# UML

—

# Unified Modeling Language

**Prof. Dr. Karl Friedrich Gebhardt**

Auflage vom 9. März 2021

Prof. Dr. K. F. Gebhardt

Tel: 0711-184945-11(16)(15)(12)
Fax: 0711-184945-10
email: kfg@lehre.dhbw-stuttgart.de

# Vorwort

Ziel des Seminars sind, dass die Teilnehmer

- die Syntax der wichtigsten Elemente von UML kennen
- wissen, wie die verschiedenen Diagrammtypen von UML angewendet werden
- eigenständig Entwürfe in UML erstellen können

# Inhaltsverzeichnis

# Kapitel 1

# Überblick und Kurzreferenz

**UML** ***(unified modeling language)*** ist eine Modellierungs-Sprache für Software-Systeme, Unternehmens-Modellierung (Geschäftsprozesse) und andere Nicht-Software-Systeme [10].

Die aktuelle Version ist UML 2.0[10]. Spezifikationen sind erhältlich von der Object management Group (OMG) Web-Seite `http://www.omg.org` .

Die UML Superstructure ist die formale Definition aller UML Elemente auf etwa 600 Seiten.

## 1.1 Modelle, Sichten, Diagramme

Im folgenden Abschnitt wollen wir erklären, was Modelle ***(model)***, Sichten ***(view)*** und Diagramme ***(diagram)*** sind.

Ein Modell ist die Repräsentation eines Teils der realen Welt durch ein bestimmtes Medium (z.B. Zeichnung oder Text auf Papier, elektronisches Medium, Holzmodell, mathematisches Simulationsmodell). Das Modell repräsentiert nicht die ganze reale Welt, sondern nur die für eine Geschäftstätigkeit relevanten Aspekte der realen Welt. Dazu wird *abstrahiert*, d.h. vereinfacht und vernachlässigt. Das Modell eines Software-Systems wird in einer Modellierungs-Sprache wie z.B. UML erstellt.

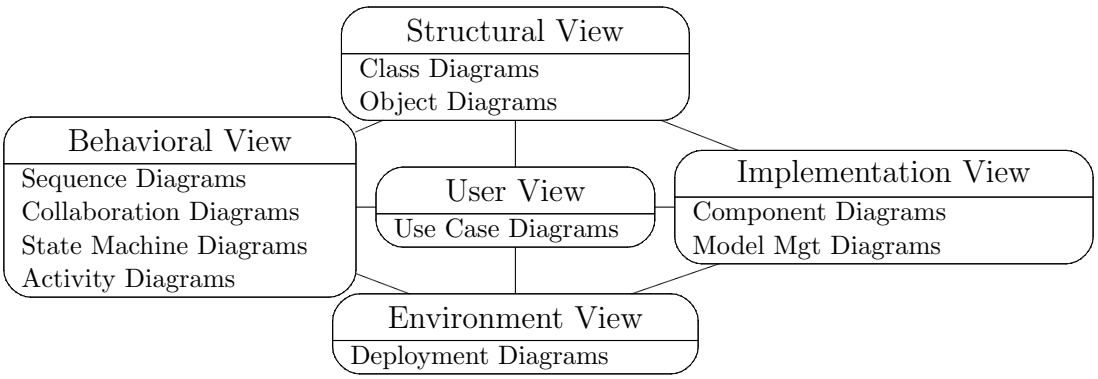
Ein Modell hat zwei wesentliche Aspekte: **Bedeutung oder Semantik** ***(semantics)*** und **Notation** ***(notation)***. Für das Verständnis des Modells oder die Verwendbarkeit (z.B. zur Code-Generierung) ist die Semantik des Modells wichtig.

Die Notation legt das äußere Erscheinungsbild ***(visual representation)*** des Modells fest (Zeichnung und/oder Text). Ein Modell ist normalerweise so kompliziert, dass es nur durch die Unterteilung in verschiedene **Sichten** ***(views)*** handhabbar wird. Eine Sicht ist daher eine Teilmenge der UML-Modellierungs-Konstrukte und repräsentiert einen Aspekt des Modells. Eine Sicht wird visuell durch i.a. mehrere **Diagramm**-Arten ***(diagram)*** dargestellt.

Da jedes Modell von der realen Welt abstrahiert und sie daher nicht exakt darstellt, ist jedes Modell eigentlich "falsch". Außerdem gibt es i.a. viele unterschiedliche Modelle. Welches dieser Modelle nun "richtig" ist, kann nur daran gemessen werden, wie gut es den aktuellen und insbesondere den zukünftigen Zustand der realen Welt repräsentiert und ihr Verhalten vorhersagt.

Die verschiedenen Sichten lassen sich folgendermaßen gliedern:

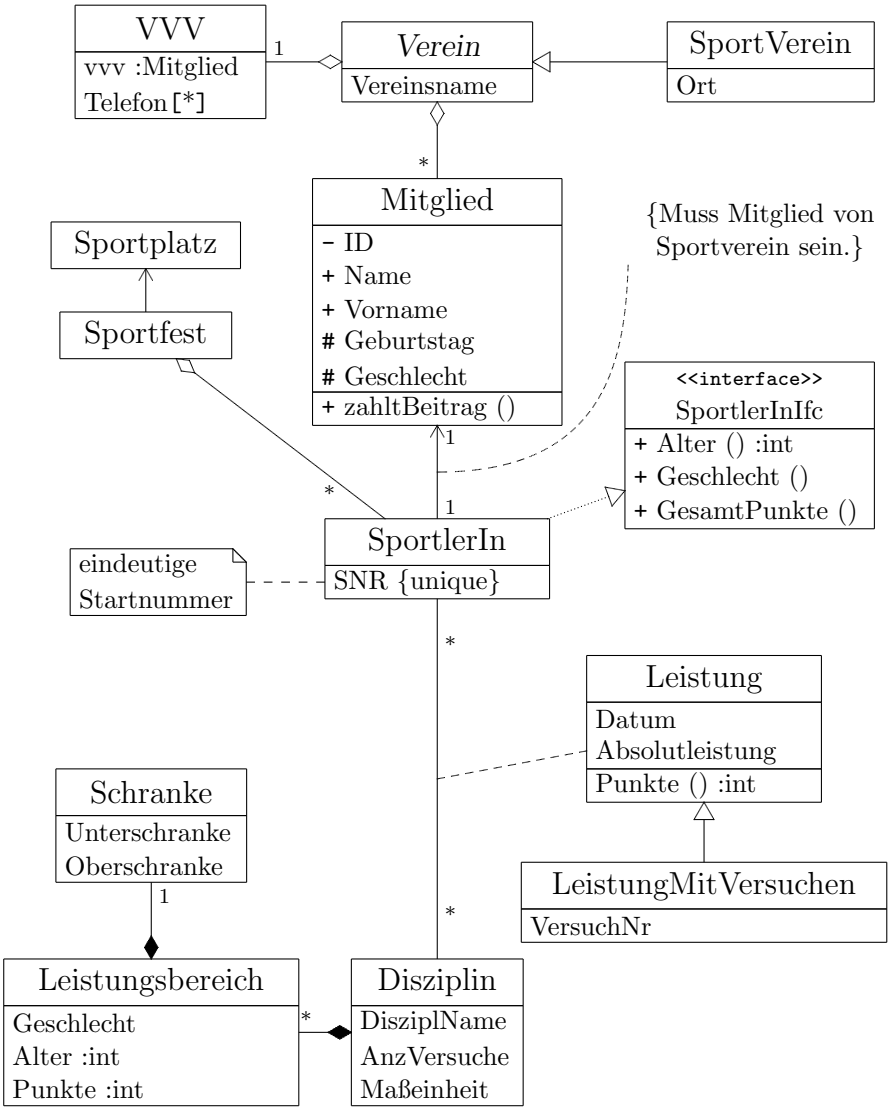
- Benutzer-Sicht *(user view, use case view)*
  - Anwendungsfall-Diagramme
- Strukturelle Sicht *(structural view, design view)*
  - Klassen-Diagramme
  - Objekt-Diagramme
- Verhaltens-Sicht *(behavioral view, process view)*
  - Sequenz-Diagramme
  - Kollaborations-Diagramme
  - Zustands-Diagramme
  - Aktivitäts-Diagramme
- Implementierungs-Sicht *(implementation view)*
  - Komponenten-Diagramme
  - Modell-Management-Diagramme
- Umgebungs-Sicht *(environment view, deployment view)*
  - Einsatz-Diagramme




Die folgenden Abschnitte stellen die verschiedenen UML-Sichten kurz vor, wobei unter Verwendung eines etwas künstlichen Beispiels versucht wird, möglichst viele (oder wenigstens die wichtigsten) Konstrukte einer Sicht in einem Diagramm unterzubringen. Wegen einer detallierten Erklärung der Konstrukte verweisen wir auf die folgenden Kapitel.

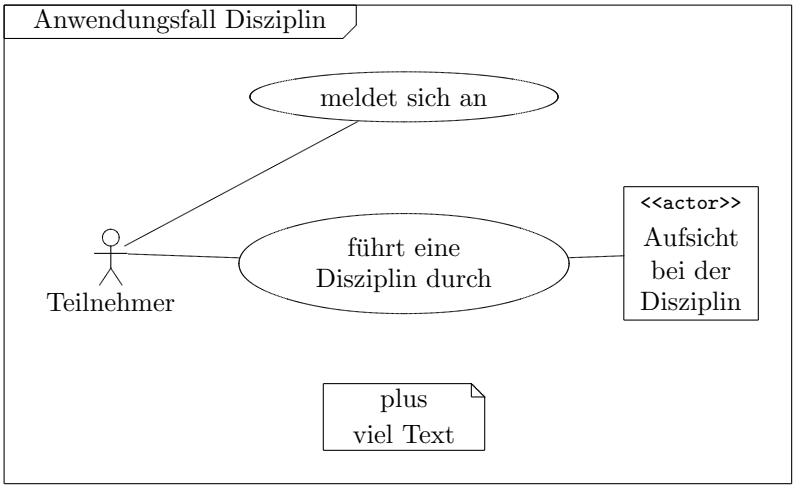
## 1.2 Strukturelle Sicht, Klassendiagramm

Die strukturelle oder statische Sicht ***(structural, static view)*** wird visuell mit einem Klassendiagramm ***(class diagram)*** dargestellt. Die Sicht ist statisch, weil sie das zeit*un*abhängige Verhalten des Systems beschreibt. Sie beschreibt die Objekte oder Klassen und ihre Beziehungen zueinander.



## 1.3 Anwendungsfall-Sicht

Die Sicht auf das System über Anwendungsfälle ***(use case view)*** modelliert die Funktionalität des Systems aus der Perspektive des externen System-Anwenders oder -Benutzers, eines sogenannten **Akteurs** ***(actor)***. Die Anwendungsfälle zeigen, welche Akteure in welchen Transaktionen oder Geschäftsprozessen mit dem System verkehren. Die visuelle Darstellung ist das Anwendungsfall-Diagramm.



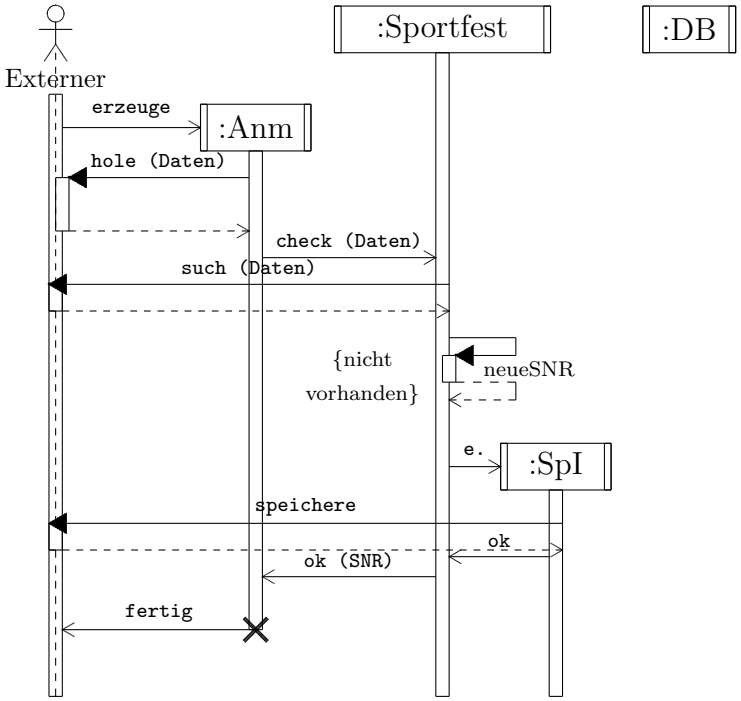

## 1.4 Interaktions-Sicht

Die Interaktions-Sicht ***(interaction view)*** beschreibt eine Folge von Botschaften, die zwischen **Rollen** ***(role)*** ausgetauscht werden. Rollen implementieren das Verhalten des Systems. Sie sind die Beschreibung eines Objekts, das innerhalb einer Interaktion eine bestimmte Rolle spielt. Zur Darstellung werden zwei Diagrammarten benutzt:

- **Sequenz-Diagramm** ***(sequence diagram)***
- **Kollaborations-Diagramm** ***(collaboration diagram)***

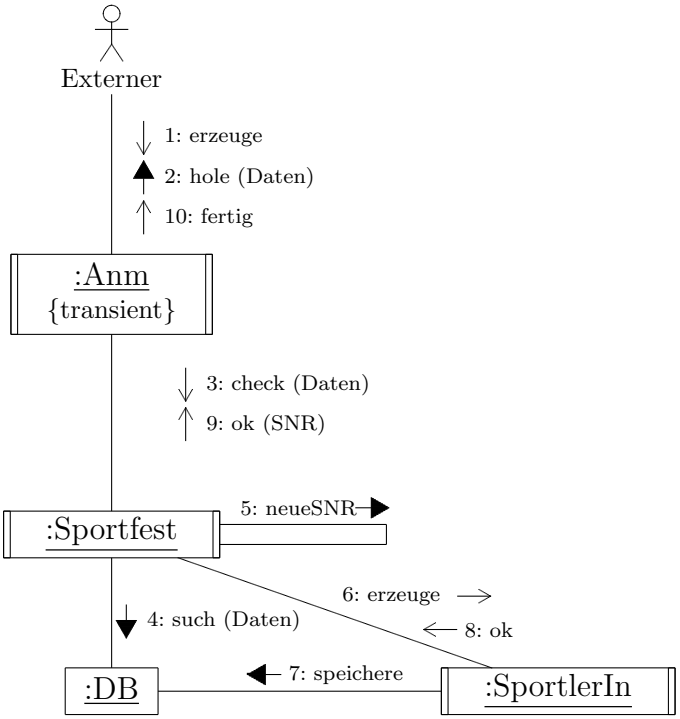
### 1.4.1 Sequenz-Diagramm

Die Botschaften werden in zeitlicher Folge als Pfeile zwischen Lebenslinien von Rollen angeordnet. Jedes Diagramm zeigt *ein* Szenario, d.h. *eine* individuelle Transaktionsgeschichte.



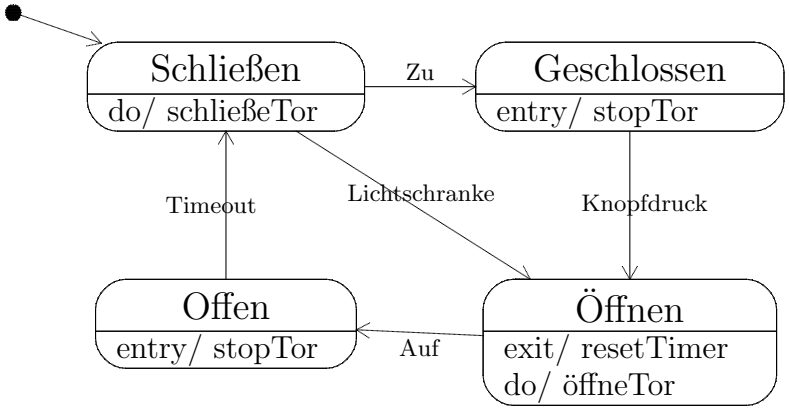

### 1.4.2 Kollaborations-Diagramm

Die Botschaften stehen – nach Reihenfolge nummeriert, mit Pfeilen versehen – auf Verbindungslinien zwischen den Objekten. Das Diagramm wird oft verwendet, um die Implementierung einer Operation zu zeigen.




## 1.5 Zustandsgraphen-Sicht

Die Zustandsgraphen-Sicht ***(state machine view)*** modelliert den Lebenszyklus von Objekten, die "leben", d.h. auf Grund von äußeren (außerhalb des Objekts) **Ereignissen** ***(event)*** ihren **Zustand** ***(state)*** ändern.

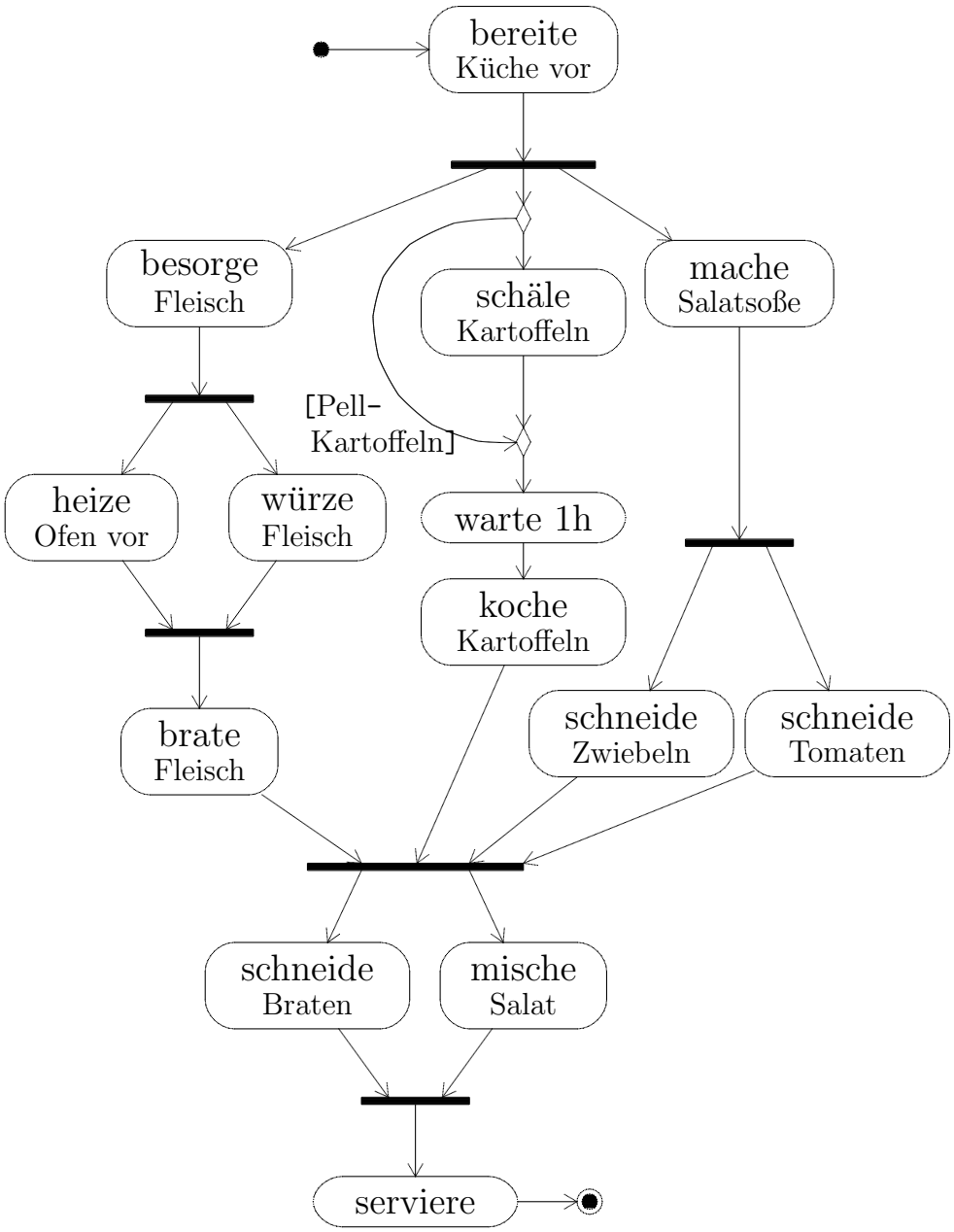



Zustandsgraphen können insbesondere für die Modellierung interaktiver Benutzeroberflächen eingesetzt werden.

## 1.6 Aktivitäten-Sicht

Die Aktivitäten-Sicht ***(activity view)*** wird als **Aktivitätsdiagramm** ***(activity graph)*** dargestellt. Das Aktivitätsdiagramm ist eine Variante des Zustandsgraphen, um etwa die – möglicherweise parallelen – Schritte einer komplexen Berechnung oder eines Geschäftsprozesses darzustellen.

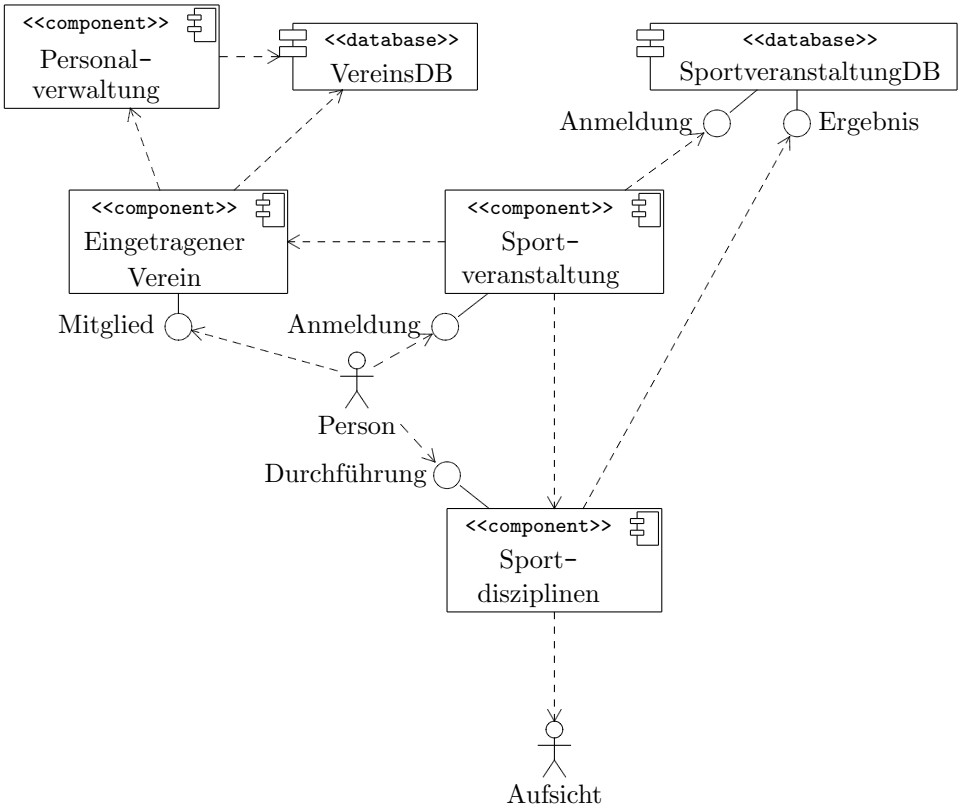
Ein **Aktivitätszustand** ***(activity state)*** repräsentiert einen ***workflow step*** oder die Durchführung einer Operation.



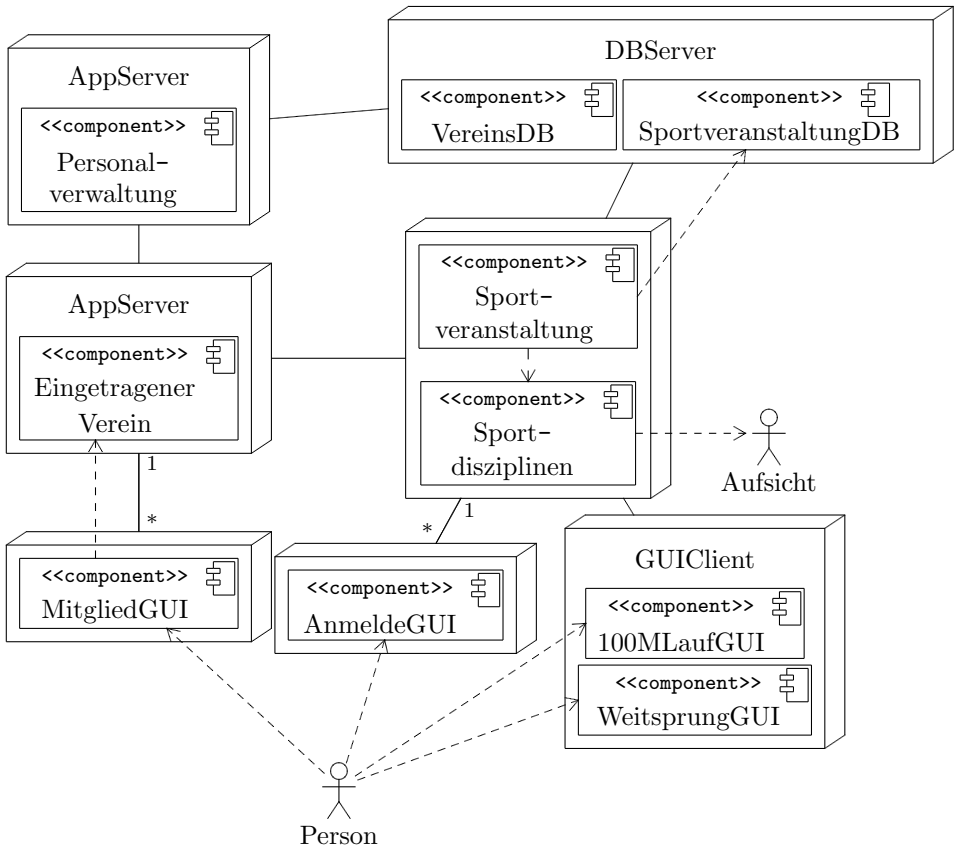
## 1.7 Physische Sichten

Die physischen Sichten ***(physical view)*** modellieren die Implementierungs-Struktur einer Anwendung, d.h. wie werden die Klassen in **Komponenten** ***(component)*** zusammengefasst **(Implementierungs-Sicht,** ***implementation view)*** und welche Komponenten laufen auf welchen **Knoten** ***(node)*** **(Einsatz-** oder **Verteilungs-Sicht,** ***deployment view)***.

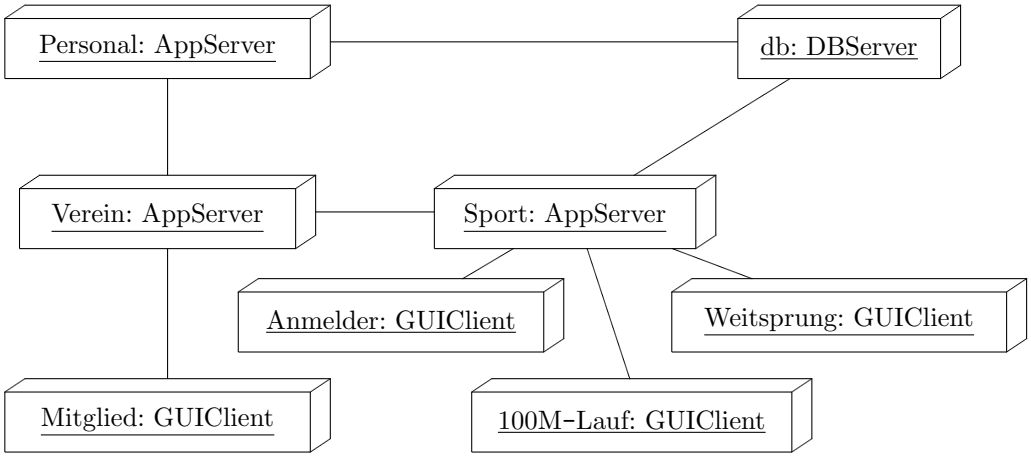
Die Implementierungs-Sicht zeigt die Komponenten (Software-Einheiten) und ihre Abhängigkeiten untereinander.




Die Einsatz-Sicht zeigt die Knoten (Hardware-Einheiten) und ihre Kommunikationsverbindungen. Auf der Beschreibungs-Ebene ***(descriptor-level)*** wird die Art der Knoten implizit durch die Komponenten beschrieben.



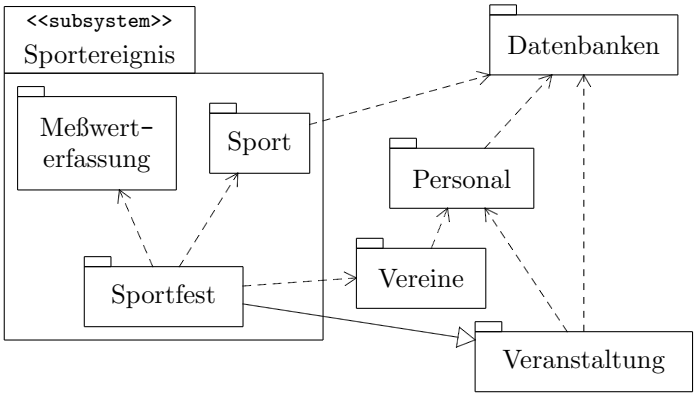

Auf der Instanzen-Ebene ***(instance-level)*** werden für eine Version des Systems individuelle Knoten mit ihren Namen und Verbindungen gezeigt.




## 1.8 Modell-Management-Sicht

Die Modell-Management-Sicht zeigt die Organisation des Modells. Ein Modell besteht aus **Paketen *(package)***, die Modell-Elemente (Klassen, Anwendungsfälle, Aktivitätsdiagramme usw) enthalten. Pakete können andere Pakete enthalten. Das Gesamtmodell ist daher ein Wurzel-Paket, das – indirekt – alle anderen Pakete enthält. Pakete sind Einheiten zur Festlegung von Zugriffsrechten und Einheiten des Konfigurations-Managements. Jedes Modellelement gehört zu genau einem Paket.

Ein **Teilsystem *(subsystem)*** ist ein spezielles Paket, das eine wohldefinierte, relativ kleine Schnittstelle nach außen hat und das als eigenständige Komponente implementiert werden kann.



## 1.9 Erweiterungs-Konstrukte

Erweiterungen sind möglich durch relativ willkürkliches Anfügen von Bedingungen oder Notizen in geschweiften Klammern oder durch die Definition von eigenen Ikonen.

## 1.10 Faustregel im Umgang mit UML

- Beinahe alles in UML ist optional.
- UML Modelle sind selten vollständig.
- UML-Konstrukte können unterschiedlich interpretiert werden. Die Interpretation hängt oft von Vereinbarungen innerhalb eines Entwicklungsteams ab.
- UML kann erweitert werden.

# 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)*

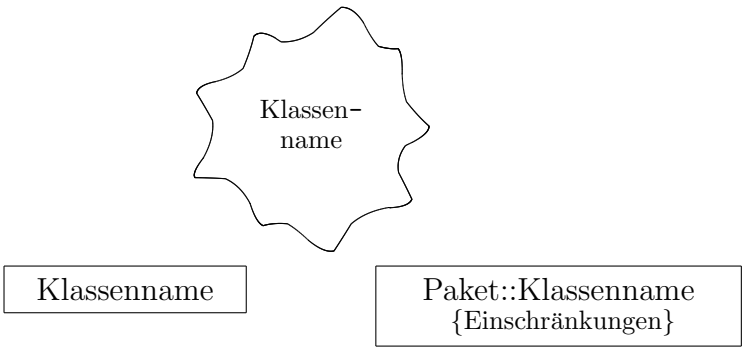
Chen definiert eine Entität als "a thing which can be distinctly identified". Eine Entität ist ein "Etwas" oder Objekt, das eindeutig identifiziert werden kann.

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**.

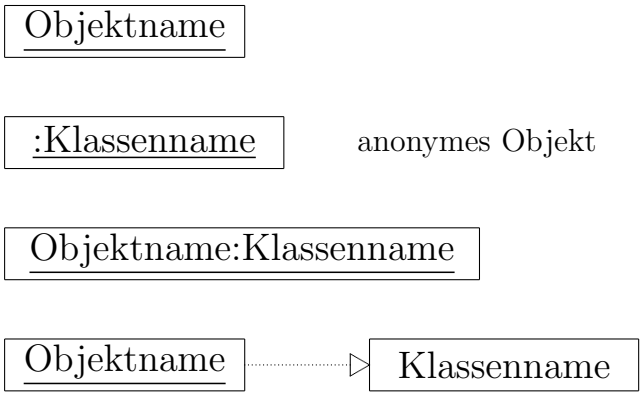
### 2.1.1 Klasse




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

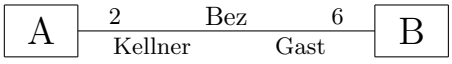
## 2.2 Multiplizitäten

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:

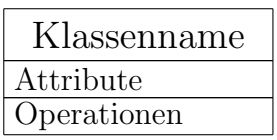
> *Ein* `A`*-Objekt steht (in der Rolle* `Kellner`*) zu genau 6* `B`*-Objekten in der Beziehung* `Bez` *(, wobei die* `B`*-Objekte dabei in der Rolle* `Gast` *auftreten).*
> *Ein* `B`*-Objekt steht (in der Rolle* `Gast`*) zu genau 2* `A`*-Objekten in der Beziehung* `Bez` *(, wobei die* `A`*-Objekte dabei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

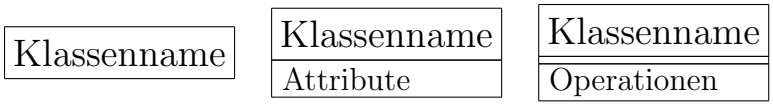
Jetzt können wir auf die verschiedenen Beziehungen einer Klasse zu anderen Klassen eingehen. Die engste Beziehung haben dabei die Eigenschaften einer Klasse.

## 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.

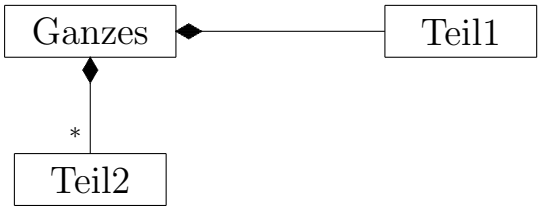
## 2.4 Teil-Ganzes-Beziehung

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.

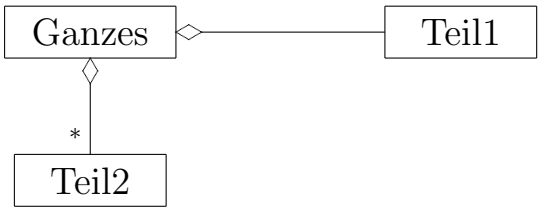
Komposition und Attribut sind kaum gegeneinander abzugrenzen. Im Hinblick auf Datenbanken haben die Teile der Komposition wie Attribute meistens keine eigene Persistenz. D.h. wenn das Ganze aus der Datenbank gelöscht wird, werden auch alle seine Teile gelöscht. Wenn ein Ganzes neu angelegt wird, werden 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.



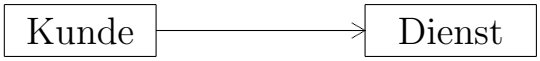

Die Teile können – zeitlich – vor und nach dem Ganzen existieren. Ein Teil kann durch ein anderes Teil desselben Typs ausgetauscht werden. Ein Teil kann von einem Ganzen zu einem anderen Ganzen transferiert werden. Aber ein Teil gehört zu einer Zeit höchstens zu *einem* Ganzen.

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.

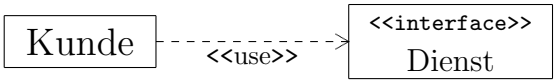
Auf der Seite des Benutzers hat man implizit die Multiplizität $*$.

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?

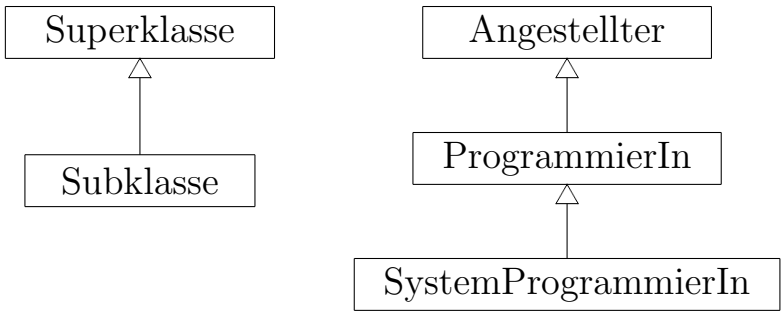
Ferner *ist* ein Programmierer ein Angestellter (ist Untertyp von Angestellter). Eigenschaften und Beziehungen werden von den Untertypen geerbt. Alles, was für die **Superklasse, (Basisklasse, Eltertyp, Obertyp, Obermenge,** ***superclass, baseclass, parent type, supertype, superset)*** gilt, gilt auch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

Die wichtigsten Tests, ob wirklich eine Erweiterung vorliegt sind:

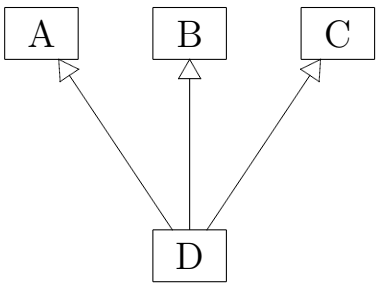
- Ist-ein-Beziehung
- Substitutionsprinzip von Liskov
- Teilmengen-Beziehung

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.

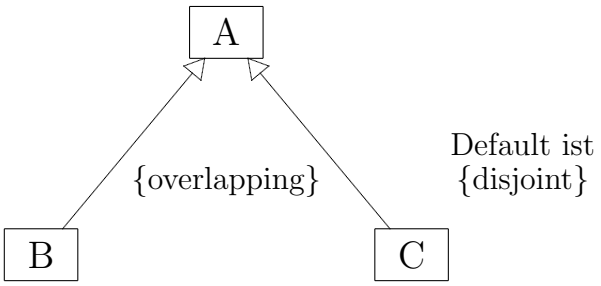
Die Erweiterung wird durch einen Pfeil mit dreieckiger Pfeilspitze dargestellt.




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

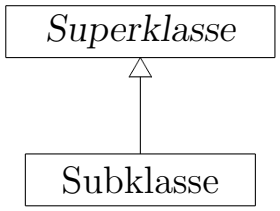


Betrachten wir Erweiterungen als Teilmengen, dann stellt sich die Frage, ob die Teilmengen **disjunkt** ***(disjoint)*** oder **überlappend** ***(overlapping)*** sind. Man spricht von der **Einschränkung durch Disjunktheit** ***(disjointness constraint)***. In der Darstellung ist *disjoint* Default.




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:

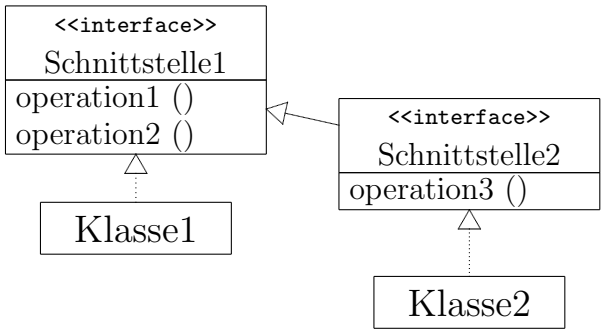
- Spezialisierung bzw Erweiterung: Wir beginnen mit einem Entitätstyp und definieren dann Untertypen durch sukzessive Spezialisierung bzw Erweiterung. Man spricht auch von einem ***top-down conceptual refinement***.
- Generalisierung: Der umgekehrte Weg beginnt bei sehr speziellen Untertypen, aus denen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

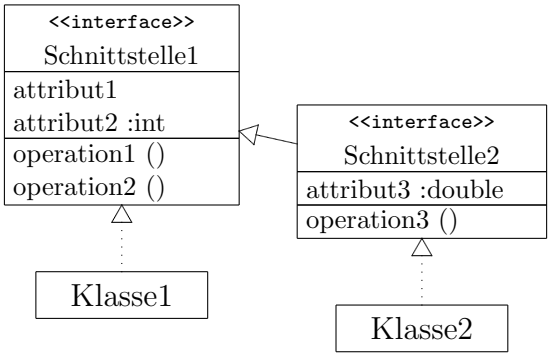
## 2.7 Realisierung

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:

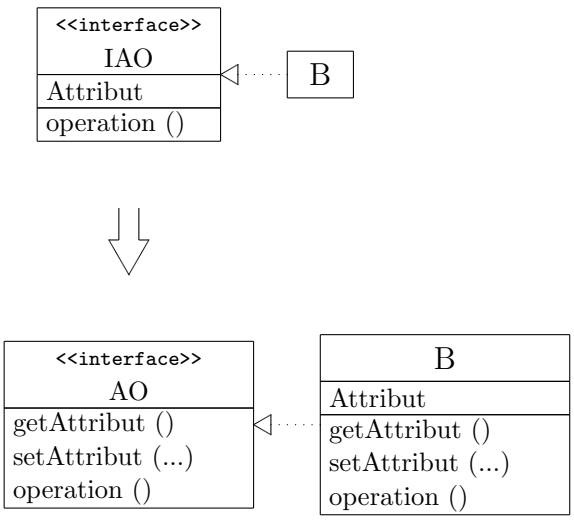



Das bedeutet, dass eine Schnittstelle auch aktiv Beziehungen zu anderen Schnittstellen und Klassen aufnehmen kann. Damit kann man dann den Klassenentwurf sehr abstrakt halten.

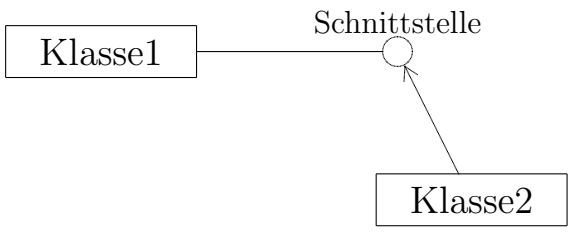
Eine Klasse kann mehr als eine Schnittstelle mit Attributen realisieren. Man hat dann natürlich eventuell Probleme der Mehrfachvererbung.

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.

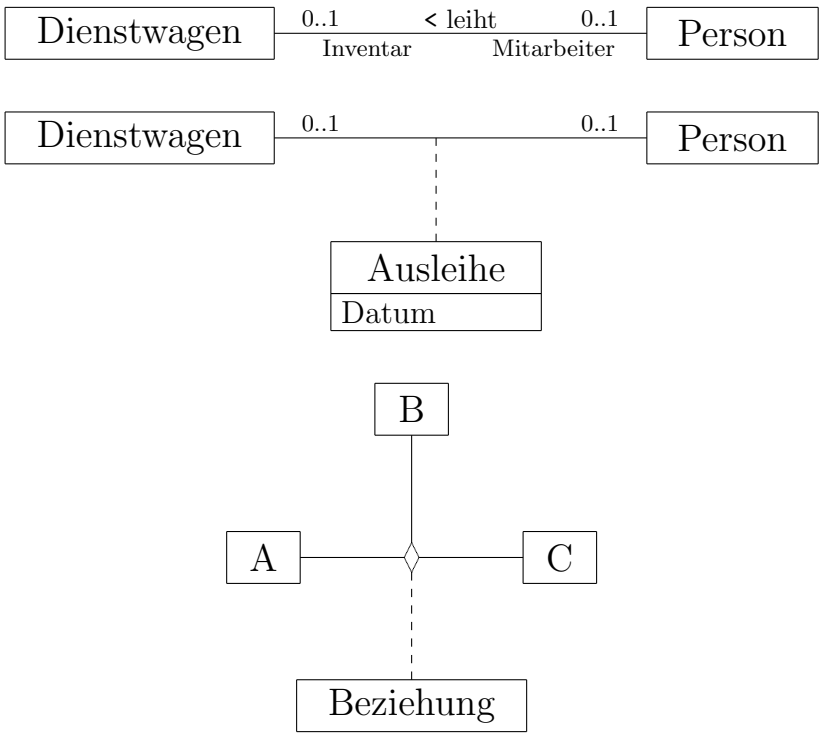
Von der Einführung von mehr als zweiwertigen Beziehungen ist abzuraten. Es sollte versucht werden, diese Beziehungen durch mehrere Zweier-Beziehungen darzustellen. Dadurch werden von vornherein Redundanzen vermieden. Eine $n$-wertige Beziehung kann immer durch eine Entität dargestellt werden, die binäre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

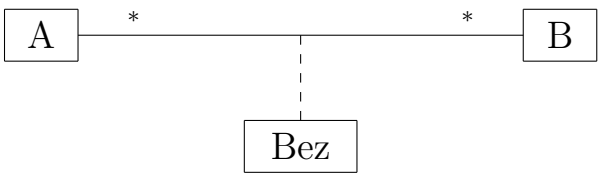
Beziehung steht – eventuell mit einer "Leserichtung" versehen – an der Linie oder in einem eigenen Rechteck, das wie eine Klasse mit Attributen und Operationen notiert werden kann (sogenannte **Assoziationsklasse**).

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.



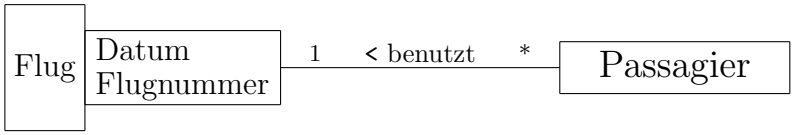

Da Assoziationen meistens als eigene Klassen implementiert werden, wird das oft schon durch eine **reifizierte** (verdinglichte, konkretisierte, ***reified***) Darstellung ausgedrückt. Anstatt von




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.

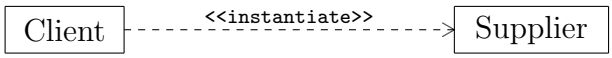
Bemerkung: Beziehungen zwischen Objekten sollten in UML-Diagrammen i.a. durch Assoziationen (also irgendwelche Linien zwischen Klassen) und nicht nur durch Attribute dargestellt werden, obwohl bei den meisten Programmiersprachen die Beziehungen zu Klassenelementen werden. Eine Assoziation macht aber 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.

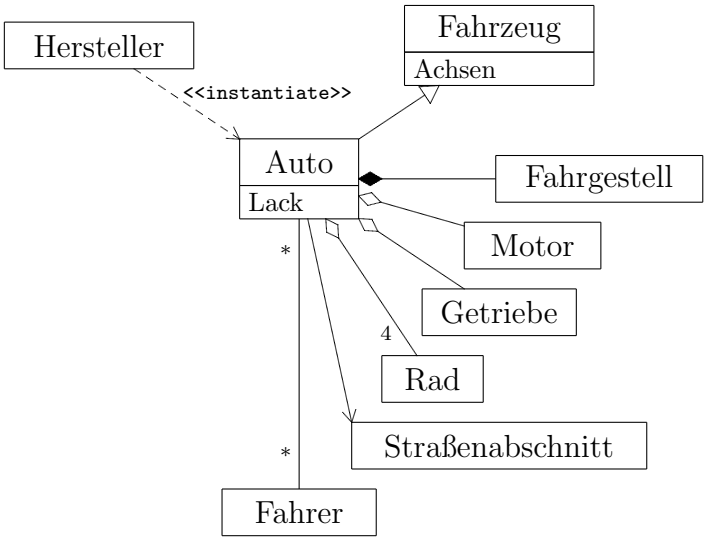
*send:* `<<send>>`
Beziehung zwischen Sender und Empfänger einer Nachricht.

*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.

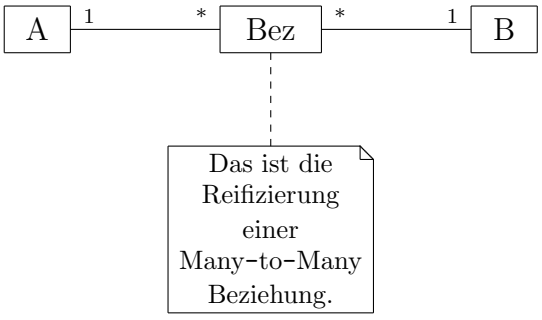
## 2.10 Zusammenfassung der Beziehungen

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




## 2.11 Notiz

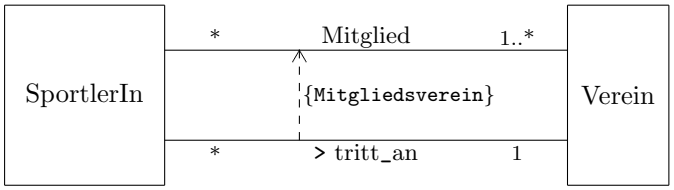
Mit dem Konstrukt der Notiz ***(note)*** können Kommentare, Bemerkungen oder andere textuelle Informationen im Diagramm untergebracht werden. Oft werden in einer Notiz Hinweise zur Implementierung einer Methode gegeben. Der Text wird in einem Rechteck mit Eselsohr dargestellt. Eine gestrichelte Linie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

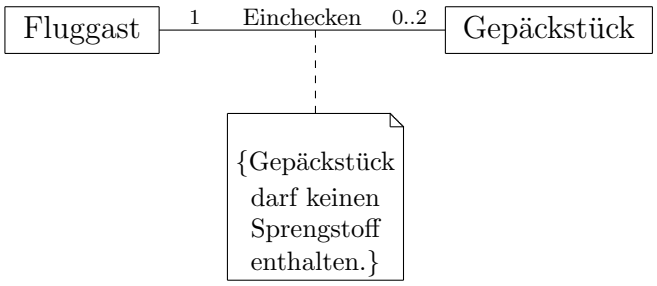



## 2.12 Einschränkung

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.



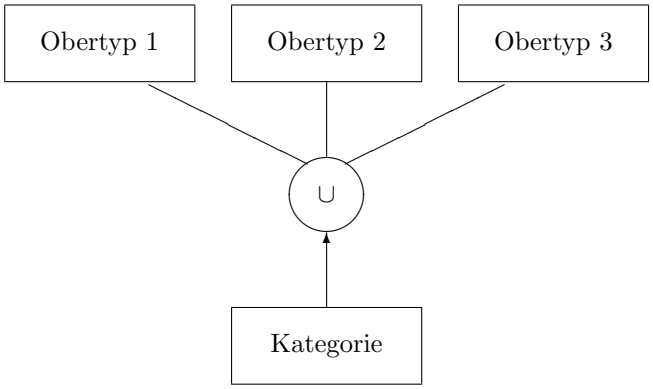

### 2.12.1 Kategorie *(category)*

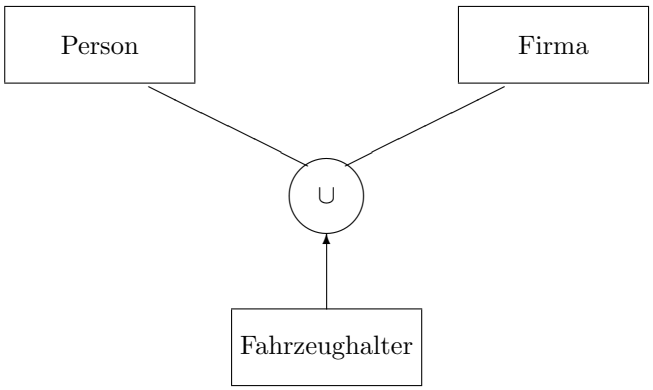
Ein Fahrzeughalter kann eine Person oder eine Firma sein. Da nicht jede Person und nicht jede Firma ein Fahrzeughalter ist, kann Fahrzeughalter kein Obertyp sein.

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.

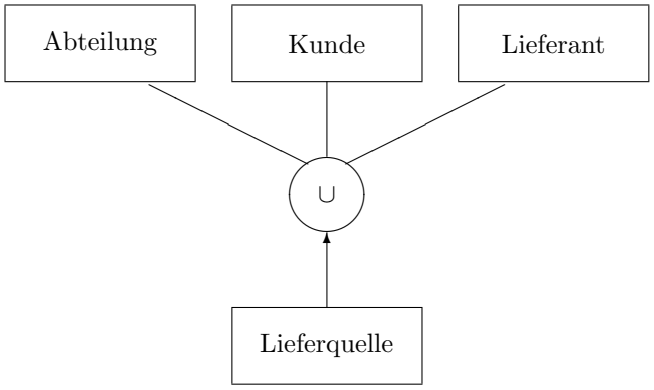



Eine Kategorie kann **partiell** oder **total** sein. Totalität bedeutet, dass jeder Obertyp der Kategorie zur Kategorie gehören muss. In diesem Fall ist eine Darstellung als Obertyp-Untertyp-Beziehung auch möglich, wobei die Kategorie der Obertyp ist, und ist meistens vorzuziehen, insbesondere wenn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

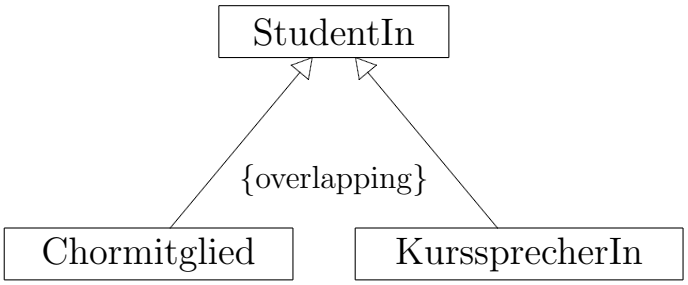
Kategorien kommen häufig vor. Sie sind eine Art von Rolle, die eine Entität spielen kann. Insbesondere Personen sind in den verschiedensten Rollen zu finden.

Beispiel:

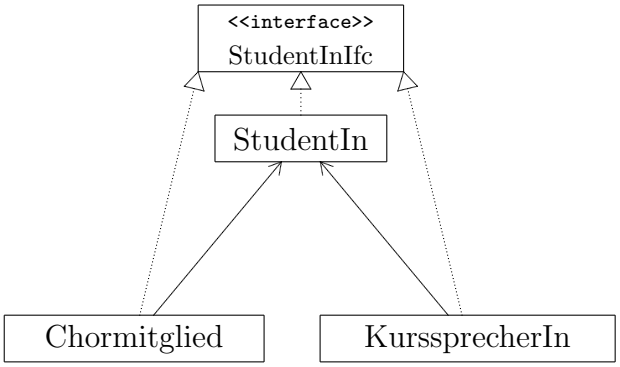



## 2.13 Beispiele

### 2.13.1 Nichtdisjunkte Untertypen



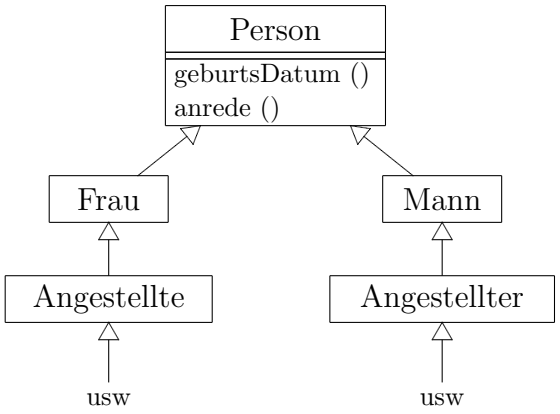

Wie wird das implementiert? Als zwei Kategorien (Chormitglied und Kursprecher):



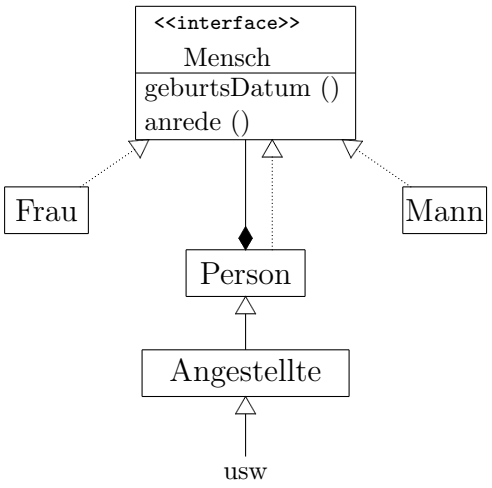

Sollte die Uses-Beziehung nicht besser zur Schnittstelle gehen? Im allgemeinen nein, denn es könnte ja sein, dass das Chormitglied oder der Kurssprecher zur Realisierung der Schnittstelle Methoden oder Datenelemente benötigt, die die Schnittstelle nicht anbietet.

### 2.13.2 Weiblich – Männlich

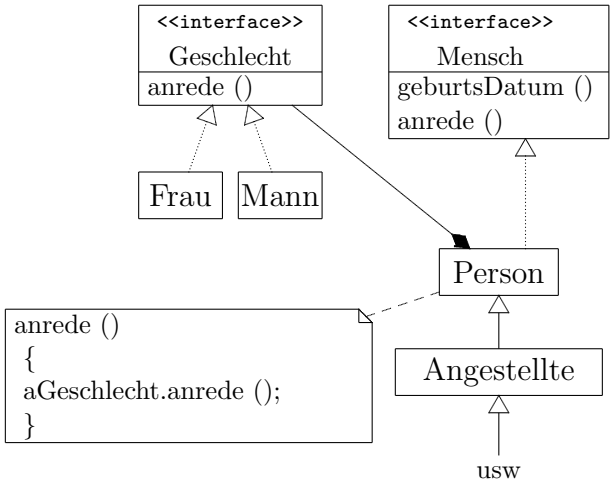
Wenn Frauen und Männer im wesentlichen gleich behandelt werden, dann ist folgende Struktur nicht günstig, weil das zu zwei redundanten Vererbungshierarchien führt:




Stattdessen sollte man eine "Person" als Kategorie einführen:



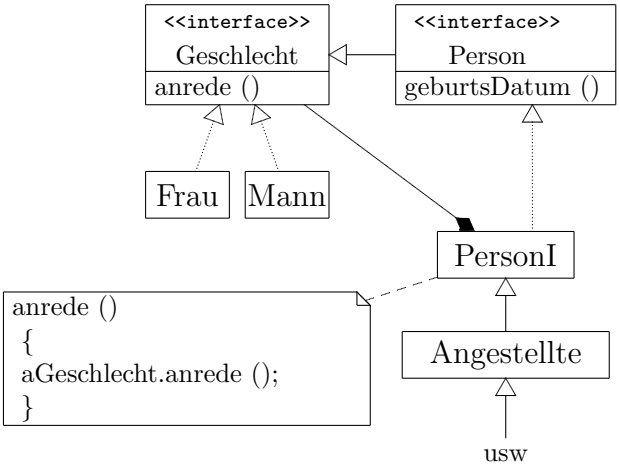

Um zu verhindern, dass sich `Person` wieder eine `Person` als Komponente nimmt, wäre folgende Struktur besser:




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:



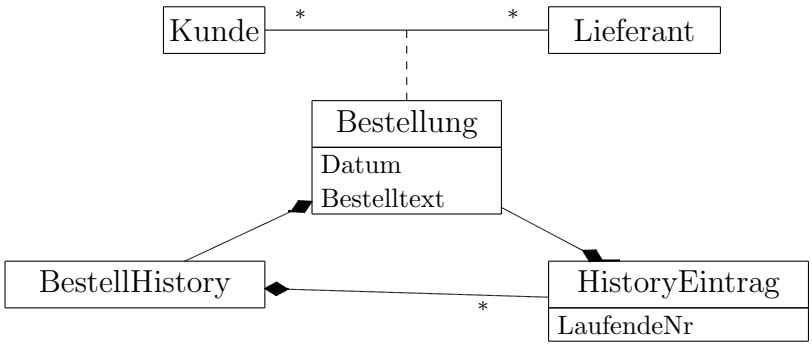

Allerdings kann eine `Person` wieder eine `Person` als Komponente erhalten, was wir aber in Kauf nehmen.

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."*

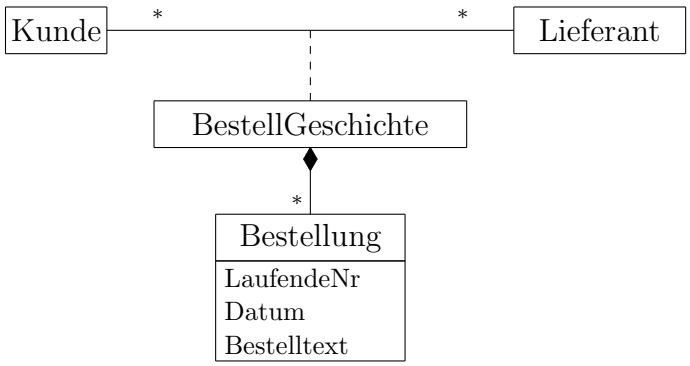
Übung: Diskutieren Sie die unterschiedlichen Optionen (Komposition, Aggregation, Benutzung) für die Beziehung zwischen `Person` und `Geschlecht`.

### 2.13.3 Many-to-Many-Beziehung mit History

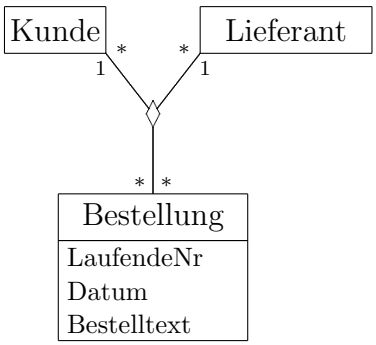
Viele Many-to-Many-Beziehungen zwischen zwei Partnern werden nicht nur einmal, sondern mehrmals eingegangen. Z.B. kann eine Person dasselbe Buch mehrmals leihen. Oder ein Kunde kann bei einer Firma mehrere Bestellungen machen. Um all diese Vorgänge zu verwalten, muss eine sogenannte ***History*** oder "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:

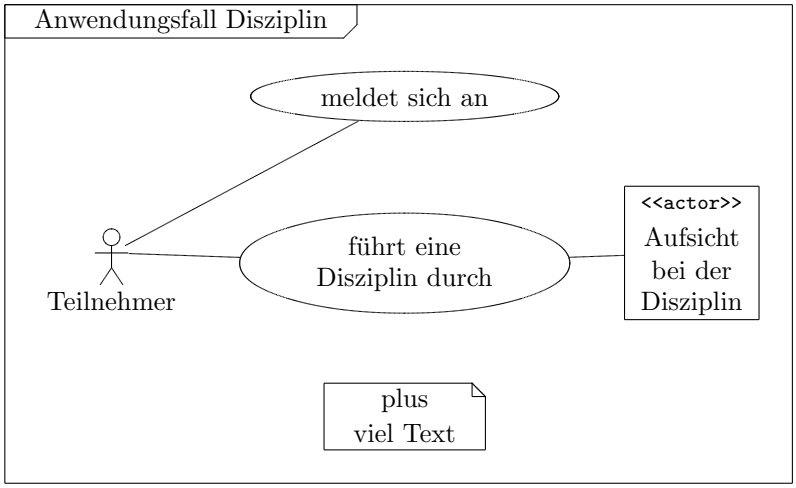

Kunde
Lieferant
1
*
*
1
*
*
Bestellung
LaufendeNr
Datum
Bestelltext

# Kapitel 3

# Anwendungsfälle

**Anwendungsfall-Diagramme** ***(use case diagram)*** oder **Szenarien (scenario)** zeigen, was ein System, Subsystem oder auch nur eine Klasse aus der Sicht des externen Beobachters oder Benutzers (Anwenders) tut.

Anwendungsfälle werden durch einen Text und ein Diagramm dokumentiert.

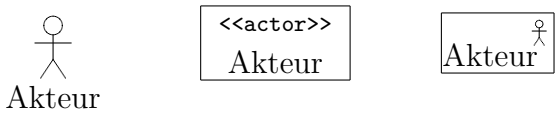



## 3.1 Notation Anwendungsfall-Diagramm

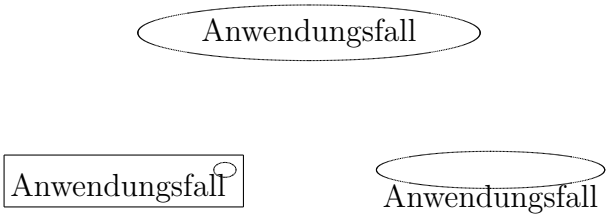
Elemente eines Anwendungsfall-Diagramms sind:

- **Akteur** ***(actor)***
  - Repräsentiert den Benutzer eines Systems, der irgenwie mit dem System wechselwirkt.

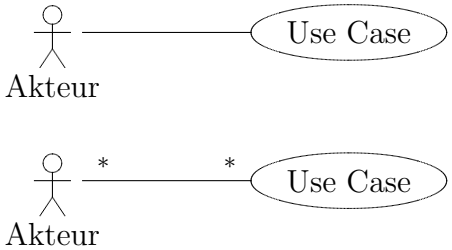
- Gehört *nicht* zum System.
- Kann menschlich oder ein anderes System oder ein Zeitereignis sein. Der Akteur ist ein idealisierter Benutzer. (Ein realer Benutzer kann sich auf mehr als einen Akteur beziehen. Umgekehrt kann sich ein Akteur auf mehrere reale Benutzer beziehen.)
- Externe Abläufe zwischen den Akteuren ohne Beteiligung des Systems werden nicht modelliert.
- Notation: Wird durch ein Strichmännchen oder ein Rechteck mit Stereotyp `<<actor>>` dargestellt oder ein Rechteck mit kleinem Strichmännchen.




- **Anwendungsfall** ***(use case)***
  - Beschreibt eine Aktivität, Funktionalität oder einen Dienst, die bzw. den das System den Akteuren zur Verfügung stellt.
  - Beschreibt eine in sich abgeschlossene Aufgabe, die das System für einen oder mehrere Akteure durchführt. Die Aufgabe sollte innerhalb des zeitlichen Rahmens einer "Sitzung" abgeschlossen sein. (Im Unterschied dazu besteht ein **Geschäftsprozess** i.a. aus mehreren Anwendungsfällen, die zeitlich eventuell weit auseinanderliegen.)
  - Notation: Wird durch eine Ellipse dargestellt, wobei der Name und ein Satz in oder unter der Ellipse steht. Ein Rechteck mit eingesetzter Ellipse ist auch möglich.



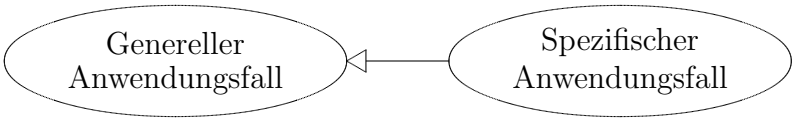

- **Beziehung** ***(relationship)***
  Bei Anwendungsfällen unterscheidet man vier Arten der Beziehung.
  - **Assoziation** ***(association)***
    Beschreibt den Kommunikationspfad zwischen einem Akteur und einem Anwendungsfall. Sie wird durch eine durchgezogene Linie dargestellt und kann auch Multiplizitäten haben, wobei die Voreinstellung auf Akteurseite 1 ist.



Akteur und Anwendungsfall tauschen Botschaften aus.

Abweichend vom Standard erlauben wir uns, einen Pfeil für die Kommunikationsrichtung anzugeben, wenn die Kommunikation sicherlich nur in einer Richtung erfolgt.

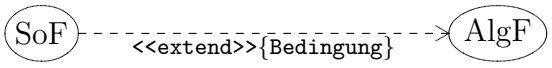
– **Generalisierung / Spezialisierung** ***(generalization / specialization)***
Hier wird eine Beziehung zwischen einem allgemeinen und einem speziellen Anwendungsfall beschrieben, der die Eigenschaften und das Verhalten des allgemeinen Falls erbt und neue Eigenschaften bzw. Verhalten hinzufügt. Sie wird mit einer durchgezogenen Linie mit Dreieckspfeil dargestellt.




Das Verhalten des allgemeinen Falls kann durch den speziellen Fall auch verändert werden (Überschreiben von Verhalten).

Bei Akteuren ist solch eine Beziehung auch möglich:


– **Erweiterung** ***(extension)***
Beschreibt eine Beziehung zwischen einem Basis-Anwendungsfall ***(base use-case)*** und einem diesen **semantisch** erweiternden Anwendungsfall ***(client use-case)***. Diese Beziehung wird durch eine gestrichelte Linie mit Pfeil zum Basis-Anwendungsfall dargestellt.

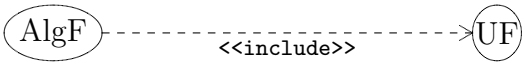



Der erweiternde Anwendungsfall wird niemals "instanziiert". D.h. es gibt keine Beziehung zu einem Akteur. Er kann nicht isoliert vom Basis-Anwendungsfall durchgeführt werden. Man kann ihn als eine Art von "Sonderfall" oder Option ansehen.

Die Optionalität kann dadurch verdeutlicht werden, dass der Pfeil mit einer geklammerten Bedingung versehen wird. Falls die angegebene Bedingung wahr ist, wird die Aktivität des Basis-Anwendungsfalls an einem sogenannten **Erweiterungspunkt** ***(extension point)*** unterbrochen und die Aktivität des Clients eingefügt. Der Basisfall hat aber keine Kenntnis, was da konkret eingefügt wird. Ohne Angabe einer Bedingung ist die Semantik so, dass die Erweiterung immer durchgeführt wird.

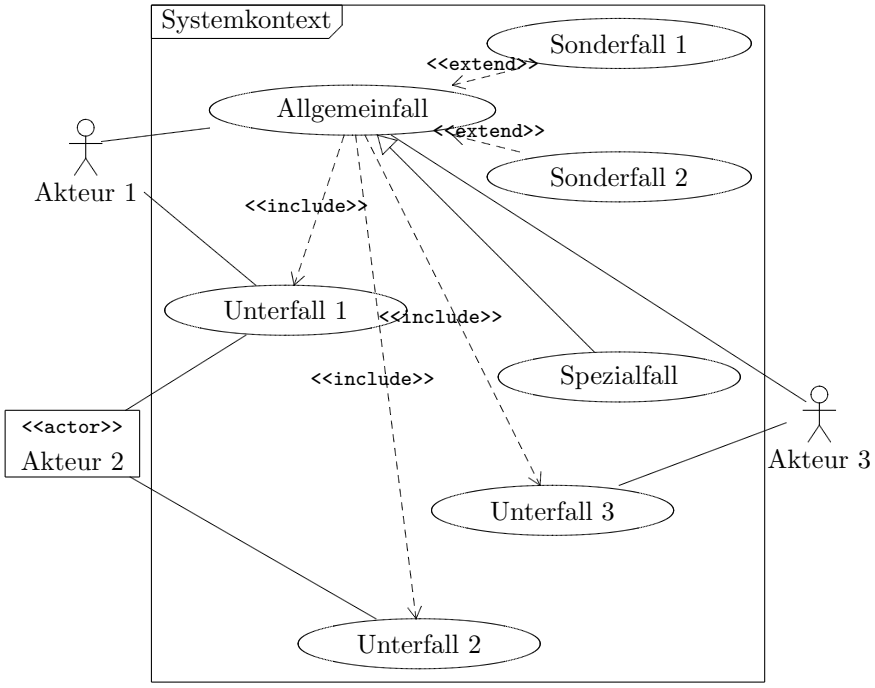
Der erweiternde Anwendungsfall ist für den zu erweiternden Anwendungsfall nicht zwingend erforderlich.

– **Inkludierung** ***(inclusion)***
Damit werden zusätzliche Eigenschaften oder Verhalten in einen Anwendungsfall ***(client use-case)*** eingefügt. Die Inkludierung wird durch eine gestrichelte Linie mit Pfeil vom Client zum inkludierten Anwendungsfall dargestellt.



Der inkludierte Anwendungsfall kann selbst instanziiert werden und hat keine Kenntnis vom Client. Der Client dagegen weiß, wen er da inkludiert. Der inkludierte Fall wird immer durchgeführt.
Der **inkludierende** Anwendungsfall erfordert zwingend den **inkludierten** Anwendungsfall.

- **Systemgrenze** ***(system boundary)***
  - Optional können Akteure und Anwendungsfälle durch eine rechteckige Systemgrenze getrennt werden, wobei eine Bezeichnung für den **Systemkontext** angegeben werden kann.




Erweiterung und Inkludierung von Anwendungsfällen verleitet oft zu einer funktionalen Zerlegung des Anwendungsfalls. Das ist aber nicht der Sinn dieser Technik. Daher sollte man diese Mechanismen nur verwenden, wenn sie sich aufdrängen. Für die funktionale Zerlegung sind Aktivitätsdiagramme wesentlich besser geeignet.

## 3.2 Beschreibung des Anwendungsfalls

Die Beschreibung eines Anwendungsfalls kann eine beliebige Form haben. Sie kann eine stichwortartige Skizze oder ein ausformulierter Text sein. Sie kann dabei auch andere Diagramm-Arten der UML verwenden (Zustandsgraph, Aktivitätsdiagramm, Sequenzdiagramm).

Es gibt zwar **keine kanonische** Liste von Punkten, die die Beschreibung eines Anwendungsfalls enthalten sollte. Aber folgende Punkte sind ein vernünftiger Vorschlag:

- **Name** des Anwendungsfalls, Informationen zur Verwaltung des Anwendungsfalls (Ansprechpartner)
- **Kurzbeschreibung** ***(short characterization)*** des Anwendungsfalls. Die Kurzbeschreibung ist oft Inhalt der Ellipse des Diagramms.
- **Ablaufbeschreibung** ***(basic course of events)***: Hier ist der eigentliche Anwendungsfall zu finden **(Primäre Szenarien)**. Wichtig ist, dass dieser Text auch vom Kunden verstanden wird. Es ist sogar sinnvoll, wenn dieser Text vom Kunden erstellt wird. Die Verwendung von Pseudo-Code ist daher wahrscheinlich nicht geeignet.

  Die Ablaufbeschreibung sollte nicht zu allgemein sein. Sie sollte so ausführlich sein, dass man eine gute Grundlage zur Entwicklung des Anwendungsfalls und zum Testen des Anwendungsfalls hat. Allerdings sollte sie auch keine "Klickanleitung" sein. Wichtige Entscheidungen der Akteure sollten repräsentiert sein.
- **Akteure** ***(actors)***:
  - **Primäre Akteure** ***(primary actors)***: Welche Akteure stoßen den Anwendungsfall an? Wie kommt es zur Ausführung des Anwendungsfalls und welcher Akteur wird dann aktiv?
  - **Sekundäre Akteure** ***(secondary actors)***: Liste aller anderen Akteure
- **Vorbedingungen** ***(preconditions)***: Beschreibung des Zustands vor der Durchführung des Anwendungsfalls
- **Nachbedingungen** ***(postconditions)***: Beschreibung des Zustands nach der Durchführung des Anwendungsfalls
- **Invarianten** ***(invariants)***: Beschreibung der Bedingungen oder Größen, die während des Anwendungsfalls erfüllt bzw. konstant sein müssen.
- **Regeln** ***(rules)***: Beschreibung von Geschäftsregeln, die eingehalten werden müssen.
- **Nicht-funktionale Anforderungen** ***(non-functional requirements)***: Gemeint sind zusätzliche Anforderungen, die durch die vier vorangehenden Punkte nicht abgedeckt werden (z.B. Performanz-Anforderungen).
- **Erweiterungspunkte** ***(extension points)***: Hier wird der Ablauf von Erweiterungen beschrieben. Diese können allerdings auch in eigenen Anwendungsfalldokumenten erscheinen.
- **Ausnahmen, Fehlersituationen** ***(exceptions, error situations)***: Der Schwerpunkt liegt nicht auf technischen Fehlern, sondern auf den **fachlichen** Fehler, die bei der Durchführung des Anwendungsfalls auftreten können. Insbesondere sind das Fehler, die auf Fehlbedienungen eines Akteurs zurückzuführen sind.

  Jeder fachliche Ausnahmefall sollte mit einem alternativen Ablauf beschrieben werden.
- **Variationen** ***(variations)***: Abläufe, die vom Normalverhalten abweichen **(Sekundäre Szenarien,** Alternativen). Das könnten dann eventuell neue, abgeleitete Anwendungsfälle sein.
- **Dienste** ***(services)***: Liste von Operationen oder Objekten, die zur Durchführung des Anwendungsfalls benötigt werden.
- **Anmerkungen** ***(remarks)***: Hier ist Platz für zusätzliche Informationen, die nicht in die oben genannten Punkte passen.

## 3.3 Identifikation von Anwendungsfällen

### 3.3.1 Identifikation der Akteure

- Wer verwendet das System?
- Wer bekommt Informationen aus dem System?
- Wer liefert Informationen an das System?
- In welchem Umfeld wird das System verwendet?
- Wer wartet das System?
- Welches andere System benutzt das System?

Ferner ist es nützlich, eine **Ereignisliste** zu erstellen, die alle Ereignisse der realen Welt enthält, die das System betreffen. Wenn ein Akteur etwas mit dem System tut, dann ist das auch ein Ereignis.

Ereignisse führen zu Anwendungsfällen und Akteuren.

### 3.3.2 Identifikation der Ziele der Akteure

Jedes Ziel kann unmittelbar in einem Anwendungsfall resultieren. Ein Ziel definiert den Titel eines Anwendungsfalls. Ein Ziel beantwortet die Frage: Was soll das System für einen Akteur tun?

## 3.4 Bedeutung der Anwendungsfälle

Anwendungsfälle gelten als die Basis der kompletten Software-Entwicklung. Sie dienen der Verständigung mit dem Kunden und bilden daher auch eine ausgezeichnete Grundlage für die Benutzerdokumentation. Schließlich wird die Software an Hand der Anwendungsfälle getestet. Der Abnahmeprozess des Kunden orientiert sich sehr wahrscheinlich an den Anwendungsfällen. Die Beschreibung der benötigten Dienste unterstützt die Entwicklung des Klassenmodells.

## 3.5 Ratschläge

- Anwendungsfälle können ineinander geschachtelt werden. Anwendungsfälle auf einer hohen Ebene *(coarse granularity)* können mehrere Detail-Anwendungsfälle enthalten. Mehr als zwei Stufen sollte man allerdings nicht verwenden.
- Man sollte Anwendungsfallentwicklung mit Problembereichsentwicklung iterieren.
- Der riskanteste Anwendungsfall sollte höchste Priorität haben.
- Die Namen von Akteuren sollten so spezifisch wie möglich sein.

- Man sollte zwischen **primären** ***(primary)*** Akteuren – Akteuren, die den Anwendungsfall anstoßen – und **sekundären** ***(secondary)*** Akteuren unterscheiden. Im Diagramm erscheinen die primären Akteure vorzugsweise links.
- Die Kurzbeschreibung eines Anwendungsfalls (Tätigkeitsausdruck, bis fünf Worte) sollte das Interesse des primären Akteurs am Anwendungsfall wiederspiegeln. Die Kurzbeschreibung sollte zeigen, was das System für den Akteur tut, nicht, was der Akteur tut.
- Ablaufbeschreibung: Es sollte die Wechselwirkung zwischen Akteur und System beschrieben werden, nicht das interne Verhalten des Systems.
- Für jeden Anwendungsfall sollten viele Szenarien beschrieben werden, insbesondere Szenarien für die erfolgreiche *und* die nicht erfolgreiche Durchführung des Anwendungsfalls.
- Test: Kann der Kunde den Anwendungsfall verstehen? Ein Benutzer sollte bei der Anwendungsfallentwicklung mitwirken.
- Offensichtliche Anwendungsfälle müssen nicht ausführlich beschrieben werden.
- Wieviele Anwendungsfälle hat ein Projekt?
- Anwendungsfälle sollten **sehr einfach** gehalten werden. Wenn es komplex wird, dann sollte man in mehrere Anwendungsfälle aufspalten. Es gibt die Tendenz, einen ganzen Geschäftsprozess als Anwendungsfall zu modellieren. Das wird unübersichtlich. Ein Geschäftsprozess besteht i.a. aus mehreren Anwendungsfällen.

# Kapitel 4

# Interaktions-Diagramme

Das dynamische Verhalten eines Systems kann auf zwei komplementäre Arten beschrieben werden: Fokusierung auf das Verhalten einzelner Objekte (Kollektion von Zustandsgraphen) oder Fokusierung auf die Interaktion zwischen Objekten (Kollektion von Interaktions-Diagrammen). Ein Zustandsgraph gibt einen tiefen, aber sehr engen Einblick in das Verhalten eines individuellen Objekts. Er ist eine genaue Spezifikation, die unmittelbar zu Code führt. Allerdings ist es schwer die Funktionsweise des Gesamtsystems zu sehen, da man viele relativ isolierte Zustandsgraphen kombinieren muss. Die Interaktions-Diagramme vermitteln nun eine eher ganzheitliche Sicht auf das System. Sie zeigen, wie die Objekte miteinander kollaborieren.

Eine Interaktion ist eine Menge von **Botschaften *(message)***, die innerhalb einer Kollaboration von Objekten ausgetauscht werden.

Eine Botschaft ist eine Kommunikation zwischen zwei Objekten in *einer* ***(one-way)*** Richtung. Sie kann Parameter haben, die als Inhalt der Botschaft geschickt werden. Eine Botschaft kann ein **Signal *(signal)*** sein (asynchrone Interobjekt-Kommunikation) oder sie kann ein **Aufruf *(call)*** sein (synchroner Aufruf einer Operation, wobei der Sender die Kontrolle abgibt und sie später wieder zurückbekommt).

## 4.1 Sequenz-Diagramm

Ein Sequenz-Diagramm zeigt die Interaktion zwischen Objekten oder (seit UML 2) Rollen von Objekten in einem zweidimensionalen Graphen. Die vertikale Dimension ist eine Zeitachse. Die Zeit nimmt nach unten hin zu. Die Zeitabstände sind willkürlich; es kommt nur auf die **zeitliche Ordnung** an.

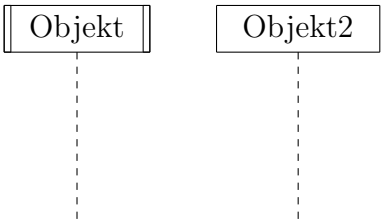
Objekte können durch ein Rechteck mit verdoppeltem rechten und linken Rand als potentiell **aktive Objekte** oder **Tasks (Prozesse, Threads)** dargestellt werden:



Da es sich bei den Objekten eventuell um komplizierte Rollen von Objekten handelt, wird der Name seit UML 2 nicht mehr unterstrichen.

Für jedes an der Interaktion beteiligte Objekt wird eine Zeitachse **(Lebenslinie, *lifeline)*** angelegt.

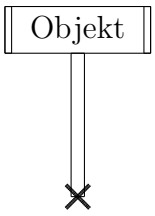
- Eine gestrichelte Lebenslinie zeigt an, dass das Objekt nicht aktiv (suspendiert) ist.




  Die Lebenslinien von Objekten, die nie aktiv werden, d.h. keine Tasks sind, werden ebenfalls gestrichelt dargestellt.

- Eine Doppellinie bedeutet, dass das Objekt aktiv ist.


- Wenn die Task eines aktiven Objekts an ihr natürliches Ende läuft, wird das durch ein `X` markiert.

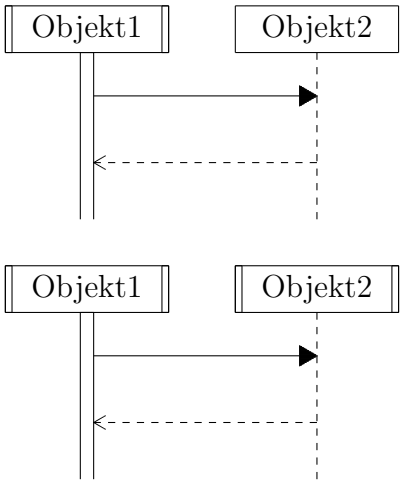



**Botschaften:**

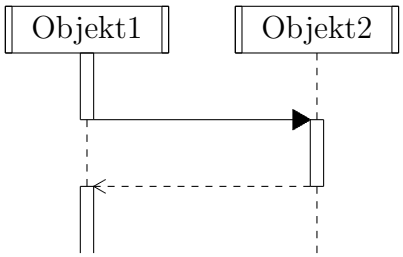
Objekte senden sich gegenseitig Botschaften. Eine Botschaft wird als ein Pfeil von einer Lebenslinie zu einer anderen notiert. Wir unterscheiden **synchrone** und **asynchrone** Botschaften.

**Synchrone Botschaft:** Bei synchronen Botschaften wird auf eine Antwort gewartet. Wir unterscheiden zwei Fälle:

- **Funktions- oder Methodenaufruf:** Die Botschaft wird im Task-Kontext des sendenden (aufrufenden) Objekts durchgeführt. Die Botschaft kann an passive oder aktive Objekte gehen. Die Notation ist:

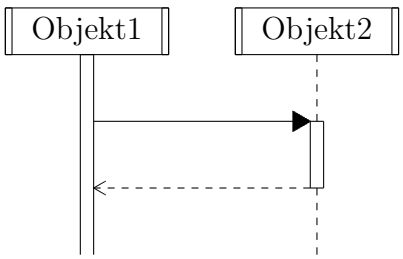



- **Eigentliche synchrone Botschaft:** Die Botschaft wird im Task-Kontext des empfangenden Objekts behandelt. Das sendende Objekt ist während dieser Zeit passiv. Das empfangende Objekt wird dabei aktiv. Die Notation ist:




Wir sprechen in diesem Fall auch von **Aktivierung** ***(activation)***.

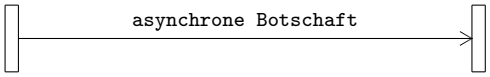
- **Sender und Empfänger aktiv:** Die Botschaft wird im Task-Kontext des empfangenden Objekts behandelt. Das sendende Objekt ist während dieser Zeit allerdings nicht suspendiert. Die Notation ist:



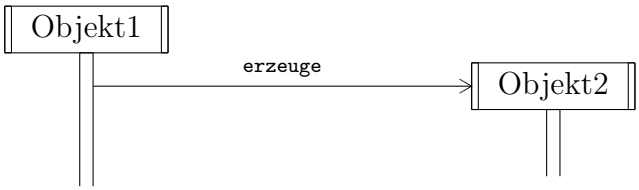
**Asynchrone Botschaft:** Bei asynchronen Botschaften wird nicht auf eine Antwort gewartet. Asynchrone Botschaften werden immer im Task-Kontext des empfangenden Objekts behandelt.

Ein durchgezogener Pfeil mit

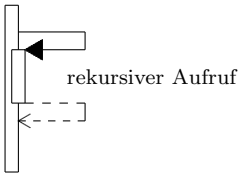
**normaler** (d.h. nicht voller) Pfeilspitze bedeutet **asynchrone** Botschaft an ein Objekt. Eine asynchrone Botschaft kann nur durch eine weitere asynchrone Botschaft beantwortet werden. Asynchrone Botschaften bedeuten eigentlich immer, dass parallele Tasks (aktive Objekte) laufen.



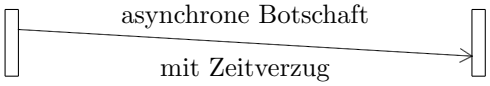

**Objekt-Erzeugung:** Ein durchgezogener Pfeil mit **normaler** Pfeilspitze wird bei Erzeugung eines Objekts verwendet. Wenn das erzeugte Objekt aktiv ist, dann sprechen wir auch von **Aktivierung** ***(activation)***.




**Rekursive synchrone Aufrufe** (Selbstdelegation) sind möglich.



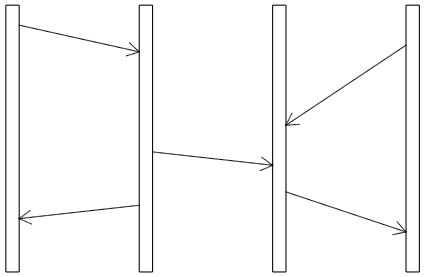

**Verzögerung:** Normalerweise sind die Pfeile waagerecht. Man kann aber die Dauer der Botschafts-Übermittlung (etwa wegen einer Netzverbindung) durch mehr oder weniger schräge Pfeile andeuten.




**Einschränkungen** (Zusicherungen oder Bedingungen) können an passenden Stellen in geschweiften Klammern angebracht werden.

{Einschränkung}

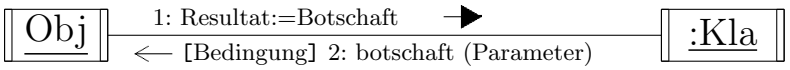
**Zeitliche Ordnung:** Sequenz-Diagramme sind nur **partiell geordnet** ***(partially ordered)***.



## 4.2 Kollaborations-Diagramm

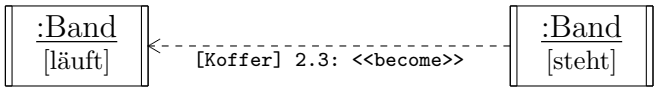
Ein Kollaborations-Diagramm ***(collaboration diagram)*** beschreibt eine Konfiguration von Objekten und ihren Verbindungen, wenn eine Interaktion ausgeführt wird. Nur die Objekte, die an der Interaktion teilnehmen, werden gezeigt.

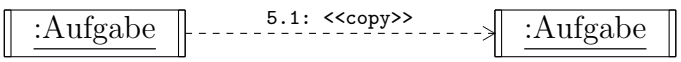
Über die Verbindungen zwischen Objekten werden Botschaften geschickt:




- Pfeile geben die Richtung und die Art (synchron, asynchron) der Botschaft an.
- Sequenz-Nummern (ganzzahlig oder mit Dezimalpunkt) geben die Reihenfolge der Botschaften an. Die Nummer kann auch mit einem Thread-Identifikator versehen werden. Alle Botschaften, die zu derselben Thread gehören, sind sequentiell. Botschaften in verschiedenen Threads sind nebenläufig ***(concurrent)***.
- Bedingungen können in eckigen Klammern angegeben werden.
- Lokale Variable können verwendet werden.
- Die Objekte können Einschränkungen tragen:
  - {`new`}: Wird während der Interaktion erzeugt.
  - {`destroyed`}: Wird während der Interaktion aufgegeben.
  - {`transient`}: Wird während der Interaktion erzeugt und wieder aufgegeben.

Objekte können während einer Interaktion ihren Zustand wesentlich ändern, d.h. neue Verbindungen eingehen. Das kann dargestellt werden durch Angabe des Zustands in eckigen Klammern und ***become flow***-Pfeile oder ***copy flow***-Pfeile.



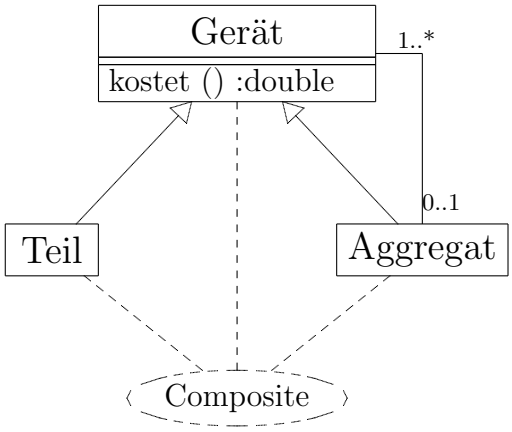
Nach einem *copy* sind die beiden Kopien unabhängig.

Mit einem Kollaborations-Diagramm kann man die Realisierung einer ganzen Klasse zeigen, indem für jede Methode ein Kollaborations-Diagramm erstellt wird. All diese Diagramme können eventuell in *einem* Diagramm dargestellt werden.

Kollaborations-Diagramme eignen sich für das detailierte Design von Prozeduren.

## 4.3 Entwurfsmuster

Ein Entwurfsmuster ***(design pattern)*** ist eine parametrisierte Kollaboration. Ein Pattern wird durch eine gestrichelte Ellipse dargestellt – hier das Pattern ***Composite*** – und wird meistens innerhalb eines Klassendiagramms verwendet, wobei die beteiligten Entitäten durch eine gestrichelte Linie mit dem Pattern verbunden werden. Die Linie kann mit einer Rollenbezeichnung versehen werden.




Die Entwurfsmuster-Notation kann in jeder Art von Diagramm verwendet werden.

# Kapitel 5

# Zustandsgraphen

Um die oft schwer verständlichen zeitlichen Abfolgen zu verstehen, entwickeln wir das **dynamisches Modell** ***(dynamic model)*** als **Zustandsgraphen (*statechart diagram,* Zustandsmaschine, Automat, Zustands-Übergangs-Automat, Zustands-Diagramm, ASM-Diagramm, *Algorithmic State Machine Diagram, state machine)***.

Für die Analyse und den Entwurf von kleinen Echtzeitsystemen oder Automaten haben sich Zustandsgraphen seit langem bewährt.

Ein Zustandsgraph beschreibt das dynamische Verhalten eines Objekts, d.h. sein zeitliches Verhalten. Das Objekt erkennt **Ereignisse** ***(event)***, die ihm von anderen Objekten geschickt werden, und reagiert darauf, indem es seinen **Zustand** ***(state)*** ändert oder auch beibehält.

Prinzipiell ist für jedes Objekt ein eigener Zustandsgraph zu erstellen. Die Menge der Zustandsgraphen bildet das dynamische Modell.

Ereignisse beschreiben Zeitpunkte, Zustände beschreiben Zeitintervalle zwischen zwei Ereignissen. Ein Zustand hat eine Dauer und beschreibt oft eine kontinuierliche Aktivität des Objekts. Ereignisse und Zustände sind *dual* zueinander.

## 5.1 Zustände

Ein Zustand ist eine Abstraktion der aktuellen Werte und Beziehungen eines Objekts. Die aktuellen Werte der Attribute eines Objekts bestimmen den Zustand eines Objekts, allerdings möglicherweise nicht vollständig oder nicht eindeutig, da der Zustand von der Geschichte des Objekts abhängen kann. (Die Geschichte ***(history)*** eines Objekts wird oft nicht durch seine Datenelemente verwaltet.) Ein Zustand kann entweder durch einen Wertesatz oder durch eine Menge von Wertesätzen oder Wertebereiche charakterisiert werden. Ein Zustand spezifiziert qualitativ die Antwort des Objekts auf ein ihm gesendetes Ereignis. Die Antwort kann quantitativ vom Wertesatz abhängen.

Ein Zustand ist ein Objekt einer bestimmten Klasse.

Die Antwort des Objekts kann eine Aktion, Aktivität und/oder eine Zustandsänderung sein.

Bei der Definition von Zuständen eines Objekts werden – der Einfachheit halber – Attribute ignoriert, die insofern keinen Einfluss auf das Verhalten des Objekts haben, als sie die Zustandsfolge nicht qualitativ beeinflussen.

Die **Granularität** von Zuständen und Ereignissen hängt ab vom Abstraktionsniveau. Z.B. kann auf einem hohen Abstraktionsniveau ein Flug von Stuttgart nach München als ein Zustand mit zwei Ereignissen (Start und Landung) charakterisiert werden, obwohl der Flug verschiedenste Zustände (Starten, Steigflug, Flug auf verschiedenen geographischen Abschnitten, Sinkflug, Landen) durchläuft.

Ein Zustand führt **Aktivitäten** ***(activity)*** und **Aktionen** ***(action)*** durch. Aktivitäten haben eine Dauer und können eventuell unterbrochen werden. Aktionen haben – für die Zwecke der Modellierung – keine Dauer und können auch nicht unterbrochen werden.

Zur Beschreibung eines Zustands gehören folgende Informationen:

**Name** des Zustands. Sollte mindestens innerhalb eines Zustandsgraphen eindeutig und aussagekräftig gewählt werden.

**Beschreibung** oder Charakterisierung des Zustands.

**Definition von drei Arten** von Aktionen und Aktivitäten:

***entry:*** Beschreibung von Aktionen oder Aktivitäten, die bei Eintritt in den Zustand auf jeden Fall durchgeführt werden. Diese Phase wird nicht durch empfangene Ereignisse unterbrochen. Es werden alle Ereignisse aufgeführt, deren Information im Zustand verarbeitet wird. Ereignisse, die zwar in den Zustand führen, aber deren Informationsgehalt für den Zustand bedeutungslos ist, sollten nicht berücksichtigt werden. Dadurch wird der Zustand und seine Beschreibung unabhängiger von den Ereignissen oder anderen Zuständen.

Im Diagramm werden diese Aktionen/Aktivitäten durch das Wort ***entry/*** gekennzeichnet.

***do:*** Beschreibung der im Zustand durchgeführten einmaligen oder kontinuierlichen Tätigkeiten (Aktivitäten und auch Aktionen). Diese Tätigkeiten können typischerweise durch Ereignisse unterbrochen werden. Eine wichtige Tätigkeit ist die Erfassung von für den Zustand bestimmten Ereignissen ***(event monitor)***. Alle empfangbaren Ereignisse müssen hier aufgelistet werden. Ereignisse, die der Zustand an andere Objekte sendet, werden typischerweise hier beschrieben.

Im Diagramm werden diese Aktionen/Aktivitäten durch das Wort ***do/*** gekennzeichnet.

***exit:*** Hier werden Tätigkeiten beschrieben, die vor Verlassen des Zustands auf jeden Fall zu erledigen sind. Hier können auch empfangene Ereignisse verarbeitet werden.

Im Diagramm werden diese Aktionen/Aktivitäten durch das Wort ***exit/*** gekennzeichnet.

***default:*** Oft werden diese drei Fälle nicht unterschieden. Dann wird das als **do** interpretiert. Die Aktionen werden sofort bei Betreten des Zustands ausgeführt. Die Aktivitäten sind durch Ereignisse unter- und abbrechbar.

## 5.2 Ereignisse

Ein Ereignis passiert zu einer bestimmten Zeit. Seine Dauer ist vernachlässigbar bezüglich der betrachteten Zeitskala.

Ein Ereignis kann logisch vor oder nach einem anderen Ereignis stattfinden. Zwischen solchen Ereignissen besteht ein Kausalzusammenhang. Bei Ereignissen, die *nicht* logisch oder kausal voneinander abhängen, oder die zeitlich ohne Beziehung sind, heißen **nebenläufige (parallele, *concurrent)*** Ereignisse. Insbesondere, wenn die Signallaufzeit zwischen den Orten zweier Ereignisse größer ist als die Zeitdifferenz zwischen den Ereignissen, dann müssen die Ereignisse unabhängig sein, da sie sich nicht beeinflussen können. Bei nebenläufigen Ereignissen werden in der Modellierung keine Annahmen über die Reihenfolge getroffen.

Ein Ereignis ist die "gerichtete" Übertragung von Information von einem Objekt zum anderen. Eine mögliche Antwort ist ein weiteres Ereignis. Alle Objekte existieren in der realen Welt parallel.

Jedes Ereignis ist ein Objekt. Diese Objekte werden nach ihrer Struktur und ihrem Verhalten in Objektklassen gruppiert, wobei auch Vererbungshierarchien gebildet werden können. Ereignisse können einfache Signale sein oder eine Struktur haben, mit der zusätzliche Information übermittelt werden kann. Insbesondere die Zeit, zu der ein Ereignis stattfindet, ist häufig eine implizit oder explizit mitgelieferte Information. Den Begriff "Ereignis" verwenden wir sowohl für Ereignis-Objekte (Instanzen) alsauch für Ereignis-Klassen.

Wenn sich Ereignisse in einer Klassen-Hierarchie strukturieren lassen, dann wird eine Transition auch von Ereignissen vom Typ der Subklassen der Klasse des eigentlich spezifizierten Ereignisses gefeuert, sofern eventuell angegebene Bedingungen erfüllt sind.

Die Beschreibung eines Ereignisses enthält

- seine Datenstruktur,
- sein Verhalten,
- *nicht* aber sendende oder empfangende Objekte bzw. Zustände.

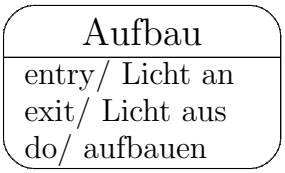
## 5.3 Notation

Die Automatentheorie ermöglicht eine formale Beweisführung von Eigenschaften wie Vollständigkeit, Vernetztheit und Determinismus von Zustandsgraphen [8] oder [11].

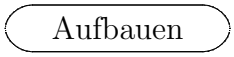
Die UML-Regeln zur Erstellung von Zustandsgraphen orientieren sich an Harel [4]. Das Objekt, für das wir den Graphen erstellen, nennen wir "Automat".

- Jeder stabile Zustand oder Operationsmode des Automaten wird durch ein Oval oder einen Kreis repräsentiert. Er hat i.A. ähnlich einer Klasse zwei Bereiche:
  - Der Name oder eine Nummer des Zustands steht im ersten Bereich. Häufig wird der Name weggelassen, wenn die Aktionen oder Aktivitäten des Zustands hinreichend aussagekräftig sind.

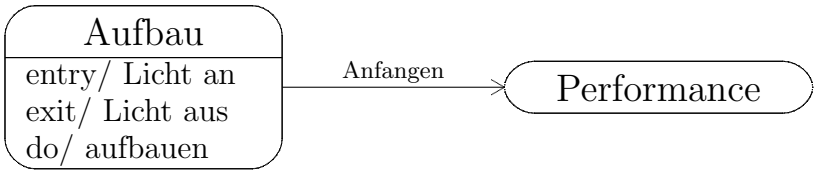
  - Im zweiten Bereich stehen die ***entry***, ***exit*** und ***do*** Prozeduren, d.h. **Aktionen *(action)*** und **Aktivitäten *(activity)*** (abgekürzt Akt., Operationen, Werte von Ausgangssignalen, Outputs, Aktuatorsignale) des Automaten, die in diesem Zustand ausgeführt werden. Die Angabe der Werte von Ausgangssignalen (Aktuatorsignalen) steht für die Aktion des Setzens dieser Werte. Aktionen haben keine Zeitdauer, d.h. sind nicht unterbrechbar. Aktivitäten haben eine Dauer und sind, falls das erlaubt wird, unterbrechbar.




  - Zustände können auch nur einen Bereich haben, entweder für den Namen oder eine aussagekräftige Aktion oder Aktivität.




- **Ereignisse *(events,*** Eingangssignale, Inputs, Sensorsignale, oft logische Verknüpfungen der Eingangssignale), die der Automat empfängt und die den nächsten Zustand bestimmen, werden auf Pfeile geschrieben, die zum nächsten Zustand führen. Pfeile beschreiben also **Zustandsübergänge *(transition)***. Ereignisse können auch durch eine Aktivität des Zustands – etwa als Resultat einer Berechnung – generiert werden.




  Ein Ereignis kann parametrisiert werden, d.h. bei einem Ereignis können Werte in runden Klammern `()` angegeben werden.

  Wenn ein Ereignis nur unter einer **Bedingung *(condition)*** in den nächsten Zustand führt, dann wird diese Bedingung in eckigen Klammern `[]` hinter das Ereignis auf den Pfeil geschrieben.

  Aktionen, die direkt auf ein Ereignis folgen, können durch `/` getrennt direkt hinter das Ereignis auf den Pfeil geschrieben werden. Man kann damit Zustände sparen.

  Zusammenfassend kann auf einem Transitions-Pfeil folgendes stehen:

```
Objekt.Ereignis (Attributwerte) [Bedingungen] / Aktionen
```

  Dabei sollte der Plural möglichst vermieden werden. "Objekt" ist hier das Ereignis sendende Objekt. (Diese Syntax entspricht leider nicht der Syntax gängiger objekt-orientierter Sprachen.)

  Beispiel:

```
Maus.gedrückt (mittlerer Knopf) [wenn im roten Feld] / lösche Feld
```

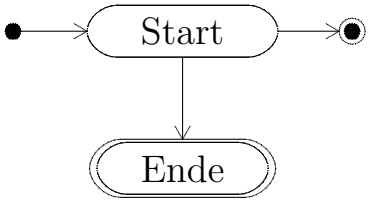
- Unbedingtes Fortschreiten (Pfeil ohne Ereignis) in den nächsten Zustand ist möglich.


- Rückführung in denselben Zustand ist möglich. Ob ein Ereignis **ignoriert** wird oder ob es in denselben Zustand zurückführt, ist i.A. ein Unterschied, da im letzteren Fall der Zustand neu betreten wird, was die Durchführung von `exit`-, `entry`- und `do`-Aktionen/Aktivitäten zur Folge hat. Daher sollten nicht-ignorierte Ereignisse auch modelliert werden. Ignorierte Ereignisse tauchen in der Modellierung normalerweise nicht auf.


- Der **Startzustand** ***(initial state)*** wird durch einen dicken Punkt zum Startzustand dargestellt. Er ist ein Pseudozustand und geht sofort in den eigentlichen Startzustand über. Es kann in einem Zustandsgraphen nur einen Startzustand geben.


- **Terminale** oder **finale** Zustände ***(terminal, final state)*** werden durch Doppelkreis oder -oval dargestellt. Es kann mehrere terminale Zustände geben. Ein terminaler Zustand ist dadurch gekennzeichnet, dass das System nicht mehr auf Ereignisse reagiert. Aus diesen Zuständen führt kein Pfeil heraus.

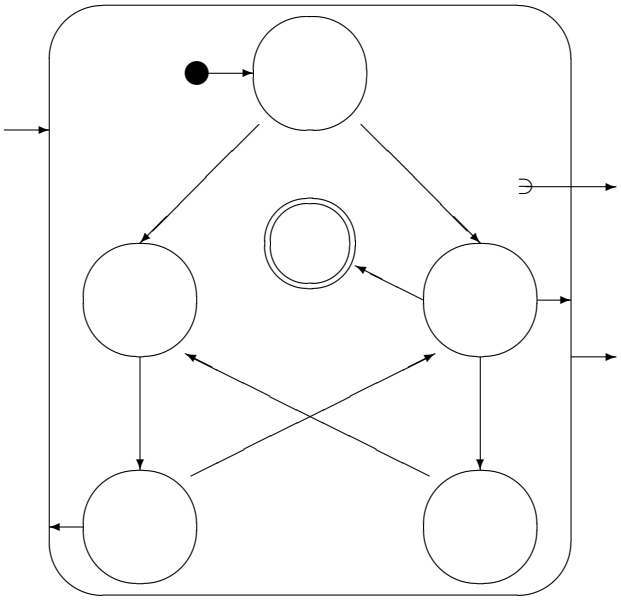
  Es gibt auch einen terminalen Pseudozustand dargestellt durch einen dicken Punkt in einem Kreis.




- Hierarchie von Zustandsgraphen ist möglich: Ein Zustandsgraph $X$ kann eine Aktivität des Zustands $Y.A$ eines übergeordneten Zustandsgraphen $Y$ sein. D.h., wenn im Zustandsgraph $Y$ der Zustand $Y.A$ erreicht wird, dann wird der Startzustand des Zustandsgraphen $X$ – gekennzeichnet durch ($\bullet\!\!\longrightarrow$) – betreten. Die Einbettung von $X$ in $Y.A$ wird dargestellt, indem der Zustandsgraph $X$ durch ein großes Oval, das den Zustand $Y.A$ darstellt, umrahmt wird. Der Zustandsgraph $X$ wird durch Pfeile verlassen, die aus einem oder mehreren Zuständen von $X$ an die Umrahmung führen. Ein terminaler Zustand in $X$ führt auch in den übergeordneten Zustand $Y.A$.

  Pfeile der Art ($\ni\!\!\longrightarrow$), die im übergeordneten Zustand $Y.A$ beginnen und aus dem Zustand herausführen, bedeuten, dass jedes auf diesem Pfeil angegebene Ereignis zur Folge hat, dass der Zustandsgraph $X$ aus jedem seiner Zustände heraus verlassen wird.

  Die Notation $\ni\!\!\longrightarrow$ eignet sich auch, wenn ein Ereignis auf alle Zustände (ausgenommen terminale Zustände) wirkt.



- Parallelität: Mehrere Automaten können gleichzeitig laufen. Das wird dadurch dargestellt, dass die entsprechenden Zustandsgraphen irgendwie nebeneinander oder übereinander, getrennt durch gestrichelte Linien plaziert werden.

  Es bleibt zu klären, wie die Interaktion zwischen parallel laufenden Zustandsgraphen darzustellen ist. Das Senden eines Ereignisses ist eine Aktion, die im sendenden Zustand ausgeführt wird und in einen Text der Art ***send event ... to ...*** eingebettet wird. Dabei kann das ***to ...*** möglicherweise weggelassen werden, da das Ereignis auf einem oder mehreren Übergangspfeilen zu finden ist.

  Um die Herkunft eines Ereignisses zu zeigen, kann der Name des Ereignisses auf dem Übergangs-Pfeil durch den Namen des sendenden Objekts qualifiziert werden ***(object.event)***.

  Eine graphische Darstellung durch Pfeile würde das Diagramm zu unübersichtlich machen.

## 5.4 Zustandsübergangsmatrix

Die **Zustandsübergangsmatrix** ***(transition matrix)*** ist eine nicht-graphische Darstellung von Zustandsgraphen. In der Matrix werden Zustände $Z_1, Z_2 \cdots Z_n$ gegen Ereignisse $E_1, E_2 \cdots E_m$ aufgetragen. Die Matrixzellen enthalten den Folge-Zustand, d.h. den Zustand, der betreten wird, wenn im Zustand $Z_i$ (Zeilenkopf) das Ereignis $E_j$ (Spaltenkopf) empfangen wird.

| | $E_1$ | $E_2$ | $\cdots$ | $E_m$ |
|---|---|---|---|---|
| $Z_1$ | $Z_4$ | $Z_6$ | $\cdots$ | $Z_1$ |
| $Z_2$ | $Z_3$ | $Z_2$ | $\cdots$ | $Z_7$ |
| $\cdots$ | $\cdots$ | $\cdots$ | $\cdots$ | $\cdots$ |
| $Z_n$ | $Z_8$ | $Z_1$ | $\cdots$ | $Z_5$ |

Wenn ein Ereignis ignoriert wird, dann bleibt die Zelle leer.

## 5.5 Formale Beschreibungssprache

Im ISO-Standard 8807 wurde die formale Beschreibungssprache LOTOS *(Language of Temporal Ordering Specification)* festgelegt. Sie wurde zur Beschreibung von verteilten Systemen, insbesondere von Kommunikationssystemen entworfen. Ein kommerzielles Produkt für die Entwicklung von Zustandsgraphen ist STATEMATE von der Firma i-Logix.

## 5.6 Strukturierung von Zustandsgraphen

Für Ereignisse können Erweiterungs-Hierarchien (Vererbungs-Hierarchien) gebaut werden.

Die Strukturierung von Zustandsgraphen erfolgt analog zur Strukturierung von Objekten, wo wir hauptsächlich die Begriffe *Erweiterung* und *Aggregation* verwendet haben.

Der Erweiterung von Objekten entspricht die Expansion der Aktivität eines Zustands als eigenen Zustandsgraphen (Hierarchie von Zustandsgraphen).

Der Aggregation entspricht die Aufteilung einer Zustandsaktivität in parallele Aktivitäten.

Bei nebenläufigen Zustandsgraphen ist zu bemerken, dass diese Graphen sich durch das Senden von Ereignissen beeinflussen können, wobei die Übergänge auch von Zuständen anderer Zustandsgraphen abhängen können.

## 5.7 Entwicklung von Zustandsgraphen

Ein erster Schritt zur Erstellung des dynamischen Modells ist ein **Szenario** ***(scenario)***. Das ist die Auflistung aller Vorgänge in *einer* möglichen Reihenfolge ohne Berücksichtigung der sendenden oder empfangenden Objekte. Bei komplizierten Problemen müssen eventuell mehrere oder besonders lange Szenarien erstellt werden.

Beispiel eines Szenarios an einem Kofferband:

1. Band hält an.
2. Koffer wird in der Mitte des Bandes aufgelegt.
3. Koffer wird links aufgelegt.
4. Band fährt nach rechts.
5. Koffer verlässt rechts das Band.
6. Koffer wird links aufgelegt.
7. Koffer wird in der Mitte vom Band genommen.
8. Koffer wird links aufgelegt.
9. Koffer wird rechts aufgelegt.
10. Koffer verlässt rechts das Band.

11. Koffer verlässt rechts das Band.
12. Koffer verlässt rechts das Band.
13. Band hält an.

Der nächste Schritt ist die Identifikation von Sendern und Empfängern. Das Resultat kann in einem Sequenz-Diagramm dargestellt werden.

Mit einem Szenario und/oder Event-trace können nur einzelne von möglicherweise Tausenden von Ereignisfolgen dargestellt werden. Mit einem Zustandsgraphen dagegen können alle Ereignis- und Zustandsfolgen dargestellt werden.

## 5.8 Übungen

### 5.8.1 Kofferband

Erstellen Sie einen Zustandsgraphen für ein Kofferband, das von beiden Seiten benutzbar ist und auf beiden Seiten jeweils genau eine Lichtschranke hat. Die Koffer sollen nicht vom Band geworfen werden. Das Band läuft eine gewisse Zeit (z.B. 8 Sekunden) nach.

Varianten:

1. Die Koffer werden vom Band geworfen.
2. Die Koffer werden nach einer gewissen Zeit (z.B. 1 Minute) vom Band geworfen.
3. Überall entlang des Bandes gibt es Lichtschranken.
4. Der Bandmotor hat eine Totmann-Schaltung.

### 5.8.2 Fahrstuhl

Das Automatisierungssystem für die Steuerung eines Fahrstuhls mit drei Stockwerken habe folgende

**digitale Eingänge:**

$s1$**,** $s2$**,** $s3$**:** Sind `true`, wenn der Haltebereich eines Stockwerks erreicht ist, sonst `false`. Wenn der Haltebereich erreicht ist, kann der Motor des Fahrstuhls abgeschaltet werden und der Fahrstuhl wird richtig an dem betreffenden Stockwerk halten.

**1, 2, 3:** Sind `true`, wenn im Fahrstuhl ein Stockwerksziel gedrückt wurde. Dabei geht auch das entsprechende Lämpchen an, ohne dass der Programmierer sich darum kümmern müsste.

$1h$**,** $2r$**,** $2h$**,** $3r$**:** Sind `true`, wenn auf einem Stockwerk eine Anforderungstaste gedrückt wurde.

**digitale Ausgänge:**

**M0:** Fahrstuhl-Motor aus

**M1:** Fahrstuhl-Motor ein

**Rr:** Fahrstuhl-Richtung runter

**Rh:** Fahrstuhl-Richtung hoch

**zero1:** Lösche Lampe 1, Signal 1 ist dann `false`.

**zero1$h$:** Lösche Lampe 1$h$, Signal 1$h$ ist dann `false`.

**entsprechend:** für die Signale 2, 3, 2$r$, 2$h$, 3$r$.

Alle Knöpfe können gleichzeitig gedrückt werden.

Aufgabe: Erstelle einen Zustandsgraphen ohne Start- und Fehlerzustände.

# Kapitel 6

# Aktivitäts-Diagramm

Das Aktivitäts-Diagramm ist eine Sonderform des Zustandsgraphen, wobei die sogenannten **Aktivitäts-Zustände** ***(activity state)*** Zwischenzustände einer Berechnung oder eines Geschäftsprozesses sind. Sie werden dargestellt durch ein Rechteck mit gerundeten linken und rechten Seiten.


Ein Startzustand kann besonders gekennzeichnet werden:


Ebenso kann die letzte Aktivität durch einen terminalen Zustand bezeichnet werden:

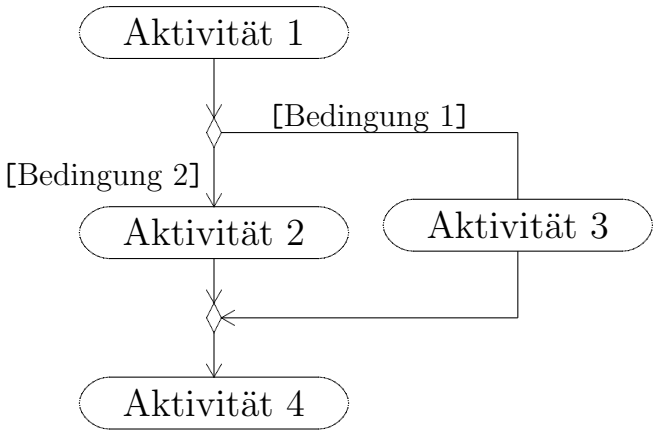

Normalerweise geht man bei einem Aktivitätsgraphen davon aus, dass die Prozesse ohne Wechselwirkung mit externen Ereignissen ablaufen. Ein Aktivitäts-Zustand wartet nicht auf ein Ereignis, sondern auf das Ende einer Aktivität (einer Berechnung oder eines Teilprozesses).

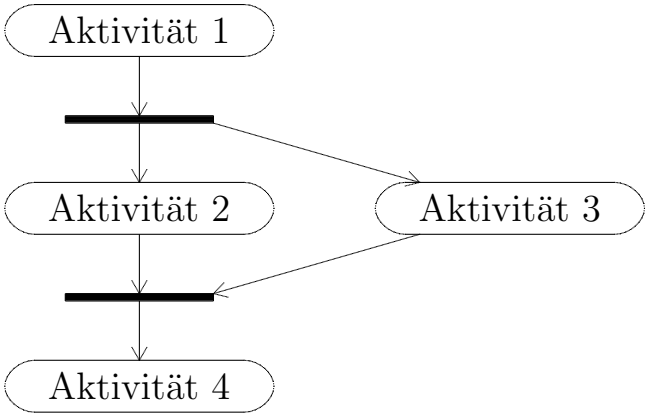
Ein Aktivitäts-Diagramm kann auch **Aktions-Zustände** ***(action state)*** (atomare Zustände, die nicht unterbrochen werden können) enthalten. Sie werden gewöhnlich für kurze Management-Tätigkeiten verwendet.

Ein Aktivitäts-Diagramm kann enthalten:

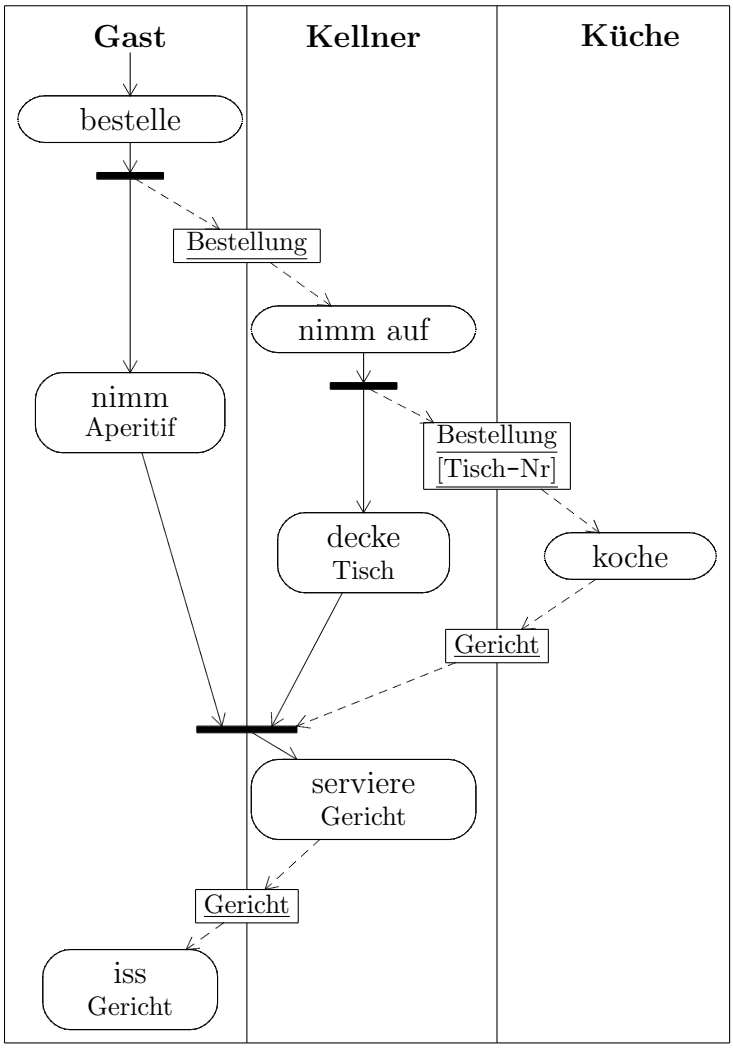
- Verzweigungen (Rauten), die den Kontrollfluss steuern
- und Verzweigungen in nebenläufige Kontrollflüsse (dicke Balken)

Die Bedingungen für die Verzweigung eines Kontrollflusses stehen auf den Pfeilen in eckigen Klammern. Ein Aktivitäts-Diagramm ist ein traditionelles Kontrollflussdiagramm mit Nebenläufigkeit. Die Nebenläufigkeitsbalken können auch senkrecht stehen.




Oft ist es günstig, ein Aktivitäts-Diagramm in verschiedene **Verantwortlichkeitsbereiche oder -grenzen (Schwimmbahn, *swimlane)*** zu unterteilen. Zwischen den Bereichen werden Botschaften in Form von **Objektflüssen** (gestrichelte Pfeile) ausgetauscht.




Ein Aktivitäts-Diagramm zeigt nicht das volle Detail eines Geschäftsprozesses oder einer Berechnung. Es zeigt den **Fluss *(flow)*** von Aktivitäten und ist ein Ausgangspunkt für das Design. Jede Aktivität muss dann mit Methoden von Klassen dargestellt werden, was dann Klassendiagramme, Zustandsgraphen und Interaktions-Diagramme als Resultat hat. Das Aktivitäts-Diagramm ist vergleichbar mit dem Datenfluss-Diagramm der Strukturierten Analyse.

# Kapitel 7

# Physische Diagramme

Der größte Teil eines System-Modells beschäftigt sich mit konzeptuellen Aspekten des Entwurfs unabhängig von der Implementation. Implementations-Überlegungen sind aber wichtig für Wiederverwendbarkeit und Performanz. UML bietet dafür zwei Notationen:

- **Implementations-Diagramm** ***(implementation diagram)***: Es zeigt austauschbare Teile des Systems, sogenannte **Komponenten** ***(component)***. Ferner zeigt es **Schnittstellen** ***(interface)*** von Komponenten und deren **Abhängigkeiten** ***(dependencies)***.
- **Einsatz-Diagramm** ***(deployment)***: Es zeigt die Anordnung von Hardware-Einheiten wie Computern, Terminalen **(Knoten,** ***node)*** und ihren **Verbindungen** ***(interconnection)***. Knoten können zur Laufzeit Komponenten und Objekte enthalten. Das kann eine statische oder dynamische Zuordnung sein. Das Diagramm kann Engpässe aufzeigen, wenn etwa Komponenten mit vielen Abhängigkeiten auf verschiedene Knoten plaziert werden.

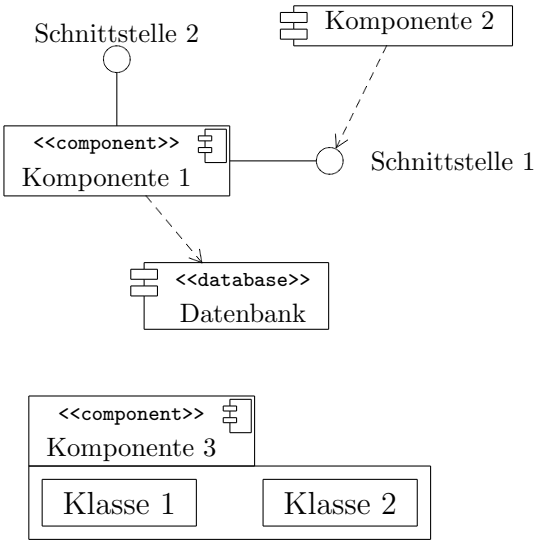
## 7.1 Komponente

Komponenten sind wiederverwendbare Software-Einheiten, mit denen Software-Systeme konstruiert werden können. Eine Komponente hat wohldefinierte, wenige Schnittstellen. Gut entworfenen Komponenten hängen nicht direkt von anderen Komponenten ab (d.h. von Klassen anderer Komponenten), allenfalls von Schnittstellen anderer Komponenten. Nur dann kann eine Komponente leicht durch eine andere Komponente, die dieselben Schnittstellen unterstützt ersetzt werden.

Eine Komponente hat Schnittstellen, die sie unterstützt, und Schnittstellen, die sie verwendet.

Ein Implementations-Diagramm kann entweder Komponenten und ihre Abhängigkeiten als eine Bibliothek zeigen, mit der Systeme gebaut werden können oder es kann ein mit Komponenten konfiguriertes System zeigen.

Eine Komponente wird als ein Stereotyp "component" mit der früheren Komponenten-Ikone dargestellt. Diese alte Form ist auch noch erlaubt, aber nicht empfohlen. Sie wird als Rechteck mit zwei kleinen seitlichen Rechtecken dargestellt. Die Schnittstellen erscheinen als kleine runde Kreise mit dem Namen der Schnittstelle. Abhängigkeiten werden durch gestrichelte Pfeile dargestellt.




Komponenten können Stereotypen sein.

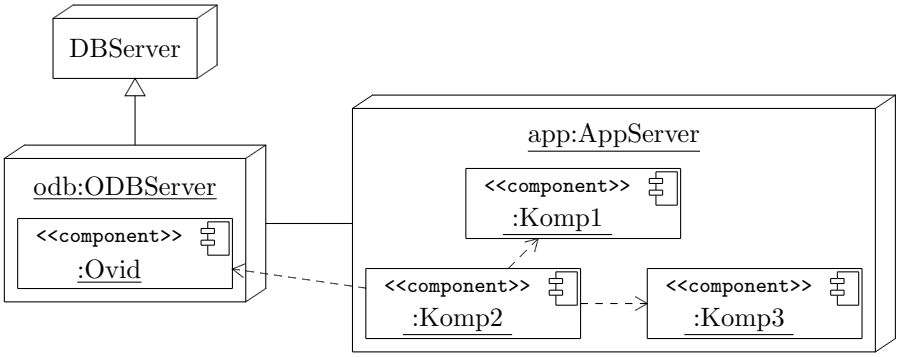
Wenn die Abhängigkeiten zwischen Komponenten durch Schnittstellen vermittelt werden, können die Komponenten leicht ausgetauscht werden.

## 7.2 Knoten

Ein Knoten ist ein zur Laufzeit existierendes physisches Objekt, das i.a. Speicher und Prozessor hat. Knoten können Stereotypen sein (z.b. Gerätetypen).

Ein Knoten wird durch einen Quader mit Knotenname, Knotentyp und seinen Komponenten dargestellt. Alle Angaben sind optional. Assoziationen (optional mit Multiplizitäten) zwischen den Knoten repräsentieren Kommunikationspfade. Zwischen Knoten kann eine Erweiterungsbeziehung bestehen, um Varianten darzustellen.

Das Diagramm kann überlagert werden durch Abhängigkeitsbeziehungen zwischen den enthaltenen Komponenten.


DBServer
app:AppServer
odb:ODBServer
<<component>>
:Komp1
<<component>>
:Ovid
<<component>>
:Komp2
<<component>>
:Komp3

# Kapitel 8

# Modell-Management-Diagramm

Ein großes System muss normalerweise in Teilsysteme aufgeteilt werden, um handhabbar zu sein. In UML bedeutet das die Definition von **Paketen** ***(package)*** und deren Abhängigkeiten.

## 8.1 Paket

Ein Paket ist ein Teil des Gesamtmodells. Jeder Teil des Gesamtmodells muss zu genau einem Paket gehören. UML gibt keine Regeln für die Erstellung von Paketen vor. Aber damit diese Unterteilung sinnvoll ist, muss sie nach vernünftigen Gesichtspunkten gemacht werden wie z.B. gemeinsame Funktionalität oder eng gekoppelte Implementation.

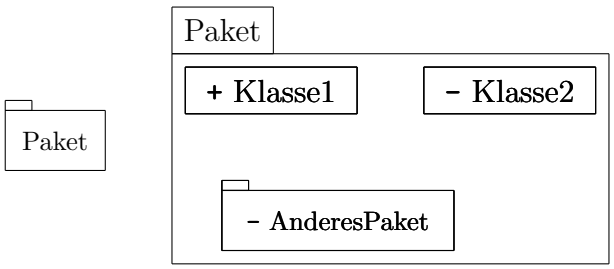
Pakete enthalten alle Arten von Diagrammen und Modellierungs-Elementen auf der höchsten Ebene, d.h. die enthaltenen Elemente sind nicht in anderen Elementen enthalten. Da eine Operation Element einer Klasse ist, kann sie nicht in einem Paket auftauchen.

Ein Element gehört nur zu **einem** Paket, dem ***home package***. Es kann aber von anderen Paketen referenziert werden. Ein Team-Mitglied muss Zugriff zum Home-Paket haben, um ein Element zu modifizieren. Mit Paketen können Zugriffsrechte definiert werden. Sie sind auch Einheiten für die Versionierung.

Pakete können andere Pakete enthalten. Es gibt ein ***root package***, das – indirekt – das Modell des Gesamtsystems enthält.

Pakete repräsentieren die Architektur des Systems auf der höchsten Ebene.

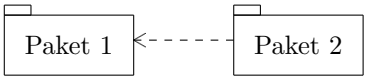
Pakete werden als Rechtecke mit Reiter dargestellt. Der Reiter enthält den Namen des Pakets, wenn Inhalte des Pakets gezeigt werden.




Die Sichtbarkeit von Elementen des Pakets kann mit "+" (public) oder "-" (private) angegeben werden.

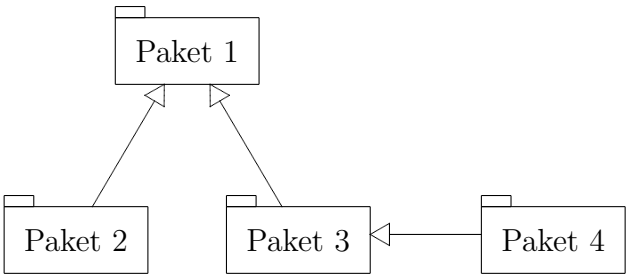
## 8.2 Abhängigkeiten zwischen Paketen

Abhängigkeiten zwischen Paketen repräsentieren summarisch irgendwelche Abhängigkeiten zwischen den enthaltenen Elementen, d.h. sind ableitbar von Abhängigkeiten zwischen Modell-Elementen.

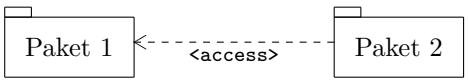



Abhängigkeiten werden als gestrichelte Pfeile dargestellt.

Pakete können auch in einer Erweiterungshierarchie zueinander stehen.




Im allgemeinen kann ein Paket nicht auf den Inhalt eines anderen Pakets zugreifen, es sei denn, dass eine `access` oder `import` Abhängigkeit das erlaubt.

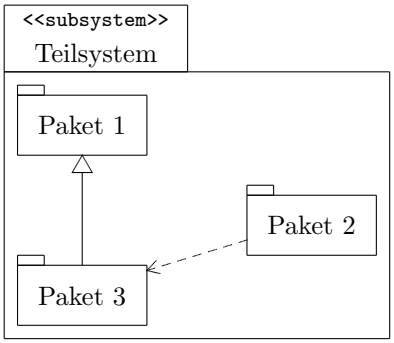



Elemente von Paket 2 dürfen auf Elemente von Paket 1 zugreifen, sofern das auf der Seite von Paket 1 gestattet ist. Auf der Supplier-Seite (Paket 1) muss die korrekte Sichtbarkeit gegeben sein, damit ein Zugriff möglich ist.

## 8.3 Teilsystem

Ein Teilsystem ***(subsystem)*** ist ein Paket, das eine eigenständige Einheit des Gesamtsystems mit wohldefinierter Schnittstelle zum Rest des Systems repräsentiert.

Ein Teilsystem wird dargestellt mit dem Stereotyp `subsystem` und seinem Namen im Reiter:



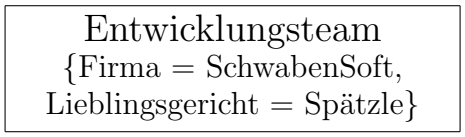
# Kapitel 9

# Erweiterungs-Konstrukte

Das wesentliche Erweiterungskonstrukt ***(extensibility construct)***: in UML ist das **Profil** ***(profile)***. Als eher schwache Erweiterungs-Konstrukte gelten noch:

- **Einschränkungen, Bedingungen** ***(constraint)***: Sie werden in geschweiften Klammern neben ein Modellelement geschrieben.


- Name-Wert-Paare ***(tagged value)***: Sie sind Kombinationen von einer Bezeichnung (oder eines Namens) und eines Wertes. Sie werden auch in geschweiften Klammern in die Nähe des Modellelements geschrieben.




- **Stereotypen** ***(stereotype)***: Sie sind neue Arten von Modellelementen, die vom Entwickler auf der Basis vorhandener Modellelemente erstellt werden. Wenn ein Stereotyp in Diagrammen verwendet wird, dann kann man z.B. alle Schnittstellen, die von ihm unterstützt werden, als bekannt voraussetzen und damit weglassen. Das macht Diagramme wesentlich übersichtlicher. Für Stereotypen können eigene Ikonen definiert werden.

  Für Stereotypen kann eine Semantik definiert werden, die dann dort gültig ist, wo der Stereotyp verwendet wird.

# Literaturverzeichnis


[1] Sinan Si Alhir, "UML in a Nutshell" O'Reilly 1998

[2] Alistair Cockburn, "Writing Effective Use Cases"

[3] Bruce Powel Douglass, "Real Time UML" Addison-Wesley

[4] D. Harel, "Statecharts: a visual formalism for complex systems", Science of Computer Programming **8** (1987), 244 – 275

[5] Martin Fowler und Kendall Scott, "UML konzentriert" Addison-Wesley 1998

[6] Hassan Gomaa, "Designing Software Product Lines with UML", Addison-Wesley

[7] Hassan Gomaa, "Designing Concurrent, Distributed, and Real-Time Applications with UML", Addison-Wesley

[8] J. E. Hopcroft und J. D. Ullman, "Introduction to Automata Theory, Languages, and Computation" Addison-Wesley 1979

[9] Bernd Oestereich, "Objekt-orientierte Softwareentwicklung", Oldenbourg 1998

[10] Object Management Group, `http://www.omg.org`

[11] Dominique Perrin, "Finite Automata" in "Formal Models and Semantics" ed. Jan van Leeuwen, Elsevier 1992

[12] Dan Pilone, "UML 2.0 in a Nutshell", OŽReilly

[13] James Rumbaugh, Ivar Jacobson und Grady Booch, "The Unified Modeling Language Reference Manual" Addison-Wesley 1999