



12. Abstrakte Klassen

ARINKO[®]

- In Vererbungsstrukturen gibt es immer folgende Fragestellung:
 - sollen mit allen Klassen in der Hierarchie sinnvoll Objekte erzeugt werden oder
 - gibt es manche Klassen, von denen Objekte unsinnig erscheinen?
- Das ist oft der Fall, wenn man in einer Hierarchie Gemeinsamkeiten oder gewisse Konzepte verankern möchte, aber ansonsten auf einer eher abstrakten Denkebene operiert.
- Klassen, die hierarchisch verankert werden, aber von denen keine Objekte existieren sollen, können als **abstrakte Klassen** gekennzeichnet werden.
- Aus abstrakten Klassen kann man keine Objekte erzeugen.
- Es können (und sollen) aber Unterklassen von abstrakten Klassen erben.
- Man bezeichnet gerne Klassen, die instanziiierbar sind, auch als **konkrete Klassen**.

- Abstrakte Klassen werden kenntlich gemacht, indem im Klassenkopf der Modifizierer `abstract` auftaucht:

```
public abstract class GeometrischesObjekt  
{  
}  
}
```

abstract kommt vor "class"

- Eine abstrakte Klasse darf Methoden deklarieren, für die sie selbst keine Implementierung anbietet.
- Sie bietet also nur die Deklaration an, d.h. dass es diese Methode geben wird.
- Eine solche Methode ist eine **abstrakte Methode** und bekommt ebenfalls den Modifizierer `abstract`:

```
public abstract class GeometrischesObjekt
{
    public abstract void zeichne();
}
```

- Abstrakte Methoden haben keinen Anweisungsblock.
- Solche Methoden sind nur in abstrakten Klassen erlaubt.
- Existiert eine abstrakte Methode \Rightarrow es muss sich um eine abstrakte Klasse handeln.
- Es existiert eine abstrakte Klasse \nRightarrow es gibt eine abstrakte Methode darin

- Konkrete Unterklassen von abstrakten Klassen müssen alle abstrakten Methoden implementieren:

```
public class Kreis extends GeometrischesObjekt
{
    public void zeichne()
    {
        // ...
    }
}
```

- Abstrakte Unterklassen *können* abstrakte Methoden implementieren, müssen es aber nicht.

- Was kann nicht abstract sein?
- Konstruktoren
- `static` Methoden
- `final` Methoden
- `private` Methoden
- Attribute dürfen ebenfalls nicht abstrakt sein.

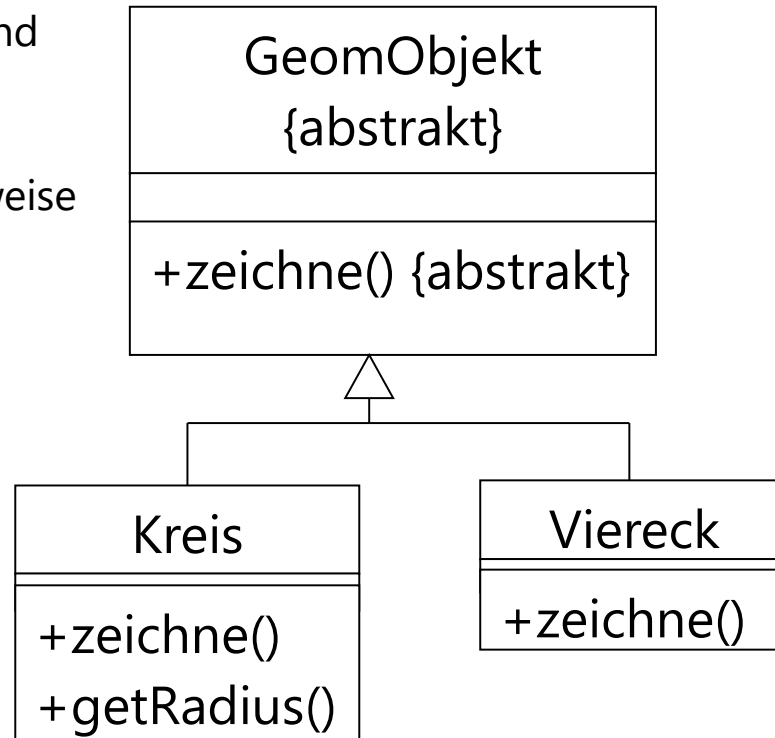
- Auch wenn man von abstrakten Klassen keine Objekte bauen kann, so dürfen trotzdem Variablen von abstrakten Typen vorkommen:

```
GeometrischesObjekt go = null;  
GeometrischesObjekt[] gos = new GeometrischesObjekt[2];
```

- Referenzvariablen eines abstrakten Typs beinhalten während der Laufzeit dann Instanzen von konkreten Subklassen (Typkompatibilität):

```
go = new Kreis();  
gos[1] = new Kreis();
```

- {abstrakt} oder {abstract} als Constraint von Klasse und Methode
- Alternativ: kursive Schreibweise *GeomObjekt* (oft schlecht erkennbar)





13. Interfaces

ARINKO[®]

- Interfaces sind abstrakten Klassen, die ausschließlich abstrakte Methoden beinhalten vergleichbar.
- Dadurch wird eine Schnittstelle (= Interface!) für Nutzer und für Implementationen vereinbart.
- Interfaces sind eine separate Typ-Art und werden von Klassen implementiert.
- Ihre Namensgebung deutet oft in Richtung einer Eigenschaft und endet im Deutschen somit auf "-bar", im Englischen auf "-able".

```
public abstract interface Transformierbar  
{  
...  
}
```

- Da es sich bei Interfaces immer um ein abstraktes Konzept handelt, darf der Modifizierer `abstract` weggelassen werden:

```
public interface Transformierbar  
{  
...  
}
```

- Interfaces besitzen [von der Grundidee her, bis einschl. Java 7] ausschließlich abstrakte Methoden.

```
public interface Transformierbar
{
    public abstract void verschiebe(int deltaX, int deltaY);
    public abstract void drehe(double grad);
    public abstract void skaliere(double faktor);
}
```

- Da Interfaces Schnittstellen darstellen, die Methoden definieren, die aufgerufen werden *sollen* und die prinzipiell abstrakt sind, dürfen diese Modifizierer ebenfalls weggelassen werden:

```
public interface Transformierbar
{
    void verschiebe(int deltaX, int deltaY);
    void drehe(double grad);
    void skaliere(double faktor);
}
```

- Interfaces beziehen sind ein objektorientiertes Konzept und beziehen sich somit auf Objekte/Instanzen.
- Daher dürfen [bis Java 7] in Interfaces keine `static` Methoden vorkommen.
- Interfaces geben nur eine Schnittstelle vor und enthalten somit keine Hinweise auf mögliche Attribute von implementierenden Klassen.
- Wenn Attribute vorhanden sind, müssen diese

`public static final`

sein.

- Alle Interface-Attribute brauchen einen Initialisierer, statische Initialisierungsblöcke sind nicht erlaubt.
- Interfaces definieren als Attribute also nur symbolische Konstanten.

- Interfaces werden von Klassen implementiert.
- Beim Übergang zwischen Schnittstelle und Klasse verwendet man das Schlüsselwort `implements`.
- Konkrete Klassen, die Interfaces implementieren, müssen alle Methoden umsetzen.
- Abstrakte Klassen müssen das nicht.
- Es darf implementiert und gleichzeitig geerbt werden!

```
public class Kreis extends GeometrischesObjekt implements Transformierbar
{
    // Realisierung der abstrakten Methode aus der Oberklasse
    public void zeichne()
    {
        // ...
    }

    // Implementierung einer der Interface-Methoden
    public void verschiebe(int deltaX, int deltaY)
    {
        // ...
    }
    ...
}
```

erst "extends", wenn
vorhanden

- Interfaces führen einen eigenen Typ ins Typsystem ein.
- Von diesen Typen dürfen ebenfalls Referenzvariablen deklariert werden.

```
Transformierbar tf = null;  
Transformierbar[] tfs = new Transformierbar[10];
```

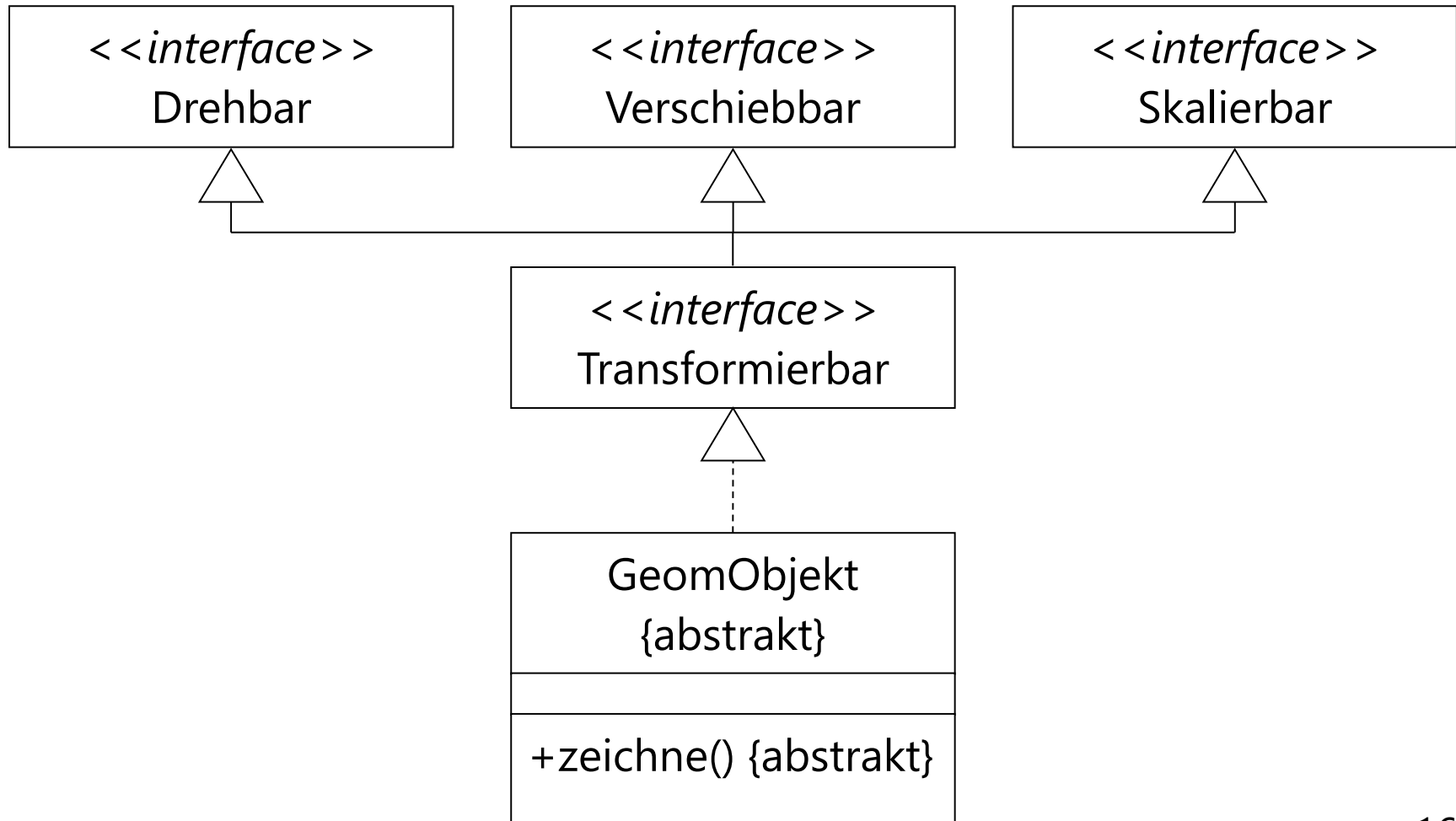
- In diesen Variablen bzw. Arrayplätzen dürfen nun alle Objekte, die das Interface direkt oder indirekt implementieren, abgelegt werden.
- Da Klassen mehrere Interfaces implementieren dürfen, kommen i.d.R. nun mehr Kandidaten in Frage.

- Interfaces dürfen voneinander ableiten.
- Hier wird das Schlüsselwort `extends` verwendet, wie bei der Ableitung zwischen Klassen.

```
public interface Transformierbar extends Verschiebbar, Drehbar, Skalierbar  
{  
}
```

- Im Gegensatz zur Einfachvererbung bei Klassen beherrschen Interfaces die Mehrfachvererbung.
- Hinter `extends` darf eine ganze Liste stehen.
- Merke: verwende `extends`, wenn Du innerhalb des selben Typsystems bleibst, verwende `implements`, wenn Du Interfaces von Klassen implementieren lassen willst.

- Interfaces werden mit Hilfe eines Stereotyps gekennzeichnet: <<interface>>
- Die Realisierungs-/Implementierungsbeziehung wird durch den gestrichelten Vererbungspfeil gezeigt.



- Seit Java 8 sind in Interfaces statische Methoden erlaubt.
- Diese *müssen* eine Implementierung enthalten.

```
public interface InterfaceMitStatic
{
    static void statischeMethode()
    {
        System.out.println("Java8 sei dank...");
    }
}
```

- Aufruf über den Interface-Typ:

```
InterfaceMitStatic.statischeMethode();
```

- Seit Java 8 sind in Interfaces ebenfalls Implementierungsvorschläge, sogenannte "default Methoden" erlaubt.

```
public interface InterfaceMitDefault
{
    default void tuwas()
    {
        System.out.println("Das ist die Default-Implementierung von tuwas");
    }
}
```

- default-Methoden wurden benutzt, um nachträglich in Interface-Hierarchien Methoden einzuführen, die in den implementierenden Klassen bislang unbekannt waren.

- Durch die Default-Methoden ist folgende Regel nun gebrochen: "Implementierung erbt man nur von Klassen (Einfachvererbung)"
- Das führt durch die Hintertür Mehrfachvererbung in Java ein.
- Damit gibt es auch z.B. folgendes Problem:

```
interface InterfaceA
{
    default int doit()
    {
        return 0;
    }
}
```

```
interface InterfaceB
{
    default int doit()
    {
        return 1;
    }
}
```

```
class AB implements InterfaceA, InterfaceB
{
    // Compilefehler
    // Duplicate default methods named doit ...
    // are inherited from the types InterfaceB and InterfaceA
}
```

- Mehrfach geerbte Methoden müssen überschrieben werden, entweder
 - mit einer eigenen Implementierung oder
 - durch den Aufruf einer der existierenden Implementierungen.

```
class AB implements InterfaceA, InterfaceB
{
    public int doit()
    {
        return 42;
    }
}
```

```
class BA implements InterfaceA, InterfaceB
{
    public int doit()
    {
        return InterfaceB.super.doit();
    }
}
```

Aufruf der Implementierung
von Interface B

- Achtung:


`super.methode()` ist etwas völlig anderes als
`InterfaceX.super.methode()`

```
class AB implements InterfaceA, InterfaceB
{
    public int doit()
    {
        return super.doit();
    }
}
```

dieses super bezieht sich auf
die Oberklasse – falsch!

```
class BA implements InterfaceA, InterfaceB
{
    public int doit()
    {
        return InterfaceB.super.doit();
    }
}
```

super bezieht sich auf das
Interface – korrekt!

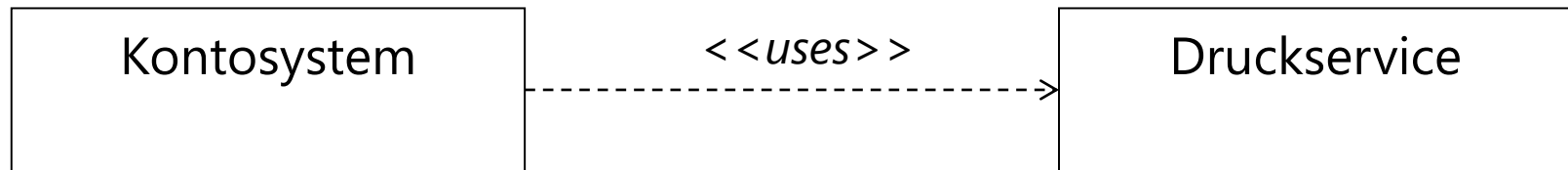


14. Beziehungen zwischen Klassen

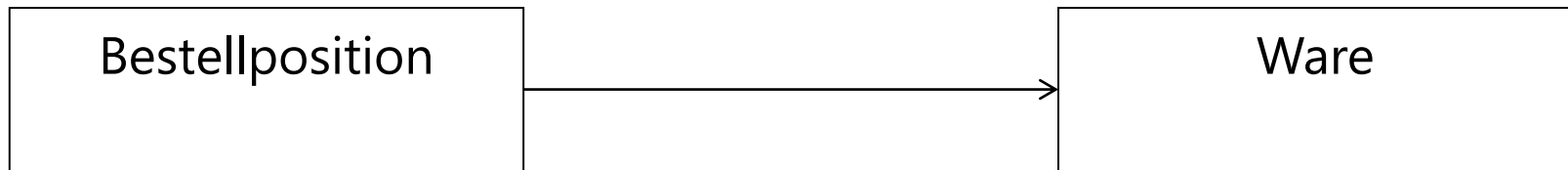
ARINKO[®]

- Klassen stehen zueinander in unterschiedlichen Beziehungen.
 - Die am häufigsten verwendeten sind
1. Vererbung (Klassen sind voneinander abgeleitet)
 2. Abhängigkeiten/Dependencies (Klassen stehen ohne Betrachtung der Objekte in einem Zusammenhang)
 3. Assoziationen (Objekte stehen in einem Zusammenhang)

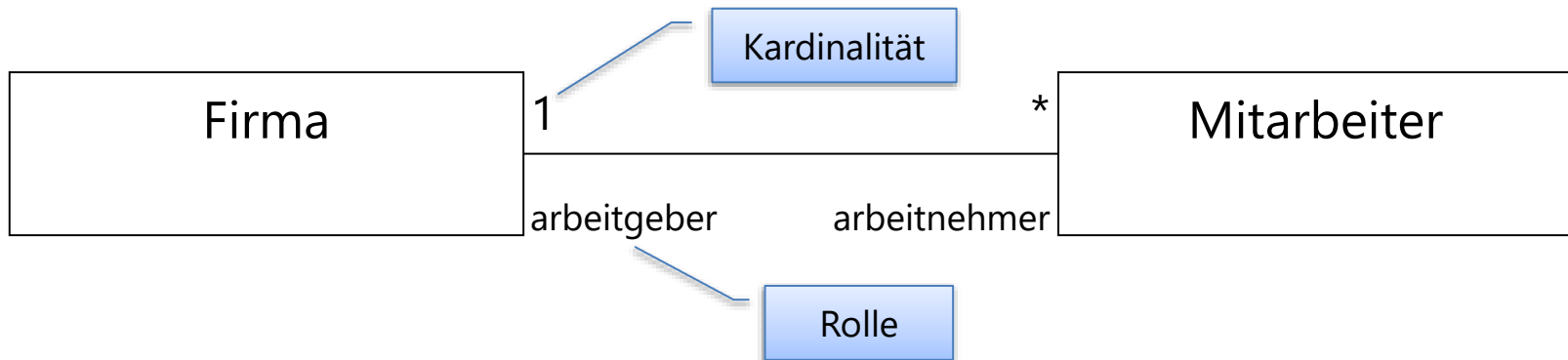
- Es gibt viele Möglichkeiten, wie Klassen zueinander eine Abhängigkeit haben können.
- Die übliche ist eine "uses"-Beziehung: eine Klasse verwendet Informationen/Methoden aus einer anderen Klasse, ohne sich dauerhaft eine Referenz darauf zu speichern (oft statische Methoden)



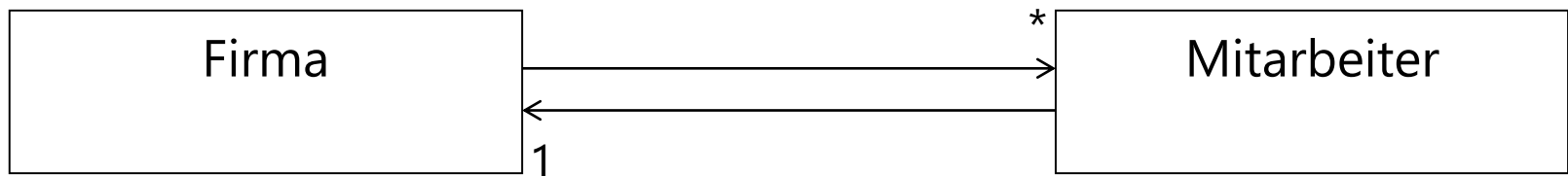
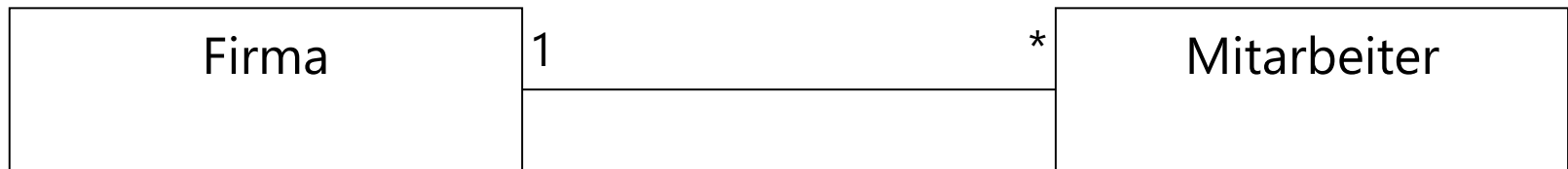
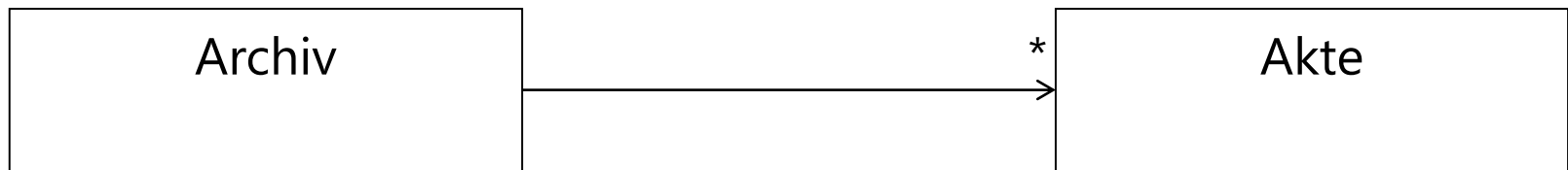
- Werden Beziehungen von Objekten zueinander betrachtet, spricht man von Assoziationen.
- Hierzu werden Referenzen in einem Objekt gespeichert (unidirektionale Beziehung) oder in beiden beteiligten Objekten (bidirektionale Beziehung).
- Wird via Referenz auf das andere Objekt Bezug genommen, spricht man von **Navigation**.



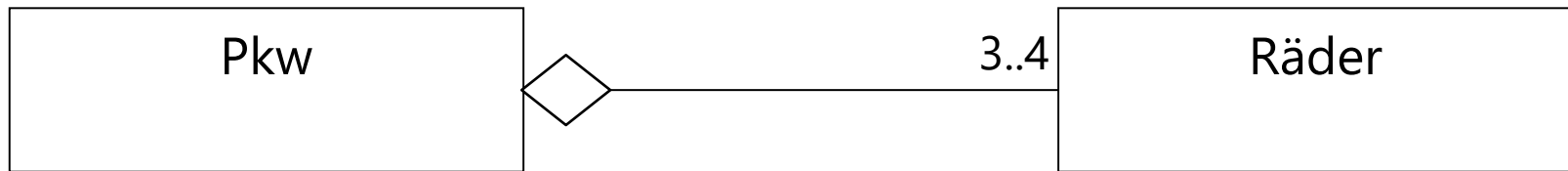
- In der UML gerne auch mit angereicherter Information:



- Assoziationen sind stets gerichtet (navigierbar).
- Die Richtung wird mit einem Pfeil angezeigt.
- Eine Linie ohne Pfeil bedeutet entweder unspezifiziert oder bidirektional, je nach Kontext und Aussage über Kardinalitäten.

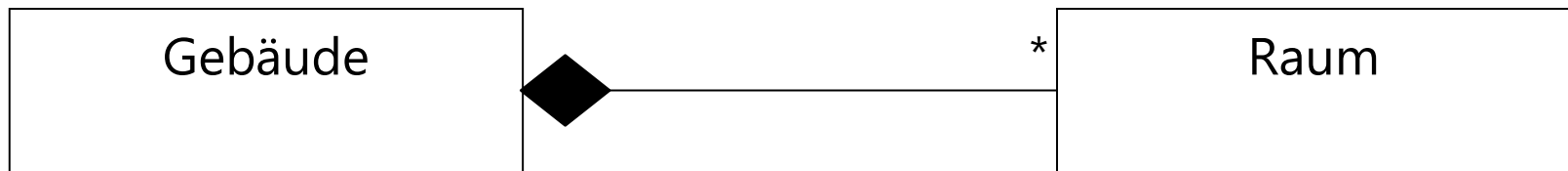


- Aggregationen stellen "stärker" verbundene Objekte dar, die aber durchaus getrennt voneinander existieren können.



- Die UML bleibt hier bewusst unspezifisch, was genau eine Aggregation ist.
- Je nach Analyst/Modellierer wird man sie unterschiedlich verwendet finden.

- Kompositionen (Ganzes-/Teile-Beziehung) sind die stärksten Verbindungen zwischen Objekten.
- Normalerweise ist es so: entfernt man das "Ganze" verschwinden auch die "Teile".



- Auch hier bleibt die UML bewusst ungenau.

- Eine (sinnvolle) Deutungsmöglichkeit liefert z.B.
<http://www.c-sharpcorner.com/UploadFile/b1df45/dependency-generalization-association-aggregation-compos/>
- 1. Association is defined as a structural relationship, that conceptually means that the two components are linked to each other. This kind of relation is also referred to as a using relationship, where one class instance uses the other class instance or vice-versa, or both may be using each other. But the main point is, the lifetime of the instances of the two classes are independent of each other and there is no ownership between two classes.
- 2. Aggregation is the same as association but with an additional point that there is an ownership of the instances, unlike association where there was no ownership of the instances.
- 3. Composition: this is the same as that of aggregation, but with the additional point that the lifetime of the child instance is dependent on the owner or the parent class instance.

- Hier wird (kurz zusammengefasst) definiert:

1. Assoziation

Von außen "zusammengesteckte" Objekte, die ihre Beziehung somit nicht selbst bestimmt haben und die auch in einer n:m Beziehung sein können

2. Aggregation

Von außen "zusammengesteckte" Objekte, die sich aber auf nur ein "Ganzes" beziehen

3. Komposition

Von innen gebaute Objektverbünde, die bei Löschung des Ganzen dann auch legitim die einzelnen Bestandteile löschen können/dürfen.

- 1:1 Beziehung



```
class Abteilung
{
    private Leiter leiter;
}

class Leiter
{
    private Abteilung abteilung;
}
```

- 1:n Beziehung



```
class Firma
{
    private Mitarbeiter[] mitarbeiter;
}
```

```
class Mitarbeiter
{
    private Firma firma;
}
```

Arrays sind hier möglich aber recht sperrig.
Man trifft hier oft Collections an.

- m:n Beziehung



```
class Firma
{
    private Mitarbeiter[] mitarbeiter;
}
```

```
class Mitarbeiter
{
    private Firma[] firmen;
}
```

Wie Objekte genau in den Behälter (Array) gelangen, ist im Diagramm nicht gezeigt.

- Häufig trifft man so oder ähnliche Strukturen an:

```
class Firma
{
    private Mitarbeiter[] mitarbeiter;

    public Mitarbeiter[] getMitarbeiter()
    {
        return mitarbeiter;
    }

    public void addMitarbeiter(Mitarbeiter einMitarbeiter)
    {
        // in Array einfügen
    }
}
```

- Idiome sind "add" (ein Element hinzufügen) oder "set" (alle Elemente ersetzen).
- Weiterhin zu beachten, ob man seine interne Datenstruktur per get nach außen gibt...

