung benutzt. Dabei sind also Speichergeräte, -strukturen, Pointer, Adressen etc. irrelevant und deshalb auch optimierbar.
Zudem wird eine rein logische (abstrakte) Sicht auf die Daten gewährleistet. Man kann in anderem Kontext auch von Datenabstraktion, Datenkapselung oder auch von Information Hiding sprechen.

Definition (Anwendungsneutralität)

Eine Datenbank heißt anwendungsneutral, wenn sie offen für die Wiederverwendung durch neue Anwendungen ist. Somit sind die Schemata nicht nur für die eine erste Anwendung angelegt, sondern so, dass auch andere Zugriffe möglich sind.
Ebenfalls gilt: Anwendungsspezifische Beschreibungsdaten sollen nicht im konzeptionellen Schema verankert sein.

Datenmodellierung: Wie kommt man zum konzeptionellen Datenbankschema?

Die gefundene Miniwelt wird auf ein Informationsmodell (auch: „semantisches Datenmodell“) abgebildet („Konzeptioneller Entwurf“). Dieses Informationsmodell wird dann im nächsten Schritt auf das vom DBMS unterstützte Datenmodell abgebildet („Logischer Entwurf“).



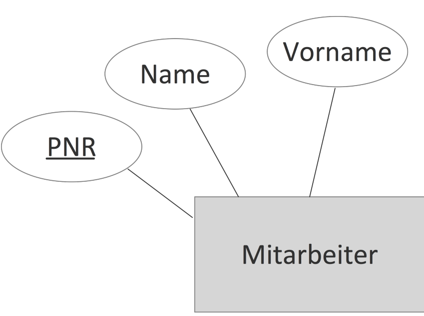
Merkblatt zur Veranstaltung

Konzeptionelle Modellierung

Merkblatt: „(E)E/R Modell“

Es gibt verschiedene Modelle um die Miniwelt zu modellieren, das wohl einfachste und auch sehr mächtige ist das (Enhanced) Entity/Relationship Modell. Dessen Schwerpunkt liegt auf Entitäten und Beziehungen.

Definition (Grundlegende Dinge über Entitäten)



Wir wollen als **Entity** das zu beschreibende **Objekt** bezeichnen. Der **Entity-Typ** einer Entität lässt sich mit der zugehörigen Klasse beschreiben, ist also eine **Beschreibung gleichartiger Objekte**. Als Entity-Menge bezeichnen wir eine Menge von gleichartigen Entitäten.
Eine Entität besteht aus **Attributen** (**Eigenschaften oder Merkmale von einzelnen Entitäten eines Entity-Typs**). Besonders hervorzuheben sind hier die **Schlüsselattribute**, ein einzelnes Attribut oder ein zusammengesetztes Attribut, **welches** eine Entität **eindeutig identifiziert**.
Für eine **Entität** sind folgende **Merkmale** besonders zu beachten:

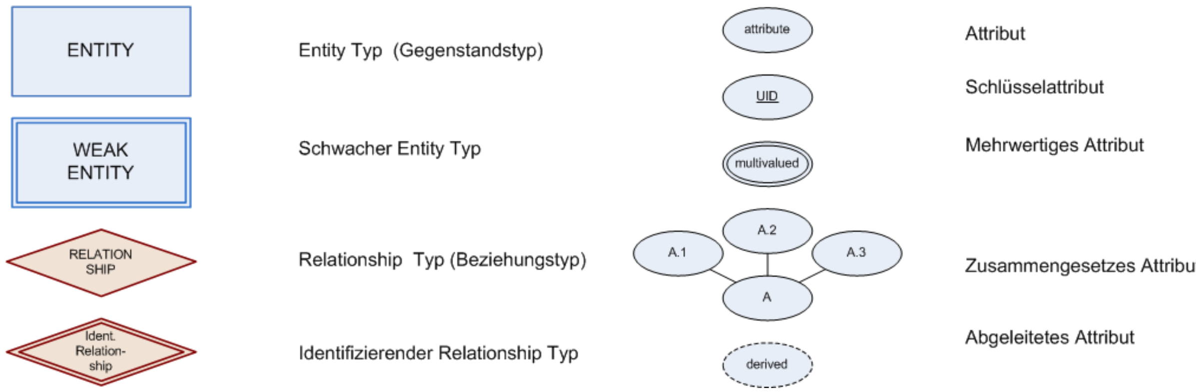
- Eine Entität beschreibt eine **eigenständige Existenz**
- Eine Entität muss **identifizierbar** sein
- Eine Entität muss **beschreibbar** über die zugeordneten Merkmale resp. **Attribute** sein
- Eine Entität muss **relevant** sein.

Wir wollen zwischen der **Intension** (dem Entity-Typen) und der **Extension** (der Entity-Menge) unterscheiden. Wichtig für die Intension ist, dass der Entity-Typ noch **keine einzelne Entität eindeutig beschreibt**. Der Entity-Typ beschreibt eine **Menge** von Entitäten mit **gleichen Attributen** aber meist **unterschiedlichen** Attribut**werten**. Der Entity-Typ wird durch den **Namen** und die **Attribute beschrieben**/identifiziert, das Entity wird durch die

Attributwerte, den **Entity-Typen** und das *Schlüsselattribut* **beschrieben**/*identifiziert* über letzteres.

Definition (Symbolik im E/R-Diagramm)

Im E/R-Diagramm ist folgende Symbolik einzuhalten.



Definition (Grundlegende Dinge zu Attributen)

Als Attribut bezeichnen wir die Eigenschaften, welche eine Entität (also eine konkrete Ausprägung eines Entity-Typens) beschreiben (siehe Definition (Grundlegende Dinge über Entitäten)). Als Attributwert wollen wir im Folgenden den **konkreten Wert eines Attributes** für eine **spezifische** Entität bezeichnen. Als Besonderheit gilt hier der Attributwert **NULL**[1]. Wir wollen die Struktur von Attributen klassifizieren in **atomare** und **zusammengesetzte** Attribute. Wir wollen zudem die mögliche Anzahl an Attributwerten pro Attribut klassifizieren in **ein-** und **mehrwertige** Attribute. Attribute können **gespeichert** werden oder aus bereits gespeicherten Werten **abgeleitet** werden.

Definition (Komplexes Attribut)

Als **komplex** wollen wir ein Attribut bezeichnen, dass sowohl aus **Komposition** als auch **Mehrwertigkeit** besteht.

Definition (Wertebereich eines Attributs)

Jedes Attribut hat per se einen **zugeordneten Wertebereich**. Wir wollen diesen als die Menge der **zulässigen Attributwerte** definieren. Wertebereiche werden im E/R-Diagramm **nicht** mit angegeben. Es ist anzumerken, dass NULL Element **keines** Wertebereichs ist.

[1] NULL bedeutet, dass der Wert entweder nicht existiert, existiert aber nicht bekannt ist oder dessen Existenz unbekannt ist. Über den Umgang mit NULL-Werten wird auch in Fachkreisen stark diskutiert.

Definition (Attribut)

Sei A ein beliebiges Attribut eines beliebigen Entity-Typen E mit dem Wertebereich V und der Entity-Menge $\mathcal{M}_E$, so gelte

$$A: E \rightarrow \mathfrak{P}(V) \text{ mit } \mathfrak{P}(V) \text{ als Potenzmenge von } V.$$

Also ist das Attribut A die Abbildung einer **spezifischen** Entität $e \in \mathcal{M}_E$ auf eine **Teilmenge des Wertebereichs**.

Wir bezeichnen den Attributwert einer spezifischen Entität $e \in \mathcal{M}_E$ mit $A(e)$. Für einwertige Attribute gilt $\#A(e) = 1$, also dass der Attributwert aus nur einem Element besteht (**Singelton**). Für **mehrwertige** Attribute ist der Attributwert eine **mehrelementige Menge**, es gilt also $\#A(e) \leqslant \#V$. Für **NULL-Werte** gilt $A(e) = \emptyset$, da sie ja kein Element des Wertebereichs sind.

Sollte A ein **zusammengesetztes** Attribut aus den Attributen $A_1, A_2, \dots, A_n$ mit den Wertebereichen $V_1, V_2, \dots, V_n$ sein, so gilt für den Wertebereich von A

$$V = \mathfrak{P}(V_1) \times \mathfrak{P}(V_2) \times \dots \times \mathfrak{P}(V_n)$$

also, dass V das **Kreuzprodukt der Potenzmengen aller Wertebereiche der zusammensetzenden Attribute** ist.

Definition (Schlüsselattribut)

Als ein **Schlüsselattribut** wollen wir ein Attribut A bezeichnet, über dessen **Attributwert** A(e) die Entität e **eindeutig identifiziert** werden kann.

Wenn eine **Gruppe** von Attributen dazu benötigt wird, so ist dies (nach Elmasri-Navathe) durch ein **zusammengesetztes** Attribut zu kennzeichnen[2].

Es gibt Entity-Typen, welche **mehr** als ein Schlüsselattribut besitzen, dann bezeichnen wir diese als **Schlüsselkandidaten**. Sogenannte **schwache Entity-Typen** besitzen **keine Schlüsselattribute**.

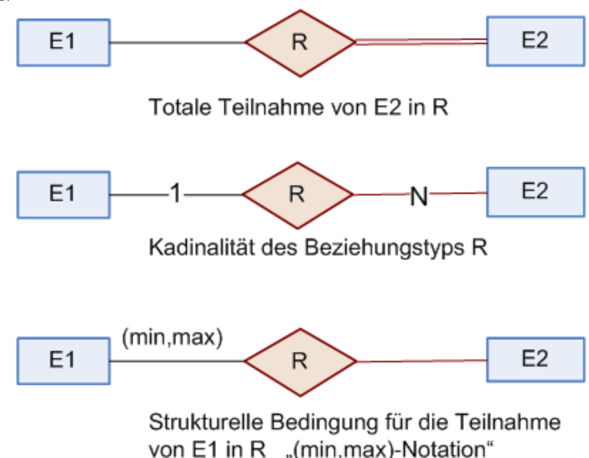
Definition (Beziehungen)

Eine **Beziehung** (auch: Relationship) besteht aus **zwei** (oder **mehr**) **Entitäten**. Wenn wir gleichartige Beziehungen beschreiben, so nennen wir diese **Beschreibung Beziehungstyp** (auch: Relationship-Typ). Als eine **Menge** gleichartiger Beziehungen bezeichnen wir die **Beziehungsmenge** (auch: Relationship-Menge). Beziehungstypen werden im E/R-Diagramm durch **Rauten** dargestellt.

[2] Wichtig: Ein zusammengesetztes Schlüsselattribut muss minimal sein.

Definition (Symbolik im E/R-Diagramm (II))

Für Beziehungen und deren Kardinalitäten gilt es gemeinhin sie im E/R-Diagramm wie folgt zu notieren:



Eine Beziehung muss also mindestens zwei und kann bis zu n Entitäten einschließen. Wichtig: Eine Beziehung besteht nur zwischen Entitäten, nicht aber zwischen Entity-Typen! Denn die Beziehung ist existenzabhängig. Heißt: Ohne beteiligte Entitäten keine Beziehung.

Es gilt grundsätzlich: Ein $\left\{\begin{matrix}\text{Entity}\\ \text{Entity-Typ}\end{matrix}\right\}$ kann an $\left\{\begin{matrix}\text{einer Beziehung}\\ \text{einem Beziehungstypen}\end{matrix}\right\}$ teilnehmen.

Ein Beziehungstyp kann durchaus Attribute definieren, allerdings keine Schlüsselattribute, da eine Beziehung immer über die beteiligten Entitäten identifiziert wird. Der Beziehungstyp und die Beziehungsmenge werden mit dem gleichen Namen bezeichnet.

Definition (Beziehungstyp)

Ein Beziehungstyp B zwischen n Entity-Typen $E_1, \dots, E_n$ definiert eine Beziehungsmenge $\mathcal{M}_B$ zwischen einzelnen Entitäten der beteiligten Entity-Typen. Somit gilt für B:

$$B \subseteq E_1 \times E_2 \times \dots \times E_n$$

Definition (Beziehungsmenge)

Eine Beziehungsmenge B ist eine Menge von individuellen Beziehungen b_i, wobei jede Instanz b_i mit n Entitäten $(e_1, \dots, e_n)$ assoziiert ist und jede Entität e_j in b_i eine Instanz des Entity-Typs E_j mit $1 \leqslant j \leqslant n$ ist.

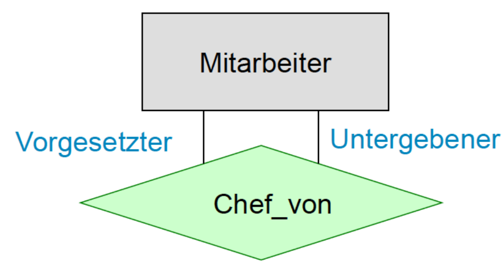
Wir stellen eine Beziehungsmenge (also die Extension eines Beziehungstyps) als Tabelle dar, indem wir für jeden beteiligten Entity-Typ E_j eine Spalte definieren und für jede Beziehungsinstanz b_i eine Zeile definieren.

Definition (Grad eines Beziehungstyps)

Als Grad eines Beziehungstyps bezeichnen wir die **Anzahl der an der Beziehung beteiligten Entity-Typen**.

Definition (Rekursive Beziehungstypen)

Sobald ein Entity-Typ E_j **mehrfach** an einem Beziehungstypen **teilnimmt**, heißt dieser Beziehungstyp **rekursiv bzgl.** E_j. Wir **benennen** jedes Vorkommen eines Entity-Typs mit einem sogenannten **Rollennamen**, mit welchem wir **ausgehende Kanten** des Beziehungstyps beschriften.



Definition (Kardinalität)

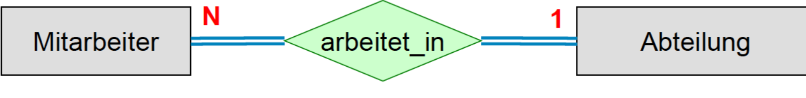
Kardinalitäten sind **Mengenangaben**, mit denen in ER-Diagrammen für jeden **Beziehungstyp** festgelegt wird**, wie viele** Entitäten eines Entity-Typs mit **genau einer** Entität **der anderen** am Beziehungstyp **beteiligten Entity-Typen** (und umgekehrt) in Beziehung **stehen können oder müssen**.

Verwendet man die Chen-Notation, so kann jeder Entity-Typ entweder mit der Kardinalität 1 oder N[3] am Beziehungstyp partizipieren. Durch die Kardinalität 1 wird eine Funktionale Abhängigkeit definiert, welche besagt, dass die Entitäten dieses Entity-Typs funktional abhängig von der Kombination der übrigen am Beziehungstyp beteiligten Entity-Typen sind.

Verwendet man die Min-Max-Notation, so wird durch die Angabe „(`min`, `max`)“ je Entitätstyp definiert, dass jede Entität dieses Typs an mindestens `min` und maximal `max` Beziehungen des Beziehungstyps teilnimmt.

Definition (Totale Teilnahme)

Die totale Teilnahme eines Entity-Typs an einem Beziehungstyp wird über eine doppelte Linie ausgedrückt. Die totale Teilnahme definiert, dass **jede** Entität eines am Beziehungstyp unter totaler Teilnahme **partizipierenden** Entity-Typens **mindestens eine Beziehung** eingehen muss. Das Äquivalent dazu in (min, max) -Notation wäre (1, N) resp. (1, 1).



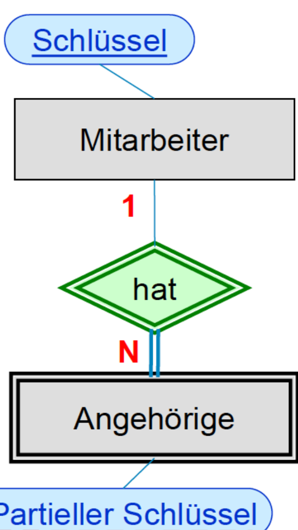
[3] N ist austauschbar mit anderen nicht festen Bezeichnern (z.B. M, O, P, A, B etc.)

Problem an (min, max) -Notation: Funktionale Abhängigkeiten gemäß Chen-Notation bei n-ären Beziehungstypen ($n \geqslant 3$) ist so nicht mehr möglich.

Problem an Chen-Notation: Aussage, eine Entität eines am Beziehungstyp partizipierenden Entity-Typen muss an mindestens m Beziehungsinstanzen teilnehmen ($m \geqslant 2$), ist so nicht auszudrücken.

Definition (Schwache Entitäten)

Als eine schwache Entität bezeichnen wir eine Entität, die von anderen Entitäten existenzabhängig ist. Der Entity-Typ einer schwachen Entität heißt ebenfalls schwach. Ein schwacher Entity-Typ hat keine Schlüsselattribute (eventuell aber partielle Schlüsselattribute) und immer einen identifizierenden Beziehungstyp, gekennzeichnet durch eine Doppellinie der Raute. An einem identifizierenden Beziehungstypen nehmen alle Entity-Typen teil, die für die schwache Entität existensabhängig sind.



Entwurfsentscheidung – Schwacher Entity-Typ ↔ Komplexes Attribut

Wichtig hierfür ist die Frage, ob mehrere identifizierende Entity-Typen beteiligt sind oder der schwache Entity-Typ mit anderen Entity-Typen in „normaler" Beziehung steht.

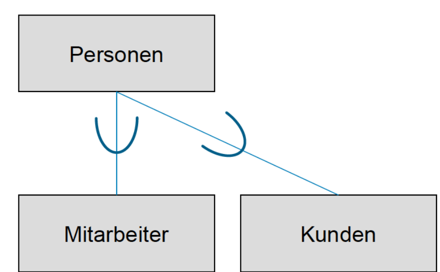
Entwurfsentscheidung – Binäre statt Ternäre Beziehungstypen

Ersetze den ternären Beziehungstyp durch einen schwachen Entitytyp ohne partiellen Schlüssel und mit drei identifizierenden Beziehungstypen. Dies geht allerdings nur dann abhängigkeitserhaltend, wenn es maximal einen Entity-Typen gibt, der mit Kardinalität 1 in den Beziehungstyp eingeht (Dieser ist dann nicht Teil der identifizierenden Beziehungstypen). Diese Entscheidung sollte immer dann getroffen werden, wenn keine ternären Beziehungstypen zur Auswahl stehen. Diese Entwurfsentscheidung sollte auch getroffen werden, wenn manche für eine Beziehung relevante Informationen erst nach Bestehen der Beziehung nachgetragen werden können sollen. Dann aber sollte der ternäre Beziehungstyp in zwei binäre Beziehungstypen umgesetzt werden.

Definition (Ober-/Unterklasse)

Man bezeichnet eine Entity-Menge (resp. deren Typ) auch allgemein als Klasse. Als Unterklasse wollen wir eine Teilmenge dieser Klasse definieren. Als Oberklasse hingegen eine Obermenge jener.

Definition (Symbolik im EE/R-Diagramm (III) – Klassenhierarchien)



Wir definieren einen neuen Beziehungstypen, die IS_A-Beziehung. Diese unterscheidet sich, da es nicht zwei verschiedene Entitäten gibt, welche eine Beziehung eingehen. Vielmehr ist die Beziehung eine Information über die Realweltobjekte.
Eine Entität kann Mitglied in mehreren Unterklassen sein, man spricht von sogenannten sich überlappenden Unterklassen (auch: „overlapping“, abgekürzt: „o“).
Ebenfalls möglich ist die Modellierung paarweise disjunkter Unterklassen. Hier sind die Unterklassen paarweise disjunkt.
Totale Teilnahmen sind auch hier wieder möglich, allerdings nur seitens der Oberklasse, da jede Entität einer Unterklasse per se Entität der Oberklasse ist. Die totale Teilnahme bedeutet dann, dass jede Person in mindestens einer Unterklasse vorkommen muss.

Definition (Vererbung)

Entitäten einer Unterklasse „erben“ alle Attribute sowie Beziehungen der Oberklasse. Ein Schlüssel der Oberklasse ist somit immer auch Schlüssel der Unterklassen. Damit müssen die Unterklassen kein eigenes Schlüsselattribut explizit definieren.

Definition (Spezialisierung)

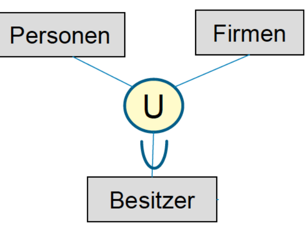
Als Spezialisierung bezeichnen wir den Prozess der Definition von Unterklassen ausgehend von einer Oberklasse. Diese Entwurfsentscheidung ist sinnvoll, wenn ein Attribut nur auf einen Teil der Entitäten eines Entity-Typens zutrifft, oder ein Beziehungstyp nur einen Teil der Entitäten einer Klasse betrifft.

Definition (Generalisierung)

Als Generalisierung bezeichnen wir den Prozess der Definition einer Oberklasse ausgehend von mehreren Unterklassen. Die Generalisierung stellt somit das Gegenstück zur Spezialisierung dar. Eine durch Generalisierung gefundene Oberklasse geht i.A. immer total in die Generalisierungshierarchie ein. Wichtig: Eine Unterklasse braucht keinen Schlüssel, die Oberklasse schon! Deswegen sollte man notfalls einen Surrogatsschlüssel bilden!

Wichtig: Der **Unterschied** zwischen Hierarchie und Netz beläuft sich auf die **Anzahl der erlaubten Oberklassen pro Unterklasse**. In einer Hierarchie ist diese Anzahl auf 1 beschränkt, in einem Netz ist diese Anzahl unbeschränkt.

Definition (Kategorie)

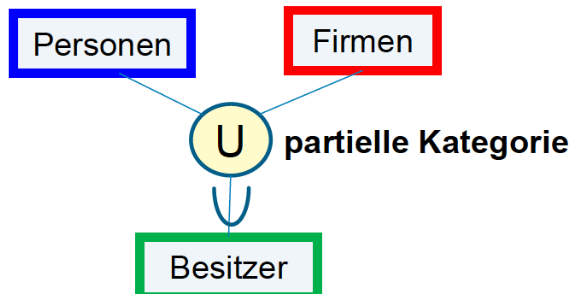


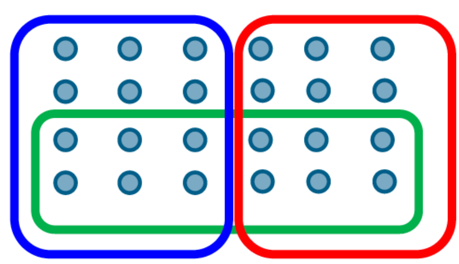
Als eine Kategorie K bezeichnen wir eine Teilmenge der Vereinigungsmenge von mehreren Entitätsmengen $\mathcal{M}_{E_1}, \mathcal{M}_{E_2}, \ldots, \mathcal{M}_{E_n}$. Diese Entwurfsentscheidung ist sinnvoll, wenn nicht jede Entity-Typ gleichzeitig dem Kategorie-Typen entspricht. Für K gilt also:

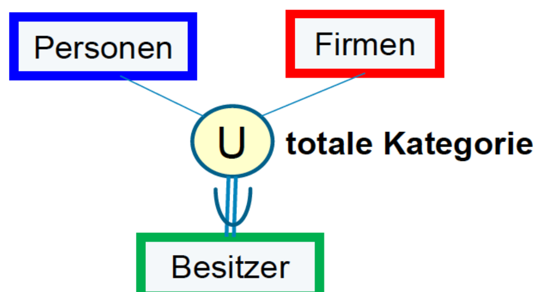
$$K \subseteq \mathcal{M}_{E_1} \cup \mathcal{M}_{E_2} \cup \ldots \cup \mathcal{M}_{E_n}$$

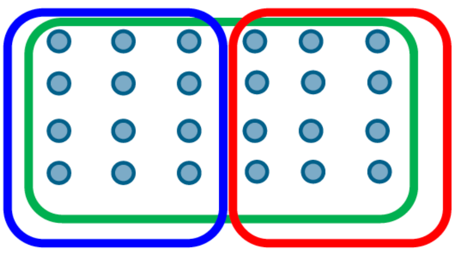
Als eine totale Kategorie bezeichnen wir eine Kategorie, welche eine totale Teilnahme an der Kategorie besitzt, hier gilt für K: $K = \mathcal{M}_{E_1} \cup \mathcal{M}_{E_2} \cup \ldots \cup \mathcal{M}_{E_n}$. Alle anderen Kategorien nennen wir partielle Kategorie.

Wichtig: Eine totale Kategorie ist das Äquivalent zu einer total, disjunkten Oberklasse.









Merkblatt zur Veranstaltung

Konzeptionelle Modellierung

Merkblatt: „Relationales Datenmodell“

Das (E)E/R-Modell ist für eine effektive Datenbeschreibung leider nicht ausreichend, es ist ein rein konzeptionelles Schema. Um für unser DBMS ein zu verwendendes Schema definieren zu können brauchen wir ein Datenmodell. Das Augenmerk liegt hier auf dem sogenannten relationalen Datenmodell, welches auf Basis von Tabellen sich bemüht die Daten der zu beschreibenden Miniwelt aufzufassen.

Definition (Datenmodell und zugehörige Begrifflichkeiten)

Als ein **Datenmodell** bezeichnen wir die „Sprache“ für die Beschreibung von Daten in einem bestimmten Schema. Das Datenmodell ist dabei immer für ein bestimmtes DBMS fest.

Der **Generische Systementwurf** erzwingt einen formalen Rahmenwert für die Beschreibung der Miniwelt. Dieser Systementwurf sieht das Datenmodell als den **formalen Rahmenwert für die Datenbeschreibung**. Es ist somit **strikt** vom **Schema** zu **trennen**, was **anwendungsspezifisch** ist und das Ergebnis der Datendefinition darstellt. Das **Datenmodell** legt die **Bedeutung** des **Schemas** fest, welches wiederum die **eigentlichen Daten beschreibt**.

Die Daten dienen der Repräsentation der Miniwelt und liefern den aktuellen Zustand der selbigen.

Definition (Aufbau eines Datenmodells)

Das Datenmodell ist aus vielen Bestandteilen aufgebaut. So besteht es zum Beispiel aus mehreren **Datentypen**, welche sich als **einfache** oder **zusammengesetzte** Datentypen klassifizieren lassen. **Zusammengesetzte** Datentypen werden über **Konstruktoren** im Datenmodell möglichgemacht.

Es besteht darüber hinaus noch aus **Konsistenzregeln** (also der **Zulässigkeitsüberprüfung**), welche sich in **inhärente** (per Konvention gesetzte) oder **explizite** (für eine spezielle Anwendung geltende) Regeln aufteilen lassen. Zudem hat jedes Datenmodell **Benennungskonventionen** für die Bezeichnung von Datenbankelementen.

Wir wollen uns nun näher mit dem relationalen Datenmodell beschäftigen, eines der heutzutage bekanntesten und weit verbreiteten Modelle. Das relationale Datenmodell liefert **logische** Datenstrukturen für einen **deskriptiven** Zugriff.
Das **Ziel** des relationalen Datenmodells ist die **Abfrage der Daten über inhaltliche Kriterien** und **nicht** mehr über **Speicherungsstrukturen**.
Man geht in diesem Modell deswegen von einer **einheitlichen logischen Datenstruktur** für alle Daten aus.
Die **Datenbank** lässt alle Daten somit in **Tabellen** ablegen. Eine Tabelle besteht aus „Zeilen" und „Spalten" und wird auch als Relation bezeichnet. Eine **Relationale Datenbank** ist eine **Menge von Relationen**.

Definition (Relation)

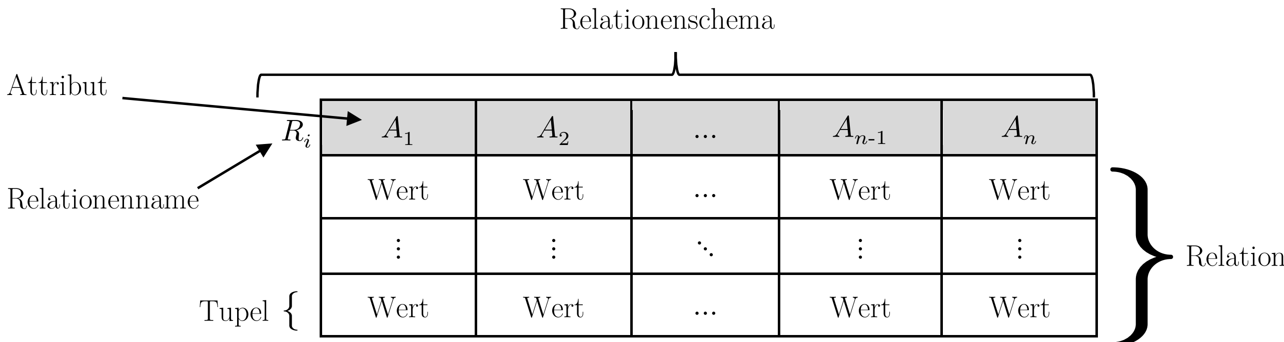
Wir wollen als eine Relation R eine Teilmenge des Kreuzprodukts der Wertebereiche der Attribute bezeichnen:

$$R(A_1, A_2, \ldots, A_n) \subseteq W(A_1) \times W(A_2) \times \ldots \times W(A_n)$$

$R(A_1, A_2, \ldots, A_n)$ entspricht dabei dem Relationenschema einer Relation R. Die Attribute A_i mit $1 \leqslant i \leqslant n$ definieren die einzelnen Komponenten der Tupel. Die i-te Komponente eines Tupels $(a_1, \ldots, a_i, \ldots, a_n)$ entspricht dem Wert des Attributs A_i in diesem Tupel.

Definition (Begrifflichkeiten in Bezug auf das Relationale Datenmodell)

Wir wollen ein **Tupel** als die **Aneinanderreihung atomarer Attributwerte** bezeichnen. Ein **Attribut** entspricht einer **Spalte** der Relation und ist **definiert** über den **Attributnamen und den Wertebereich**. Der **Wertebereich** ist die Menge **aller zulässigen Attributwerte** für ein Attribut. Als ein **Relationenschema** bezeichnen wir die **Beschreibung** einer Relation über den **Namen** und einer zur Relation gehörenden **Menge an Attributen**. Die Menge der Relationenschemata bezeichnet man auch als **Relationales Datenbankschema**.



Wichtig: Eine Relation ist eine Menge, es kann also per Definition niemals **zwei gleich Zeilen** in einer Relation geben. Zudem ist die **Reihenfolge** der Tabellen**zeilen** wie –**spalten unerheblich**.

Definition (Schlüssel)

Wir definieren als **Schlüsselkandidaten** das(die) Attribut(kombination), welche(s) ein Tupel **eindeutig identifizieren** kann. Ein **Superschlüssel** ist ein **minimaler Schlüsselkandidat**. Der **Primärschlüssel** ist der **ausgewählte** Schlüsselkandidat.
Als einen **Fremdschlüssel** bezeichnen wir ein **Attribut**, das mit einem **Primärschlüsselwert** einer (nicht notwendigerweise verschiedenen) Relation **auf** ein **bestimmtes Tupel** jener Relation verweist.

Weiteres zu Attributen – Wertebereiche

Für uns gilt: Alle Attributwerte sind **atomar**. Es sind somit nur skalare Werte ohne Struktur (aus Sicht der DB) und Mehrfachbelegung erlaubt. Dies schließt „große" Datentypen nicht aus!
Jeder Wertebereich hat einen Datentyp und unterstützt die Konsistenzsicherung (*Similia similibus comparantur*).

Weiteres zu Attributen – NULLwerte

NULL ist ein besonderer Attributwert (siehe Merkblatt (E)E/R-Modell). Für alle Wertebereiche gilt: NULL ist nicht Element davon. Wir müssen unsere Attributdefinition somit erweitern um die Fragestellung „Ist NULL erlaubt oder nicht?".
Für Schlüsselkandidaten gilt somit trivialerweise NULL ist nicht erlaubt (NOT NULL).

Weiteres zu Attributen – Erweiterte Attributdefinition

NOT NULL in der Attributdefinition bedeutet, dieses Attribut darf keine NULL-Werte enthalten.

UNIQUE in der Attributdefinition bedeutet, diese(s) Attribut(-kombination) darf keine Duplikate enthalten (NULL-Werte sind aber grundsätzlich erlaubt!)

PRIMARY KEY ($\approx$ NOT NULL + UNIQUE) definiert den Primärschlüssel. Achtung: Pro Relation genau einmal! NOT NULL und UNIQUE reichen dem DBMS nicht aus, um einen Primärschlüssel zu definieren, da dies die Definition eines Schlüsselkandidaten ist. Mit dem Schlüsselwort PRIMARY KEY wird der Schlüsselkandidat als Primärschlüssel ausgewählt.

Definition (Schlüsselkandidat)

Wir wollen eine Kombination K_{PK} als Schlüsselkandidat bezeichnen gdw. K_{PK} **eindeutig** (d.h. $\forall t_1, t_2 \in K_{PK}: t_1 \neq t_2$) und **minimal** (d.h. $\nexists k_{PK} \subset K_{PK}: \forall t_1, t_2 \in k_{PK}: t_1 \neq t_2$) ist. Wichtig: Dies ist nicht Teil der mathematischen Definition → **inhärente Konsistenzregel**.

Das Ziel ist es ein Tupel **eindeutig** zu **identifizieren**. Dies ist **per Definition** einer Relation **immer möglich**, grundsätzlich über das ganze Tupel. Wunsch ist es jetzt dieses **Identifikationsmerkmal zu verkleinern**, also ein Tupel über ein(e) Attribut(-kombination) zu identifizieren. Diese **Attributkombination nennen wir Primärschlüssel**. Es gilt somit: Jede Relation in unserem Sinne hat **immer** einen Primärschlüssel.

Definition (system-enforced-integrity)

Als **system-enforced-integrity (SEI)** wollen wir **Integritätsbedingungen** bezeichnen, deren **Einhaltung vom DBS garantiert** werden. Dadurch gelangen falsche Daten gar nicht erst in den Datenbestand, man muss sich also nicht mehr auf alle Anwendungsentwickler blind verlassen. Zudem bietet die SEI hohe Zuverlässigkeit.

Definition (Fremdschlüssel und Referenzielle Integrität)

Wir wollen als **Fremdschlüssel** ein Attribut, das einen **Primärschlüssel** einer (nicht unbedingt verschiedenen) **Relation** aufnimmt (als sogenannten „**logischen Zeiger**"), bezeichnen.

Dies ist notwendig, um **Beziehungen im relationalen Datenmodell zu modellieren**. Denn **Beziehungen** wie im (E)E/R-Modell **gibt es** so per se **nicht**! Dafür definieren wir logische Zeiger, mit denen diese Beziehungen imitiert werden können.

Wichtige Bedingung: Der **Wertebereich** des Fremdschlüsselattributs muss mit dem des **referenzierten** Primärschlüsselattributs **übereinstimmen**.

Wir wollen nun **folgende Bedingung** garantieren: **Kein Verweis darf ins Leere gehen**, anders ausgedrückt: **Jeder Fremdschlüsselattributwert muss** so auch tatsächlich als **Primärschlüsselattributwert vorkommen**, **NULL** ist aber grundsätzlich **OK** (im Fremdschlüsselattribut). Diese Bedingung wollen wir **referenzielle Integrität** nennen.

Diese Integritätsbedingung ist auch über die SEI im DBS garantiert, sofern wir ein Attribut explizit als **FOREIGN KEY** bekannt machen.

Die **referenzielle Integrität** kann nur beim **Ändern oder Löschen** eines Datensatzes **verletzt** werden (beim Einfügen blockiert die SEI). Es gibt **verschiedene** Maßnahmen, wie man auf das **Verletzen reagieren** kann:

- **RESTRICTED** – Das DBS lehnt die Operation ab
- **CASCADES** – Alle referenzierenden Tupel werden auch gelöscht/modifiziert
- **NULLIFY** – Alle Referenzen werden auf NULL gesetzt (sofern dies erlaubt ist)
- **SET DEFAULT** – Alle Referenzen werden auf den Standardwert gesetzt (sofern definiert)

Definition (Benutzerdefinierte Integrität)

Als benutzerdefinierte Integriätsbedingungen bezeichnen wir Bedingungen aus der Anwendungsdomäne, die explizit formuliert werden müssen. Diese werden durch das DBS kontrolliert. Beispiele hierfür: Prädikate auf Attributen und Wertebereicheinschränkungen.

Merkblatt zur Veranstaltung

Konzeptionelle Modellierung

Merkblatt: „Mapping“

Bisher wurde das (E)E/R-Modell vom Relationalen Datenmodell getrennt behandelt. Wie kommt man jetzt aber vom (E)E/R-Modell in das Relationale Datenmodell? Dafür „mappen“ wir das (E)E/R-Modell. Dieses Merkblatt beschreibt die 9 Schritte, welche grundsätzlich nötig sind, um jedes (E)E/R-Modell zu überführen.

Schritt 1 – Starke Entity-Typen

Für jeden starken Entity-Typen E im EE/R-Diagramm erzeuge eine Relation R, welche alle einfachen Attribut von E umfasst. Übernimm für jedes zusammengesetzte Attribut nur die Komponenten als eigenständige Attribute.
Wähle von den Schlüsselattributen des Entity-Typs E eines als Primärschlüssel von R aus. Ist der ausgewählte Schlüssel zusammengesetzt, bilden die Komponenten den Primärschlüssel zusammen.
Kennzeichne jedes im EE/R-Diagramm als Schlüssel markierte, aber nicht als Primärschlüssel ausgewählte, Attribut als UNIQUE und NOT NULL.

Schritt 2 – Schwache Entity-Typen

Erzeuge für jeden schwachen Entitätstyp W mit identifizierendem/n Typen $E_{1,\ldots,n}$ eine Relation R, die alle einfachen Attribut von W umfasst. Übernimm für jedes zusammengesetzte Attribut nur die Komponenten als eigenständige Attribute. Füge als Fremdschlüsselattribute die Primärschlüsselattribute aller identifizierenden Typen $E_{1,\ldots,n}$ hinzu. Dabei ist zur Sicherstellung der referenziellen Integrität CASCADE zu wählen. Der Primärschlüssel von R ist die Kombination aller Fremdschlüsselattribute der identifizierenden Typen zusammen mit dem partiellen Schlüssel des schwachen Entity-Typs.

Für Schritt 3 mit 5 gibt es verschiedene Optionen, welche zu wählen ist, hängt von der Semantik der Daten, der Dynamik der Anwendung (Anzahl an Änderungen) und der relativen Anzahl an Beziehungen im Verhältnis zur Kardinalität der beteiligten Relationen ab. Man wähle mit Bedacht und dokumentiert!

Schritt 3 – M:N-Beziehungen

Erzeuge für jeden binären M:N-Beziehungstyp B eine Relation R, die alle einfachen Attribute (resp. deren einfachen Komponenten) von B umfasst. Füge als Fremdschlüsselattribut die Primärschlüsselattribute der beiden Relationen ein, die zu den an B beteiligten Entitätstypen gehören. Dabei ist zur Sicherstellung der referenziellen Integrität CASCADE zu wählen. Der Primärschlüssel der Relation R ist die Kombination der Fremdschlüsselattribute.

Schritt 4 – N:1-Beziehungen

Variante 1 – Beziehungsrelation

Man erstelle für den binären N:1-Beziehungstyp B eine Relation R_B mit zwei Fremdschlüsselattributen zzgl. aller einfachen Attribute (resp. deren einfachen Komponenten) von B. Das Fremdschlüsselattribut der Relation S, welche den Entitätstyp der N-Seite repräsentiert, wird zum Primärschlüssel. Das andere Fremdschlüsselattribut der Relation T muss als NOT NULL deklariert werden.
Problem an dieser Variante: Totale Teilnahme ist nicht darstellbar!
Vorteil dieser Variante: NULL-Werte im Fremdschlüsselattribut werden vermieden.

Variante 2 – Fremdschlüssel

Identifiziere für jeden regulären binären N:1-Beziehungstyp B die Relation R, die dem Entity-Typ E auf der N-Seite des Beziehungstyps entspricht (gemäß Chen-Notation). Füge den Primärschlüssel der anderen Relation S, die den Entity-Typ F auf der 1-Seite von B repräsentiert, als Fremdschlüssel in R ein. Sollte E eine totale Teilnahme an B haben, so ist dieser Fremdschlüssel als NOT NULL zu deklarieren.
Füge dann alle einfachen Attribute (resp. deren einfachen Komponenten) von B als Attribute in R ein.

Schritt 5 – 1:1-Beziehungen

Variante 1 – Beziehungsrelation

Man erstelle für den binären 1:1-Beziehungstyp B eine Relation R_B mit zwei Fremdschlüsselattributen zzgl. aller einfachen Attribute (resp. deren einfachen Komponenten) von B. Das Fremdschlüsselattribut einer beliebigen Teilhaber-Relation S wird zum Primärschlüssel. Das andere Fremdschlüsselattribut der Relation T muss als NOT NULL und UNIQUE deklariert werden.
Problem an dieser Variante: Totale Teilnahme ist nicht darstellbar!
Vorteil dieser Variante: NULL-Werte im Fremdschlüsselattribut werden vermieden.

Variante 2 – Vereinigung der Relationen

Man beachte, dass diese Variante nur möglich ist, sobald beide beteiligte Entitätstypen an der Beziehung total teilnehmen.

Man verschmelze beide Relationen S und T in eine neue Verbundrelation R_{ST}. Man wählen nun einen beliebigen Primärschlüssel der Relationen S und T als neuen Primärschlüssel für R_{ST}. Der andere Primärschlüssel geht als Schlüsselkandidat in die neue Relation ein.

Problem: Konzeptionell schwer zugänglich (quasi „Müll"), da verschiedene eigenständige Entitätstypen in einen neuen Verbund überführt werden, der nichts mehr mit den Entitätstypen zu tun hat.

Vorteil: Performance und weniger Relationen im Schema

Variante 3 – Fremdschlüssel

Identifiziere für jeden regulären binären 1:1-Beziehungstyp B die zugehörigen Relationen S und T, die den beteiligten Entitätstypen entsprechen. Wähle eine Relation aus[1] (hier: T) und füge den Primärschlüssel von S als Fremdschlüssel in T mit dem Zusatz UNIQUE (bei totaler Teilnahme von T noch: NOT NULL) hinzu. Füge alle einfachen Attribut (resp. deren einfachen Komponenten) des Beziehungstyps B als Attribute in T ein.

Schritt 6 – Mehrwertige Attribute

Erzeuge für jedes mehrwertige Attribut A_M eines Entitätstyps E eine Relation R. Füge ein Attribut A zu R hinzu[2]. Füge den Primärschlüssel der zu E gehörenden Relation T als Fremdschlüssel zu R hinzu. Der Primärschlüssel der Relation R ist die Kombination aus dem Fremdschlüssel und A.

Für die Sicherstellung der logischen Integrität, ist hier die Bedingung CASCADE für das Ändern und Löschen zu verwenden.

Schritt 7 – Mehrstellige Beziehungstypen

Erzeuge für jeden mehrstelligen Beziehungstyp B eine Relation R, die alle einfachen Attribute (resp. deren einfache Komponenten) umfasst. Füge als Fremdschlüssel die Primärschlüssel aller Relationen ein, die zu den an B beteiligten Entitätstypen gehören. Der Primärschlüssel der Relation R ist i.A. die Kombination all dieser Fremdschlüssel.

---

[1] Wichtig: Man wähle am besten den Entitätstypen aus, der eine totale Teilnahme am Beziehungstyp besitzt. Sollte es davon keinen geben, so wählt man den Entitätstypen aus, dessen Extension öfters in die Beziehung eingeht (um NULL-Werte) zu vermeiden.

[2] Dieses Attribut entspricht dem abzubildenden Attribut A_M

Achtung: Funktionale Abhängigkeiten müssen abgebildet werden.
Geht genau ein Entitätstyp mit Kardinalität 1 in B ein, so wird der zugehörige Fremdschlüssel nicht mit in den Primärschlüssel aufgenommen.
Gehen mehrere Entitätstypen mit Kardinalität 1 in B ein, so wird aus diesen einer ausgewählt, dessen Fremdschlüssel nicht mit in den Primärschlüssel aufgenommen wird. Die restlichen funktionalen Abhängigkeiten sind über Integritätsbedingungen ebenfalls sicherzustellen.

Schritt 8 – Abbildung der Generalisierung/Spezialisierung

Überführe jede Spezialisierung mit m Unterklassen $\{S_1, S_2, \dots, S_m\}$ und Oberklasse C mit den Attributen $\{k, a_1, \dots, a_n\}$ (mit PK(C)[3] = k) in eine der folgenden 4 Varianten:

Variante 8A Nachbildung der IS_A-Semantik

Erzeuge eine Relation L für C mit Attrs(L)[4] = $\{k, a_1, \dots, a_n\}$ und PK(L) = k. Erzeuge eine Relation L_i für jede Unterklasse S_i $(1 \leqslant i \leqslant m)$ mit den Attributen Attrs(L_i) = $\{k\} \cup$ Attrs(S_i) und PK(L_i) = k. Dabei ist der Primärschlüssel von L_i gleichzeitig Fremdschlüssel von L. Es muss also gelten: $\pi_{<k>}(L_i) \subseteq \pi_{<k>}(L)$.

Variante 8B Disjunkte totale Spezialisierung

Erzeuge eine Relation L_i für jede Unterklasse S_i $(1 \leqslant i \leqslant m)$ mit den Attributen Attrs(L_i) = $\{k, a_1, \dots, a_n\} \cup$ Attrs(S_i) und PK(L_i) = k.

Problem: Funktioniert nur bei disjunkter, totaler Vererbung. Bei partieller Vererbung: Tupel der Oberklasse, welche in keiner Unterklasse sind, gehen verloren. Bei überlappender Spezialisierung werden die Attribute zu einer Entität redundant in mehreren Relationen abgelegt. Ein OUTER JOIN ist erforderlich um die Menge C „wiederherzustellen“.

Variante 8C Disjunkte Spezialisierung

Erzeuge eine **einzelne** Relation L mit den Attributen Attrs(L) = $\{k, a_1, \dots, a_n\} \cup$ Attrs(S_1) $\cup \dots \cup$ Attrs(S_m) $\cup \{t\}$ und PK(L) = k.

Vorteil: Alle Attribute einer Entität finden sich in einem Tupel, somit ist keine JOIN-Operation notwendig.

Problem: Ein diskriminierendes Attribut „t“ → viele NULL-Werte.

Problem: Funktioniert nur bei disjunkter Spezialisierung. Bei partieller Spezialisierung können im diskriminierenden Attribut NULL-Werte stehen → noch mehr NULL-Werte ↯ (aber prinzipiell möglich)

Bei überlappender Spezialisierung – im diskriminierenden Attribut kann nur ein Wert eingetragen werden!

[3] PK(R) beschreibt den Primärschlüssel der Relation R
[4] Attrs(R) beschreibt die Attribute der Relation/des Entitätstyps R

Variante 8D Diskriminierende Attribute

Erzeuge eine **einzelne** Relation L mit den Attributen Attrs(L) = $\{k, a_1, \dots, a_n\} \cup$ Attrs(S_1) ∪ ... ∪ Attrs(S_m) $\cup \{t_1, \dots, t_m\}$ und PK(L) = k.

Vorteil: Alle Attribute einer Entität finden sich in einem Tupel, somit ist keine JOIN-Operation notwendig.

Problem: Viele diskriminierende Attribute „t" → viele NULL-Werte, in den Attrs(S_i).

Für die disjunkte Spezialisierung ist ein logischer Ausdruck als Regel für den gegenseitigen Ausschluss erforderlich (sicherlich nicht optimal).

Diese Variante ist 8A zu bevorzugen, wenn die Unterklassen nur wenig spezifische Attribute haben.

Schritt 9 – Kategorien

Hier ist die Definition der Kategorie nochmal zu betrachten:

> *Eine Kategorie ist eine Teilmenge der Vereinigungsmenge von mehreren Entitätsmengen.*

Wichtig: Obermengen können verschiedene Schlüssel haben! Somit:

Bilde für jede Kategorie K eine Relation R. Führe ein Surrogat s als Primärschlüssel von R ein. Füge s als Fremdschlüssel in alle Oberklassen ein.

Wichtig: Für Kategorien, deren Oberklassen den selben Schlüssel besitzen, braucht es kein Surrogat!

Merkblatt zur Veranstaltung

Konzeptionelle Modellierung

Merkblatt: „Normalisierung“

Das wohl stärkste Problem an nicht normalisierten Relationen ist das eventuelle Auftreten von Anomalien. Die Hauptursache für das Auftreten von Anomalien ist, dass **Informationen, die eigentlich zu verschiedenen Entity-Typen gehört, in einer Relation zusammengefasst** wird.
Man unterscheidet zwischen drei Anomalie-Typen:

1) Einfüge-Anomalie (INSERT)
   Eine Information vom Typ B kann nicht eingefügt werden, ohne zuvor eine Information vom Typ A einzufügen.
2) Lösch-Anomalie (DELETE)
   Durch Löschen einer Information vom Typ A kann u.U. auch Information vom Typ B mitgelöscht werden.
3) Änderungs-Anomalie (UPDATE)
   Änderungen an einem Attribut B_i müssen ggf. in vielen Tupeln durchgeführt werden.

Man merkt auch, dass sich die Anomalien nur auf die Methoden der DML beziehen. Sollten also Relationen sich nicht ändern, so kann man gezielt denormalisieren, um somit performanter zu arbeiten. Solche Entwurfsentscheidungen sind aber immer ausreichend zu begründen und ebenfalls zu dokumentieren!
Um dem Auftreten von Anomalien und der Redundanz vorzubeugen möchte man nun die Relationen in möglichst redudanzfreie Relationen aufspalten. Dies geschieht mittels der Analyse der funktionalen Abhängigkeiten zwischen Attributen.

Definition (Funktionale Abhängigkeit (FA))

Sei $R(A_1, A_2, \dots, A_n)$ eine beliebige Relation, und X, Y Teilmengen der Attributmenge $\{A_1, A_2, \dots, A_n\}$. Wir wollen Y als **funktional abhängig** von X bezeichnen ($X \rightarrow Y$), wenn es keine zwei Tupel geben darf, in denen für gleiche X-Werte verschiedene Y-Werte auftreten.
Man spricht auch X bestimmt / determiniert Y. X bekommt zudem den Namen „Determinatne“

Für funktionale Anhängigkeiten gelten folgende Amstrong-Axiome sowie die Folgerung aus dessen:

Satz (Armstrong-Axiome und Folgerungen)

Sei $R(A_1, A_2, \dots, A_n)$ eine beliebige Relation, und $\alpha, \beta, \gamma, \delta$ Teilmengen der Attributmenge $\{A_1, A_2, \dots, A_n\}$.

Für $\alpha, \beta, \gamma, \delta$ gilt nun:

- $\beta \subseteq \alpha \Rightarrow \alpha \to \beta$ (Reflexivität)
- $\alpha \to \beta \Rightarrow \alpha\gamma \to \beta\gamma$ (Verstärkung)
- $\alpha \to \beta \land \beta \to \gamma \Rightarrow \alpha \to \gamma$ (Transitivität)
- $\alpha \to \beta \land \alpha \to \gamma \Rightarrow \alpha \to \beta\gamma$ (Vereinigung)
- $\alpha \to \beta\gamma \Rightarrow \alpha \to \beta \land \alpha \to \gamma$ (Dekomposition)
- $\alpha \to \beta \land \delta\beta \to \gamma \Rightarrow \alpha\delta \to \gamma$ (Pseudotransitivität)

Definition (Volle Funktionale Abhängigkeit)

Sei $R(A_1, A_2, \dots, A_n)$ eine beliebige Relation, und X, Y Teilmengen der Attributmenge $\{A_1, A_2, \dots, A_n\}$. Wir wollen Y als **voll funktional abhängig** von X bezeichnen, wenn $X \to Y$ und es keine echte Teilmenge Z von X gibt, für die gilt: $Z \to Y$.
Man braucht also alle Attribute aus X um Y zu bestimmen.

Definition (Superschlüssel)

Als Superschlüssel einer Relation R wollen wir ein Attribut oder eine Kombination aus Attributen bezeichnen, von der alle Attribute einer Relation funktional abhängen. Als ein minimaler Superschlüssel bezeichnen wir Schlüsselkandidaten.
Attribute, die Teil eines Schlüsselkandidaten sind, werden als Schlüsselattribut bezeichnet, entsprechendes gilt für Attribute, die Teil keines Schlüsselkandidaten sind.

Definition (Normalformen)

Als Normalform bezeichnen wir den Zustand einer Relationendefinition auf der Basis der funktionalen Abhängigkeiten, welche dem Ziel dienen Anomalien zu vermeiden. Die Normalformen bauen aufeinander auf. Es ist deswegen trivial zu folgern, dass die (i+1)-te NF die (i)-te NF voraussetzt.

Definition (Normalisierung)

Wir wollen nun den Begriff der Normalisierung definieren als den Vorgang der Überführung eines Relationenschematas R in eine höhere Normalform durch Aufspaltung in Schemata R_1, R_2, ..., R_n.

Wichtig sind hierbei die Eigenschaften der Verlustlosigkeit und der Abhängigkeitserhaltung.

(a) **Verlustlosigkeit**
Wir bezeichnen eine Aufspaltung als verlustlos(-frei), wenn die in den Tupeln der Relation R enthaltenen Informationen genauso aus den entsprechenden Tupeln der neuen Relationen R_1, R_2, ..., R_n rekonstruierbar sind.

(b) **Abhängigkeitserhaltung**
Als abhängigkeitserhaltend bezeichnen wir eine Aufspaltung dann, wenn die für R geltenden funktionalen Abhängigkeiten, genauso für die neuen Schemata R_1, R_2, ..., R_n gelten.

Definition (Mehrwertige Abhängigkeit (MVD))

Ein Attribut B bezeichnen wir als mehrwertig abhängig von einem Attribut A ($A \twoheadrightarrow B$), wenn durch den Wert von A eine Menge von Werten in B bestimmt wird.

Genauer: Wir bezeichnen B als mehrwertig abhängig von A, wenn zwei Tupel t_1 und t_2 in R existieren, so dass $t_1[A] = t_2[A]$ und daraus folgt, dass zwei weitere Tupel t3 und t4 in R existieren[1] mit den folgenden Eigenschaften:

- $t_3[A] = t_4[A] = t_1[A] = t_2[A]$
- $t_3[B] = t_1[B] \wedge t_4[B] = t_2[B]$
- $t_3[Z] = t_1[Z] \wedge t_4[Z] = t_2[Z]$, mit $Z - (A \cup B)$

Die Mehrwertige Abhängigkeit nennt man auch **trivial**, wenn $B \subseteq A$ oder $B \cup A = R$.

Definition (Erste Normalform (1NF))

Eine Relation ist in erster Normalform (1NF), wenn sie **nur atomare** Attributwerte besitzt. Es gilt somit allgemein für eine Relation in 1NF, dass **jeder** Attributwert **atomar** ist, **keine** zwei **gleichen** Tupel existieren, die **Reihenfolge** der Tupel **irrelevant** ist. Zudem muss jeder Attributwert zum **Wertebereich** des jeweiligen Attributs **passen**, jedes Attribut einen **eindeutigen** Namen haben und die **Reihenfolge** der Attribute ebenfalls **irrelevant** sein.

[1] Die Tupel t1 mit t4 sind nicht zwingenderweise paarweise disjunkt.

Definition (Zweite Normalform (2NF))

Eine Relation ist in zweiter Normalform (2NF), wenn sie in erster Normalform ist und alle Nicht-Schlüsselattribute **voll funktional** von jedem Schlüsselkandidaten **abhängen**.

Aufspalten: Für jede Determinante einer FA, die ein Nicht-Schlüsselattribut bestimmt und Teil eines Schlüsselkandidaten ist, leg eine neue Relation an.

Definition (Dritte Normalform (3NF))

Eine Relation ist in dritter Normalform (3NF), wenn kein Nicht-Schlüsselattribut transitiv abhängig von einem Schlüsselkandidaten ist. C heißt transitiv abhängig von A, wenn es ein B gibt, so dass gilt: $A \rightarrow B \rightarrow C$ und $B \nrightarrow A$.

Aufspalten: Für jede Determinante einer FA, die ein Nicht-Schlüsselattribut bestimmt und nicht Teil eines Schlüsselkandidaten ist, leg eine neue Relation an.

Definition (Boyce-Codd Normalform (BCNF))

Eine Relation ist in Boyce-Codd Normalform (BCNF), wenn jede Determinante einer FA ein Superschlüssel ist (impliziert damit 3NF).

Algorithmus: Liste alle FA auf und erzeuge für jede FA, deren Determinante kein Superschlüssel ist, eine neue Relation

Wichtig: $3NF \subset BCNF$, das heißt jede Relation R, die in BCNF ist, ist immer auch in 3NF.
Eine Relation R, die in 3NF ist, ist immer auch in BCNF, wenn R keine überlappenden Schlüsselkandidaten hat.

Definition (Vierte Normalform (4NF))

Eine Relation R ist in vierter Normalform (4NF), wenn für jede nicht-triviale MVD $X \twoheadrightarrow A$ in R gilt: X ist Superschlüssel von R.

Eine Relation R ist auf jeden Fall nicht in 4NF, wenn folgende Prädikate gelten:

- Die Relation muss mindestens drei Attribute haben
- A bestimmt mehrere Werte von B und A bestimmt mehrere Werte von C.
- B und C sind unabhängig voneinander.

Definition (Denormalisierung)

Wir wollen als **Denormalisierung** den Prozess der **gezielten Zusammenführung** mehrerer Relationen, in **höherer** Normalform, in eine neue Relation **niederer** Normalform bezeichnen. Denormalisierung ist **sinnvoll**, wenn es aufgrund der Daten zu **wenig/keinen Anomalien** kommen kann (es gibt **keine SELECT-Anomalie**) oder aufgrund von Performancegründen. Diese Entscheidung sollte immer genau dokumentiert werden, da die DBA mit den eventuell auftretenden Redundanzen zurechtkommen müssen.

Merkblatt zur Veranstaltung

Konzeptionelle Modellierung

Merkblatt: „Relationenalgebra“

Bislang haben wir nur grundlegende Begrifflichkeiten (z.B. den der Relation) und somit die Struktur der Daten definiert. Ein Datenmodell erfordert jedoch auch Operationen auf diesen Daten. Dafür definieren wir den Begriff der Relationenalgebra.

Definition (Relationenalgebra)

Als **Relationenalgebra** bezeichnen wir eine **Menge an elementaren Operationen** auf **Relationen**. Wichtig ist die **Eigenschaft** der **Abgeschlossenheit**, denn nur mit ihr können wir garantieren, dass jede Operation auch von Nutzen ist.
Wir wollen zudem innerhalb der Relationenalgebra zwischen **unären** und **binäre** Operationen unterscheiden.

Wichtig: Relationen sind Mengen ⇒ Die Mengenoperationen **Verbund, Durchschnitt und Differenz** sind also anwendbar.
Einschränkung: Relationen R und S als Operanden müssen *vereinigungsverträglich*[1] sein, da das Ergebnis wieder eine Relation sein muss.
Als Konvention hat sich festgelegt, dass die Ergebnisrelation die Attributnamen des ersten Operanden verwendet[2].

Definition (Vereinigung)

$R \cup S = union(R, S) := \{r \mid r \in R \vee r \in S\}$

Definition (Durchschnitt)

$R \cap S = intersection(R, S) := \{r \mid r \in R \wedge r \in S\}$

Definition (Differenz)

$R - S = difference(R, S) := \{r \mid r \in R \wedge r \notin S\}$

Definition (Symmetrische Differenz)

$R \Delta S := (R - S) \cup (S - R)$

---

[1] als vereinigungsverträglich bezeichnen wir zwei Relationen $\mathrm{R}(\mathrm{A}_1, \mathrm{A}_2, \ldots, \mathrm{A}_\mathrm{n})$ und $\mathrm{S}(\mathrm{B}_1, \mathrm{B}_2, \ldots, \mathrm{B}_\mathrm{m})$ gdw. $\mathrm{n} = \mathrm{m} \wedge \forall\ 1 \leqslant \mathrm{i} \leqslant \mathrm{n}\colon \mathfrak{W}(\mathrm{A}_\mathrm{i}) = \mathfrak{W}(\mathrm{B}_\mathrm{i})$.
[2] Es ist auch möglich die Attributnamen explizit neu festzulegen.

Definition (Konkatenation)

Wir wollen ein auf Tupelebene definitorisches Hilfsmittel definieren. Seien $a = (a_1, a_2, \dots, a_n)$ und $b = (b_1, b_2, \dots, b_m)$ Tupeln beliebiger Relationen. Dann ist die Verkettung von a und b definiert als:

$$concat(a, b) := (a_1, a_2, \dots, a_n, b_1, b_2, \dots, b_m)$$

Die Konkatenation ist trivialerweise sicherlich keine Operation der Relationenalgebra, dient aber als Hilfsoperation für andere.

Definition (Kreuzprodukt)

$R \times S = crossproduct(R, S) := \{concat(r, s) \mid r \in R \ \wedge \ s \in S\}$

Definition (Selektion oder auch Restriktion)

$\sigma_p(R) = select_p(R) := \{r \mid r = (a_1, a_2, \dots, a_n) \in R \wedge P(a_1, a_2, \dots, a_n)$

$P(r)$ ist dabei das **Selektionsprädikat** über den Attributen von R (ist also ein logischer Ausdruck). Dieses hat für ein konkretes Tupel entweder den Wert „⊤“ oder „⊥“ und muss dazu nicht alle Attributwerte berücksichtigen. $P(r)$ darf nur solche Operationen anwenden, die mit den Wertebereichen verträglich sind.

Es gilt: $\sigma_p(R) \subseteq R$

Definition (Projektion)

Sei R (A_1, A_2, ..., A_n) Relation mit A = (A_1, A_2, ..., A_n) als der Menge der Attribute von R und mit B = (A_{i1}, A_{i2}, ..., A_{ik}) mit $k \leqslant n$ eine Folge von Attributen aus A gegeben. Für $r = (a_1, a_2, \dots, a_n) \in R$ sei $project[B](r) := (a_{i1}, a_{i2}, \dots, a_{ik})$.

Die Projektion von R auf B ist definiert als:

$$\pi_B(R) := \{project[B](r) \mid r \in R\}$$

Es gilt: $|\pi_B(R)| \leqslant |R|$

Zudem werden nach den **Regeln der Relationenalgebra** die Tupel von R, die in den Attributen B übereinstimmen, zusammen auf ein **einziges** Tupel in $\pi_B(R)$ abgebildet (**Eliminierung von Duplikaten**).

Problem (Multimengensemantik)

SQL als Standardanfragesprache liefert als **Ergebnis** von Anfragen eine **Multimenge** zurück, also eine Menge, die **Elemente mehrfach enthalten** kann. Dies geschah aufgrund der **Performance**. Man kann für die Relationenalgebra ebenfalls eine Funktion bag-to-set spezifizieren, und die Basisoperationen dann als Multimenge spezifizieren. Dies wird hier nicht näher verfolgt.

Seien R (A_1, A_2, ..., A_n) und S (B_1, B_2, ..., B_m) Relationen und P (A_1, A_2, ..., A_n, B_1, B_2, ..., B_m) ein Prädikat über den Attributen von R und S

Definition (Verbund (Theta-Join))

Als Verbund von R und S über P definieren wir $R \bowtie_P S = join(R, S) := \sigma_P(R \times S)$[3].

Definition (Natürlicher Verbund (Nat-Join / Equi-Join))

Beim weitaus häufigsten Fall des Verbunds gibt es Attribute mit gleichem Namen in R und S ($\exists 1 \leqslant i \leqslant n, 1 \leqslant j \leqslant m\colon A_i = B_j$) und gleichem Wertebereich. Wir definieren dann das Prädikat P (A_1, A_2, ..., A_n, B_1, B_2, ..., B_m) als A_i=B_j.
Der Nat-Join oder Equi-Join ist dann definiert durch:

$$natjoin(R, S) := \pi_{(A_1,\ldots,A_n,B_1,\ldots,B_{j-1},B_{j+1},\ldots,B_m)}(R \bowtie_{A_i=B_j} S)$$

Achtung: Es werden immer alle gleichnamigen Attribute verknüpft!

Definition (Alias (Umbenennung))

Um die Relation R (A_1, A_2, ..., A_n) in eine Relation S (A_1, A_2, ..., A_n) umzubenennen, definieren wir folgende unäre Operation:

$$\rho_{S(\mathrm{A}_1,\mathrm{A}_2,\ldots,\mathrm{A}_\mathrm{n})}(R)$$

Diese Operation nennt die Relation R in die Relation S um.

Definition (Selbstverbund (Auto-Join))

Bei dem Verbund einer Relation mit sich selbst muss man ihr Aliasnamen geben um ersten und zweiten Operanden zu unterscheiden. Der Auto-Join ist dann definiert als:

$$R \bowtie_P \rho_S(R)$$

Definition (Semi-Join)

Wir definieren die Projektion aller Attribute aus S auf R als Semi-Join:

$$R \rtimes_\mathrm{P} S := \pi_S(R \bowtie_P S)$$

[3] Dass dies eine rein formale Definition ist, sollte klar sein. Aus Performance-Gründen ist es schon nicht ratsam, zuerst ein sehr großes Kartesisches Produkt zu bilden und danach wieder mit dem Prädikat zu verkleinern.

Definition (Äußerer Verbund)

Bei einem normalen Verbund befinden sich die Tupel, die im Verbund keinen Partner finden, nicht mit in der Verbunds Relation. Problem: Beispielsweise Umsetzung von Variante 8B des Mapping-Prozesses in Zusammenhang mit der Frage nach allen Entitäten der Oberklasse. Dies ist so noch nicht möglich.
Wir wollen also eine Verbundart definieren, als Verbund, bei dessen **Ergebnisrelation auch die Tupel, die keinen Partner finden mit aufgenommen werden und diese notfalls mit NULL-Werten ergänzt** werden.
Wir wollen dies einen **äußeren** Verbund (auch OUTER JOIN) nennen.
Wir wollen zudem zwischen einem **LEFT**, **RIGHT** und **FULL** OUTER JOIN unterscheiden. Die Ergebnisrelation des **LEFT** OUTER JOINs beinhaltet alle Werte der **linken** Relation und füllt diese mit NULL-Werten auf. Die Ergebnisrelation des **RIGHT** OUTER JOINs beinhaltet alle Werte der **rechten** Relation und füllt diese mit NULL-Werten auf. Die Ergebnisrelation des **FULL** OUTER JOINs beinhaltet alle Werte der **beiden** Relationen und füllt diese mit NULL-Werten auf.

Definition (Linker Äußerer Verbund)

$$R ⟕ S := \big(natjoin(R,S)\big) \cup ((R - (S \rtimes_{R.A=S.B} R) \times (NULL, \dots))$$

Definition (Rechter Äußerer Verbund)

$$R ⟖ S := \big(natjoin(R,S)\big) \cup ((NULL, \dots) \times (S - (R \rtimes_{R.A=S.B} S)))$$

Definition (Voller Äußerer Verbund)

$$R ⟗ S := R ⟕ S \cup R ⟖ S$$

Definition (Division)

Seien $R_1(A_1, \dots, A_n, B_1, \dots, B_m)$ und $R_2(B_1, \dots, B_m)$ Relationen mit der Eigenschaft $R_2 \subseteq R_1$. So definieren wir als Division von R_1 durch R_2:
$R_1 \div R_2 := \{t \mid t = \pi_{[A_1,\dots,A_n]}(r_1) \land r_1 \in R_1 \land \forall r_2 \in R_2 \colon concat(t, r_2) \in R_1\}$
Das Ergebnis enthält die Attribute des Dividierenden R_1, die nicht im Divisor R_2 vorkommen. Es kommen nur solche Attributwertkombinationen ins Ergebnis, bei denen für **jede Zeile** des Divisors R_2 eine **entsprechende kombinierte** Zeile im Dividenden R_1 vorkommt.

Die Division ist die **inverse Operation zum kartesischen Produkt** es gilt $R_1 \times R_2 \div R_2 = R_1$. Die Division ist sicherlich **nicht kommutativ**. Die Division kann aus Projektion, Kreuzprodukt und Mengendifferenz zusammengesetzt werden:

$$R_1 \div R_2 = \pi_{[A_1,\dots,A_n]}(R_1) - \pi_{[A_1,\dots,A_n]}\big((\pi_{[A_1,\dots,A_n]}(R_1) \times R_2) - R_1\big)$$

Erweiterung (NF^2 – Non-First-Normalform)

Eine Erweiterung des relationalen Datenmodells (und somit der relationalen Algebra) ist das NF^2-Modell. Der Name steht für Non-First-Normalform (NFNF) und bedeutet, dass die Bedingung atomarer Attributwerte der 1. Normalform aufgebrochen wird. Dies bedeutet, dass Mengen von Attributen und Mengen von Mengen erlaubt sind $\Rightarrow$ Ein Attribut kann also wieder eine Relation sein. Der Wertebereich eines zusammengesetzten Attributs ist somit das Kreuzprodukt der Wertebereich der beteiligten Attribute.
Die Erweiterung der relationalen Algebra finden dort hingehend statt, als dass zwei neue Operationen hinzukommen, die Festung ν und die Entnestung μ. Die Nestung fasst eine Menge von Attributen in eine Unterrelation zusammen, die einen neuen Attributnamen erhält. Die Entnestung hebt Schachtelungen auf. Dies dient der Transformation in die/von der 1NF.
Achtung: NF^2 braucht **keinen** Fremdschlüssel!

Merkblatt zur Veranstaltung

Konzeptionelle Modellierung

Merkblatt: „SQL“

Allgemein wollen wir zwischen zwei Typen von Datenbanksprachen, den Tupel- resp. satzorientierten prozeduralen Datenbanksprachen und den mengenorientierten deskriptiven Datenbanksprachen, unterscheiden. Der Hauptunterschied ist die Spezifikation, denn in **prozeduralen** DB-Sprachen wird spezifiziert **wie**, in *deskriptiven was* das DBS zu suchen hat. Mit SQL ist uns ein mächtiges Werkzeug an die Hand gelegt, was über viele Funktionen verfügt. SQL zerfällt grundsätzlich in die Bereiche der DDL (Data Definition Language), DML (Data Manipulation Language), DQL (Data Query Language) und DCL (Data Control Language).

Data Definition Language (DDL)

Mit der Datendefinition werden Relationen sowie Views kreiert.

(a) Erstellen einer Relation

In SQL gibt es verschiedene elementare Datentypen:

INT[EGER]	SHORTINT	FLOAT [REAL]
DECIMAL (i, j)	CHAR(n)	BIT(n)
VARCHAR (n)	BIT VARYING (n)	DATE / TIME ...

Der Unterschied zwischen CHAR(n) und VARCHAR(n) ist die flexible Länge von VARCHAR. Ein Attribut des Datentyps CHAR(n) hat genau n Zeichen. Ein String vom Typ VARCHAR(n) hat variable, aber eine maximale Länge n.

Allgemeiner Aufbau des CREATE TABLE-Befehls:

```
CREATE TABLE Relationname (
      <Spaltendefinition>
      {, <Spaltendefinition>}
      [, <Zusatzangaben>]
);
```

Die Spaltendefinition besteht aus:

```
<Spaltendefinition> := attr_name <DATENTYP> {<ZUSATZ>}
```

Mit dem Zusatz als Auswahl zwischen `NOT NULL`, `UNIQUE`, `PRIMARY KEY` und `REFERENCES Relation(Attribut)`.
Im Zusatz kann auch eine Integritätsbedingung explizit bekanntgemacht werden, dies geschieht mit:

```
CHECK (Integritätsbedingung, logischer Ausdruck)
```

Die Integritätsbedingungen des Primär- und Fremdschlüssels können Teil der Attributdeklaration sein (siehe oben) oder aber auch extra Teil der Zusatzangaben sein:

```
<Zusatzangaben> := CONSTRAINT cstrN PRIMARY KEY
(Attributliste)
<Zusatzangaben> := CONSTRAINT cstrN FOREIGN KEY (Attributname)
REFERENCES Relation(Attribut) [ON DELETE ...]
```

(b) Löschen einer Relation

Um eine Relation komplett zu löschen, und damit auch ihre Definition gibt es den Befehl `DROP TABLE`:

```
DROP TABLE Relation
```

(c) Verändern der Definition einer Relation

Mit dem Befehl ALTER TABLE lassen sich Relationen in ihrer Definition verändern. Diese Befehle sind nicht im jedem DBMS unbedingt erlaubt.

```
ALTER TABLE relation <ALTER_OPT>
<ALTER_OPT> := ADD col_name <DATATYPE> | DROP COLUMN col_name
```

(d) **Erstellen einer Sicht**

Sichten dienen dazu, „virtuelle“ Relationen zu erstellen. Diese sind nicht physisch gespeichert, sondern sind logische Konstrukte, die bei der Rechtevergabe helfen sollen. So kann man zum Beispiel für verschiedene Benutzergruppen verschiedene Sichten definieren. Eine Sicht ist immer basierend auf einer (Multi-)Menge einer SQL-Anfrage.

Wichtig: Eine View kann **nicht** (sic!) dazu genutzt werden um Datensätze zu ändern oder löschen oder hinzuzufügen! Eine View ist immer auf dem aktuellsten Stand, da sie bei jeder Anfrage neu generiert wird.

Eine View wird erstellt mit dem `CREATE VIEW`-Befehl:

```
CREATE VIEW view_name AS <DQL-Anfrage>
```

(e) **Löschen einer Sicht**

Um eine View und damit auch die Definition jener zu löschen gibt es den Befehl `DROP VIEW`:

```
DROP VIEW view_name
```

Data Manipulation Language (DML)

Im Gegensatz zur Relationenalgebra hat SQL auch Funktionen zum bequemen Einfügen, Ändern und Löschen in/von dem Datenbestand.

(a) **Einfügen eines/mehrerer Tupel**

Das Einfügen von Werten geschieht mit dem `INSERT INTO`-Befehlkonstrukt:

```
INSERT
INTO relation_name [(<Attributliste>)]
VALUES (<Werteliste>)
```

Wird die Attributliste weggelassen, so muss die Werteliste vollständig nach der Definition und auch in der gleichen Reihenfolge angegeben sein. Andernfalls dürfen nur die in der Attributliste spezifizierten Attributwerte angegeben werden, die anderen Attributwerte erhalten NULL-Werte!

Ebenfalls möglich `INSERT INTO SELECT`: Statt einzelner Tupel werden hier ganze Mengen eingefügt:

```
INSERT INTO relation_name [(<Attributliste>)]
SELECT <Attributliste>  ⎫
FROM relation(en)        ⎬ Allgemeines SFW-Konstrukt der DQL
WHERE ...                ⎭
```

(b) Verändern von Tupeln

Um ein Tupel zu verändern gibt es die UPDATE-Anweisung. Sie hat folgenden Syntax:

```
UPDATE relation_name
SET <Attribut> = <Wert> {, <Attribut> = <Wert>}
[WHERE <Bedingung>]
```

Die zu verändernden Tupel werden über die optionale `WHERE`-Klausel bestimmt. Der Wert muss nicht fest sein, auch Verbindungen mit Attributwerten (z.B. `Gehalt = Gehalt * 1.1`) sind möglich. Pro Update ist nur eine Relation anzugeben, nicht mehr.

(c) Löschen von Tupeln

Um einzelne/mehrere Tupel zu löschen gibt es die `DELETE`-Anweisung. Zu löschende Tupel werden über die optionale `WHERE`-Klausel bestimmt.

```
DELETE FROM relation_name
[WHERE <Bedingung>]
```

(d) Löschen von allen Tupeln

Die `TRUNCATE`-Anweisung entspricht der `DELETE`-Anweisung ohne `WHERE`-Klausel. Sie wurde im Ansi-SQL-Standard:2008 zuerst eingeführt.

```
TRUNCATE relation_name
```

Data Query Language (DQL)

Die Data Query Language dient der Datenanfrage. Sie wird oft als Teil der DML gesehen. Mit den in ihr definierten Sprachelementen lassen sich **deskriptive** Anfragen an die Datenbank stellen. Dabei formuliert der Benutzer präzise die von den gewünschten Daten

aufzuweisenden Eigenschaften und das DBMS liefert dann die Menge der sich qualifizierenden Daten zurück.

(a) **Grundmuster einer SQL-Anfrage**

```
SELECT <was?>            (≙ <Attributliste>)
FROM   <woher?>          (≙ <Relation(en)>)
WHERE  <Eigenschaften?>  (≙ <Bedingung>)
```

Als Bezug zur Relationenalgebra sehen wir hier: `SELECT <L_A>` $\approx \pi_{<L_A>}$ und `WHERE <P>` $\equiv \sigma_{<P>}$
Falls keine Projektion gewünscht ist, so schreiben wir einen *.

(b) **Duplikateliminierung**

Wichtig: SQL liefert Multi-Mengen und muss keine Relationen liefern!
Dies geschah aus reinen Performancegründen (siehe Relationenalgebra). Man kann aber in SQL die Mengeneigenschaft erzwingen. Dafür fügen wir unserer Anfrage das Schlüsselwort `DISTINCT` hinzu.

```
SELECT DISTINCT ...
```

Wir sehen jetzt in Bezug auf die Relationenalgebra:

`SELECT DISTINCT <L_A>` $\equiv \pi_{<L_A>}$

(c) **Mengenvergleiche**

In der Restriktion können wir auch Mengen vergleichen. Dafür können wir explizite Mengen angeben, Abfragen schachteln und Quantoren verwenden.

(i) *Explizite Mengenangabe*

```
WHERE col_name IN (val₁, val₂, val₃, ...)
```

(ii) *Vergleichsmenge als Ergebnis einer Anfrage*

Unteranfragen dürfen durchaus auch in der `FROM`-Klausel stehen, dies ist sinnvoll bei Verbund-beziehungen zwischen Relationen und Anfragen.

```
WHERE col_name IN (SELECT col_name ....)
```

oder aber man überprüft ob überhaupt ein Element vorhanden ist ...

```
WHERE EXISTS (<unteranfrage>)
```

(iii) *Quantoren*
Quantoren sind ALL, ANY oder SOME, wobei ANY und SOME gleichbedeutend sind. Syntax:

```
<Attribut> θ ( ALL | ANY | SOME ) (<unteranfrage>)
```

ALL steht dabei für den Allquantor ($\forall$), SOME und ANY stehen dabei für den Existenzquantor ($\exists$). $\theta \in \{<, >, =, \leqslant, \geqslant, \neq\}$. Diese sind in der WHERE-Klausel einsetzbar.

(d) Aliasoperation
Für den Eigenverbund wurde diese unäre Operation eingeführt, auch SQL hat diese implementiert. Das optionale Schlüsselwort AS gibt der Relation einen Alias-Namen.

```
FROM Relation [AS] R
```

(e) Verbundoperationen
Bislang: nur unäre Operationen, jetzt auch binäre Operationen → Verbundoperationen.

(i) *Kreuzprodukt (Cross-Join)*
Das Kreuzprodukt ist auf verschiedene Weisen in SQL umzusetzen. Entweder man trennt die zu multiplizierenden Relationen mit einem Komma oder gibt dies explizit mit dem Schlüsselwort CROSS JOIN an.

```
FROM R, S   (≡ R×S)
FROM R CROSS JOIN S   (≡ R×S)
```

(ii) *Verbund (Join)*
Der Verbund zweier Relationen kann ebenfalls durch ein Schlüsselwort umgesetzt werden. Auch kann er durch das Anwenden der Definition (zuerst Kreuzprodukt, danach Selektion) umgesetzt werden. Letzteres ist

aber nicht zu empfehlen, denn diese Variante ist ineffizient! Denn das DBS muss erkennen, dass ein Verbund vorliegt, damit es dann die internen Optimierer auf die Anfrage ansetzt.
Das Schlüsselwort in SQL für den JOIN ist `JOIN ... ON ...`

```
FROM R JOIN S ON P (≡ R ⋈_P S)
```

(iii) *Eigenverbund (Auto-Join)*
Dieser kann wie ein normaler Join entsprechend der Definition durch die Relationenalgebra umgesetzt werden.

(iv) *Gleichverbund (Equi-Join)*
Das Schlüsselwort hier ist `JOIN ... USING ...`

```
FROM R JOIN S USING (Attr) (≡ R ⋈_{R.Attr=S.Attr} S)
```

(v) *Natürlicher Verbund (Nat-Join)*
Das Schlüsselwort hier ist `NATURAL JOIN ...`

```
FROM R NATURAL JOIN S (≡ natjoin(R,S))
```

(vi) *Äußerer Verbund (Outer-Join)*
Das Schlüsselwort hier ist `(LEFT | RIGHT | FULL) OUTER JOIN ...`

```
FROM R LEFT OUTER JOIN S  (≡ R ⟕ S)
FROM R RIGHT OUTER JOIN S (≡ R ⟖ S)
FROM R FULL OUTER JOIN S  (≡ R ⟗ S)
```

Achtung: Der Äußere Verbund ist auch realisierbar mit NOT EXISTS und UNION. Dies geschieht über die Auflösung der Definition des äußeren Verbunds.

(f) Sortierung
Die Sortierung ist nicht Teil des mathematischen Konzepts einer Menge, dennoch ist die Sortiermöglichkeit in SQL für den Anwendungsbezug relevant und von Nöten.

Das Schlüsselwort lautet ORDER BY. Syntax:

```
ORDER BY col_i [ASC | DESC] {, col_j [ASC | DESC]} (i ≠ j)
```

Der Standardwert für die Art der Sortierung ist **ASC**. Bei mehreren Werten wird zuerst nach dem Ersten, bei gleichem Wert im ersten Attribut nach dem zweiten, ... sortiert.

(g) Mengenoperationen

Genauso wie in der Relationenalgebra gibt es auch in SQL (zumindest in ANSI-SQL) die Mengenoperationen UNION, INTERSECTION und EXCEPT. Ein **ALL** hinter den Schlüsselbegriffen bedeutet, dass Duplikate **nicht** eliminiert werden. Voraussetzung für Mengenoperationen ist die Vereinigungsverträglichkeit der selbigen. Syntax:

```
<Relation> ( UNION | INTERSECTION | EXCEPT ) [ALL] <Relation>
```

(h) Ausdrücke der Projektionsliste

(i) Definition neuer Spalten

In der Projektionsliste können auch neue Spalten basierend auf den vorhandenen Spalten definiert werden.

```
SELECT (Gehalt + Werbeeinhamen) AS Einkünfte
SELECT (Name, Vorname, NULL AS Gehalt)
```

Dies ist für das Schaffen von vereinigungsverträglichen Relationen sehr wichtig.

(ii) Aggregationsfunktionen

Aggregationsfunktionen sind anwendbar auf ein Attribut der Ergebnisrelation. Insgesamt gibt es **fünf** verschiedene Aggregationsfunktionen: Minimum, Maximum, Anzahl, Durchschnitt und Summe.

Minimum und Maximum setzten eine totale Ordnung auf dem Wertebereich des Attributs voraus, der Syntax:

```
MIN(<Attribut>) resp. MAX(<Attribut>)
```

Die Anzahl gibt die Anzahl der Attributwerte zurück, `DISTINCT` hat dieselbe Bedeutung wie sonst auch. Ein * als Argument bedeutet, dass die Kardinalität der Relation zurückgegeben wird resp. die Anzahl der unterschiedlichen Attributwerte sollte ein `DISTINCT` auftauchen. Der Syntax:

```
COUNT([DISTINCT] (<Attribut> | *))
```

Durchschnitt und Summe erlauben nur numerische Attribute, `DISTINCT` hat dieselbe Bedeutung wie sonst auch. Der Syntax:

```
AVG([DISTINCT] <Attr>) resp. SUM([DISTINCT] <Attr>)
```

(i) Gruppierung und Restriktion auf Gruppen

Gewollt ist ein Anwenden von Aggregationsfunktionen auf Partitionen der Eingaberelation. Mit dem Schlüsselwort `GROUP BY` können wir Partitionen des jeweiligen Arguments erstellen. Die Ergebnisrelation erhält dann nur für jede Gruppe ein Tupel, nicht für jedes Tupel der Ausgangsrelation(en). Syntax:

```
GROUP BY col_name {, col_name}
```

Wichtig: Attribute, die **nicht** im `GROUP BY` genannt werden, dürfen nur **aggregiert** im `SELECT` verwendet werden!
SQL unterscheidet nun zwischen der Restriktion **vor** und **nach** der Partitionsbildung. Für die Restriktion **vor** ist die **`WHERE`-Klausel** verantwortlich, für die Restriktion nach der Partitionsbildung die `HAVING`-Klausel. Der Syntax ist mit der der `WHERE`-Klausel zu vergleichen.

(j) TOP(n) Anfragen

Ein gern gesehenes Beispiel für die `HAVING`-Klausel ist die Frage nach der TOP(n). Schema der Abfrage „Ermittle die TOP(n) der p(x) für jedes y und gib den Rang und (L_C) aus“:

```
SELECT y, COUNT(*) AS Rang, L_C
FROM Relation AS R, Relation AS S
WHERE R.y = S.y AND p(x) (Vorsicht: hier ist das <-Z. umg. zu setzten)
GROUP BY R.ID, R.L_C , R.y
HAVING COUNT(*) <= n
```

(k) **Abarbeitung einer SQL-Anfrage**

Für die Abarbeitungsreihenfolge einer SQL-Anfrage gilt **in der Theorie**:
`FROM` → `WHERE` → `GROUP BY` → `HAVING` → `SELECT` → `ORDER BY`

(l) **Allgemeiner Syntax für eine SQL-Anfrage**

```
SELECT [ALL | DISTINCT] column1 {,column2}
FROM table1 {, table2} | {((CROSS | NATURAL |
(LEFT|RIGHT|FULL) OUTER) JOIN table2) | JOIN table2 (ON P |
USING (Attr))}
[WHERE „conditions“]
[GROUP BY „column-list“]
[HAVING „conditions“]
[ORDER BY „column-list“ [ASC | DESC]]
```

Data Control Language (DCL)

Die DCL dient der Zugangskontrolle. Mit ihr Können Rechte vergeben aber auch genommen werden.

(a) **Rechtevergabe**

Für die Rechtevergabe gibt es den `GRANT`-Befehl. Der Syntax lautet wie folgt:

```
GRANT   <privilege_name>
ON      <object_name>
TO    { <user_name>  |  PUBLIC  |  <role_name> }
[WITH GRANT OPTION]
```

Mit der optionalen Option `WITH GRANT OPTION` darf der Rechteinhaber das Recht an andere Nutzer weitergeben. **Vorsicht**: Nimmt man dem Nutzer danach das Recht weg, so sind die durch ihn an das Recht gelangten nicht unbedingt vom Recht ausgeschlossen.

(b) **Rechteentzug**

Für den Rechteentzug gibt es den `REVOKE`-Befehl. Der Syntax lautet wie folgt:

```
REVOKE    <privilege_name>
ON        <object_name>
FROM    { <user_name>  |  PUBLIC  |  <role_name> }
```

Allgemeine Bedingungen und Trigger

Problem mit Check Constraints: Es sind Bedingungen, die für jedes Tupel einer Tabelle erfüllt sein müssen, welche nur beim Einfügen und Ändern überprüft werden. Bedingungen, die durch das Löschen eines Tupels invalidiert werden können (bspw. Totale Teilnahme), sind so nicht formulierbar!

(a) Assertions

Assertions sind Bedingungen, die für die ganze Datenbank erfüllt sein müssen! Problem: Kann leider kein reales System! (Zu aufwändig!) Syntax:

```
CREATE ASSERTION ass_name
CHECK (<Bedingung>)
```

(b) Trigger

Trigger sind ECAs, (Event-Condition-Action), sie werden beim Auftreten des Events getriggert, treten dann nur ein, wenn eine Bedingung gilt und führen somit eine Aktion aus. Syntax:

```
CREATE TRIGGER trigger_name                          (Name des Triggers)
(BEFORE | AFTER)                                     (Aktivierungszeit)
(INSERT | UPDATE [OF <attrs>]) ON <table>            (Auszulösendes Ereignis)
REFERENCING NEW ROW AS n
FOR EACH ROW [WHEN <Bedingung>]                      (Granularität)
<Aktion>                                             (Aktion)
```

Merkblatt zur Veranstaltung

Konzeptionelle Modellierung

Merkblatt: „XML“

Während bislang die Modellierung strukturierter Daten im Vordergrund stand, soll es bei XML nun um „semi-strukturierte“ Daten gehen. Daten haben ein Schema im Allgemeinen, das heißt viele Datensätze haben die gleiche Struktur.

Definition (XML)

Vom Ausgangspunkt der Standard Generalized Markup Language (SGML), welche als Metasprache zur Definition von Auszeichnungssprachen und der Annotation von Textdokumenten dient, entsteht die Metasprache XML. Sie wurde 1996 vom W3C entwickelt und hat es sich zum Ziel gesetzt Textdaten semantisch zu annotieren.

XML ist somit eine textbasierte Auszeichnungssprache. Sie besteht im Grunde aus strukturierenden „Tags“, welche im Kollektiv Markup heißen. Ein Textausschnitt wird immer durch Beginn- und Ende-Tags markiert:

```
<tag> Das ist ein Text </tag> (Element mit Bezeichnung ,,tag‘‘)
```

Die Daten, welche durch ein Beginnendes und ein Schließendes Tag eingeschlossen werden, bilden ein Element. Alleinstehende Tags ohne Inhalt können aber als Flag verkürzt werden:

```
<flag></flag>  ≡  <flag />
```

Beispiel (Hierarchische Datenstrukturen in XML)

($\approx$ Baumstruktur) Tags können per se geschachtelt werden. **Wichtig**: Zu **jedem** vorhandenen Anfangs-Tag muss es auch ein **passendes** Ende-Tag geben. Zudem sind Überlappungen **nicht** zulässig!

```
<a> ... <b> ... <c> ... </c> ... </b> ... </a>
```

Definition (Attribute)

Genau wie im Relationalen Datenmodell dienen Elemente dazu Attribute genauer zu beschreiben. Attribute zu einem Element stehen immer im zugehörigen Anfangs-Tag. Jedem Attribut wird ein Wert zugewiesen. Syntax:

```
<tag attribut1=‘‘attributwert‘‘> ... </tag>
```

Definition (Wohlgeformtheit)

Ein XML-Dokument soll **wohlgeformt** heißen, wenn es **alle syntaktischen Regeln** von XML einhält. Syntaktische Regeln sind, aber beschränken sich nicht auf:

- Jedes Dokument hat genau ein Wurzelelement (nicht mehr, nicht weniger!)
- Zu jedem Anfangs-Tag gibt es ein Ende-Tag.
  $\Rightarrow$ Jedes XML-Dokument bildet eine Baumstruktur.
- Elemente dürfen nicht überlappen.
- Verschiedene Attribute in einem Element haben verschiedene Namen.

Definition (Gültigkeit)

Ein XML-Dokument soll **gültig** heißen, wenn das XML-Dokument **einem Schema** zugeordnet ist und das **Wurzelelement** mit den **untergeordneten** Elementen den **Deklarationen** im Schema entspricht.
Voraussetzung: Wohlgeformtheit

Aufbau eines XML-Dokuments

Ein XML-Dokument ist im **Allgemeinen** aus einem *Prolog*, der *DTD* (eine Möglichkeit der Schemadefinition) und dem *Wurzelelement* aufgebaut.

Definition (Kommentar)

Einen Kommentar leiten wir mit `<!--` ein und beenden ihn mit `-->`. Beispiel:

```
<!-- Also das ist jetzt ein Kommentar. -->
```

Definition (Verarbeitungsanweisung)

„Processing Instructions“ dienen der Beeinflussung des XML-Prozessors (Beispiel: Prolog) Syntax:

```
<?Name Daten ?>
```

Definition (Zeichendaten)

Ohne Escape-Zeichen zu verwenden sind Zeichenketten in der Form

```
<![CDATA[ ... ]]>
```

anzugeben. Sie werden nicht als Zeichen für XML-Anweisungen verstanden.

Definition (Prolog)

Der Prolog einer XML-Datei beginnt mit `<?xml version=“1.0“` und endet mit `?>`. Das Prolog-Element hat verschiedene Attribute, so zum Beispiel noch **`encoding`** (Der zu verwendende Zeichensatz) und **`standalone`** (Werden externe Entities oder DTDs referenziert?)

Definition (Schema)

Ziel von XML ist es eigene, anwendungsspezifische Auszeichnungssprachen (XML-Applikationen) zu erstellen. Dies geschieht über das Festlegen von Regularien über Element- mit Attributkombinationen, die wir als zulässig resp. notwendig bezeichnen. Die Regeln wollen in kollektiv **Schema** nennen. Es gibt zwei Varianten ein Schema zu definieren: Über DTDs und XML-Schemata.

Document Type Definition (DTD)

(a) **DTD-Deklaration**

Wir wollen zwischen der internen und externen DTD-Deklaration unterscheiden. Mit der internen Deklaration beginnen wir die Deklaration direkt nach dem Prolog und vor dem Wurzelelement:

```
<?xml version=“1.0“ ?>
<!DOCTYPE name_des_wurzelelements [
      <!-- Hier folgen die DTD-Definitionen -->
]> ...
```

Bei der externen DTD-Deklaration erstellen wir eine Datei mit der Endung .dtd, in der die DTD-Definitionen stehen. Zum Importieren einer externen DTD gibt es folgende Syntax:

```
<?xml version=“1.0“ standalone=“no“?>
<!DOCTYPE name_des_wurzelelements SYSTEM “<pfad zur dtd>“>
```

(b) Elementtypdefinition

Der Syntax der Elementtypdefinition ist Folgender:

```
<!ELEMENT name_d_elements ( <Inhaltsmodell> ) >
<Inhaltsmodell> := EMPTY | ANY | #PCDATA | <Unterelemente>
```

Wichtig: Verwende ANY nur dann, wenn es nicht anders geht. ANY ist per se unschön i.Vgl.z. anderen Definitionen. Wenn man schon alles zulassen möchte, so ist die DTD überflüssig. **Vorsicht**: Alles ist hier auch nicht zugelassen, die Elemente müssen in der DTD definiert worden sein!
#PCDATA steht für **reinen** Text (Parsed Character Data), das Element darf dann nur Text als Inhalt besitzten.
EMPTY steht für ein **leeres** Element, dieses Element **darf keinen** Inhalt haben. **Wichtig**: Attribute sind hier **nicht** ausgeschlossen!
Das Unterelement kann aus einer Sequenz (zu trennen mit einem Komma), einer Auswahl an Elementen (zu kennzeichnen mit einem |) oder einer Kombination aus den Typen bestehen.
Die Anzahl an Einheiten kann reguliert werden über „?“ (kann maximal einmal vorkommen, muss aber nicht), „+“ (muss mindestens einmal, maximal n mal vorkommen) oder „*“ (Vokommen zwischen 0 und n)

(c) Attributtypdefinition

Nicht immer ist es sinnvoll alles bis in Unterelemente aufzuteilen. Manchmal ist es durchaus sinnvoll Texte in Attribute zu extrahieren. **Wichtig**: Attributnamen müssen für den betreffenden Elementtyp eindeutig sein. Syntax für die Definition:

```
<!ATTLIST elementname
      {  Attributname   Typ   Attributbedingungen  }
>
```

Attributbedingungen sind #REQUIRED (muss da sein), #IMPLIED (kann da sein) und [#FIXED] “Defalut-Wert“ (Standardwert, mit #FIXED eine Konstante).

Attributtypen können sein:

Datentyp in XML	Bedeutung
`CDATA`	Zeichenkette
`ID`	Identifikationstyp[1]
`IDREF`	Referenz auf eine ID
`IDREFS`	Menge an Referenzen (zu trennen durch Leerzeichen)[2]
`NMTOKEN`	Namens-Token[3]
`NMTOKENS`	Liste an Namens-Token
`name ( Option_1 | ... | Option_n )`	Aufzählungstyp[4]

Definition (Namensräume in XML)

Ziel ist es ein gemeinsames Vokabular für verschiedene XML-Dokumente zu schaffen. Man könnte eine externe DTD definieren, dies ist suboptimal, da ein Dokument nur einer DTD genügen kann. Gewünscht ist aber:

- Wiederverwenden von Vokabularen für verschiedene Zwecke
- Verschiedene Vokabulare innerhalb eines Dokuments
- Auflösen von Namenskonflikten

Dafür gibt es sogenannte Namensräume. Sie haben eine weltweit eindeutige URI und bestehen aus einem gültigen XML-Schema. Innerhalb eines Dokuments wird für einen Namensraum ein Kürzel definiert. Dieses Kürzel ist dann als Präfix zu verwenden um Elementen eines Namensraumes zu verwenden. **Wichtig**: Weder Name noch Präfix dürfen einen Doppelpunkt enthalten und müssen gültig sein!

Ein Namensraum ist gültig in dem Element, in dem die Namensraumdefinition vorgenommen wird, es sei denn in einem Unterelement wird ein anderer Namensraum definiert. Sollte kein Präfix definiert werden so gilt der Namensraum als Vorbelegung.

Definition über Attribut xmlns:

```
<tag xmlns:kuerzel="<namespace_uri>"> ... </tag>
```

---

[1] Achtung: Der Wert muss mit einem Buchstaben oder Unterstrich beginnen, danach dürfen Buchstaben, Ziffern, Unterstriche, Punkte und Bindestriche kommen. Leerzeichen sind verboten!

[2] Achtung: Es können auch mehrere gleiche IDs angegeben werden. Ist dies nicht gewünscht → XML-Schema

[3] Achtung: Der Wert muss mit einem Buchstaben oder Unterstrich beginnen, danach dürfen Buchstaben, Ziffern, Unterstriche, Punkte und Bindestriche kommen. Leerzeichen sind verboten!

[4] Der Wert kann nur eine der im Aufzählungstyp definierten Optionen annehmen

Definition (Entitäten in XML)

Eine Entität in XML ist eine separate Dateneinheit, auf welche innerhalb eines Dokuments verwiesen werden kann. Bei der Verarbeitung werden Entitäten vor der Validierung aufgelöst. Man unterscheidet zwischen „Parsed“ und „Unparsed“ Entitäten:

(a) *„Parsed Entity“ – XML-Fragment*

„Parsed“ Entitäten werden in der DTD definiert. Syntax:

```
<!ENTITY name ‘‘Inhalt der Entität‘‘>
```

definiert eine Entität `name`, so dass der XML-Prozessor jedes vorkommen der Entität mit »Inhalt der Entität« ersetzt. Aufzurufen über `&name;`

Eine externe Definition ist ebenfalls möglich in einer Datei mit der Endung .ent. Einzubinden über:

```
<!ENTITY name SYSTEM ‘‘datei.ent‘‘>
```

(b) *„Unparsed Entity“ – andere Daten*

Wert eines Attributs vom Typ ENTITY oder ENTITIES, das auf eine externe Datei verweist. Man definiert Entitäten mit nicht analysierten Inhalten in der DTD wie folgt:

```
<!ENTITY name SYSTEM ‘‘referenz‘‘ NDATA ntn>
<!NOTATION ntn SYSTEM ‘‘was_ist_die_datei z.b. image/jpeg‘‘>
```

Man definiert also zuerst eine externe Entität und setzt dahinter eine Notationsidentifikation. Über diese ist dann der Typ der Entität zu definieren. Die Notation definiert man ebenfalls der Entität ähnlich hier ist die Inhaltsinformation identifizierend für die einzubettenden Daten. Wichtig: Es gibt kein offizielles Format dafür!

(c) *Vordefinierte Entitäten*
Bestimmte Zeichen haben eine definierte Bedeutung in XML. Was tun, wenn ein Element ein solches Zeichen enthalten soll? Dafür gibt es in XML genau fünf vordefinierte Entitäten:

Entität	Inhalt der Entität
<	<
>	>
&	&
'	'
"	″

Verweise in XML – Xlink und Xpointer

(a) Xlink – Verweise zwischen Dokumenten
Um Xlink zu verwenden gibt es einen vorgegebenen Namensraum: http://www.w3.org/1999/xlink. Jedes Element kann grundsätzlich die Rolle eines Verweises einnehmen. Dazu bedarf es des Imports des Namensraum und dann die Belegung spezieller xlink-Attribute:

```
<link xlink:type=“simple“[5] xlink:href=“datei.xml“
xlink:title=“quickinfo zum link“
[xlink:show=“replace|embedded|new“[6] xlink:role=“zweck des
verweises“ xlink:actuate=“onRequest | onLoad“]>
```

(b) Xpointer („XML Pointer Language“) – Verweise auf spezifische Stellen in Dokumenten
Mit Xpointer sind auch Referenzen auf einzelne Elemente eines spezifischen Dokuments erlaubt. Dabei wird nur das Attribut xlink:href verändert!

```
xlink:href=“test.xml#ID9“   (Verweis auf Element mit ID „ID9“)
xlink:href=“test.xml#xpointer(/Projekt/Person/Name)“ (Der
Ausdruck muss dem Schema entsprechen)
```

Allgemein werden hier XPath-ähnliche Ausdrücke erwartet!

[5] `extended` ist rein theoretisch auch möglich, soll hier aber nicht näher behandelt werden (nur so viel: dadurch sind dann Linksets möglich)
[6] `replace`: Standard in HTML, `embedded`: wird direkt eingebunden, `new`: neues Fenster

XPath – Adressierung von Inhalten in XML-Dokumenten

Grundsätzlich gilt: XML-Dokumente werden als Bäume interpretiert, dabei gibt es unterschiedliche Knotentypen:

- **Wurzelknoten** – Repräsentiert kein Element → wird zusätzlich erzeugt![7]
- **Elementknoten** – Für jedes Element, kann ID besitzen
- **Attributknoten** – Für jedes Attribut
- **Textknoten** – Für zusammenhängenden Text
- **Verarbeitungsanweisungsknoten**
- **Kommentarknoten**
- **Namensraumknoten**

XPath-Ausdrücke werden ausgehend von dem sogenannten **Kontext-Knoten** ausgewertet. Lokalisierungspfade dienen dabei der abstrakten Beschreibung einer Menge von Informationsknoten innerhalb eines Dokuments.

(a) Lokationspfade und Schritte

XPath-Ausdruck	Bedeutung
`[/]{locationStep[[expr]]/}*`	Geht vom Wurzelelement aus (mit / dem Dateiwurzelelement, ohne dem von XPath) und schreitet dann den durch die locationStep angegebenen Pfad im Baum hierarchisch herab und wählt alle Elemente aus auf die der Pfad zutrifft.
`[expr]`	Kann eine Funktion der XPath Core Bibliothek sein oder eine eindeutige Zahl n für das n-te Kind sein.
`locationStep`	Besteht aus Achsenspezifikation, Node Test, null oder mehr Prädikate
`@`	⇔ attribute::
`//`	Hierarchieunabhängiges Suchen eines spezifischen Knotenwerts

[7] **Achtung:** Der Dokumentknoten ist Kind des Wurzelknotens!

(b) Achsenspezifikation

XPath-Ausdruck	
ancestor::	ancestor-or-self::
attribute::	child::
descendant::	descendant-or-self::
following::	following-sibling::
namespace::	parent::
preceding::	preceding-sibling::
self::	Achsenspezifikationen (triviale Bed.)

(c) Node-Tests

XPath-Ausdruck	
name	*
prefix:name	prefix:*
node()	text()
comment()	processing-instruction()
processing-instruction(*literal*)	„Node-Tests“ (triviale Bed.)

(d) Node Set Funktionen

XPath-Ausdruck	Bedeutung
last()	Gibt die Position des letzten Knotens zurück
position()	Gibt die Position des aktuellen Knotens zurück
id(object)	Gibt die ID des Objekts zurück
count(node-set)	Zählt die Knoten

Anomalien bei XML – XML als universelles Speicherformat?

Analog zu nicht-normalisierten Relationen treten Änderungs-, Einfüge- und Löschanomalien auf. Der Grund: Es kommen redundante Daten vor.
Wichtig: XML dient als Speicherformat, hat auch seine Vorteile muss aber ebenso mit Sorgfalt erstellt werden.

Merkblatt zur Veranstaltung

Konzeptionelle Modellierung

Merkblatt: „Multidimensionales Datenmodell“

Im Gegensatz zu der OLTP (Online Transaction Processing) geht es nun um ein analytisches Auffassen der Daten. Das grundlegende Ziel ist es Datenanalyse zu betreiben, also zum Beispiel Berichte über vorhandene Daten zu erstellen. Genau dafür wurde die Data-Warehouse-Umgebung geschaffen.

Vergleich Relationenmodell ↔ Multidimensionales Datenmodell

Im Vergleich zum Multidimensionalen Datenmodell fällt vor allem die Anwendungsneutralität, sowie die fehlende Anwendungssemantik auf. Es gibt zudem nur wenige sehr einfache Modellierungskonstrukte.
Das Multidimensionale Datenmodell ist komplexer und hat mehr Modellierungskonstrukte, ist aber speziell auf Anwendungen zur Datenanalyse zugeschnitten.

Definition (OLAP)

Das Online Analytical Processing (OLAP) beschreibt eine Data-Warehouse-Umgebung, welche insbesondere in betrieblichen Informationssystemen zum Einsatz kommt. Das Ziel ist, durch multidimensionale Betrachtung dieser Daten, zu einem entscheidungsträchtigen Analyseergebnis zu gelangen. Die im Vordergrund stehenden mehrdimensionalen Analysevorhaben verursachen ein hohes Datenaufkommen. Man unterscheidet zwischen ROLAP[1] („relational OLAP“), MOLAP[2] („multidimensional OLAP“), HOLAP[3] („hybrid OLAP“) sowie DOLAP („Desktop-OLAP“).

Definition (OLTP)

Das Online Transaction Processing (OLTP) beschreibt eine Produktionsumgebung, welche über eine transaktionale Umgebung vordefinierte Formulare verwendet.

---

[1] ROLAP: Zugriff erfolgt auf eine Relationale Datenbank. Dazu wird der Würfel als RDB dargestellt.
[2] MOLAP: Zugriff erfolgt auf Multidimensionale Datenbank (MDDB)
[3] HOLAP: Hybrides OLAP, ist ein Kompromiss aus ROLAP und MOLAP

Definition (Multidimensionales Datenmodell)

Beim Multidimensionalen Datenmodell unterscheiden wir zwischen **Fakten**, **Dimensionen** und **Klassifikationshierarchien**. Als Metapher für die letztendliche Speicherung überlegt man sich einen **Hyperkubus**, welcher von den Dimensionen aufgespannt wird. Bei n Dimensionen ist dies folglich ein n-dimensionaler Hyperkubus. **Ein Punkt** in diesem n-dimensionalen Raum repräsentiert dann genau **ein Faktum**. Die Klassifikationshierarchien repräsentieren sich in den Koordinaten in unterschiedlichem Vertiefungsgrad.
Im Hyperkubus unterscheidet man zwischen Qualifizierenden und Quantifizierenden Daten.
Wichtig ist die Eigenschaft der **Stabilen Daten**. Hier werden Daten (fast) nie geändert und nur neue Daten hinzugefügt!

In einem MDDM werden dann die analytischen Operationen auf dem zugehörigen Hyberkubus definiert.

Definition (Qualifizierende Daten)

Als Qualifizierende Daten bezeichnen wir die Dimensionen des Würfels, somit den Klassenhierarchien mit qualifizierenden Attributen. So dienen als Startpunkt der Datenanalyse.

Definition (Quantifizierende Daten)

Als Quantifizierende Daten bezeichnen wir die Zellen des Datenwürfels (Fakten). Diese sind die Elementarbausteine für das Ergebnis der Datenanalyse.

Definition (mE/R – Multidimensionales Entity/Relationship-Modell)

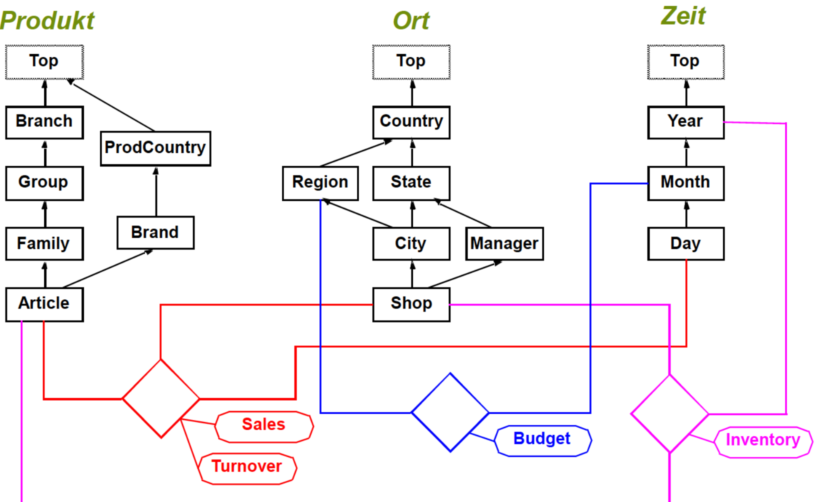
Als das Beispiel des Konzeptionellen Schemas gibt es das E/R-Modell für die OLTP spezifischen Modelle. Für die besonderen Elemente der multidimensionalen Modellierung erweitern wir das E/R-Modell für jene. Die Erweiterung mit den Elementen der multidimensionalen Modellierung des E/R-Modells nennen wir mE/R.

Es gilt die Spezialisierung (Modellierungskonstrukte sind reine Spezialfälle), Erweiterung (Leichte Erlenbarkeit und Minimale Zahl an neuen Konstrukten) und die Multidimensionale Semantik.

Definition (Logisches Schema)

Im logischen Schema werden die Klassifikationsstufen ähnlich der Relationen modelliert und die Fakten durch eine Beziehungsraute. Kenngrößen sind Attribute der Fakten. Alle Klassifikationsstufen einer Dimension sind durch Pfeile[4] miteinander verbunden. Alle Klassifikationshierarchien münden in einem Top.

Wichtig: Es muss das mathematische Prinzip der Orthogonalität gelten, d.h. verschiedene Dimensionen sind voneinander unabhängig!



Definition (Dimensionsschema)

Wir wollen das Dimensionsschema einer Dimension als eine partiell geordnete Menge D an Dimensionsattributen bezeichnen. Wir schreiben $(\{\mathrm{D}_1, \dots, \mathrm{D}_n, \mathrm{Top}_D\}; \rightarrow)$ mit:

- $\rightarrow$ als funktionale Abhängigkeit auf D
- Top_D als das maximale Element bezüglich $\rightarrow$ $(\forall\ \mathrm{D}_i \in \mathrm{D}: \mathrm{D}_i \rightarrow \mathrm{Top}_D)$

Zudem gibt es ein einzigartiges, minimales Element D_i, welches alle anderen funktional bestimmt. $(\exists \mathrm{D}_i \in \mathrm{D}: \forall \mathrm{D}_k \in \mathrm{D}, k \neq i: \mathrm{D}_i \rightarrow \mathrm{D}_k)$

Durch die partielle Ordnung kann es parallele Hierarchien geben (siehe Bild oben). Das Prinzip der Orthogonalität gilt gemeinhin $(\nexists D \neq D' \colon D.\mathrm{D}_i \rightarrow D'.\mathrm{D}_j)$.

---

[4] Ein Pfeil repräsentiert eine Funktionale Abhängigkeit ($\rightarrow$ Merkblatt 6 – FA). Zudem sind parallele Hierarchien erlaubt.

Definition (Instanz einer Dimension)

Durch die Funktionalen Abhängigkeiten entsteht eine balancierte Baumstruktur auf Instanzebene. Jeder Pfad im Schema einer Dimension definiert eine Klassenhierarchie, welche einen balancierten Baum repräsentiert.
Als die Instanz einer Dimension bezeichnen wir die Menge aller Klassenhierarchien.

Definition (Schema eines Datenwürfels / Hyperkubus)

Wir wollen einen Datenwürfel C als Kombination der Menge von Fakten M und der Granularität G wie folgt definieren:

$$\mathrm{C}\ [\mathrm{G}, \mathrm{M}] \text{ mit } \mathrm{M} = (\mathrm{M}_1, \dots, \mathrm{M}_\mathrm{m}) \text{ und } \mathrm{G} = (\mathrm{G}_1, \dots, \mathrm{G}_\mathrm{n})$$

Dabei gilt für alle G_i mit $1 \leqslant i \leqslant n$, G_i ist ein dimensionales Attribut. Im Allgemeinen gibt es für jede Dimension ein G_i.

Wichtig: Instanz eines Datenwürfels

Grundsätzlich gilt hier das „closed world“-Prinzip **nicht**! Denn: **Alle** Zellen aus dem Definitionsbereich des Datenwürfels werden als **existierend angenommen**, **unabhängig** von ihrer **physischen** Existenz. Im **Gegensatz** dazu: **Relationales Datenmodell** → Nichts, was nicht explizit als Datensatz vorkommt, wird angenommen.

Wichtig: „Datenwürfel“ als Metapher

Ebenso wichtig ist die Aussage, dass der Würfel **nur** als Metapher dient, denn es sind so gut wie **nie alle Zellen physisch** vorhanden! Denn auf **physischer** Ebene **nicht** vorhandene Werte sind auf **logischer** Ebene entweder **NULL** oder **numerisch 0**!

Definition (Fakten)

Als Fakten (Kenngrößen) wollen wir als ein Analogon zu Wertebereichen verstehen. Diesen können auch Eigenschaften zugesprochen werden, sind aber keine Datenstruktur an sich.

Definition (Aggregationstyp)

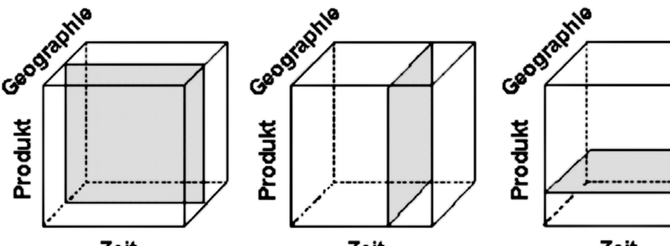
Neben dem Namen und Wertebereich ist der Aggregationstyp eine ebenso nicht-triviale und sehr wichtige Eigenschaft. Sie soll die erlaubten Aggregationsoperationen auf einer Kenngröße definieren. Wir unterscheiden zwischen:

1. FLOW – Beliebig aggregierbar
   Beispiel: Verkäufe, Umsatz, ...
2. STOCK – Nicht temporal summierbar
   Beispiel: Lager, Inventar, ...
3. VPU – Nicht summierbar
   Beispiel: Preis, Steuer, Faktoren, ...

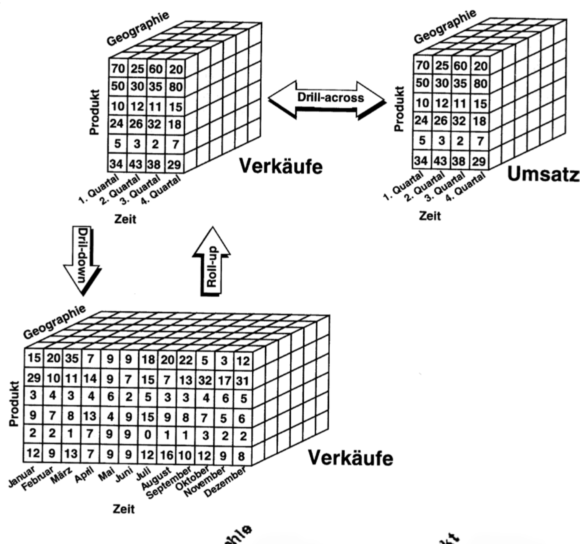
Definition (Multidimensionale Operatoren)

Wir wollen verschiedene Operatoren im Folgenden definieren:

- **Slicing und Dicing**
  Selektion eines Teilwürfels: **Dice**
  Selektion von Scheiben aus einem Würfel: **Slice**



- **Drill-Down und Roll-Up**
  Abstieg in der Klassifikationshierarchie zu feinerem Granulat: **Drill-Down**
  Aufstieg in der Klassifikationshierarchie zu gröberem Granulat: **Roll-Up**[5]
- **Drill-Across**
  Verknüpfung mehrerer Datenwürfel mit gemeinsamen Dimensionen (sog. „Cubic Join"): **Drill-Across**
- **Drill-Through**
  Wechsel von der feinsten Granularität zu den Original-daten: **Drill-Through**
- **Pivotierung**
  Wechsel der Darstellung in einer Pivot-tabelle/Drehen des Würfels: **Pivotierung**

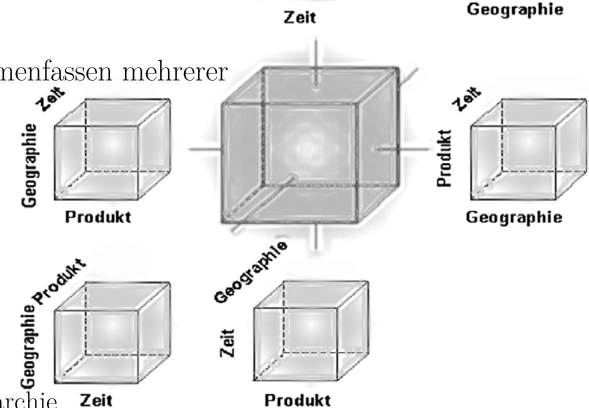


Definition (Aggregation)

Als **Aggregation** bezeichnen wir das Zusammenfassen mehrerer Zellen[6].

Als **Operationen** stehen zur Verfügung:

- SUM, AVG, MIN, MX, COUNT
- Ordnungsbasierte Aggregation: Cumulating, ranking



[5] (Fast) inverse Operation zu Drill-Down, ggf. zuerst Expansion des Datenraums auf den Zielknoten in der Hierarchie

[6] Unter anderem notwendig beim Roll-Up

Auftretendes Problem: Nicht alle Funktionen sind summierbar (Median, Standardabweichung). Unter Umständen dürfen manche Aggregationsfunktionen nicht angewendet werden (siehe Aggregationstyp!).

Multidimensionaler Schemaentwurf nach Kimball

1. Auswahl eines Geschäftsprozesses
   → *Subjektorientierung*
2. Auswahl der Erfassungsgranularität
   ***Achtung**: Je feiner, desto mehr Daten!*
3. Auswahl der Dimensionen
   ***Achtung**: Orthogonalität und Funktionale Abhängigkeiten*
4. Auswahl der Kennziffern
   ***Achtung**: Aggregationstyp*

Implementierungsvarianten (MOLAP ↔ ROLAP)

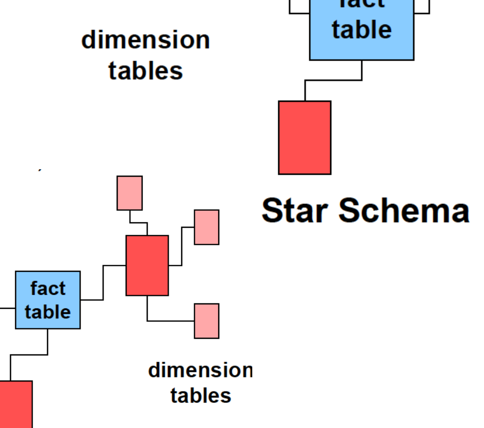
MOLAP (Multidimensional)	**ROLAP** (Relational)
▪ **Vorteil**: Straightforward ▪ Nachteil: Dünne Besetzung (weniger als 5% der Zeilen besetzt!) ▪ Nachteil: Skalierbarkeit (Arrays mit > 100 GB schwer zu speichern)	▪ **Vorteil**: Skalierbare, ausgereifte Technologie ▪ Nachteil: RDB sind für OLTP geeignet ▪ Nachteil: Schlechtere Performance ▪ Nachteil: Mangeln an Operationen

Deswegen nimmt man sich zum Ziel beide Vorteile zu vereinen ⇒ HOLAP (Hybrid)

ROLAP – Relationale Abbildung

Die Idee der Relationalen Abbildung ist eine Tabelle mit zusammengesetztem Primärschlüssel aus den Dimensionen („Fact Table“) und für jede vorhandene Datenzelle wird ein Tupel abgespeichert.
Neben der „Fact Table“ noch für Dimensionen je eine Tabelle.
Man unterscheidet die verschiedenen Schemata:

- **Star Schema**
  Pro Dimension eine Tabelle
- **Snowflake Schema**
  Pro Klassifikationsstufe eine Tabelle (oder: normalisiertes Star Schema)



Merkblatt zur Veranstaltung

Konzeptionelle Modellierung

Merkblatt: „Modellierung mit UML – Augenmerk: Verhaltensdiagramme“

Während bislang die konzeptionelle Modelliereung und damit die Umsetzung deklarativer Anforderungen auf ein relationales Datenbankschema im Vordergrund stand, geht es jetzt um die funktionalen und deklarativen Anforderungen. Dabei gehen wir von objektorientierten Programmiersprachen aus und suchen Methoden für die objektorientierte Analyse sowie den Entwurf solcher Systeme.

Historisches (UML)

Die Unified Modeling Language (UML) sollte von Beginn an der Vereinheitlichung und Erweiterung bereits bestehender Objektorientierter Modellierungstechniken dienen. Hauptsächlich entwickelt wurde sie von Grady Booch[1], James Rumbaugh[2] und Ivar Jacobsen[3]. Bereits im Januar 1997 wurde die Version 1.0 von der OMG als Standard gewählt. Version 2 folgte 2005 mit grundlegenden Neuerungen. Aktuell (zu diesem Zeitpunkt) ist die Version 2.5 (Juni 2015).

Definition (Unified Modeling Language UML)

Die Unified Modeling Language (UML) ist eine graphische Modellierungssprache zum **Spezifizieren**, **Konstruieren**, **Visualisieren** und **Dokumentieren** von ***Softwaressystemen***. Die Modelle von UML werden graphisch in Form von Diagrammen repräsentiert. Wir wollen UML in zwei grundlegend verschiedene Teile auftrennen, in die **Verhaltensdiagramme** und *Strukturdiagramme*.

Anwendungsfalldiagramm (Use-Case-Diagramm)

Das Ziel eines jeden Anwendungsfalldiagramms ist die funktionale Beschreibung eines Systems. Dabei beschreiben die Use Cases die **Benutzeranforderungen**.

Modellierungselemente	Bedeutung
Akteur / Actor	Akteure: • Akteure benutzen Funktionen • Akteure sind immer außerhalb jeglicher Systeme • Akteure werden **nicht** (sic!) implementiert (sie stellen den Benutzer dar)
Anwendungsfall / Use Case	Konkreter Anwendungsfall (Funktionen), stellt eine Aktivität oder Aktion dar.

[1] bekannt durch für die Booch-Methode, eine frühere Variante des **o**bjekt**o**rientierten **D**esigns (OOD)
[2] verantwortlich für die **O**bjekt-**M**odellierungs**t**echnik (OMT) (vor allem Objekt- und Zustandsdiagramme)
[3] verantwortlich für die Use-Cases, des OOSE (**O**bjekt**o**rientiertes **S**oftware-**E**ngenieering)

Modellierungselemente	Bedeutung
Systemkontext	Systemkontext: Alles innerhalb dieses Rechtecks gehört zu einem System.
genereller Akteur; spezialler Akteur A; spezialler Akteur B; genereller Anwendungsfall; spezieller Anwendungsfall A; spezieller Anwendungsfall B	Generalisierung: Im Prinzip ähnlich der Vererbung bei Klassen, so werden bei Akteuren alle Eigenschaften weiter vererbt und alle möglichen Anwendungsfälle. Bei Anwendungsfällen ist dies eine IS_A-Beziehung.
Anwendungsfall A <<include>> Anwendungsfall B	Beziehungen zwischen Anwendungsfällen: «include» (auch «benutzt»): Ein Anwendungsfall wird in mehreren anderen Anwendungsfällen wiederverwendet. Wichtig: Diese Beziehung ist ein **MUSS**! (**Funktionale Dekomposition**)
Anwendungsfall A <<extend>> Anwendungsfall B	Beziehungen zwischen Anwendungsfällen: «extends» (auch «erweitert»): Beschreibt eine Variation des normalen Verhaltens. Nutzen: Beschreiben von Ausnahmefällen.
Anwendungsfall A <<extend>> Anwendungsfall B; Bedingung Erweiterungsstelle	Beziehungen zwischen Anwendungsfällen: «extends» (auch «erweitert») (II): Bei notwendigen Bedingungen für Ausnahmen steht ein Extensionpoint zur Verfügung.
Anwendungsfall; extension points: Erweiterungsstelle	Alle «extend» Anwendungsfälle stehen an gegebener Stelle
Anwendungsfall {abstract}	Abstrakter Anwendungsfall. Ähnlich der Klassendefinition kann man diesen Anwendungsfall nicht ausführen. Er dient der Definition der Wiederverwendbarkeit eines gemeinsamen Verhaltens.

Aktivitätsdiagramm

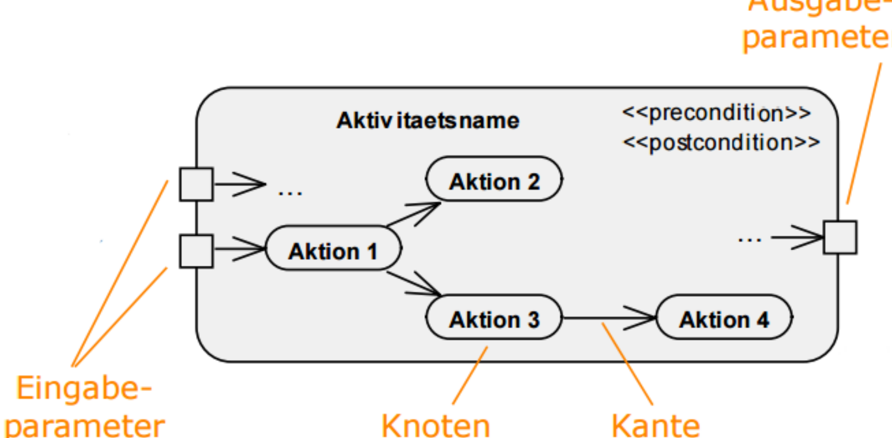
Das Aktivitätsdiagramm hat den Nutzen der Definition von prozeduralen Verarbeitungsaspekten, dem Kontroll- und Datenfluss zwischen verschiedenen Aktionen sowie der Modellierung objektorientierter und nicht-objektorientierter Systeme. Seit UML 2 ist auch die Modellierung von Geschäftsprozessen möglich, wird aber in der Praxis nur selten genutzt und wurde durch das mächtigere BPMN abgelöst.

Definition (Aktivität – Begriffsdefinition)

Als eine Aktivität bezeichnen wir eine von Mensch oder Maschine auszuführende Aufgabe. Dabei kann eine Aktivität unterschiedliche Granulate aufweisen[4]. Diese Aktivitäten können ebenfalls geschachtelt vorkommen. Einen **atomaren** Bestandteil einer Aktivität nennen wir fortan **Aktion**.

Definition (Inhalt einer Aktivität)

Bei UML-Aktivitätsdiagrammen beschreibt eine Aktivität einen gerichteten Graphen mit Aktivitätsknoten und -kanten. Die Knoten beschreiben dabei die einzelnen Aktionen, die Kanten hingegen die Kontroll- und Datenflüsse.



Definition (Verschiedene Knotenarten)

Im Aktivitätsdiagramm unterscheiden wir zwischen vielen verschiedenen Knoten.

Definition (Aktion/Aktionsknoten)

Als **Aktion** bezeichnen wir einen elementaren Baustein für ein beliebig benutzerdefiniertes Verhalten. Aktionen sind per se **atomar**, können aber **abgebrochen** werden.

Als Aktionsknoten bezeichnen wir die graphische **Repräsentation** einer **vordefinierten** UML-Aktion. Sie empfangen dabei Eingaben und können diese in Ausgaben für andere Knoten umwandeln.

Definition (Kontrollknoten)

Wir bezeichnen die Knoten, welche Aktivitätsabläufe steuern als Kontrollknoten. Es gibt eine Vielzahl unterschiedlicher Kontrollknoten, welche in der unteren Tabelle zusammen mit ihren graphischen Repräsentationen zu finden sind.

Definition (Objektknoten)

Wir wollen gewisse Knoten als Objektknoten bezeichnen. Diese speziellen Knoten stehen durch Objektflüsse miteinander in Beziehung und haben Datentoken als Inhalt. Sie können als Ein- und Ausgabeparameter dienen und bilden somit ein Bindeglied zwischen Verhaltens- und Strukturmodellierung. Ebenfalls können sie einen zentralen Puffer bilden.

[4] Dies reicht von Methodenaufrufen bis hin zu Arbeitsschritten im Ablauf eines Anwendungsfalls

Definition (Kanten)
Als Kanten bezeichnen wir die Verbindungen zwischen einzelnen Knoten, welche **mögliche Abläufe** einer Aktivität festlegen. Wir wollen dabei zwischen Kontrollflusskanten und Objektflusskanten unterscheiden.

Definition (Kontrollflusskante)
Eine Kontrollflusskante drückt eine reine Kontrollabhängigkeit zwischen den beteiligten Knoten aus (Reihenfolgeabhängigkeiten).

Definition (Objektflusskante)
Eine Objektflusskante transportiert zusätzlich noch Daten/Objekte und drückt somit eine **Datenabhängigkeit** aus.

Zusammenfassung (Elemente eines Aktivitätsdiagramms)

Elemente	Name und Bedeutung
Aktion A	**Aktionsknoten**: Repräsentieren eine Aktion
	Initialknoten: Kennzeichnen den Beginn eines Aktivitätsablaufs
	Aktivitätsendknoten: Kennzeichnen des Endes ***aller*** Abläufe einer Aktivität
	Ablaufendknoten: Kennzeichnen des Endes ***eines*** Ablaufs einer Aktivität
	Entscheidungs-/Vereinigungsknoten: Aufspaltung/-Zusammenführung von alternativen Abläufen

Elemente	Name und Bedeutung
	Parallelisierungs-/Synchronisationsknoten: Aufspaltung eines Ablaufs in mehrere nebenläufige Abläufe und umgekehrt
Aktion	**Aktivitätsaufruf**: Schachtelung von Aktivitäten
Object1 Aktion A	**Objektknoten,Pins**: Beinhalten Daten und Objekte
	Transition: Verbindung der Knoten einer Aktivität.

Interaktionsdiagramme – Sequenzdiagramme

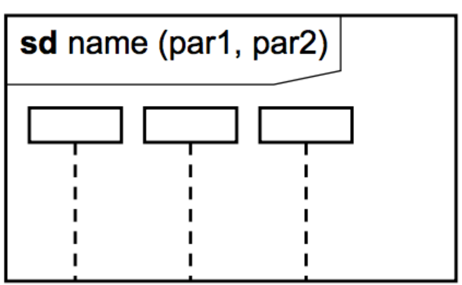
Definition (Interaktionsdiagramm)

Wir definieren ein Interaktionsdiagramm als das Diagramm, welches das **Interobjektverhalten** in Form von Nachrichten zwischen Objekten in bestimmten Rollen spezifizieren.

Definition (Interaktion)

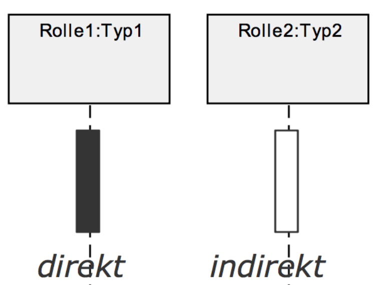
Wir bezeichnen die Art und Weise, wie **Nachrichten** und Daten zwischen verschiedenen Interaktionspartnern in einem bestimmten Kontext ausgetauscht werden, um eine bestimmte Aufgabe zu erfüllen als **Interaktion**. Der Nachrichtenaustausch wird im Allgemeinen auf Typebene modelliert. Interaktionen erfolgen durch Nachrichten in Form von Signalen und Operationsaufrufen oder wird durch Bedingungen und Zeitereignisse gesteuert.
In anderen Worten: Interaktion ≡ Wer tauscht mit wem welche Informationen in welcher Reihenfolge aus?

Das Sequenzdiagramm als solches hat viele verschiedene Einsatzmöglichkeiten, so soll es die Interaktionen eines Systems mit seiner Umwelt, die Realisierung eines Anwendungsfalls, das Zusammenspiel der inneren Struktur einer Klasse, Komponente oder Kollaboration, die Spezifikation von Schnittstellen zwischen Systemteilen oder die Operationen einer Klasse modellieren.
Wir unterscheiden im Sequenzdiagramm zwischen der horizontalen Interaktionspartnerachse und der vertikalen Zeitachse. Die Notation sieht wie folgt aus:



Definition (Interaktion, Trace und Ausführungsspezifikation im Sequenzdiagramm)

Als **Interaktion** bezeichnen wir eine Abfolge so genannter Ereignisspezifikationen. Diese partitionieren eine Lebenslinie in Zeitsegmente und deren Reihenfolge ist durch die Lebenslinie festgelegt. Sie definieren einen potenziellen Ereigniseintritt zur Laufzeit.
Als Folge von konkreten Ereigniseintritten bezeichen wir einen **Trace**. Er gibt den Ablauf einer Interaktion zur Laufzeit wieder
Die Periode, in der ein Interaktionspartner direkt oder indirekt ein bestimmtes Verhalten ausführt, nennt man Ausführungsspezifikation. Notation:



Definition (Nachrichtentypen)

Wir wollen zwischen synchronen, asynchronen, verlorenen sowie gefundenen Nachrichten unterscheiden.

Definition (Synchrone Nachricht)

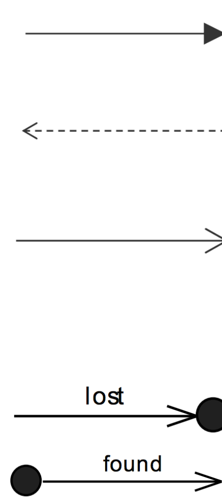
Bei einer **synchronen Nachricht** (ausgefüllte Pfeilspitze) wartet der Sender, bis die durch die Nachricht ausgelöste Interaktion beendet ist. Meist geschieht dies über eine **Antwortnachricht** (gestrichelter Pfeil).

Definition (Asynchrone Nachricht)

Bei einer **asynchronen Nachricht** (leere Pfeilspitze) wartet der Sender eben nicht auf die Antwort des Empfängers.

Definition (Verlorene und Gefundene Nachricht)

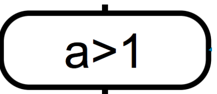
Von verlorenen Nachrichten spricht man dann, wenn man die Nachricht an einen unbekannten, außenstehenden oder nicht relevanten Interaktionspartner verschicken möchte (siehe Pfeil «lost»). Außenstehend meint hier außerhalb des zu modellierenden Systems. Von einer gefundenen Nachricht spricht man, wenn man eine Nachricht von einem solchen Interaktionspartner empfängt.

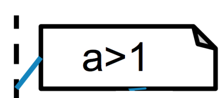


Definition (Zustandsinvariante)

Als **Zustandsinvariante** bezeichnen wir eine Zusicherung, dass eine bestimmte Bedingung zu einem bestimmten Zeitpunkt erfüllt ist. Dabei bezieht sich jene immer auf eine ganz bestimmte Lebenslinie und wird vor Eintritt des darauffolgenden Ereignisses ausgewertet. **Wichtig**: Falls die Invariante nicht erfüllt ist kommt es zu einem Fehler!

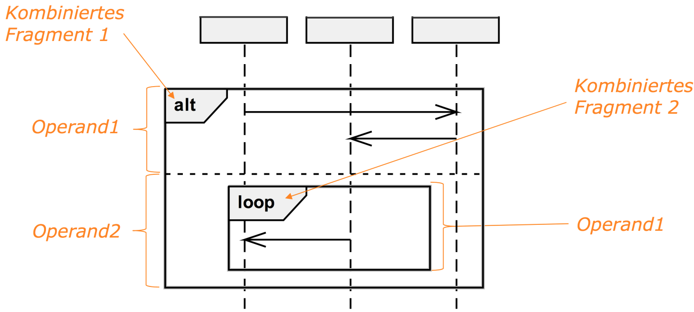
Notation:





Definition (Kombinierte Fragmente)

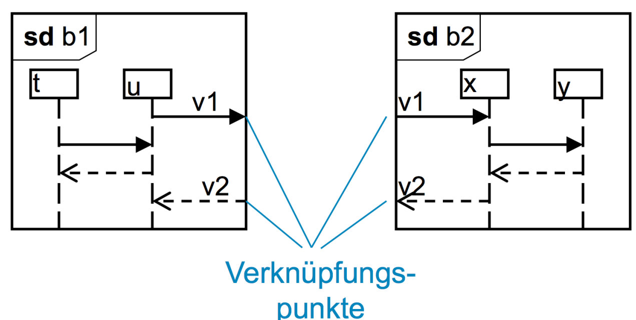
Ein kombiniertes Fragment wird wie ein Sequenzdiagramm als Rechteck mit einem Pentagon oben links dargestellt. Es dient der Modellierung komplexer Kontrollstrukturen und beinhaltet einen Interaktionsoperator und einen oder mehrere Interaktionsoperanden. Die Operanden werden durch gestrichelte Linien voneinander getrennt.



Operator	Zweck
`alt`	Alternative Interaktionen
`opt`	Optionale Interaktionen
`break`	Ausnahme Interaktionen
`loop`	Iterative Interaktionen
`seq`	Sequentielle Interaktionen mit schwacher Ordnung
`strict`	Sequentielle Interaktionen mit strenger Ordnung
`par`	Nebenläufige Interaktionen
`critical`	Atomare Interaktionen
`ignore`	Irrelevante Interaktionen
`consider`	Relevante Interaktionen
`assert`	Zugesicherte Interaktionen
`neg`	Ungültige Interaktionen

Definition (Interaktionsreferenzen)
Als Interaktionsreferenz wird die Referenzierung anderen Sequenzdiagramme bezeichnet, wodurch ganze Interaktionsabläufe als auch einzelne Lebenslinien zerlegt werden. Notation:

Auch möglich sind Start- und Zielmarken (ähnlich der Zustandsinvarianten bloß runder), sie sind das äquivalent zu der goto-Anweisung.
Definition (Verknüpfungspunkte)
Als Verknüpfungspunkt bezeichnen wir die Verbindung von Nachrichten zwischen Sequenzdiagrammen, Interaktionsreferenzen oder kombinierten Fragmenten. Dies können durchaus mal verlorene und gefundene Nachrichten gewesen sein, sie sind nun aber benannt und verknüpft!



Merkblatt zur Veranstaltung

Konzeptionelle Modellierung

Merkblatt: „Modellierung mit UML – Augenmerk: Strukturdiagramme und Metamodellierung“

Um die Struktur eines Systems in ihren verschiedenen Aspekten zu beschreiben, bietet UML eine Reihe von Diagrammarten an, welche in den folgenden Abschnitten definiert werden.

Definition (Klassendiagramm)
Klassendiagramme entstammen der konzeptionellen Datenmodellierung und entsprechen somit den Entitätstypen im E/R-Modell. Der einzige **Unterschied**: „Objekte“ stellen im Gegensatz zu den Entitäten nun **Software-Artefakte** dar. Wir verwenden es deswegen für die Modellierung statischer Strukturen eines Systems.

Basisnotation für Klassen

≪Stereotyp≫
C
{abstract}

+ öffentlichesAttribut : Datentyp = Default
# protectedAttr : DTMMultiplizität[0..*]
~ /abgeleitet : DT {readOnly}
+ Attr : DTMMulti[0..*] {unique, ordered}
– statischesAttribut : Datentyp

– method(in par1: DT, out par2: DT) : RT
method(inout par2: DT[0..*]) : void
~ method(par: DT[0..*] = Default {set})
+ statisch()

Namensfelder
Innerhalb des Namensfelds findet sich der Klassenname. Zudem können Stereotype (Schlüsselwörter) angegeben werden, welche die Klasse in eine Kategorie einteilen. Eigenschaftsangaben sind ebenfalls erlaubt. Hier wird gerade eine abstrakte Klasse definiert ({abstract}).

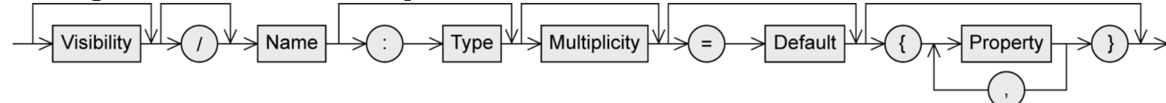
Attribute
Innerhalb der zweiten Kategorie finden sich die Attribute wieder. Allgemein gilt auch hier: Attribute beschreiben Eigenschaften. Während Instanzattribute nur für ein Objekt spezifisch sind, sind Klassenattribute (unterstrichen darzustellen) für eine ganze Klasse spezifisch.

Abgeleitete Attribute
Abgeleitete Attribute sind Attribute, deren Wert sich durch andere Attribute bestimmen lässt. Man kennzeichnet sie durch einen vorangestellten Schrägstrich /.

Mehrwertige Attribute
Mittels der Multiplizitätsangabe lassen sich auch mehrwertige Attribute modellieren. So beschreibt jene die Anzahl der Werte, die ein Attribut annehmen kann.

Eigenschaften bei Attributen
Bei Attributen kann ein Datentyp sowie ein Initialwert angegeben werden. Zusätzliche Eigenschaften ergeben sich über die Eigenschaftsliste zu notieren in geschweiften Klammern. Achtung: Die Eigenschaft **unique** unterscheidet sich von der Eigenschaft `UNIQUE` im Relationenmodell! In UML heißt unique, dass ein Attributwert in **einer** Instanz in **einem mehrwertigen** Attribut nur einmal vorkommen darf!

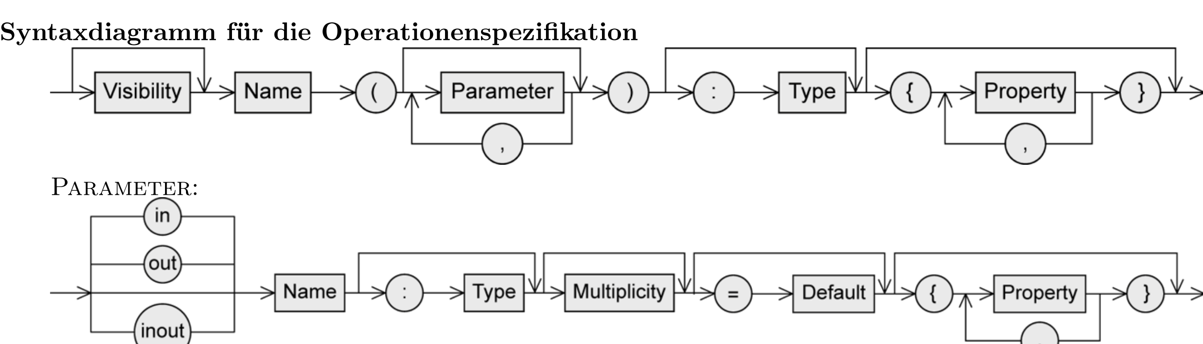
Syntaxdiagramm für die Attributspezifikation



Operationen

Neben Attributen haben die meisten Klassen noch Operationen (↯ Unterschied zum EE/R-Modell, denn dort gab es **keine** Operationen). Neben dem Namen und der Sichtbarkeit gibt es hier im Gegensatz zu der Attributspzifikation noch die Möglichkeit eine Parameterliste anzugeben. Jeder Parameter ist ähnlich wie ein Attribut aufgebaut (von der Spezifikation her zumindest), statt der Sichtbarkeit lässt sich hier aber eine Richtung angeben (`in`, `out`, `inout`).

Syntaxdiagramm für die Operationenspezifikation



Sichtbarkeiten

Notation	Name	Bedeutung
+	`public`	Der Zugriff ist durch Objekte beliebiger Klassen erlaubt.
−	`private`	Der Zugriff ist nur innerhalb des Objekts selbst erlaubt.
#	`protected`	Der Zugriff ist nur durch Objekte derselben Klasse und deren Subklassen erlaubt.
∼	`package`	Der Zugriff ist nur durch Objekte, deren Klassen sich im selben Paket befinden, erlaubt.

Aktive Klassen

Wir wollen grundsätzlich zwischen passiven und aktiven Klassen unterscheiden. Als aktiv bezeichnen wir eine Klasse, für welche ein eigenes Verhalten (bspw. mittels Zustands- oder Aktivitätsdiagrammen) definiert ist. Deren aktive Instanzen sind in ihrem eigenen Operationsfaden resp. können den Kontrollfluss starten oder modifizieren. Aktive Objekte sind somit sequenziell und machen etwas (bspw.: Variablenmodifizieren, Verhaltensänderung etc.). Sie sind zu kennzeichnen mit der Eigenschaft {active} oder einem vertikal doppelten Rahmen.

Definition (Objekt)

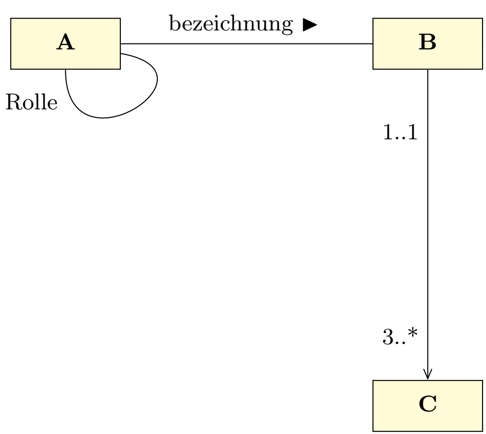
object:Type

Wir wollen die Instanzen einer Klasse als **Objekt** bezeichnen. Die Ausprägungen der von der Klasse definierten Struktur (Attribute) weisen ein definiertes Verhalten auf (Operationen). Attributwerte können unter dem Namen in einem Kasten vermerkt werden. Wir notieren ein Objekt als dass der Objektname dem Klassennamen vorangestellt wird. Entweder der Objektname oder der Klassenname kann aber entfallen. Bei letzterem lässt man auch den Doppelpunkt weg. Fällt der Objektname weg, so spricht man auch von einem **anonymen Objekt**. Beides wird aber unterstrichen.

Sollten zwei **Objekte** eine Beziehung eingehen, so spricht man von einem LINK. Dieser ist durch eine einfache Kante zwischen zwei Objekten darzustellen.

Definition (Assoziationen)

Als Assoziation bezeichnen wir eine Beziehung auf Klassenebene. Zu vergleichen ist dies mit der eigentlichen Definition des Beziehungstyps im E/R-Modell. Wir stellen eine solche dar als einfache Kante zwischen zweier Klassen und geben eine Leserichtung mit einer Bezeichnung an.



Definition (Navigierbare und Rekursive Assoziationen)

Für rekursive Assoziationen benötigt es Rollen, damit die einzelnen Beziehungspartner voneinander getrennt werden können. Eine Pfeilspitze zeigt grundsätzlich die Navigationsrichtung an. Salopp gesagt: Ein Objekt kennt seine Partnerobjekte und kann dadurch auf die sichtbaren Merkmale zugreifen. Ein nicht navigierbares Assotiationsende kann durch die explizite Angabe eines X spezifiziert werden. Dies bedeutet, dass dann das Partnerobjekt nicht auf die Attribute und Operationen des Objektes – inklusive der öffentlichen – zugreifen kann.

Multiplzitäten

Die Multiplizitäten sind bei UML-Klassendiagrammen im Prinzip genauso zu lesen wie deren Chen-Notationsäquivalente (heißt: "gegenüberliegende"Klasse!). An sich sind sie aber der (min,max)-Notation von der Schreibweise ähnlicher. Zu notieren ist eine Multiplizität wie folgt:

`zahl [.. max] {, zahl [.. max]}`

Wichtig: Bei mehrwertigen Assoziationsenden kann man Eigenschaften angeben (Ordnung, Eindeutigkeit). Anders als im E/R-Diagramm gibt es die Möglichkeit, dass ein Objekt mehrere Links zum selben Partnerobjekt hat (Expizite Angabe von {nonunique} erforderlich).

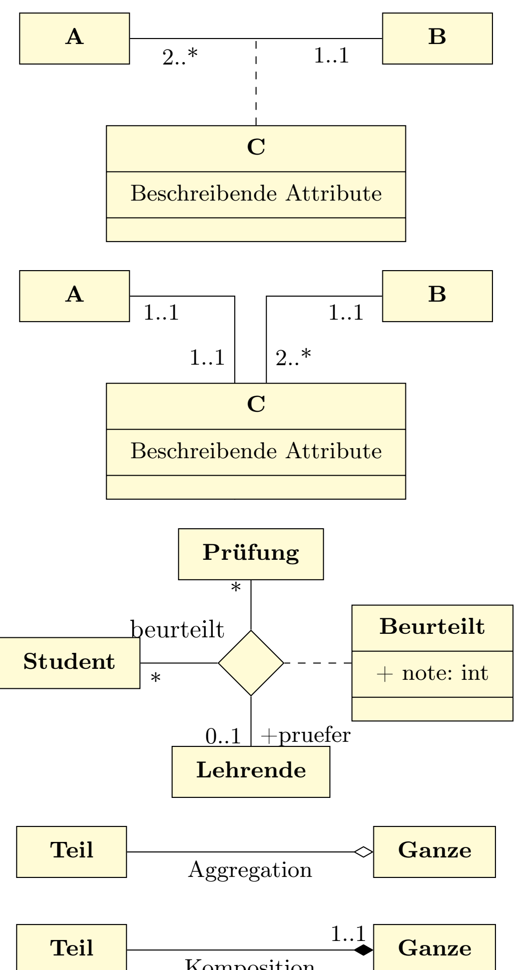
Definition (Assoziationsklasse)
Sollte eine Assoziation durch Attribute näher beschrieben werden, so erstellen wir eine Assoziationsklasse (zu sehen rechts oben). Diese kann dann, wie rechts mitte dargestellt, auch in eine normale Klasse umgewandelt werden.

Definition (N-äre Assoziationen)
Sind mehr als 2 Partnerobjekte an einer Beziehung beteiligt, so kann dies durch eine n-äre Assooziation modelliert werden. Notiert wird eine solche Assoziation durch eine ungefüllte Raute im Zentrum, welche mit allen Partnerobjekten verbunden ist. Während es hier keine Navigationsrichtungen gibt, kann es durchaus Multiplizitäten und Assoziationsklassen geben (siehe Beispiel rechts unten).

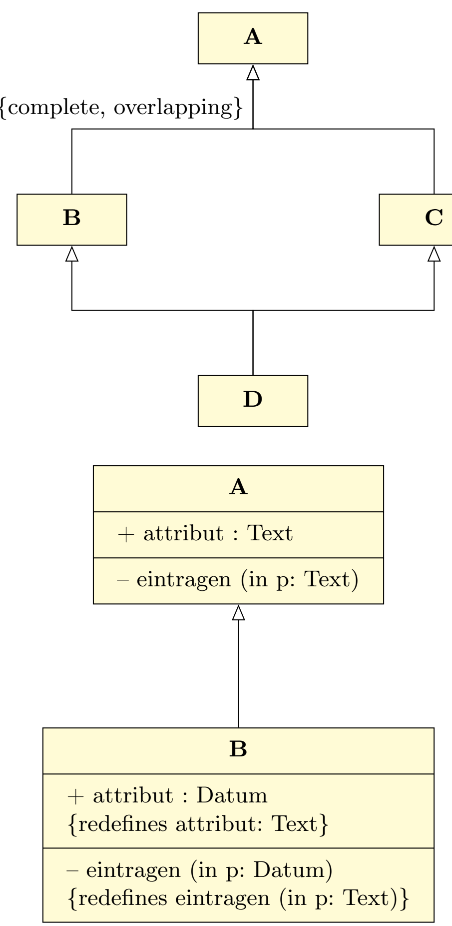
Definition (Aggregation)
Als eine Aggregation bezeichnen wir eine spezielle Assoziation, mit der eine Teil-Ganze-Beziehung ausgedrückt werden kann. Wir unterscheiden zwischen der schwachen und starken Aggregation. Beide werden durch eine Raute am Assoziationsende des Ganzen ausgedrückt. Beide Aggregationstypen sind reflexive und a(nti)symmetrische Assoziationen.

Definition (Schwache Aggregation)
Die schwache Aggregation drückt eine schwache Zugehörigkeit der Teile zum Ganzen aus, heißt ein Teil existiert auch unabhängig vom Ganzen und kann Teil mehrerer Ganzer sein. Die Raute ist **nicht** ausgefüllt!

Definition (Komposition)
Im Gegensatz zur schwachen Aggregation darf bei der starken Aggregation (auch: Komposition) ein Teil in maximal einem Kompositum enthalten sein. Mögliche Multiplizitäten sind somit: 0..1 und 1..1. Bei letzterem ist das Teil vom Ganzen existenzabhängig. Bei ersterem kann das Teil zwar ohne Ganzes existieren, aber sobald es zu einem Ganzen zugeordnet ist ist es auch existenzabhängig.



Definition (Generalisierung)



Die IS_A-Beziehung kann ebenfalls in UML modelliert werden. Wir unterscheiden generell zwischen direkten und indirekten Instanzen. Als **direkte Instanz** bezeichnen wir das Objekt in Bezugnahme auf den direkten Typen, also des „dynamischen Typen“. Eine Instanz einer Klasse ist aber **indirekte Instanz** ihrer Oberklassen.
Totale/Disjukte/Überlappende Vererbung kann über Eigenschaften modelliert werden. Zur Auswahl stehen hier: `complete` und `incomplete` sowie `disjoint` und `overlapping`.
Von **Mehrfachklassifikation** sprechen wir, wenn ein Objekt direkte Instanz mehrerer Klassen ist (bspw. bei „overlapping“).
Von **Mehrfachvererbung** sprechen wir, wenn eine Klasse Unterklasse mehrerer Oberklassen ist.
Die Klassifikation kann durch einen **Klassifikationstypen** näher beschrieben und modelliert werden. Dazu schreibt man einen Diskriminator-Typ an die Generalisierungsbeziehung (Wichtig: Notation ähnlich dem anonymen Objekt).

Redefinition

Geerbte Merkmale können per se in Unterklassen redefiniert werden. Dies ist entsprechend der Abbildung links zu notieren. Geerbte Merkmale umfassen, beschränken sich aber nicht auf: Navigierbare Enden von Assoziationen, Attribute oder Operationen.

Definition (Paket-Diagramm)

Für nichttriviale Problemstellungen können die einzelnen Diagramme, wie z.B. das Klassendiagramm, schnell sehr unübersichtlich werden.
Für solche Fälle bietet UML einen Strukturierungsmechanismus in Form von Paketen an.
Ein solches Paket erlaubt es, eine **beliebige Anzahl** von paketierbaren **(semantisch zusammengehörigen) UML-Modellelementen zu gruppieren**, dazu zählen u.a. **Klassen, Datentypen, Aufzähltypen und Komponenten.**
Beachte: Pakete können selbst wieder Pakete enthalten, da Pakete paketierbare Elemente sind.
Das Paket-Diagramm dient nicht zur Spezifikation der eigentlichen Systemstruktur, sondern der Unterstützung der Modellstrukturierung.
Hierbei wird v.a. auf die Gruppierung und die hierarchische Anordnung von Modellelementen Wert gelegt.

Definition (Komponenten-Diagramm)

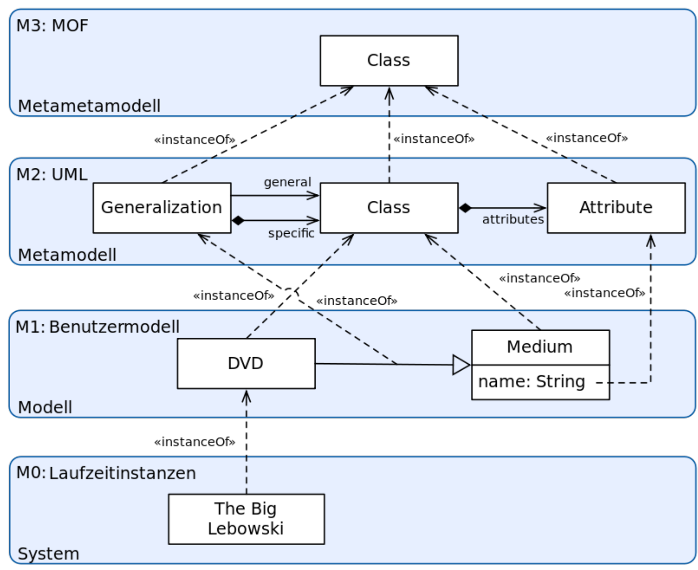
Das Komponentendiagramm unterstützt das Paradigma der komponentenorientierten Softwareentwicklung und eignet sich somit bestens zur Beschreibung von Architekturen von Softwaresystemen.
Eine **Komponente** ist ein modularer Teil **(eigenständig ausführbare Einheit)** eines Systems, der zur Abstraktion und Kapselung einer beliebig komplexen Struktur dient, die nach außen wohldefinierte Schnittstellen zur Verfügung stellt.
Eine Komponente kann hierin im gesamten Softwareentwicklungszyklus modelliert und zunehmend

verfeinert werden, bis eine **Software-Installation** vollständig abgeschlossen ist.
Somit kann eine Komponente z.B. die Modellierung der Funktionalität eines Systems mittels Anwendungsfällen (logische Modellierungsaspekte) bis hin zu derer Realisierung durch physische Artefakte umfassen.
Das Komponentendiagramm zeigt also die Definition von Komponenten und Abhängigkeiten zwischen diesen.

Definition (Metamodellierung)
Ein Metamodell ist ein Modell, das zur Beschreibung eines anderen Modells in Form einer Modellierungssprache dient. Das UML-Metamodell ist ein Modell der abstrakten Syntax der UML, formuliert in UML. Der Zweck des Metamodells ist die Beschreibung der Architektur von UML und der Ermöglichung einer besseren Interpretation der UML-Modelle sowie einer konsistenten Erweiterbarkeit der Sprache. Zudem kann damit die syntaktische Korrektheit von Modellen überprüft werden.

Ebenen der Metamodellierung

Ebene	Beschreibung	Beispiel
Meta-Metamodell	Sprache zur Beschreibung von Metamodellen (sog. Kernel)	Metaklasse, Metaoperation, Namespaces, Multiplicities, ...
Metamodell	Ausprägung eines Meta-Metamodells	Actor, UseCase, ExtensionPoint, LifeLine, CombinedFragment, ...
	Sprache zur Beschreibung von Modellen	Activity, Gate, Join-/ForkNode,...
Modell	Eine Ausprägung eines Metamodells Beschreibung eines Anwendungsbereichs	Bestellung, pruefe_Zahlung, ...
Anwendungsobjekte	Ausprägung eines Modells	Bestellung4712, pruefe_Zahlung(objekt9311)



Merkblatt zur Veranstaltung

Konzeptionelle Modellierung

Merkblatt: „Ontologien“

Während in UML- und E/R-Diagrammen es immer um konkrete (Software-)objekte geht, solle es dieses mal um die Möglichkeit eines Referenzmodells gehen. Problemstellung: Ein systemübergreifender Datenaustausch ist in Folge einer größeren Fusion erforderlich. Das Problem, welches sich nun stellt, ist das der **heterogenen Systeme**. So wurden die Systeme getrennt voneinander entworfen, basieren wahrscheinlich auf unterschiedlichen Modellen oder haben unterschiedliche Begriffe und Konzepte[1], was den Datenaustausch merklich erschwert. Wir wollen deshalb ein allgemeines Referenzmodell kreieren, welches anwendungsdomänentypische Konzepte und Begrifflichkeiten definiert. Dies führt dann auf den Begriff der ONTOLOGIE.

Im Allgemeinen muss man die Begrifflichkeiten der Ontologie strikt trennen. So wird als erstes kurz der Ontologiebegriff in der Philosophie betrachtet, welcher notwendig ist um den Begriff in der Informatik zu definieren.

Definiton (Ontologie in der Philosophie)

Die Ontologie (ὄν „seiend“ und λόγος „Lehre, Wort“, quasi die Lehre des Seiens) ist eine Disziplin der theoretischen Philosophie. Die Hauptfrage ist dabei unter anderem: Wie kann die Natur der Dinge beschrieben werden? oder Wie kann die Natur der Dinge in fundamenatale Kategorien aufgeteilt werden?
Wir erforschen in der Ontologie die Kategorien, die **allem** Sein zugrunde liegen. Dabei gibt es zwei unversöhnliche Lehrmeinungen, die des Realismus und die des Konstruktivismus.
In einem Satz zusammengefasst: Während beim Realismus die Sprache die menschliche Erfahrung reflektiert und damit die Grundstruktur des Existierenden über die Sprache erfahrbar ist, lehrt der Konstruktivismus, dass die Sprache nur Projektion des Denkens sowie abhängend von der menschlichen Erkenntnis ist und damit die Grundstruktur nicht unmittelbar erfahrbar ist.
Eng damit verknüpft ist die Epistemologie (Erkenntnistheorie), hier fragt man sich ob die Ontologie die Welt beschreibt, wie sie wirklich ist oder nur wie sie uns erscheint.

Man merkt also unmittelbar, dass der Konsens auf eine gemeinsame Sprache und damit verbundene Zeichen und Symbolik hinausläuft. Man eint inzwischen bei einem Mittelwert zwischen Realismus und Konstruktivismus.

Definition (Semiotik)

Als Semiotik bezeichnen wir die Lehre von den Zeichen und Symbolen (σημεῖον „Zeichen“ → Zeichentheorie). Wir beziehen dabei die Zeichen zu Gegenständen mittels des semiotischen Dreiecks. Für die Aufspaltung der Semiotik siehe Abbildung 1.

Definition (Terminologische Kontrolle)

Als Terminologische Kontrolle bezeichnet man alle Maßnahmen, welche entweder direkt oder indirekt der Definition und Abgrenzung der Begriffe und der Zuordnung von Benennungen und Begriffen dienen.

[1] Diesen Sachverhalt bezeichnen wir auch als **semantische Heterogenität**

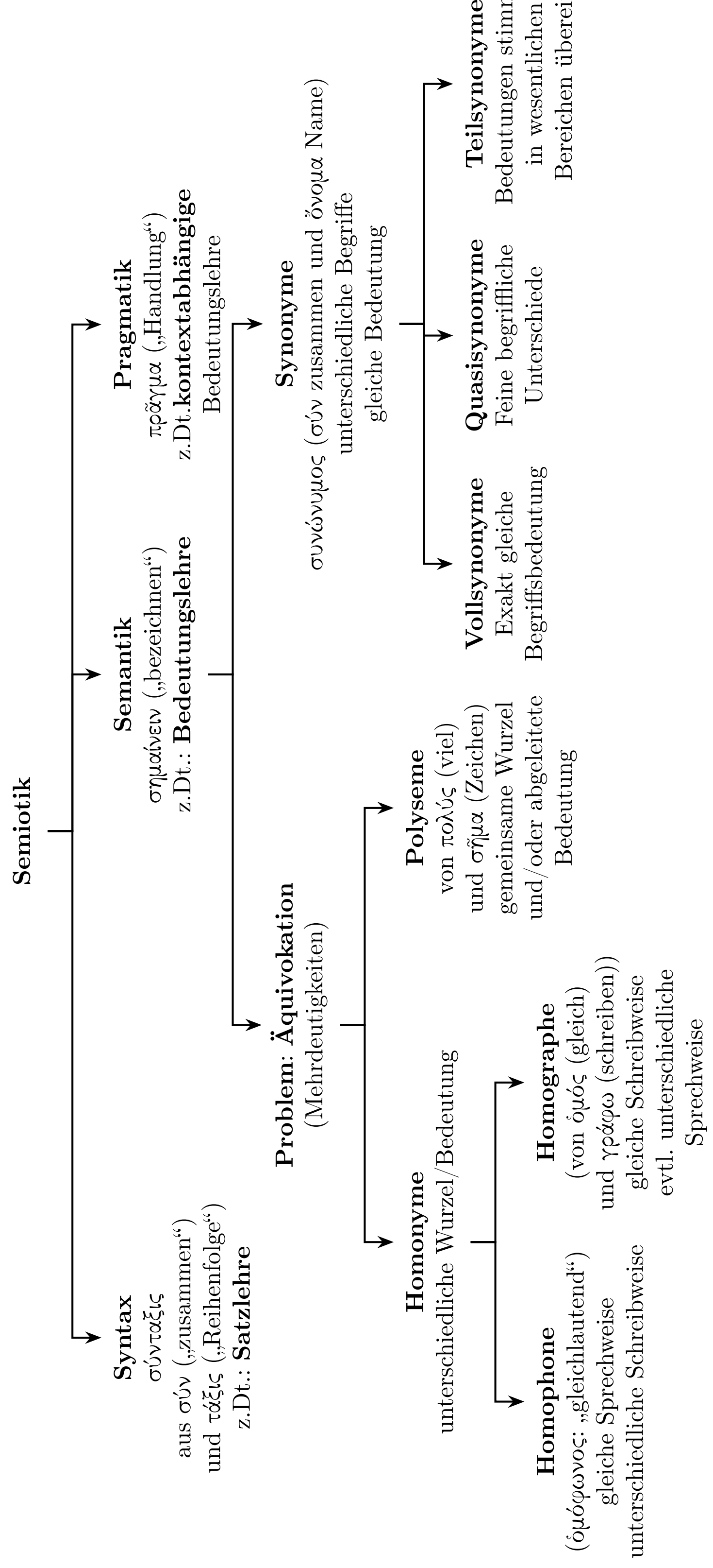


Abbildung 1: Semiotik mit ihren Kindestheorien

Definition (Ontologie)[2]
Eine Ontologie ist eine formelle explizite Spezifikation einer geteilten/gemeinsamen Konzeptualisierung (quasi ein Referenzmodell).

Man kann nun Ontologien in verschiedene Kategorien unterteilen: Top-Level-Ontologien, Kern-Ontologien (Top-Domänen-Ontologien) und Domänen-Ontologien. Ebenso kann man mit der Formalisierung statt der Allgemeingültigkeit vorgehen.

Definition (Formal Ontology)
Das Ziel der „Formal Ontology“ ist die formale Beschreibung von Dingen, Modi, Strukturen und Prozessen der Welt auf der Grundlage von Kategoriesystemen. (Interdisziplinär!)

Sinn und Zweck von Ontologien
Aufgrund der Vielzahl an Ontologien stellt sich die Frage welche verwendet wurde. Deswegen gibt es eine Ontologische Übereinkunft, in der man sich auf die Spezifikation einer bestimmten Ontologie festlegt (zum Beispiel: Namespaces bei XML).
Es gibt dabei verschiedene Methoden zur Spezifikation von Ontologien, als Beispiel sei hier das „Semantic Web“ genannt. Wichtig: Open-World-Assumption und keine globale Konsistenz, sowie die Identifizierbarkeit und Typisierung über URIs. Beispiel Ontologien: RDF (Resource Description Framework).

Definition (RDF-Grundbegriffe)
Anfragesprache: SPARQL
RDF und XML sind komplementär!

Definition (Ressource)
Jedes Ding, welches durch einen RDF-Ausdruck beschreibbar ist, nennt man **Ressource**. Jede Ressource wird durch einen URI[3] identifiziert.

Definition (Eigenschaft)
Eine Eigenschaft ist ein Attribut, eine Charakteristik, ein Aspekt oder eine Beziehung. Diese wird für die **Identifikation** der Ressource benötigt und hat eine **spezifische Bedeutung**, definiert die **zugelassenen Werte**, die **Typen** der Ressourcen und ihre **Beziehungen** zu anderen Eigenschaften.

Definition (Objekt)
Ein Objekt kann ein String, ein in XML definierter **primitiver** Datentyp oder eine sonstige Ressource sein.

[2]nach Tom Gruber 1993
[3]URI := Universal Resource Identifier