Kapitel 2

Klassendiagramm

Die wichtigste Komponente objekt-orientierter Softwareentwicklung ist das Klassen- oder Objekt-Modell **(statische Sicht, strukturelle Sicht, *static view, structural view)***, das graphisch als Klassendiagramm ***(class diagram)*** dargestellt wird. In diesem Diagramm werden die Klassen und ihre Beziehungen zueinander dargestellt. Beim Datenbankdesign werden wesentliche Teile dieser Technik mit großem Erfolg schon seit 1976 (Peter Pin-Shan Chen) als Entity-Relationship-Diagramm verwendet.

Das Verhalten eines Objekts wird beschrieben, indem Operationen benannt werden, ohne dass Details über das dynamische Verhalten gegeben werden.

2.1 Objekt *(object)*, Klasse *(class)*, Entität *(entity)*

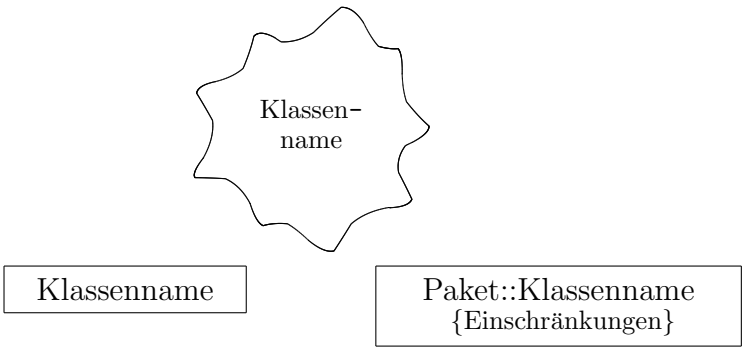
Chen definiert eine Entität als "a thing which can be distinctly identified". Eine Entität ist ein "Etwas" oder Objekt, das eindeutig identifiziert werden kann.

Entitäten können eine physikalische oder auch begriffliche, strategische Existenz haben, z.B. "Person", "Firma", "Rezeptur", "Vorgehensweise" sind Entitäten.

Den Begriff Entität verwenden wir sowohl für Typen oder **Klassen** von Objekten – dann eventuell genauer **Entitätstyp *(intension of entity)*** – alsauch für die **Objekte** selbst – **Instanzen, Exemplare, Ausprägungen *(extension of entity)*** eines Entitätstyps.

Entitäten werden im Diagramm durch Rechtecke dargestellt, die den Namen der Entität enthalten. Normalerweise werden in den Diagrammen immer Klassen verwendet. Objekte werden **unterstrichen**.

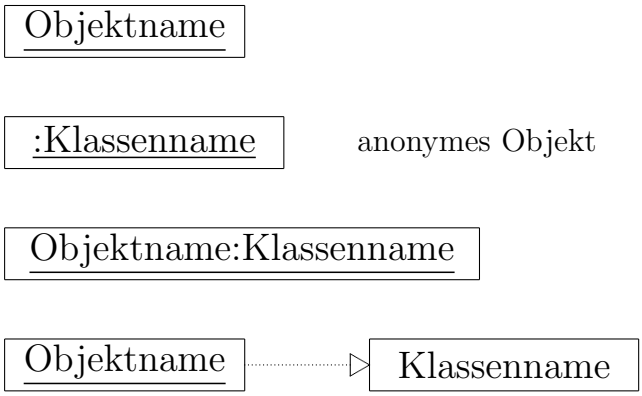
2.1.1 Klasse



Das ist die einfachste Notation für eine Klasse. Dem Namen der Klasse kann man einen Paketnamen und Einschränkungen mitgeben. Wie man Attribute und Operationen einer Klasse notiert, folgt in einem späteren Abschnitt.

2.1.2 Objekt

Objekte werden in Klassendiagrammen relativ selten verwendet. Trotzdem gibt es ziemlich viele Notationsmöglichkeiten.



Instanziierung eines Objekts

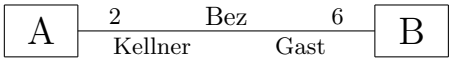
2.2 Multiplizitäten

Bevor wir auf die Notation der verschiedenen Beziehungen zwischen Klassen eingehen, müssen wir zeigen, wie die **Multiplizität** ***(multiplicity)*** oder **Ordinalität** ***(ordinality)*** zu notieren ist. Mit der Multiplizität wird die Anzahl der an einer Beziehung beteiligten Objekte angegeben.

(Nach UML darf dafür nicht mehr der Begriff **Kardinalität** ***(cardinality)*** verwendet werden, der für die Anzahl der Objekte einer Klasse reserviert ist.)

Multiplizität	Bedeutung
1	genau ein
*****	viele, kein oder mehr, optional
1..*	ein oder mehr
0..1	kein oder ein, optional
m..n	m bis n
m..*	m bis unendlich
m	genau m
m,l,k..n	m oder l oder k bis n

Auf welcher Seite einer Beziehung müssen Multiplizitäten stehen? Da das in UML leider nicht sehr vernünftig definiert wurde und daher immer wieder zu Missverständnissen führt, soll das hier an einem Beispiel verdeutlicht werden. Folgende Abbildung zeigt eine Beziehung `Bez` zwischen zwei Klassen `A` und `B`. Die Objekte der Klassen treten dabei auch noch in **Rollen** ***(role)*** auf:



Das bedeutet:

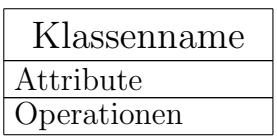
> *Ein* `A`*-Objekt steht (in der Rolle* `Kellner`*) zu genau 6* `B`*-Objekten in der Beziehung* `Bez` *(, wobei die* `B`*-Objekte dabei in der Rolle* `Gast` *auftreten).*
> *Ein* `B`*-Objekt steht (in der Rolle* `Gast`*) zu genau 2* `A`*-Objekten in der Beziehung* `Bez` *(, wobei die* `A`*-Objekte dabei in der Rolle* `Kellner` *auftreten).*

Bemerkung: Der Rollenname bedeutet oft nur, dass in unserem Beispiel die Klasse `A` eine Referenz (oder Pointer) mit Name `Gast` hat, und `B` eine Referenz mit Name `Kellner` hat.

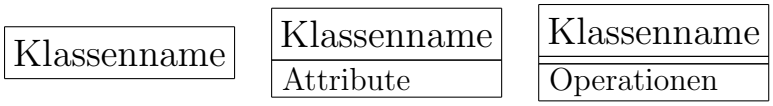
Jetzt können wir auf die verschiedenen Beziehungen einer Klasse zu anderen Klassen eingehen. Die engste Beziehung haben dabei die Eigenschaften einer Klasse.

2.3 Eigenschaften *(properties)*

Zu den Eigenschaften einer Klasse gehören die **Attribute** ***(attribute)*** und die **Operationen** ***(operation)*** oder Methoden.



Die Angabe von Attributen oder Operationen ist optional. Sie erscheinen meistens nur in *einem* Diagramm. (Eine Klasse kann in vielen Diagrammen vorkommen.)



Bemerkung: Das die Klasse repräsentierende Rechteck wird in ***compartments*** eingeteilt. Eine Klasse kann beliebig viele Compartments haben, wenn das Design es erfordert.

2.3.1 Attribute

Die Darstellung eines Attributs hat folgende Form:

$$\begin{aligned}\text{Sichtbarkeit}_{\text{opt}}/_{\text{opt}}\text{Name}_{\text{opt}} : \text{Typ}_{\text{opt}}[\text{Multiplizität}]_{\text{opt}} \\ = \text{Anfangswert}_{\text{opt}}\{\text{Eigenschaften}\}_{\text{opt}}\end{aligned}$$

Alle Angaben sind optional (opt). Wenn das Attribut aber überhaupt erscheinen soll dann muss wenigsten entweder der Name oder der :Typ angegeben werden. (Ein Attribut kann ganz weggelassen werden.) Die Sichtbarkeit kann die Werte

"`+`" ***(public)***,
"`#`" ***(protected)***,
"`-`" ***(private)*** und
"`~`" ***(package)***

annehmen.

"`/`" bezeichnet ein abgeleitetes Attribut, ein Attribut, das berechnet werden kann.

Die "Eigenschaften" sind meistens Einschränkungen, wie z.B.:

- Ordnung:**ordered** oder **unodered**
- Eindeutigkeit: **unique** oder **not unique**
- Es können hier auch die Collection-Typen verwendet werden: **bag, orderedSet, set, sequence**
- Schreib-Lese-Eigenschaften, z.B.: **readOnly**
- Einschränkungen beliebiger Art

Im allgemeinen sind Attribute Objekte von Klassen. (Wenn sie primitive Datentypen sind, dann kann man sie als Objekte einer Klasse des primitiven Datentyps auffassen.)

Statische Attribute (Klassenattribute) werden unterstrichen.

2.3.2 Operationen

Die Darstellung einer Operation hat folgende Form:

$$\text{Sichtbarkeit}_{\text{opt}}\text{Name (Parameterliste)} : \text{Rückgabetyp}_{\text{opt}} \quad \{\text{Eigenschaften}\}_{\text{opt}}$$

Mit Ausnahme des Namens und der Parameterliste der Operation sind alle Angaben optional (opt). Die Sichtbarkeit kann ebenfalls die Werte

"`+`" ***(public)***,
"`#`" ***(protected)***,
"`-`" ***(private)*** und
"`~`" ***(package)***

annehmen.

Statische Operationen (Klassenmethoden) werden unterstrichen.

Beispiel:

Konto
- anzKonten :int
- kontoStand :double
- transaktion :Transaktion [*]
+ getAnzKonten () :int
+ getKontoStand () :double
+ letzteTransaktion () :Transaktion
+ transaktion (datum:Date) :Transaktion[*]

Als "Eigenschaften" bieten sich bei Operationen irgendwelche Vor-, Nachbedingungen und Invarianten an ***(preconditions, postconditions, invariants)***. Exceptions und sogenannte Bodyconditions (Bedingungen, die durch Subklassen überschreibbar sind) gehören auch dazu. Oft ist es vernünftiger diese Eigenschaften in einer Notiz unterzubringen.

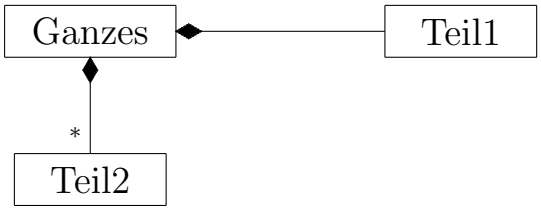
2.4 Teil-Ganzes-Beziehung

Die **Teil-Ganzes-** oder **Hat-ein-**Beziehung ***(whole-part, has-a)*** (auch Komponente / Komponentengruppe) hat große Ähnlichkeit mit Attributen. D.h. viele Attribute können auch als eine Hat-ein-Beziehung modelliert werden. In den objektorientierten Programmiersprachen gibt es da oft keine Implementierungsunterschiede.

Bei der Hat-ein-Beziehung unterscheidet man noch **Komposition** ***(composition)*** und **Aggregation** ***(aggregation)***.

2.4.1 Komposition

Die Komposition wird mit einer gefüllten Raute auf der Seite des Ganzen notiert (mit impliziter Multiplizität 1). Auf der Seite des Teils kann eine Multiplizität angegeben werden.



Bei der Komposition besteht zwischen Teil und Ganzem eine sogenannte **Existenzabhängigkeit**. Teil und Ganzes sind nicht ohne einander "lebensfähig". Ein Teil kann nur zu *einem* Ganzen gehören.

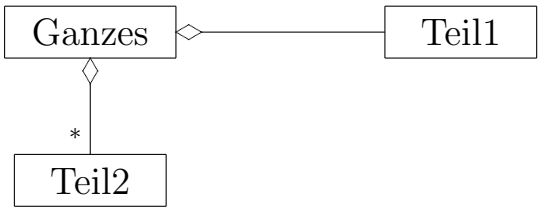
Komposition und Attribut sind kaum gegeneinander abzugrenzen. Im Hinblick auf Datenbanken haben die Teile der Komposition wie Attribute meistens keine eigene Persistenz. D.h. wenn das Ganze aus der Datenbank gelöscht wird, werden auch alle seine Teile gelöscht. Wenn ein Ganzes neu angelegt wird, werden auch alle seine Teile neu erzeugt. Ein Attribut beschreibt den Zustand eines Objekts und kann verwendet werden, um ein Objekt etwa in einer Datenbank zu suchen. Das Teil einer Komposition wird in dem Zusammenhang eher nicht verwendet.

Attribute und Teile (Komposition) können zwar Beziehungen zu Objekten außerhalb des Ganzen haben, wobei aber nur in der Richtung nach außen navigierbar ist. D.h. das Teil kann von außerhalb des Ganzen nicht direkt referenziert werden.

Die Komposition wird manchmal auch so interpretiert, dass die Teile nur einen einzigen Besitzer haben.

2.4.2 Aggregation

Die Aggregation wird mit einer leeren Raute auf der Seite des Ganzen notiert (mit impliziter Multiplizität 0..1). Auf der Seite des Teils kann eine Multiplizität angegeben werden.



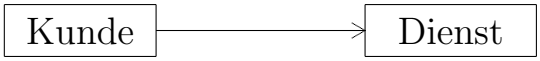
Die Teile können – zeitlich – vor und nach dem Ganzen existieren. Ein Teil kann durch ein anderes Teil desselben Typs ausgetauscht werden. Ein Teil kann von einem Ganzen zu einem anderen Ganzen transferiert werden. Aber ein Teil gehört zu einer Zeit höchstens zu *einem* Ganzen.

Im Hinblick auf Datenbanken haben hier die Teile meistens eine eigene Persistenz. Sie werden nicht automatisch mit dem Ganzen gelöscht oder angelegt oder aus der Datenbank geladen.

Oft wird die Aggregation auch so interpretiert, dass die Teile mehr als einen Besitzer haben, wobei die Besitzer aber unterschiedlichen Typs sein müssen.

2.5 Benutzung

Die **Benutzung** oder **Navigierbarkeit** oder **Delegation** ***(uses-a)*** ist eine Beziehung zwischen Benutzer ***(user)*** und Benutztem ***(used)***, bei der der Benutzte meistens von vielen benutzt wird und nichts von seinen Nutzern weiß.



Das kann man folgendermaßen lesen:

- Benutzung: Ein Kunde *benutzt* ein Dienst-Objekt.
- Delegation: Ein Kunde *delegiert an* ein Dienst-Objekt.
- Navigierbarkeit: Ein Kunde *hat eine Referenz* auf ein Dienst-Objekt. Ein Kunde kennt ein Dienst-Objekt.

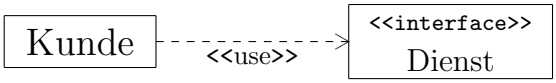
Auf der Seite des Benutzers hat man implizit die Multiplizität $*$.

Auf der Dienst-Seite ist auch eine Multiplizität möglich, wenn der Kunde etwa mehr als ein Dienstobjekt benutzt.

Der Dienst hat immer eine eigene Persistenz. Sein Lebenszyklus ist unabhängig von dem der Kunden. Er kennt typischerweise seine Kunden nicht, d.h. es gibt keine Navigationsmöglichkeit vom Dienst zum Kunden.

Die Benutzungs-Beziehung impliziert folgende Multiplizitäten:

Wenn eine Schnittstelle benutzt wird, dann kann auch folgende Notation verwendet werden:



2.6 Erweiterung, Vererbung

Die Instanz einer Entität ist mindestens vom Typ *eines* Entitätstyps. Sie kann aber auch Instanz *mehrerer* Entitätstypen sein, wenn eine "Ist-ein"-Typenhierachie vorliegt. Z.B. müssen folgende Fragen mit "ja" beantwortbar sein:

- Ist ein Systemprogrammierer ein Programmierer?
- Ist ein Systemprogrammierer eine **Erweiterung** ***(extension)*** von Programmierer?
- Ist ein Systemprogrammierer eine **Subklasse** ***(subclass)*** von Programmierer?
- Ist ein Systemprogrammierer ein **Untertyp** ***(subtype)*** von Programmierer?
- Sind alle Elemente der Menge Systemprogrammierer Elemente der Menge Programmierer?

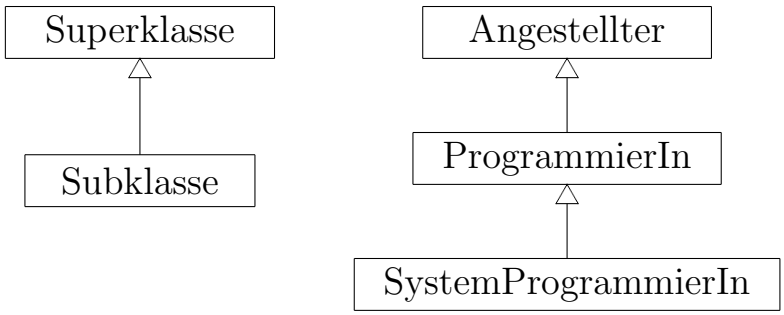
Ferner *ist* ein Programmierer ein Angestellter (ist Untertyp von Angestellter). Eigenschaften und Beziehungen werden von den Untertypen geerbt. Alles, was für die **Superklasse, (Basisklasse, Eltertyp, Obertyp, Obermenge,** ***superclass, baseclass, parent type, supertype, superset)*** gilt, gilt auch für die **Subklasse (Kindtyp, Subtyp, Teilmenge,** ***subclass, child type, subtype, subset)***.

Die wichtigsten Tests, ob wirklich eine Erweiterung vorliegt sind:

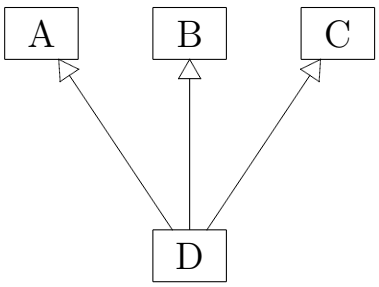
- Ist-ein-Beziehung
- Substitutionsprinzip von Liskov
- Teilmengen-Beziehung

Der Untertyp ist eine **Spezialisierung** des Obertyps. Der Obertyp ist eine **Generalisierung** des Untertyps oder verschiedener Untertypen. Generalisierung und Spezialisierung sind *inverse* Prozeduren der Datenmodellierung.

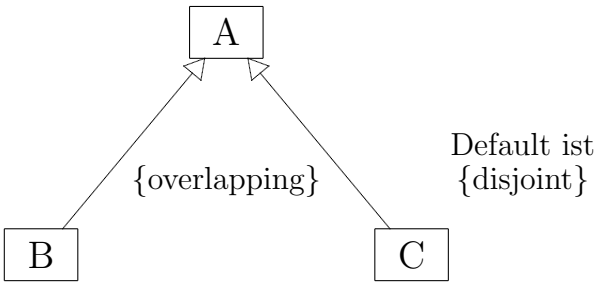
Die Erweiterung wird durch einen Pfeil mit dreieckiger Pfeilspitze dargestellt.



Mehrfachvererbung ist diagrammatisch leicht darstellbar. Auf die damit zusammenhängenden Probleme sei hier nur hingewiesen.

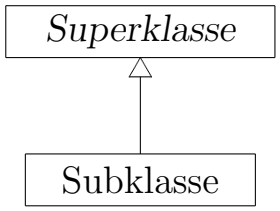


Betrachten wir Erweiterungen als Teilmengen, dann stellt sich die Frage, ob die Teilmengen **disjunkt** ***(disjoint)*** oder **überlappend** ***(overlapping)*** sind. Man spricht von der **Einschränkung durch Disjunktheit** ***(disjointness constraint)***. In der Darstellung ist *disjoint* Default.



Eine weitere Einschränkung der Erweiterung bzw Spezialisierung/Generalisierung ist die der **Vollständigkeit** ***(completeness constraint)***, die **total** oder **partiell** sein kann. Bei der totalen Erweiterung muss jede Entität der Superklasse auch Entität einer Subklasse sein. Die Superklasse ist dann eine **abstrakte** ***(abstract)*** Klasse, d.h. von ihr können keine Objekte angelegt werden. Eine Generalisierung ist üblicherweise total, da die Superklasse aus den Gemeinsamkeiten der Subklassen konstruiert wird.

Eine abstrakte Klasse wird mit **kursivem** Klassennamen dargestellt:



Die abtrakten (nicht implementierten) Methoden einer abtrakten Klasse werden auch kursiv dargestellt.

Da jeder Untertyp auch seinerseits Untertypen haben kann, führt das zu **Spezialisierungshierarchien** und zu **Spezialisierungsnetzen** ***(specialization lattice)*** bei Untertypen, die mehr als einen Obertyp haben **(Mehrfachvererbung,** ***multiple inheritance, shared subclass)***.

Generalisierung und Spezialisierung sind nur verschiedene Sichten auf dieselbe Beziehung. Als Entwurfs-Prozesse sind das allerdings unterschiedliche Methoden:

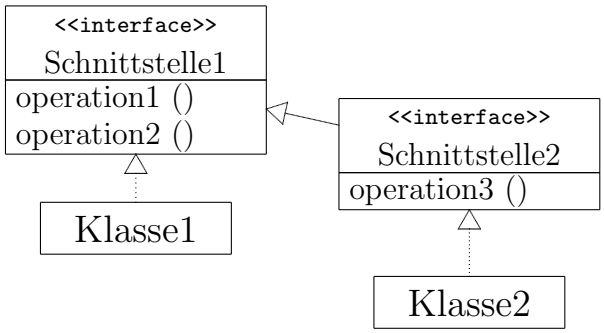
- Spezialisierung bzw Erweiterung: Wir beginnen mit einem Entitätstyp und definieren dann Untertypen durch sukzessive Spezialisierung bzw Erweiterung. Man spricht auch von einem ***top-down conceptual refinement***.
- Generalisierung: Der umgekehrte Weg beginnt bei sehr speziellen Untertypen, aus denen durch sukzessive Generalisierung eine Obertypenhierarchie bzw ein Obertypennetz gebildet wird. Man spricht von ***bottom-up conceptual synthesis***.

In der Praxis wird man irgendwo in der Mitte beginnen und beide Methoden anwenden.

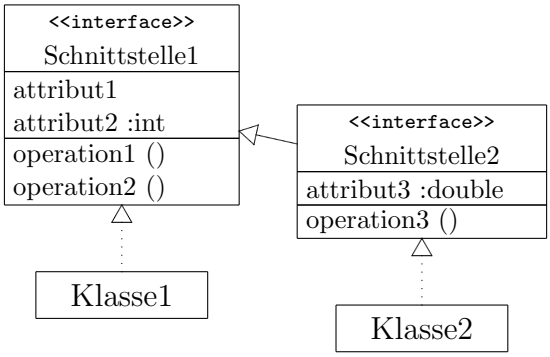
2.7 Realisierung

Die Beziehung **Realisierung** ***(realization)*** ist die Beziehung zwischen einer **Spezifikation** ***(specification)*** oder einem **Spezifikator** ***(specifier)*** und einer **Implementierung** ***(implementation)*** oder **Implementor** ***(implementor)***. Meistens ist das die Beziehung zwischen einer Schnittstelle und einer sie realisierende Klasse. Die Realisierung wird durch einen gestrichelten Erweiterungspfeil dargestellt.

Eine **Schnittstelle** ***(interface)*** ist eine total abstrakte Klasse. Alle ihre Operationen sind abstrakt. Eine Schnittstelle kann andere Schnittstellen erweitern.



Seit UML 2 kann eine Schnittstelle auch Attribute haben:

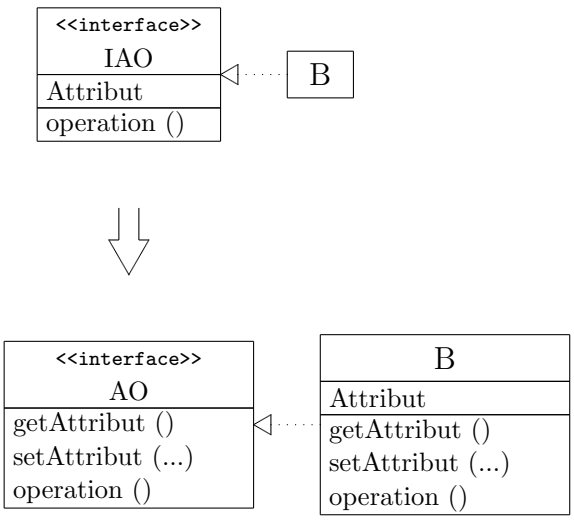


Das bedeutet, dass eine Schnittstelle auch aktiv Beziehungen zu anderen Schnittstellen und Klassen aufnehmen kann. Damit kann man dann den Klassenentwurf sehr abstrakt halten.

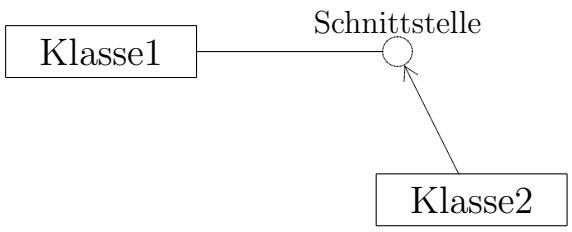
Eine Klasse kann mehr als eine Schnittstelle mit Attributen realisieren. Man hat dann natürlich eventuell Probleme der Mehrfachvererbung.

In C++ kann eine Schnittstelle mit Attributen einfach als abstrakte Klasse realisiert werden.

In Java müsste man die Attribute durch die entsprechenden set/get-Methoden repräsentieren. Die eigentlichen Attribute müssten in der realisierenden Klasse Aufgenommen werden.



Klassen können Schnittstellen anbieten, die dann von anderen Klassen benutzt werden. Die Benutzung einer Schnittstelle wird auch durch eine Socket-Notation dargestellt.



2.8 Assoziation

Wenn eine Beziehung zwischen Klassen nicht eine der in den vorhergehenden Abschnitten diskutierten Beziehungen ist (Attribut, Komposition, Aggregation, Benutzung, Erweiterung, Realisierung), dann kann sie als Assoziation ***(association)*** modelliert werden. Die Assoziation ist die allgemeinste Form einer Beziehung, die bestimmte Objekte semantisch eingehen.

Die an einer Beziehung beteiligten Entitäten heißen **Teilnehmer** ***(participants)*** einer Beziehung. Die Anzahl der Teilnehmer ist der **Grad** ***(degree)*** der Beziehung. Je nach Grad kann die Beziehung **binär, ternär** oder n**-wertig** ***(binary, ternary, n-ary)*** sein.

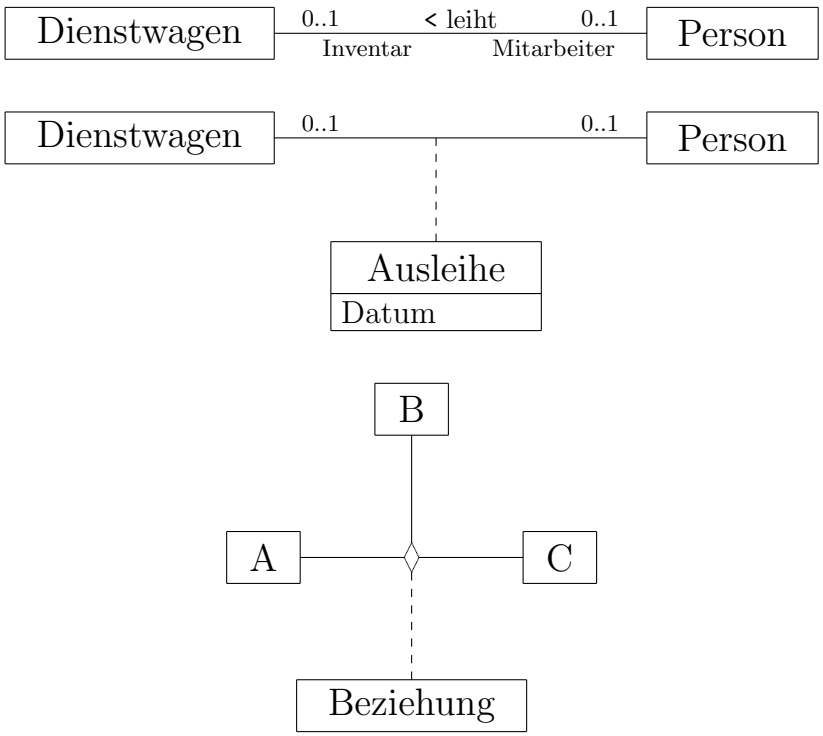
Von der Einführung von mehr als zweiwertigen Beziehungen ist abzuraten. Es sollte versucht werden, diese Beziehungen durch mehrere Zweier-Beziehungen darzustellen. Dadurch werden von vornherein Redundanzen vermieden. Eine n-wertige Beziehung kann immer durch eine Entität dargestellt werden, die binäre Beziehungen zu den n beteiligten Entitäten unterhält.

Zweiwertige Beziehungen werden durch Linien dargestellt, mehrwertige Beziehungen durch eine zentrale Raute. Dabei können Rollen und Multiplizitäten angegeben werden. Der Name der

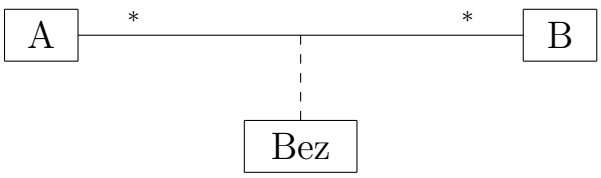
Beziehung steht – eventuell mit einer "Leserichtung" versehen – an der Linie oder in einem eigenen Rechteck, das wie eine Klasse mit Attributen und Operationen notiert werden kann (sogenannte **Assoziationsklasse**).

Bei einer mehrwertigen Beziehung müssen im Prinzip die Multiplizitäten zwischen je zwei beteiligten Partnern geklärt werden. Bei einer ternären Beziehung ist das graphisch noch vernünftig machbar.

Assoziationen können Navigationspfeile haben. (Bemerkung: Oft werden bei einer Assoziation ohne Pfeile die Navigationspfeile auf beiden Seiten impliziert und entspricht damit der ***Relationship*** des Modells der ODMG. Es ist selten, dass eine Assoziation ohne Navigation existiert.) Wenn eine Navigationsrichtung ausgeschlossen werden soll, dann wird dies mit einem Kreuz "×" auf der entsprechenden Seite notiert.



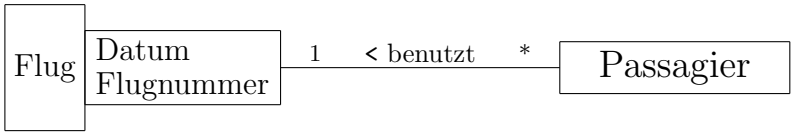
Da Assoziationen meistens als eigene Klassen implementiert werden, wird das oft schon durch eine **reifizierte** (verdinglichte, konkretisierte, ***reified***) Darstellung ausgedrückt. Anstatt von



wird

notiert.

Ein **Assoziations-Endpunkt** ***(association end)*** kann außer einem Rollennamen und einer Multipliziert auch noch einen **Qualifikator** ***(qualifier)*** haben, der eine Teilmenge von Objekten einer Klasse bestimmt, meistens genau ein Objekt.



Normalerweise wäre die Beziehung zwischen `Passagier` und `Flug` eine Many-to-Many-Beziehung. Hier drücken wir aus, dass, wenn `Datum` und `Flugnummer` gegeben sind, es für jeden Passagier genau einen Flug gibt.

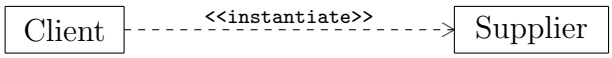
Bemerkung: Beziehungen zwischen Objekten sollten in UML-Diagrammen i.a. durch Assoziationen (also irgendwelche Linien zwischen Klassen) und nicht nur durch Attribute dargestellt werden, obwohl bei den meisten Programmiersprachen die Beziehungen zu Klassenelementen werden. Eine Assoziation macht aber deutlich, dass eventuell eine referentielle Integrität zu beachten ist, d.h. eine Abhängigkeit zwischen Objekten zu verwalten ist.

2.9 Abhängigkeit

Mit der Abhängigkeit ***(dependency)*** können Beziehungen allgemeiner Art zwischen einem Client und Supplier dargestellt werden.

Dargestellt wird die Abhängigkeit durch einen gestrichelten Pfeil vom Client zum Supplier mit stereotypem Schlüsselwort ***(keyword)***. Die Pfeilrichtung ist oft nicht sehr intuitiv. Dabei hilft folgende Überlegung (Navigation): Welche Seite kennt die andere Seite? **Die Client-Seite kennt die Supplier-Seite.**

Der folgende Konstrukt



bedeutet, dass ein Client-Objekt ein Supplier-Objekt erzeugt oder instanziiert. Die Supplier-Klasse stellt ihr Objekt zur Verfügung.

Folgende Abhängigkeiten sind in UML definiert:

access: `<<access>>`
: Ein **Paket** ***(package)*** kann auf den Inhalt eines anderen (Ziel-)Pakets zugreifen.

binding: `<<bind>>`
: Zuweisung von Parametern eines Templates, um ein neues (Ziel-)Modell-Element zu erzeugen.

call: `<<call>>`
Die Methode einer Klasse ruft eine Methode einer anderen (Ziel-)Klasse auf.

derivation: `<<derive>>`
Eine Instanz kann aus einer anderen (Ziel-)Instanz berechnet werden.

friend: `<<friend>>`
Erlaubnis des Zugriffs auf Elemente einer (Ziel-)Klasse unabhängig von deren Sichtbarkeit.

import: `<<import>>`
Ein Paket kann auf den Inhalt eines anderen (Ziel-)Pakets zugreifen. Aliase können dem Namensraum des Importeurs hinzugefügt werden.

instantiation: `<<instantiate>>`
Eine Methode einer Klasse kann ein Objekt einer anderen (Ziel-)Klasse erzeugen. Die (Ziel-)Klasse stellt als Supplier das neue Objekt zur Verfügung.

parameter: `<<parameter>>`
Beziehung zwischen einer Operation und ihren (Ziel-)Parametern.

realization: `<<realize>>`
Beziehung zwischen einer (Ziel-)Spezifikation und einer Realisierung.

refinement: `<<refine>>`
Beziehung zwischen zwei Elementen auf verschiedenen semantischen Stufen. Die allgemeinere Stufe ist das Ziel.

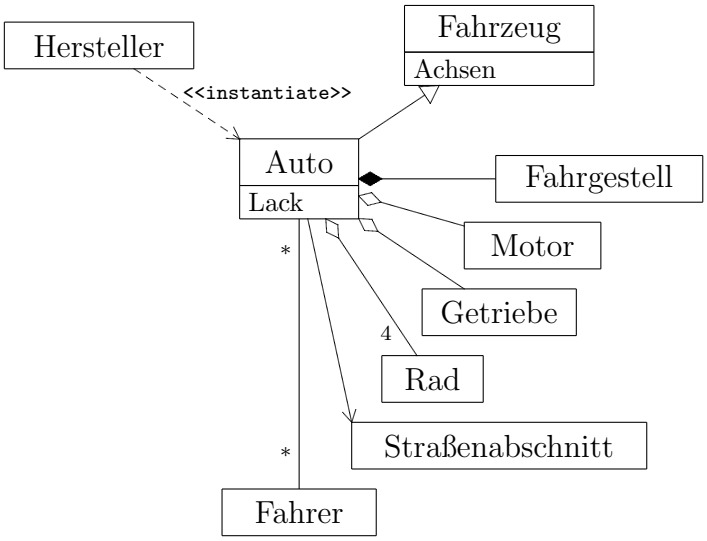
send: `<<send>>`
Beziehung zwischen Sender und Empfänger einer Nachricht.

trace: `<<trace>>`
Es besteht eine gewisse – etwa parallele – Beziehung zwischen Elementen in verschiedenen Modellen.

usage: `<<use>>`
Ein Element setzt das Vorhandensein eines anderen (Ziel-)Elements für seine korrekte Funktionsfähigkeit voraus.

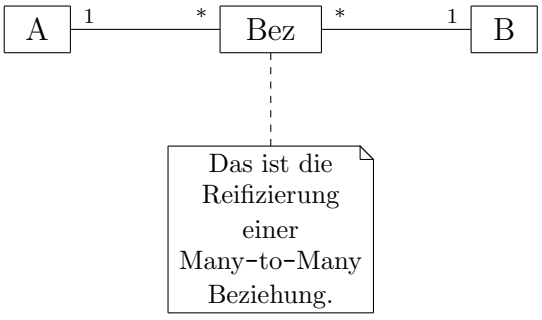
2.10 Zusammenfassung der Beziehungen

Folgendes Beispiel soll nochmal die Unterschiede zwischen Erweiterung, Attribut, Komposition, Aggregation, Benutzung, Abhängigkeit und Assoziation zeigen:



2.11 Notiz

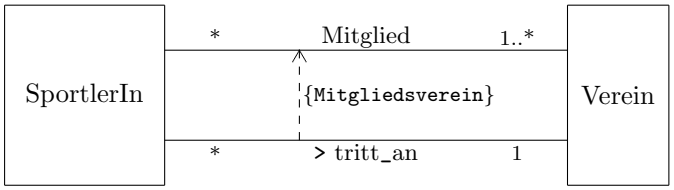
Mit dem Konstrukt der Notiz ***(note)*** können Kommentare, Bemerkungen oder andere textuelle Informationen im Diagramm untergebracht werden. Oft werden in einer Notiz Hinweise zur Implementierung einer Methode gegeben. Der Text wird in einem Rechteck mit Eselsohr dargestellt. Eine gestrichelte Linie gibt an, worauf sich die Notiz bezieht.

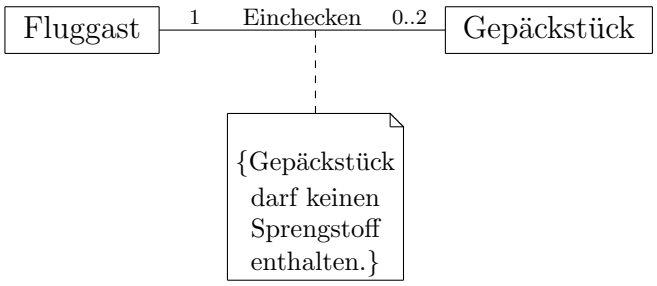


2.12 Einschränkung

Eine Einschränkung ***(constraint)*** ist ein Text in einer natürlichen oder formalen Sprache in geschweiften Klammern. Der Text kann auch in einer Notiz stehen (mit oder ohne geschweifte Klammern).

Dieser Text wird mit einer gestrichelten Linie oder einem gestrichelten Pfeil mit dem Element verbunden, für das die Einschränkung gilt. Wenn ein Pfeil verwendet wird, dann soll der Pfeil vom eingeschränkten zum einschränkenden Element zeigen.





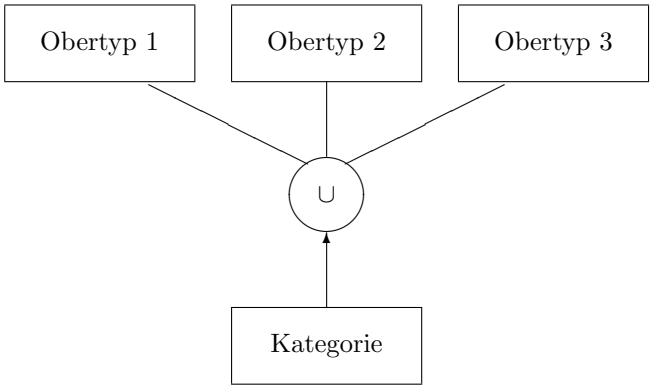
2.12.1 Kategorie *(category)*

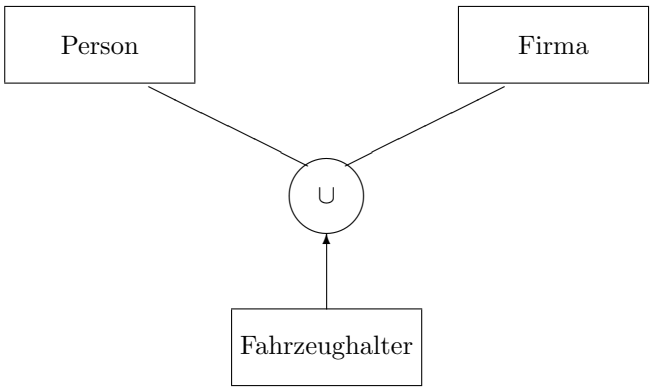
Ein Fahrzeughalter kann eine Person oder eine Firma sein. Da nicht jede Person und nicht jede Firma ein Fahrzeughalter ist, kann Fahrzeughalter kein Obertyp sein.

Fahrzeughalter kann auch kein Untertyp sein. Denn von wem sollte Fahrzeughalter ein Untertyp sein – Person oder Firma?

Fahrzeughalter ist ein Beispiel für eine **Kategorie**, die von verschiedenen Obertypen **exklusiv** erben kann. Mit "exklusiv" ist gemeint, dass die Entität Fahrzeughalter nur von genau einem Obertyp erben kann, dass sie nicht die Kombination mehrerer Obertypen ist, wie das bei Mehrfachvererbung der Fall ist. Eine Kategorie erbt *einen* Obertyp aus einer Vereinigung von Obertypen.

Die graphische Darstellung sieht folgendermaßen aus, wobei der Kreis ein Vereinigungszeichen enthält.

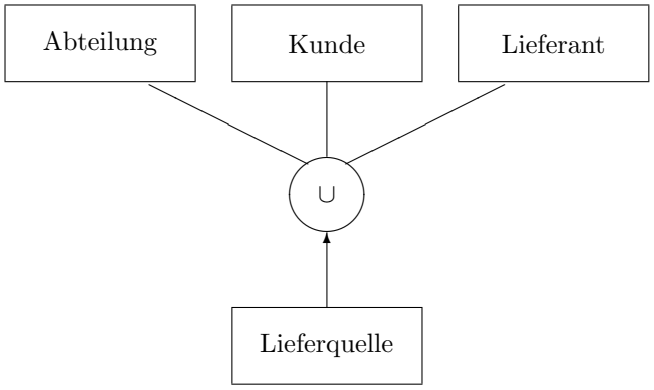




Eine Kategorie kann **partiell** oder **total** sein. Totalität bedeutet, dass jeder Obertyp der Kategorie zur Kategorie gehören muss. In diesem Fall ist eine Darstellung als Obertyp-Untertyp-Beziehung auch möglich, wobei die Kategorie der Obertyp ist, und ist meistens vorzuziehen, insbesondere wenn die Entitäten viele Attribute teilen.

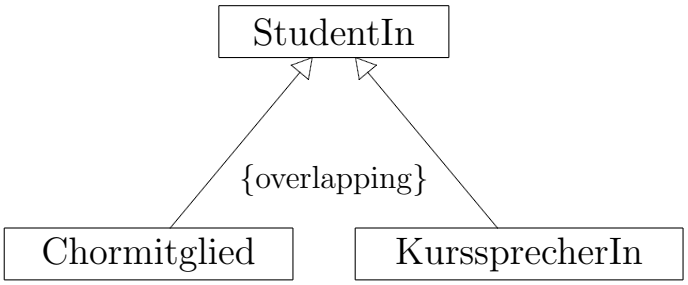
Kategorien kommen häufig vor. Sie sind eine Art von Rolle, die eine Entität spielen kann. Insbesondere Personen sind in den verschiedensten Rollen zu finden.

Beispiel:

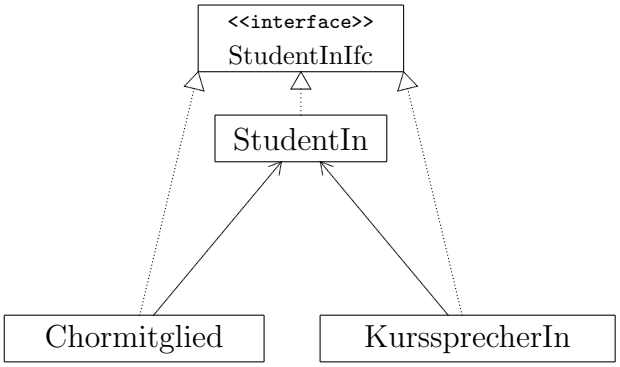


2.13 Beispiele

2.13.1 Nichtdisjunkte Untertypen



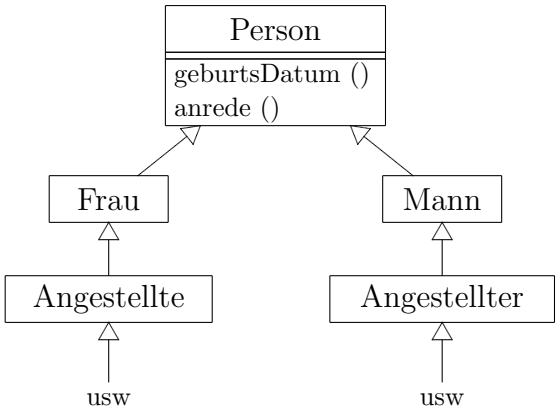
Wie wird das implementiert? Als zwei Kategorien (Chormitglied und Kursprecher):



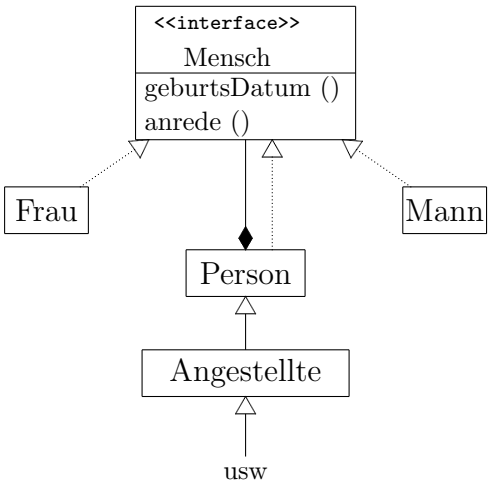
Sollte die Uses-Beziehung nicht besser zur Schnittstelle gehen? Im allgemeinen nein, denn es könnte ja sein, dass das Chormitglied oder der Kurssprecher zur Realisierung der Schnittstelle Methoden oder Datenelemente benötigt, die die Schnittstelle nicht anbietet.

2.13.2 Weiblich – Männlich

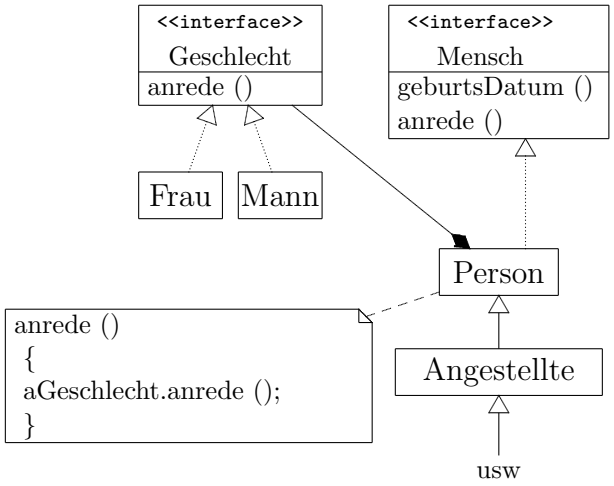
Wenn Frauen und Männer im wesentlichen gleich behandelt werden, dann ist folgende Struktur nicht günstig, weil das zu zwei redundanten Vererbungshierarchien führt:



Stattdessen sollte man eine "Person" als Kategorie einführen:



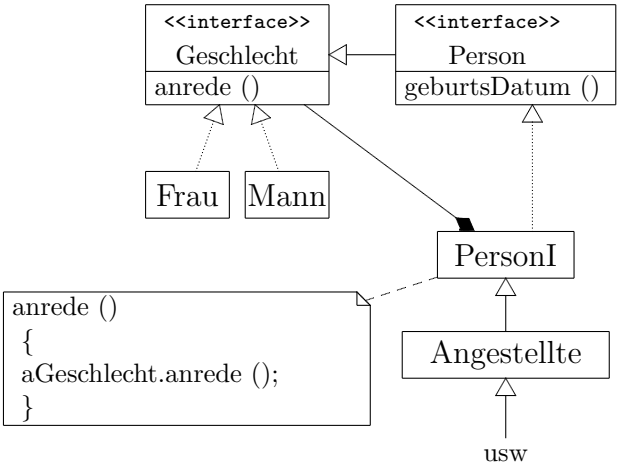
Um zu verhindern, dass sich `Person` wieder eine `Person` als Komponente nimmt, wäre folgende Struktur besser:



Übung: Diskutieren Sie die unterschiedlichen Optionen (Komposition, Aggregation, Benutzung) für die Beziehung zwischen `Person` und `Geschlecht`.

Kategorien haben eine große Ähnlichkeit zum Bridge-Pattern.

Um die "anrede"-Redundanz zu entfernen, wäre noch schöner:



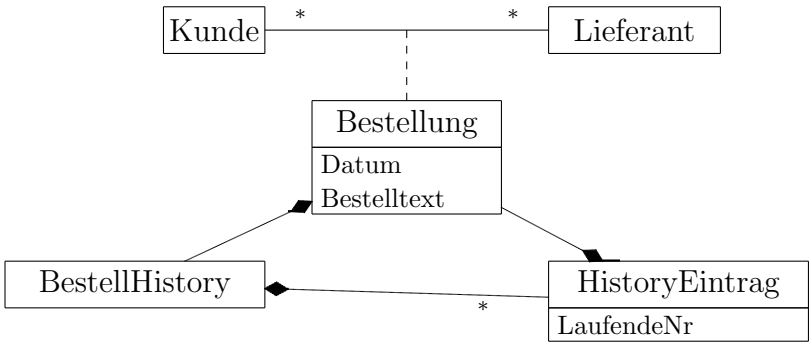
Allerdings kann eine `Person` wieder eine `Person` als Komponente erhalten, was wir aber in Kauf nehmen.

Das ist übrigens eine treue Abbildung der – eventuell in einer Problembeschreibung erscheinenden – Sätze *"Ein Angestellter ist eine Person. Eine Person ist ein Mensch und hat ein Geschlecht."*

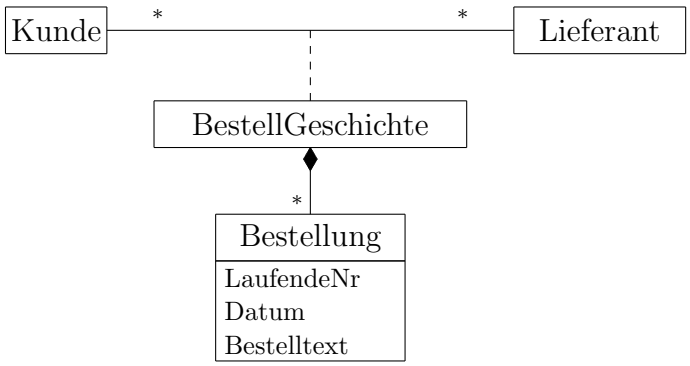
Übung: Diskutieren Sie die unterschiedlichen Optionen (Komposition, Aggregation, Benutzung) für die Beziehung zwischen `Person` und `Geschlecht`.

2.13.3 Many-to-Many-Beziehung mit History

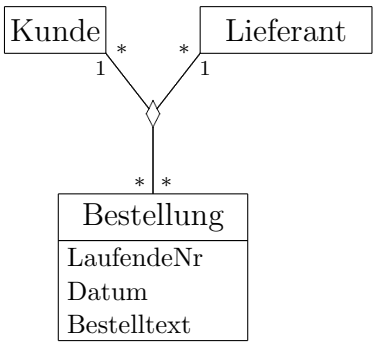
Viele Many-to-Many-Beziehungen zwischen zwei Partnern werden nicht nur einmal, sondern mehrmals eingegangen. Z.B. kann eine Person dasselbe Buch mehrmals leihen. Oder ein Kunde kann bei einer Firma mehrere Bestellungen machen. Um all diese Vorgänge zu verwalten, muss eine sogenannte ***History*** oder "Geschichte" für ein Beziehungsobjekt angelegt werden. Dabei mag es sinnvoll sein, den verschiedenen Vorgängen eine laufende Nummer zu geben. Das könnte folgendermaßen modelliert werden:



Hier wurde darauf geachtet, dass die ursprüngliche Struktur möglichst erhalten bleibt. Das macht es aber ziemlich umständlich. Wahrscheinlich würde man folgende einfachere Struktur vorziehen, die sich auch wesentlich bequemer in ein RDB übersetzen lässt.



Wir können das Ganze auch als eine ternäre Beziehung betrachten, wobei wir ein gutes Beispiel für die unterschiedlichen Multiplizitäten zeigen können:


Kunde
Lieferant
1
*
*
1
*
*
Bestellung
LaufendeNr
Datum
Bestelltext