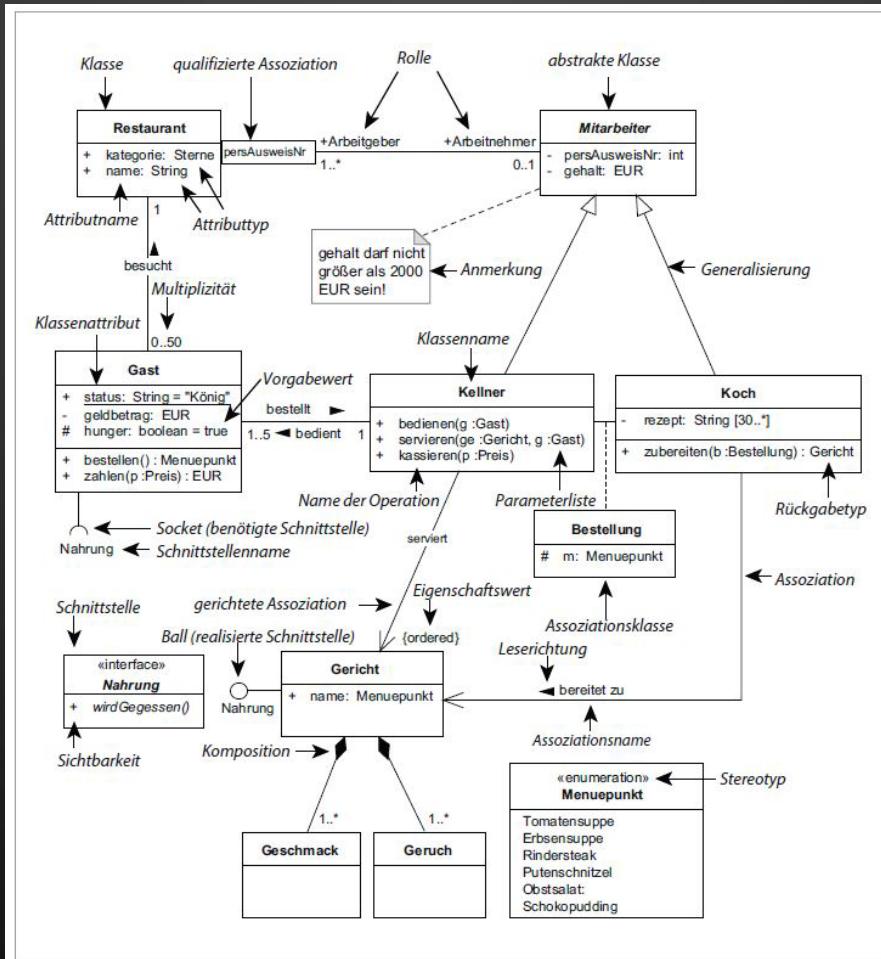




# KLASSENDIAGRAMM

MARKUS SZYNSKA

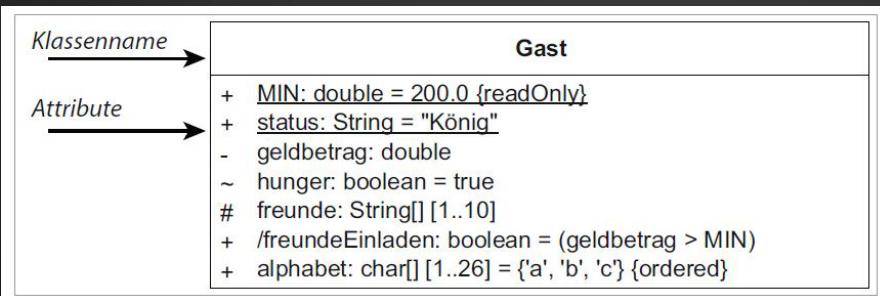
# ÜBERSICHT



MARKUS SZYNSKA

# KLASSE

- Eine **Klasse** (engl. *Class*) beschreibt eine Art **Bauplan für Objekte** mit der gleichen Struktur (Attribute) und dem gleichen Verhalten (Operationen).
- Als **Instanzen** oder **Ausprägungen** einer Klasse werden die nach ihrem Bauplan erstellten **Objekte** bezeichnet. Die **Erstellung eines Objekts** nach dem **Bauplan einer Klasse** nennt man **Instanziierung**.
- **Attribute** (engl. *Attributes*) stellen **strukturelle Eigenschaften** einer Klasse dar.

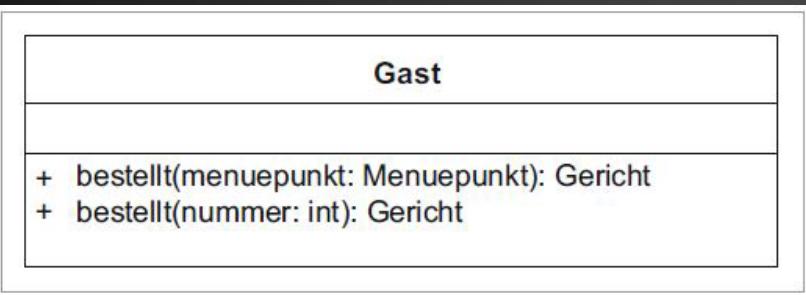


# OPERATIONEN(METHODEN) EINER KLASSE

Klassenname	Gast
Attribute	<ul style="list-style-type: none"><li>+ <u>MIN: double = 200.0 {readOnly}</u></li><li>+ <u>status: String = "König"</u></li><li>- geldbetrag: double</li><li>~ hunger: boolean = true</li><li># freunde: String[] [1..10]</li><li>+ /freundeEinladen: boolean = (geldbetrag &gt; MIN)</li><li>+ alphabet: char[] [1..26] = {'a', 'b', 'c'} {ordered}</li></ul>
Operationen	<ul style="list-style-type: none"><li>+ setGeldbetrag(geldbetrag: double): void</li><li>+ getGeldbetrag(): double</li><li>+ <u>getMINAsString():String</u></li><li>+ setFreund(in freund: String): void</li><li>+ getBesterFreund(out freund: String): void</li><li>+ getFreunde(): String[]</li><li>+ getAlphabet(start: int, end:int): String {ordered}</li><li>+ bestellt(menupunkt: Menuepunkt): Gericht</li><li>+ bestellt(nummer: int): Gericht</li><li>+ zahlt(d: double): double</li></ul>

# OPERATIONEN(METHODEN) EINER KLASSE

- **Operationen** (engl. *Operations*) spezifizieren die von einer Klasse angebotenen Dienstleistungen.



```
public Gericht bestellt(Menuepunkt menuepunkt)
{
    Koch koch = new Koch();
    Gericht gericht = koch.kocht(menuepunkt);
    return gericht;
}

public Gericht bestellt(int nummer)
{
    Menuepunkt menuepunkt = new Menuepunkt(numero);
    Koch koch = new Koch();
    Gericht gericht = koch.kocht(menuepunkt);
    return gericht;
}
```

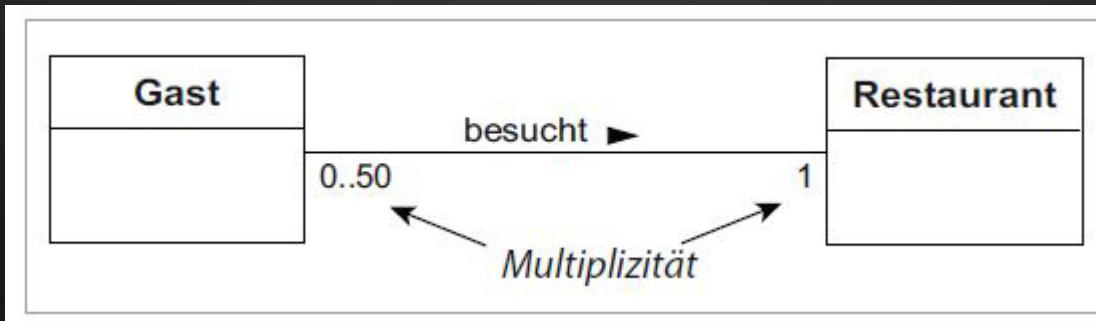
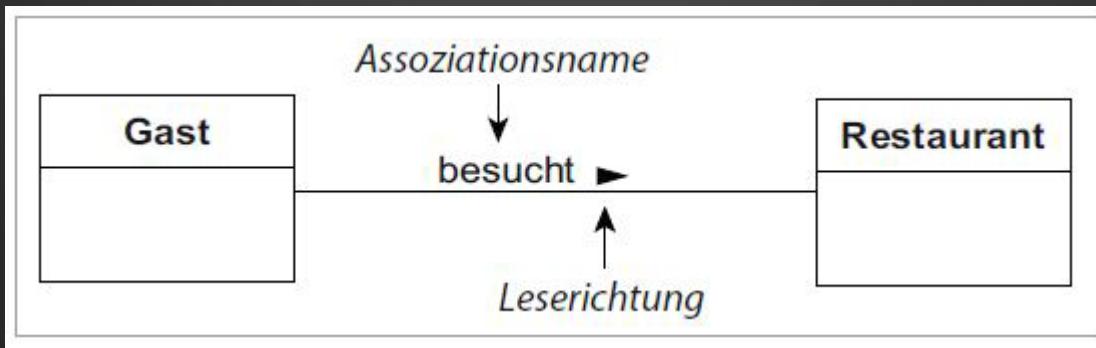


# ASSOZIATIONEN

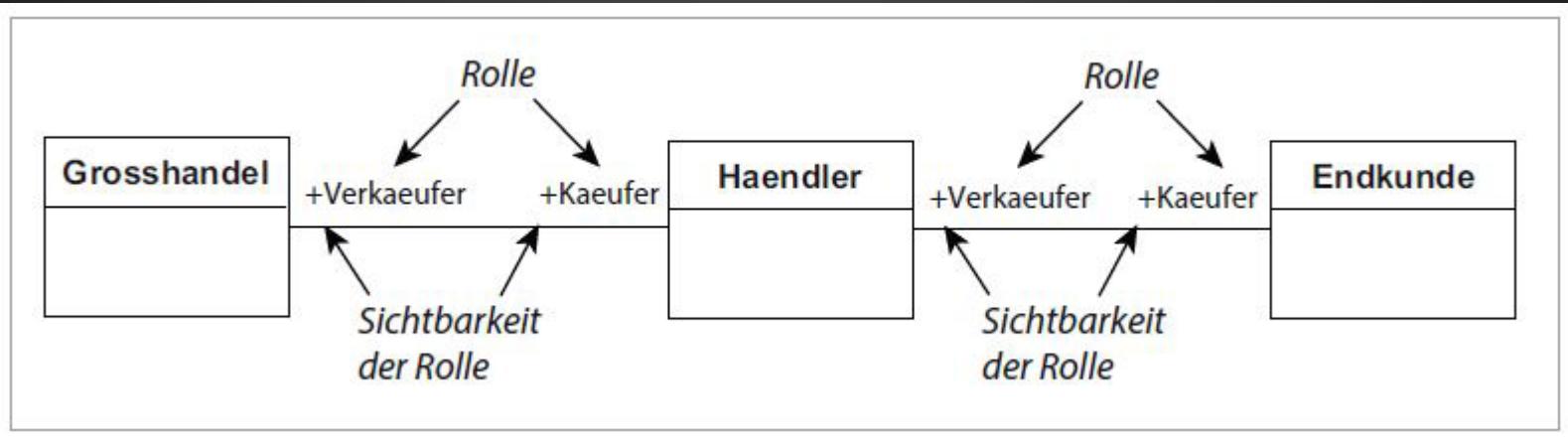
MARKUS SZYNSKA

# ASSOZIATIONEN MIT ASSOZIATIONSNAME UND MULTIPLIZITÄT

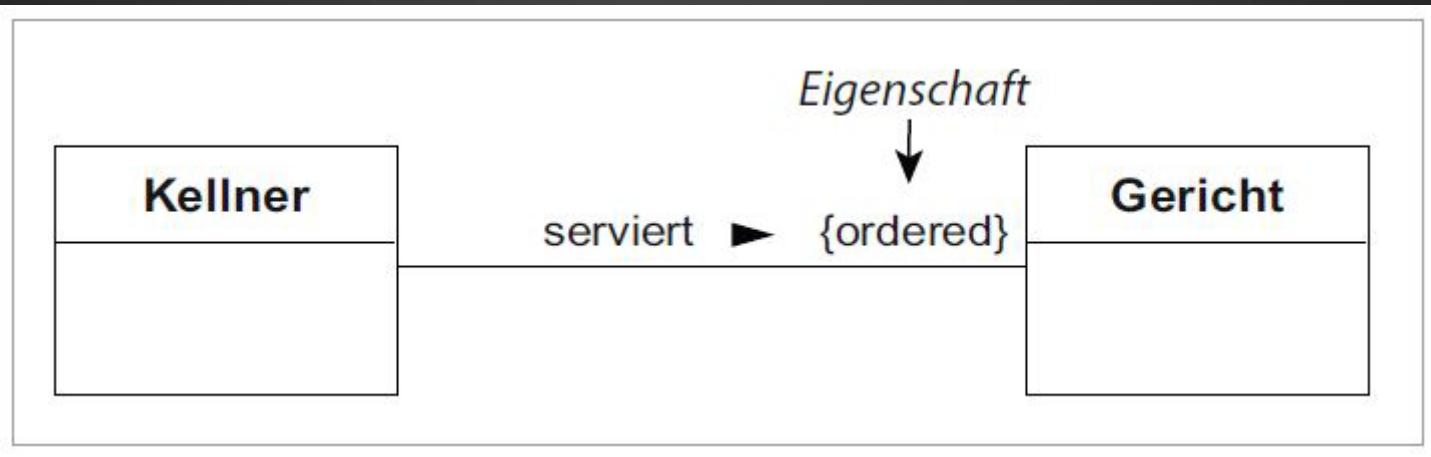
Eine **binäre Assoziation** (engl. *Binary Association*) spezifiziert eine semantische **Beziehung** zwischen zwei Klassen.



# ASSOTIATIONEN MIT ROLLENANGABEN

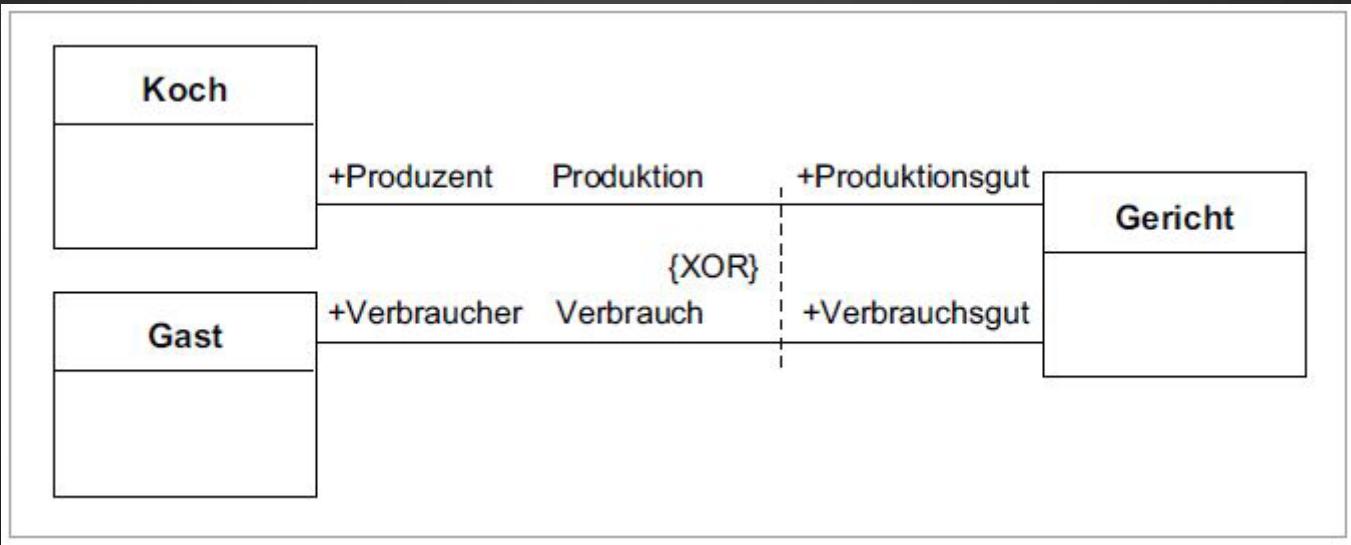


# ASSOTIATIONEN MIT EIGENSCHAFT



MARKUS SZYNSKA

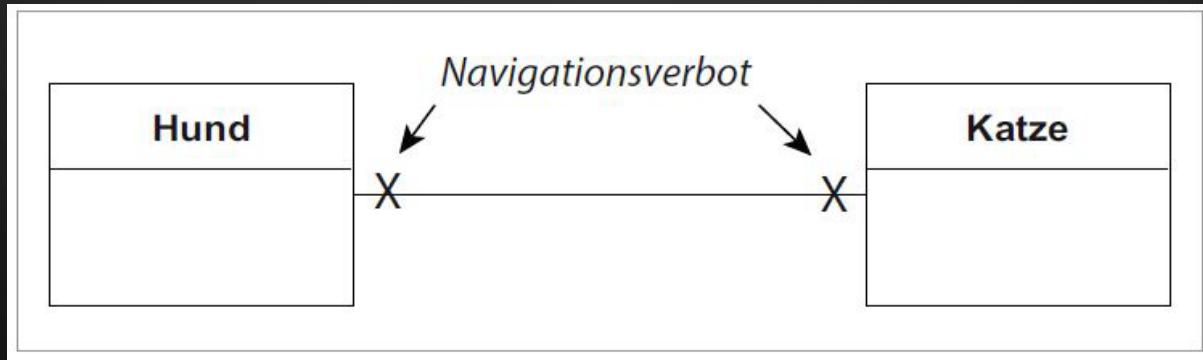
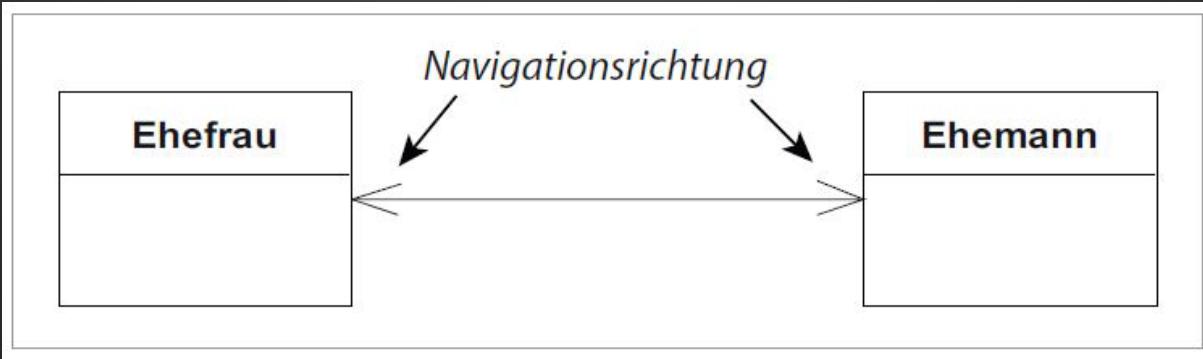
# ASSOTIATIONEN MIT EINSCHRÄNKUNG



MARKUS SZYNSKA

10

# ASSOTIATIONEN MIT BIDIREKTIONALER NAVIGIERBARKEIT



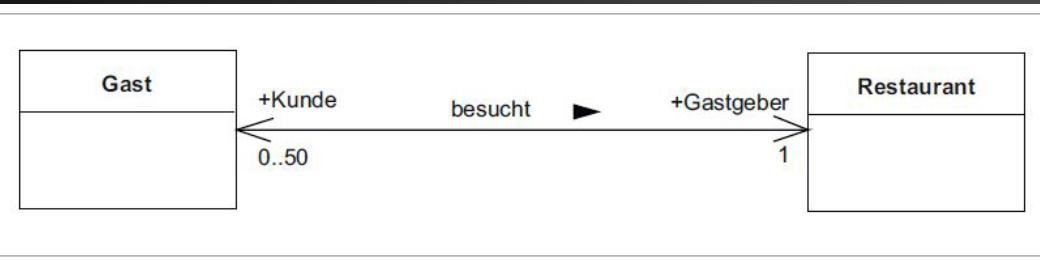
# ASSOTIATIONEN MIT UNSPEZIFIZIERTER UND UNIDIREKTIONALER NAVIGIERBARKEIT



# ASSOTIATIONEN MIT TEILWEISER SPEZIFIKATION DER NAVIGIERBARKEIT UND BESITZANZEIGE



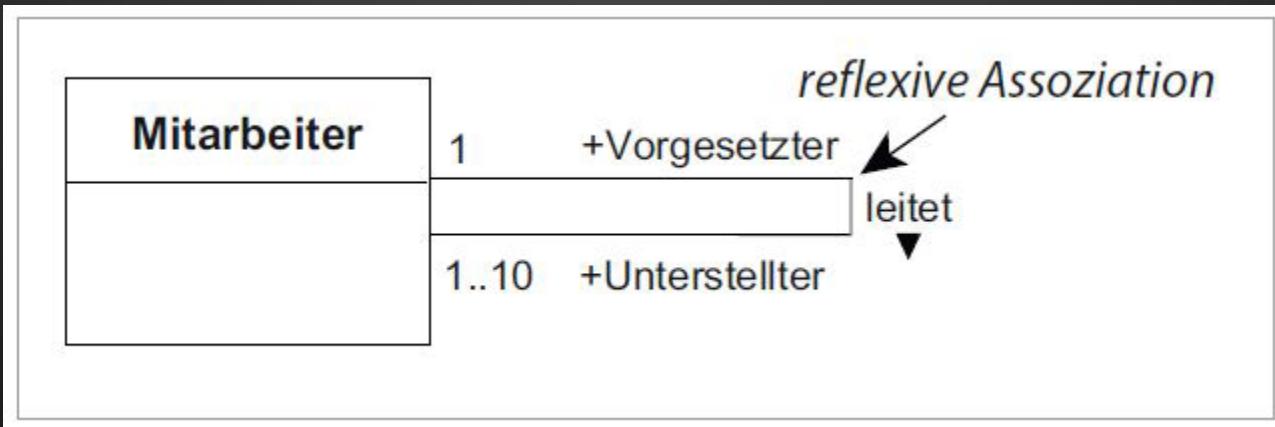
# BEISPIEL ASSIOZIATIONSDEFINITION



```
class Gast
{
A public Restaurant gastgeber;
B public Gast(Restaurant r)
{
    gastgeber = r;
}
class Restaurant
{
C public Gast[] kunde;
public Restaurant()
{
D kunde = new Gast[50];
}
```

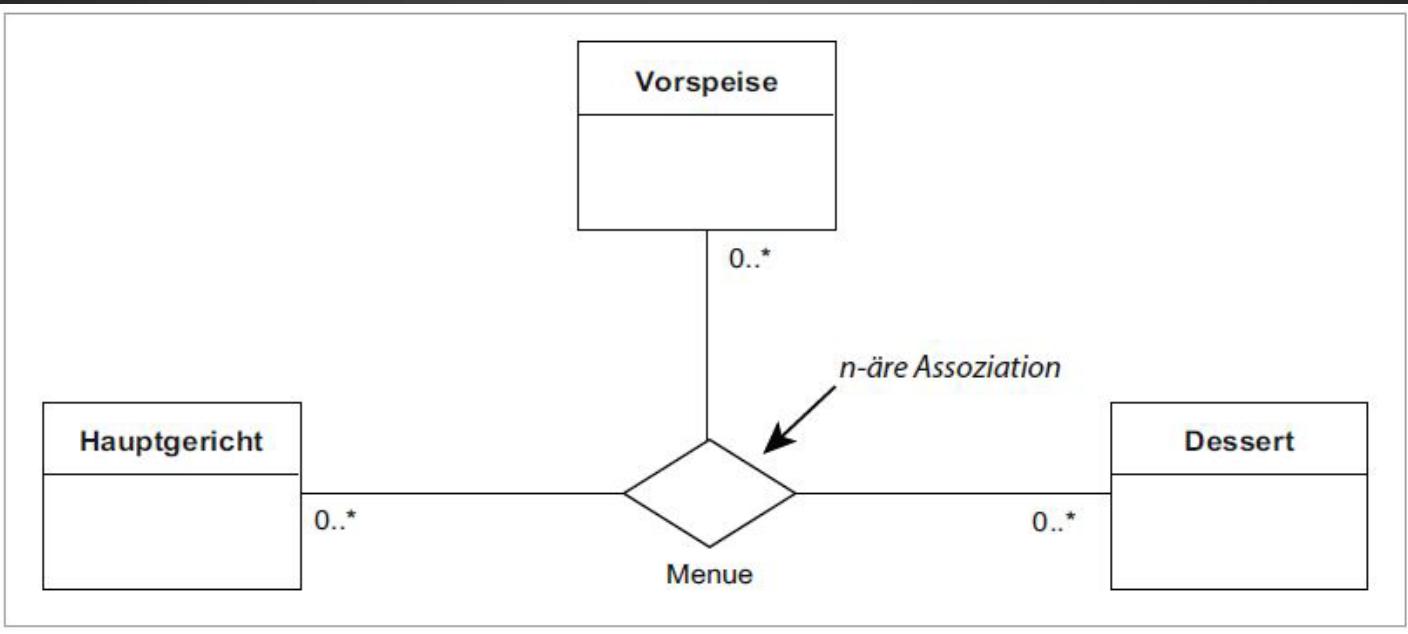
# REFLEXIVE ASSOZIATION

Reflexive Assoziationen sind Assoziationen von Klassen mit sich selbst.

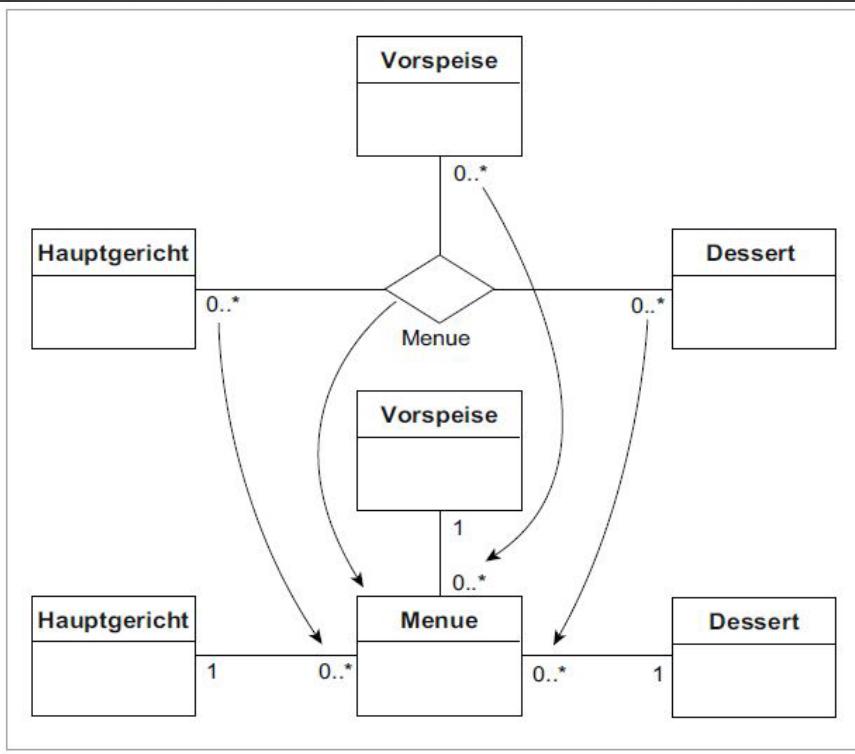


# N-ÄRE ASSOZIATION

Eine **n-äre Assoziation** (engl. *n-ary Association*) beschreibt allgemein **eine Beziehung** zwischen **n Klassen**.



# N-ÄRE ASSOZIATION – UMWANDLUNG IN BINÄRE ASSOZIATION



# N-ÄRE ASSOZIATION – UMWANDLUNG IN BINÄRE ASSOZIATION BEISPIEL JAVA

```
A class Menue
{
    B public Vorspeise vorspeise;
    public Hauptgericht hauptgericht;
    public Dessert dessert;

    C public Menue(Vorspeise v, Hauptgericht h, Dessert d)
    {
        vorspeise = v;
        hauptgericht = h;
        dessert = d;
    }

}
```

```
class Vorspeise
{
    A public ArrayList menue;

}

class Hauptgericht
{
    A public ArrayList menue;

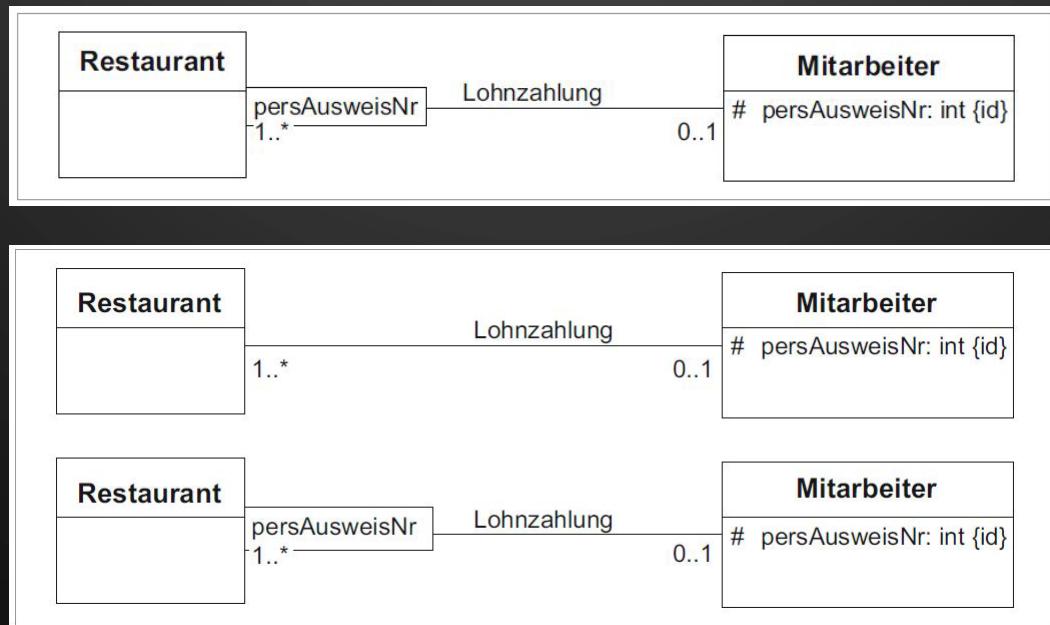
}

class Dessert
{
    A public ArrayList menue;

}
```

# QUALIFIZIERTE ASSOZIATION

Bei einer **qualifizierten Assoziation** (engl. *Qualified Association*) definiert der **Qualifizierer** (engl. *Qualifier*) eine Menge von Attributen, die **Objekte** der gegenüberliegenden Klasse **eindeutig referenzieren**. Qualifizierer sind nur für binäre Assoziationen definiert.



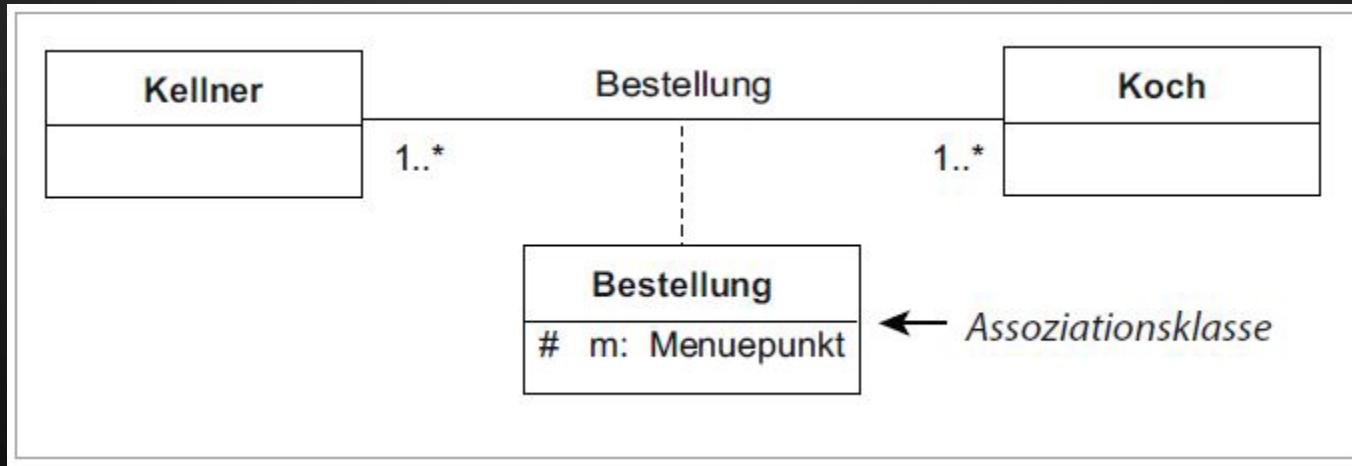
# QUALIFIZIERTE ASSOZIATION IN JAVA

```
class Mitarbeiter
{
    A protected int persAusweisNr;

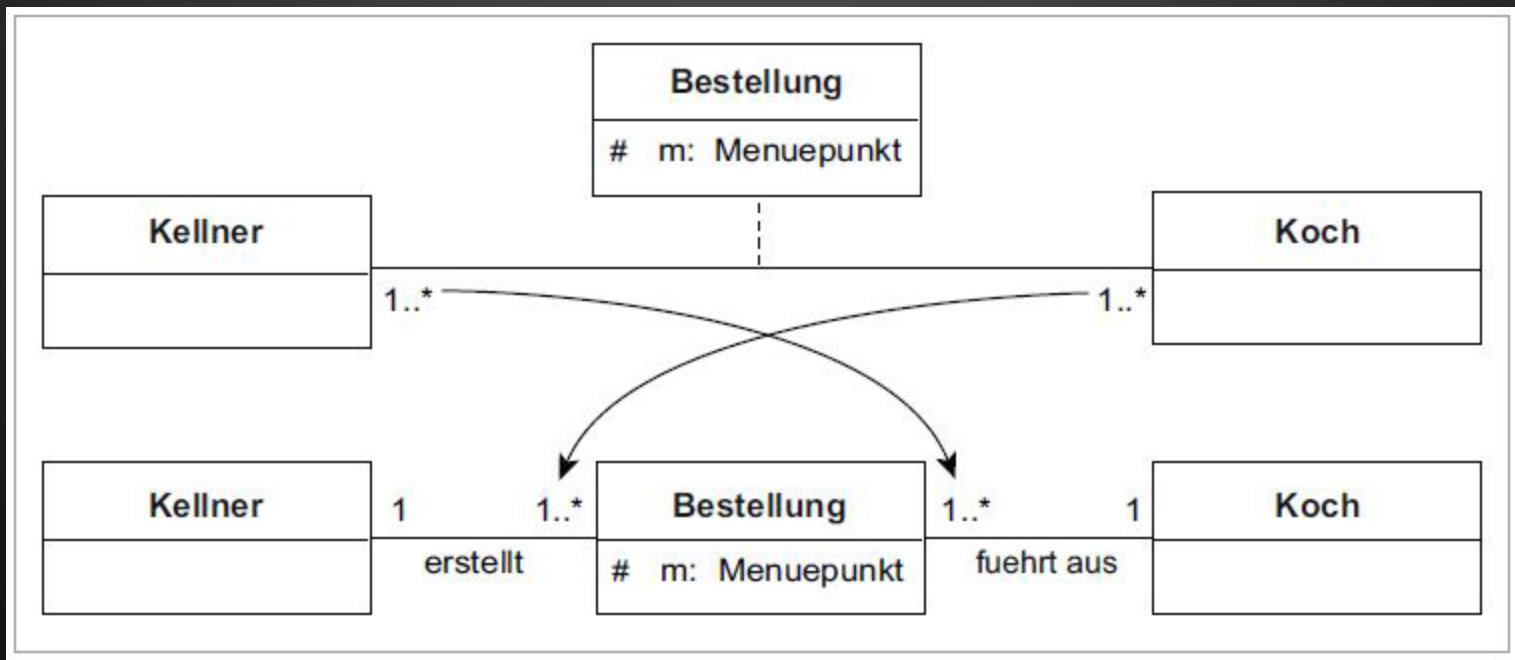
    }
class Restaurant
{
    B protected ArrayList mitarbeiter;
}
```

# ASSOZIATIONSKLASSE

Eine **Assoziationsklasse** (engl. *AssociationClass*) vereint die **Eigenschaften einer Assoziation und einer Klasse**. Sie stellt nicht nur eine Beziehung zwischen Klassen dar, sondern ordnet der Beziehung alle Eigenschaften einer Klasse zu. Das können sowohl Attribute als auch Operationen sein. Sowohl die Assoziation als auch die Klasse müssen den gleichen Namen tragen.



# ASSOZIATIONSKLASSE IN JAVA



# ASSOZIATIONSKLASSE IN JAVA

```
class Kellner
{
    A  public ArrayList bestellung;

}

class Koch
{
    B  public ArrayList bestellung;

}

class Bestellung
{
    C  protected Menuepunkt m;
    public Kellner ke;
    public Koch ko;
}
```

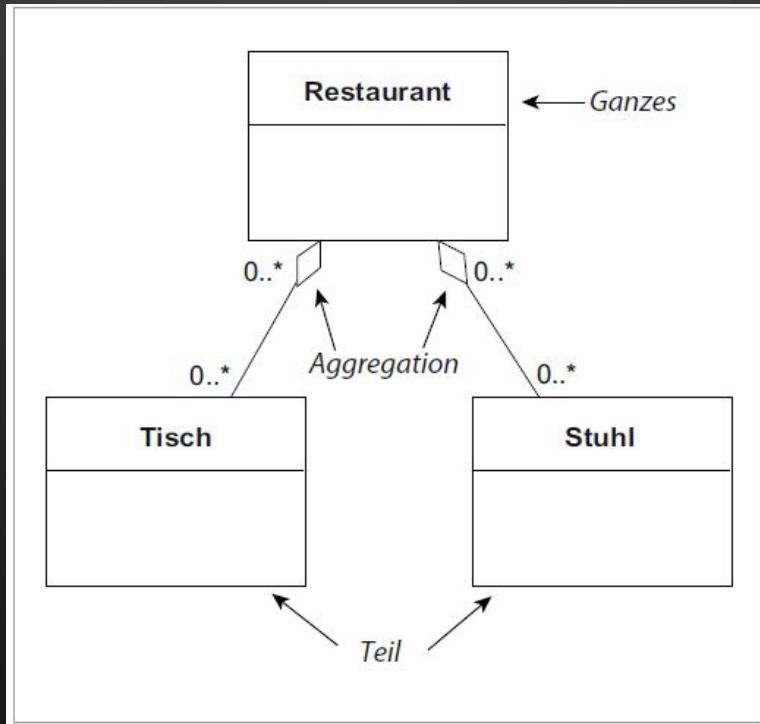


# AGGREGATIONEN

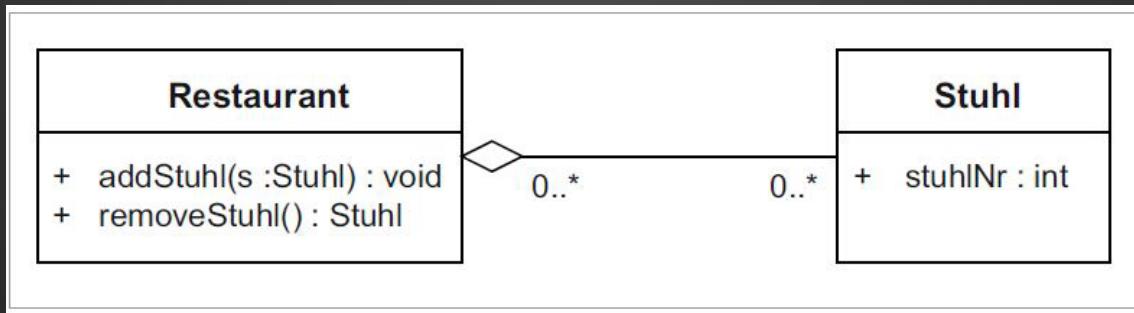
MARKUS SZYNSKA

# AGGREGATION

Die **Aggregation** (engl. Aggregation) ist eine spezielle Form der binären Assoziation und beschreibt eine **Ganzes-Teile-Beziehung**. Aggregationen mit mehr als zwei Enden (n-är) sind nicht definiert.



# AGGREGATION



```
class Stuhl  
{  
    public int stuhlNr;  
    public Stuhl(int nr)  
    {  
        stuhlNr = nr;  
    }  
}
```

# AGGREGATION

```
class Restaurant
{
    A public ArrayList stuhl;

    B public Restaurant()
    {
        stuhl = new ArrayList();
    }

    C public void addStuhl(Stuhl s)
    {
        stuhl.add(stuhl.size(), s);
    }

    D public Stuhl removeStuhl()
    {
        return (Stuhl)stuhl.remove(stuhl.size()-1);
    }
}
```

```
public static void main(String[] args)
{
    E Restaurant r = new Restaurant();
    Stuhl[] s = new Stuhl[10];
    for(int i = 0; i < 10; i++)
        s[i] = new Stuhl(i);

    F for(int i = 0; i < 10; i++)
        r.addStuhl(s[i]);

    G for(int i = 0; i < 10; i++)
        s[i] = r.removeStuhl();

}
}
```



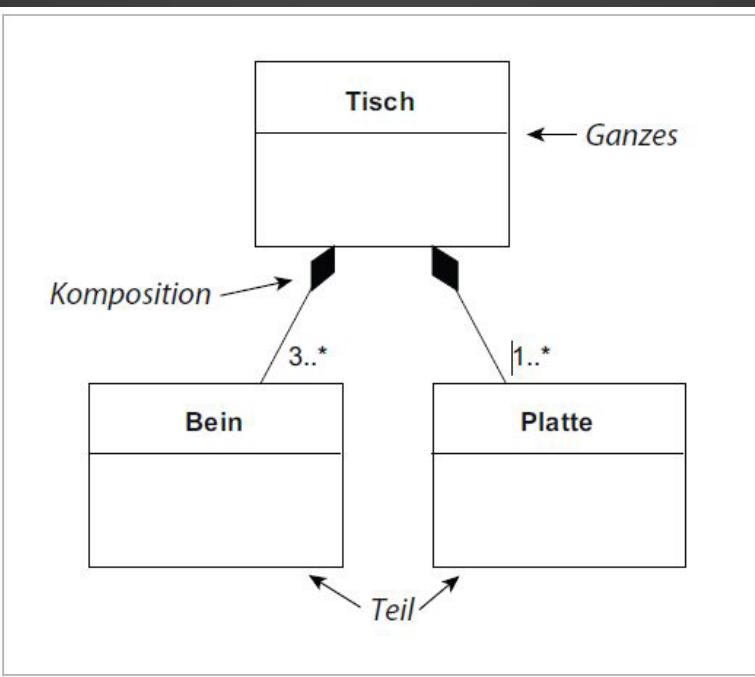
# KOMPOSITION

MARKUS SZYNSKA

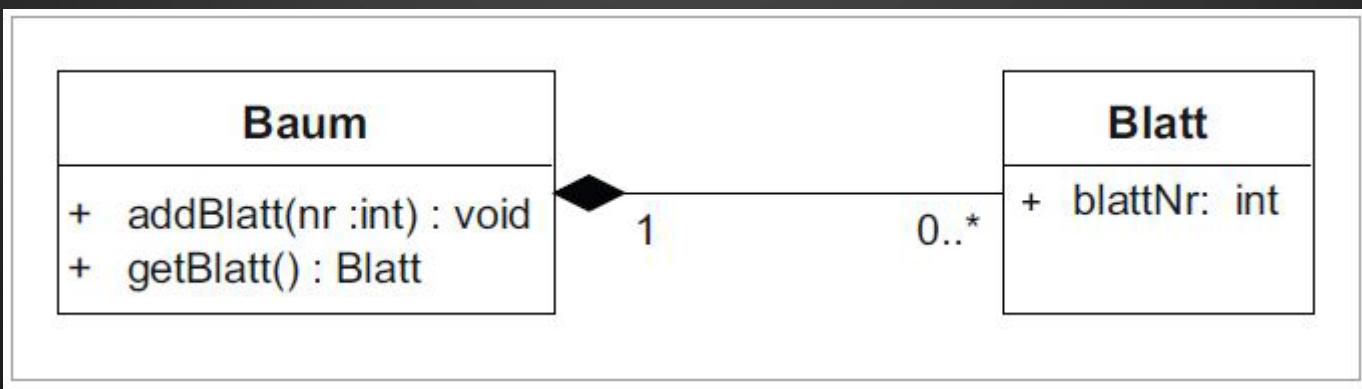
28

# KOMPOSITION

Die **Komposition** (engl. *Composition*) ist eine starke Form der Aggregation, die ebenfalls eine **Ganzes-Teile-Beziehung** definiert. Die **Verbindung** zwischen den Teilen und dem Ganzen wird jedoch als **untrennbar** definiert.



# KOMPOSITION



# KOMPOSITION

```
class Blatt
{
    public int blattNr;
    public Blatt(int nr)
    {
        blattNr = nr;
    }
}
```

# KOMPOSITION

```
class Baum
{
A private ArrayList bl;

B public Baum()
{
    bl = new ArrayList();
}

C public void addBlatt(int nr)
{
    bl.add(new Blatt(nr));
}

D public Blatt getBlatt()
{
    Blatt neuesBlatt = new Blatt(0);
    neuesBlatt.blattNr=((Blatt)bl.get(bl.size()-1)).blattNr;
    return neuesBlatt;
}
}
```

MARKUS SZYNSKA

32

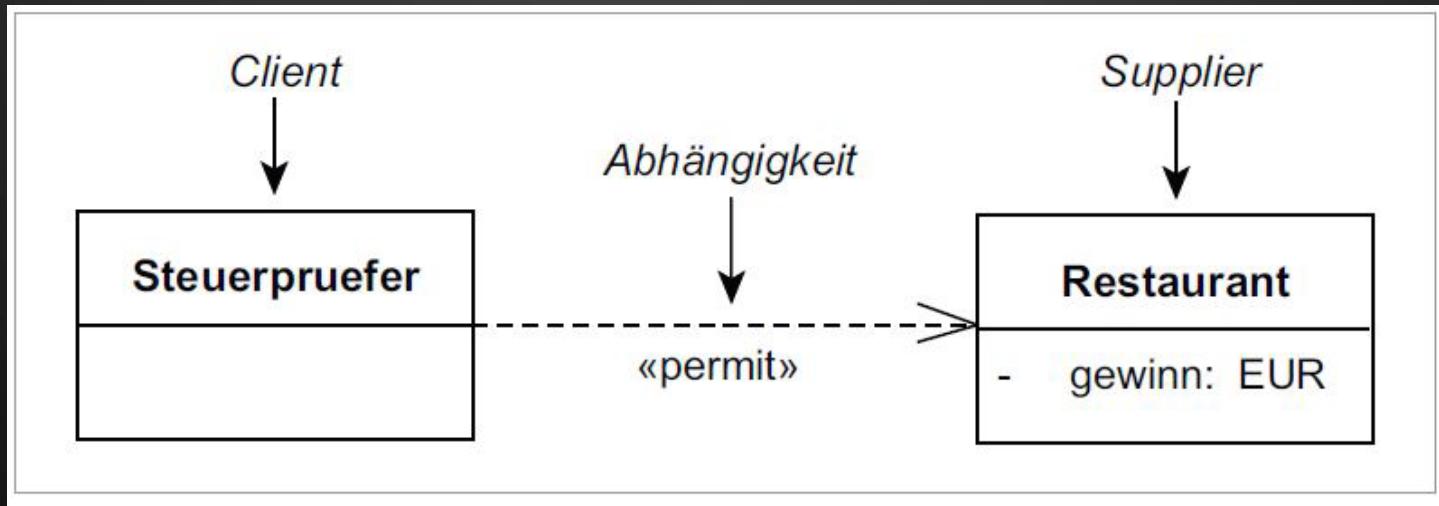


# ABHÄNGIGKEIT

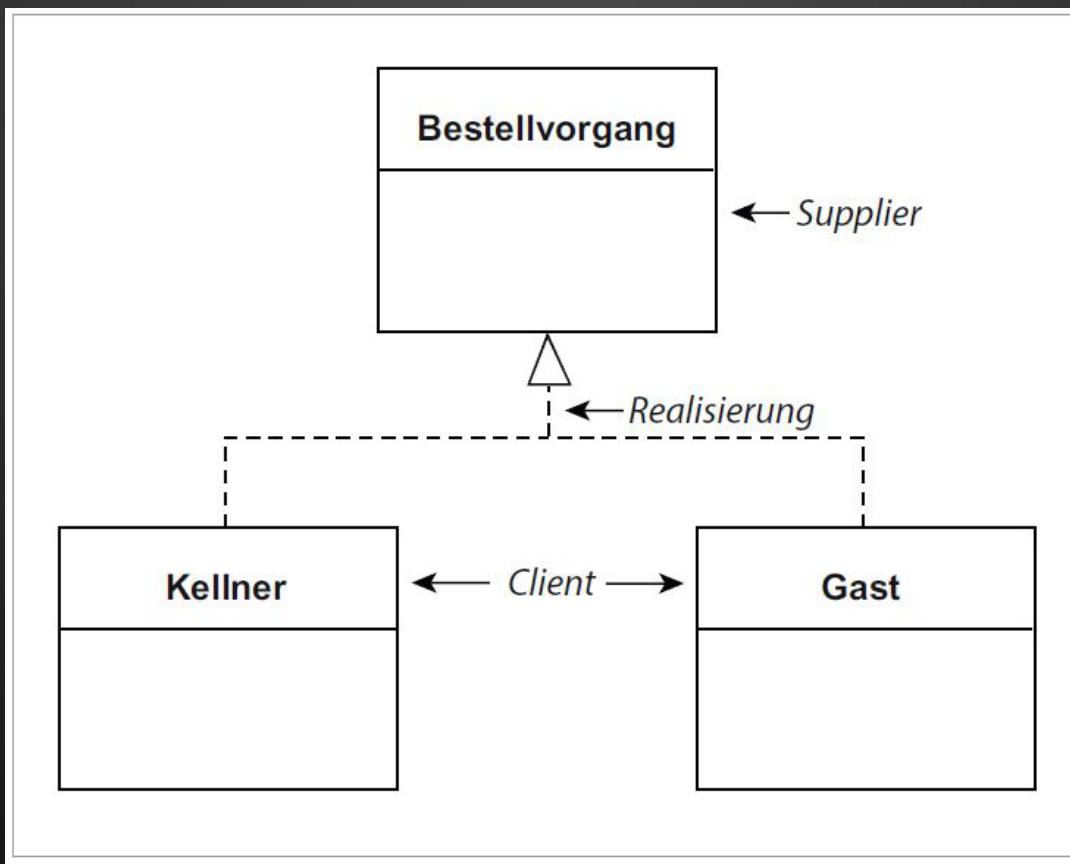
MARKUS SZYNSKA

# ABHÄNGIGKEIT

Eine **Abhängigkeit** (engl. *Dependency*) signalisiert, dass eine Klasse eine andere für ihre eigene Spezifikation oder Implementierung benötigt. Sie wird auch als **Client-Supplier-Beziehung** bezeichnet (**Kunde-Dienstleister**) und dient lediglich der Dokumentation.



# ABHÄNGIGKEIT



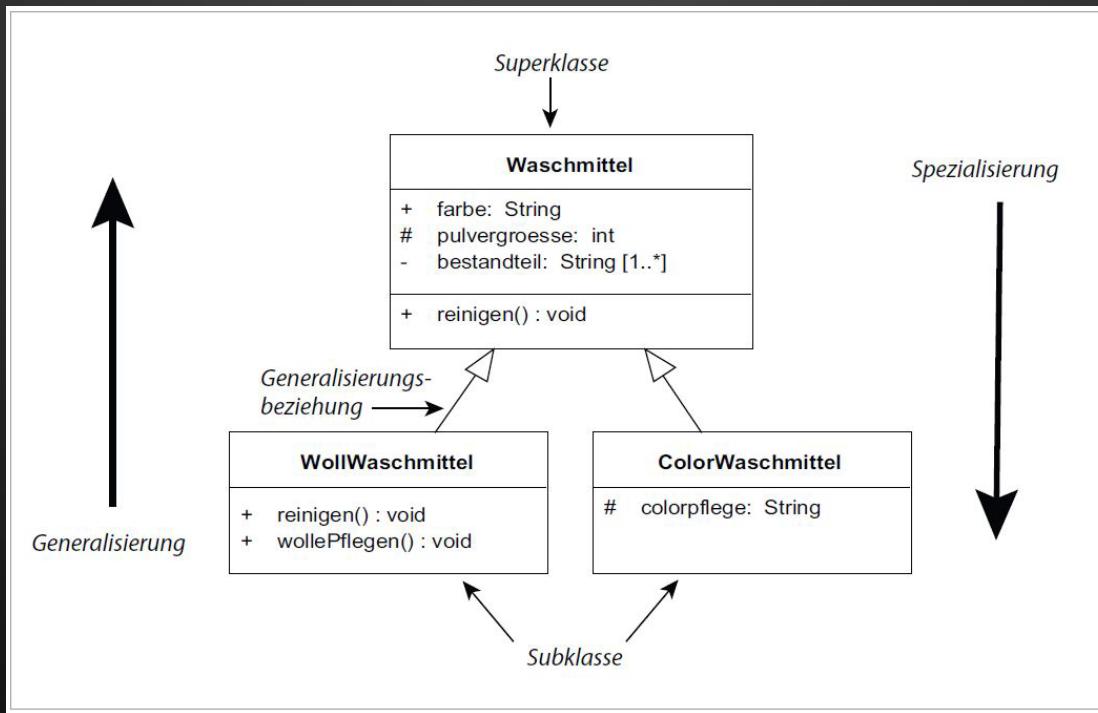


# GENERALISIERUNG

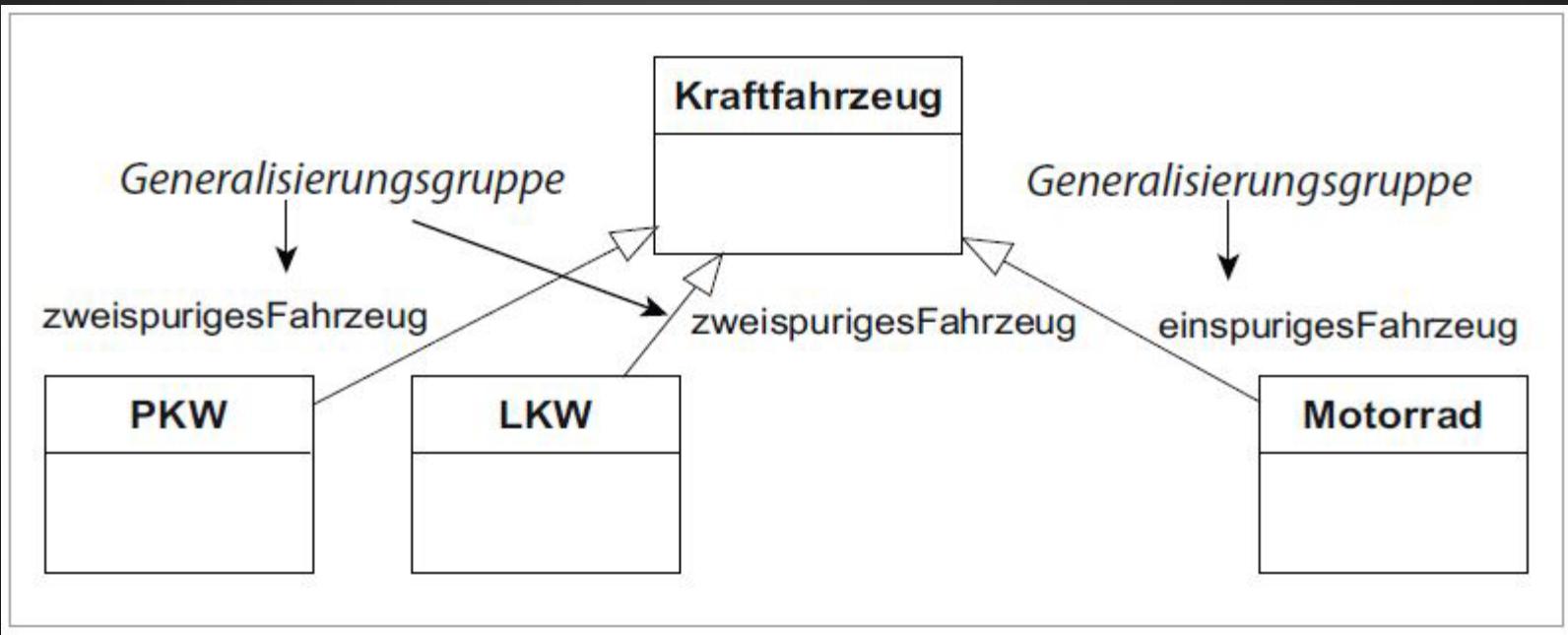
MARKUS SZYNSKA

# GENERALISIERUNG

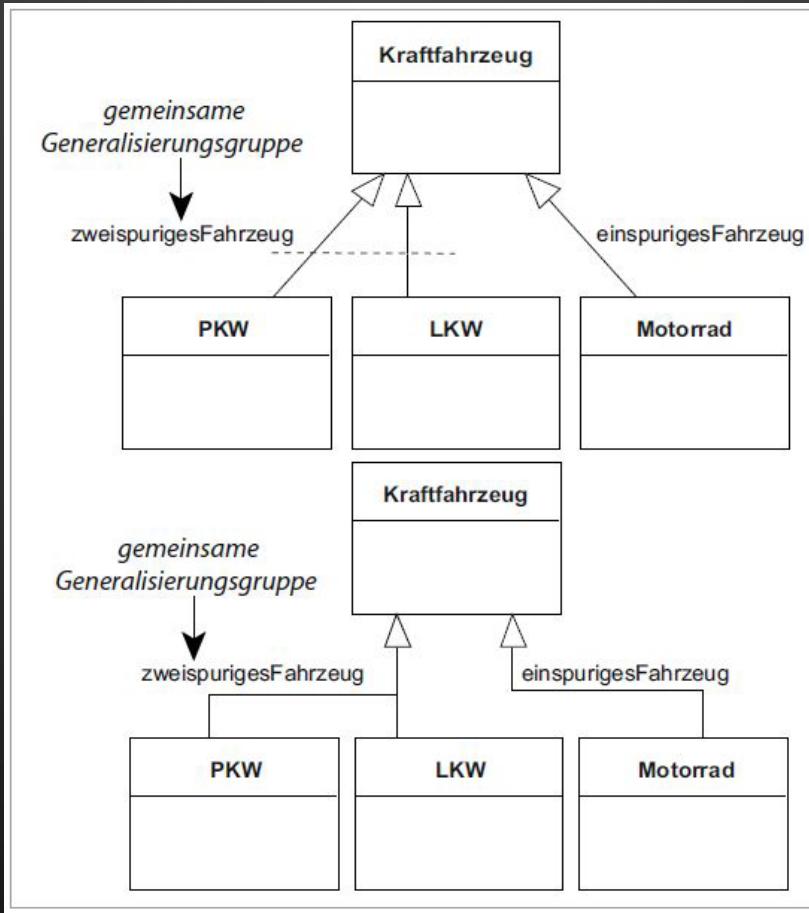
Die **Generalisierung** (engl. *Generalization*) modelliert eine **Beziehung zwischen einer spezifischen Subklasse und einer allgemeinen Superklasse** und definiert damit eines der zentralen Konzepte der Objektorientierung.



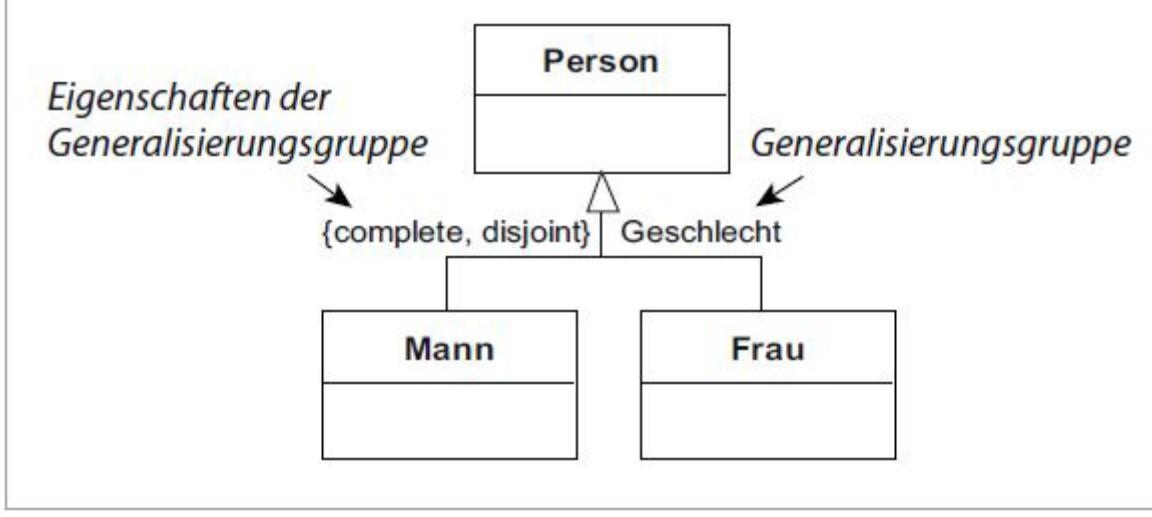
# GRUPPIEREN VON GENERALISIERUNGEN



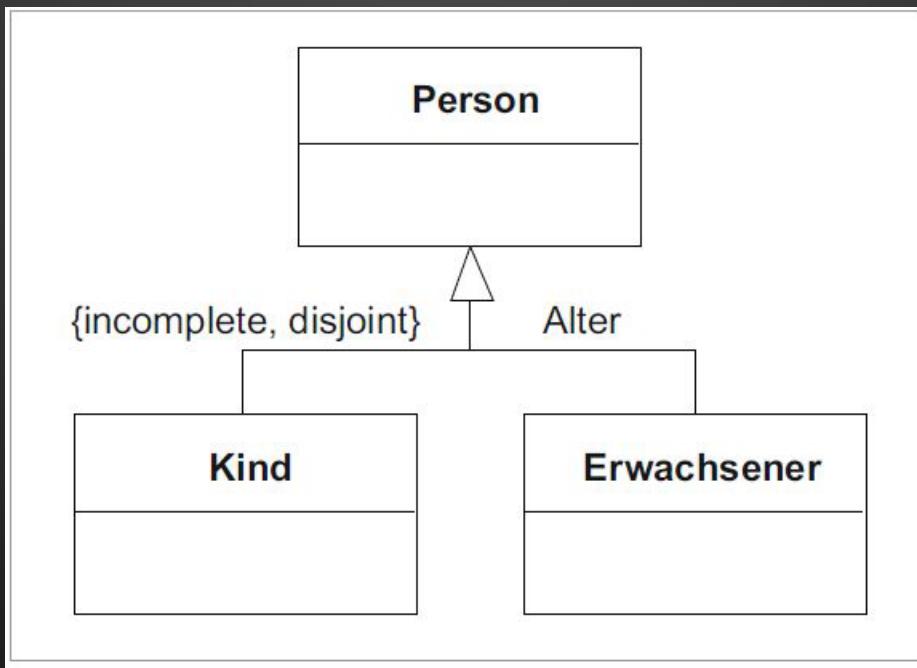
# GEMEINSAME GENERALISIERUNGSGRUPPEN



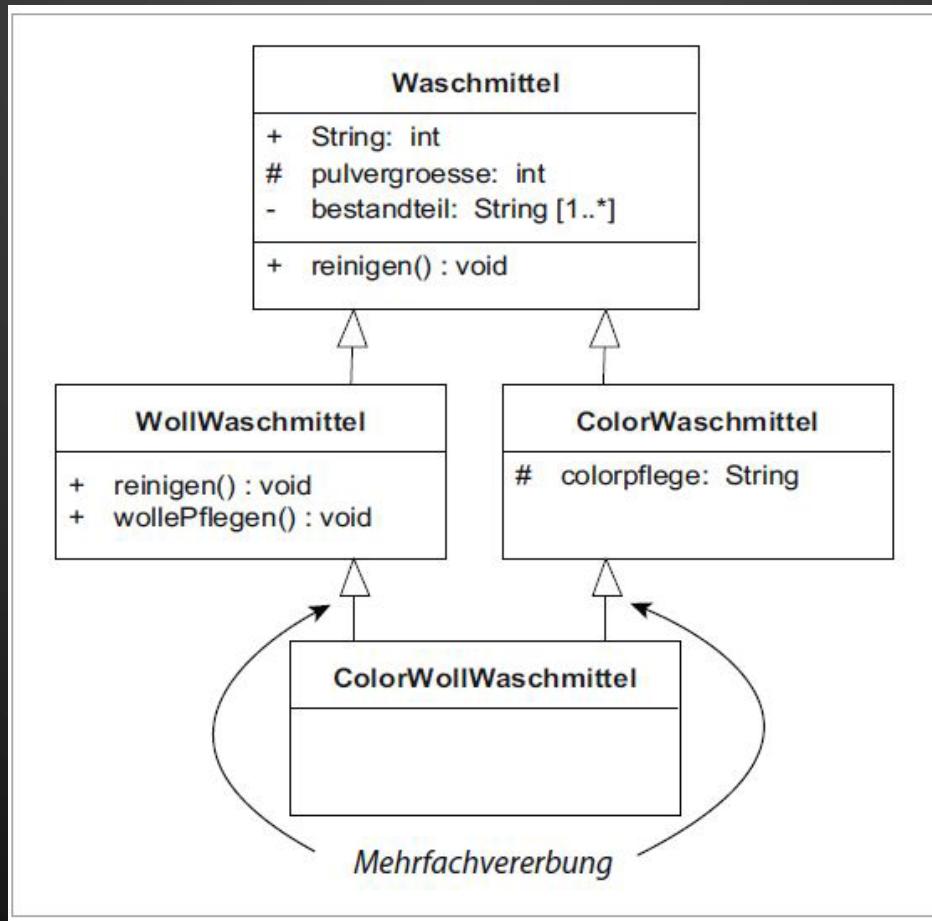
# {complete, disjoint}-Eigenschaft



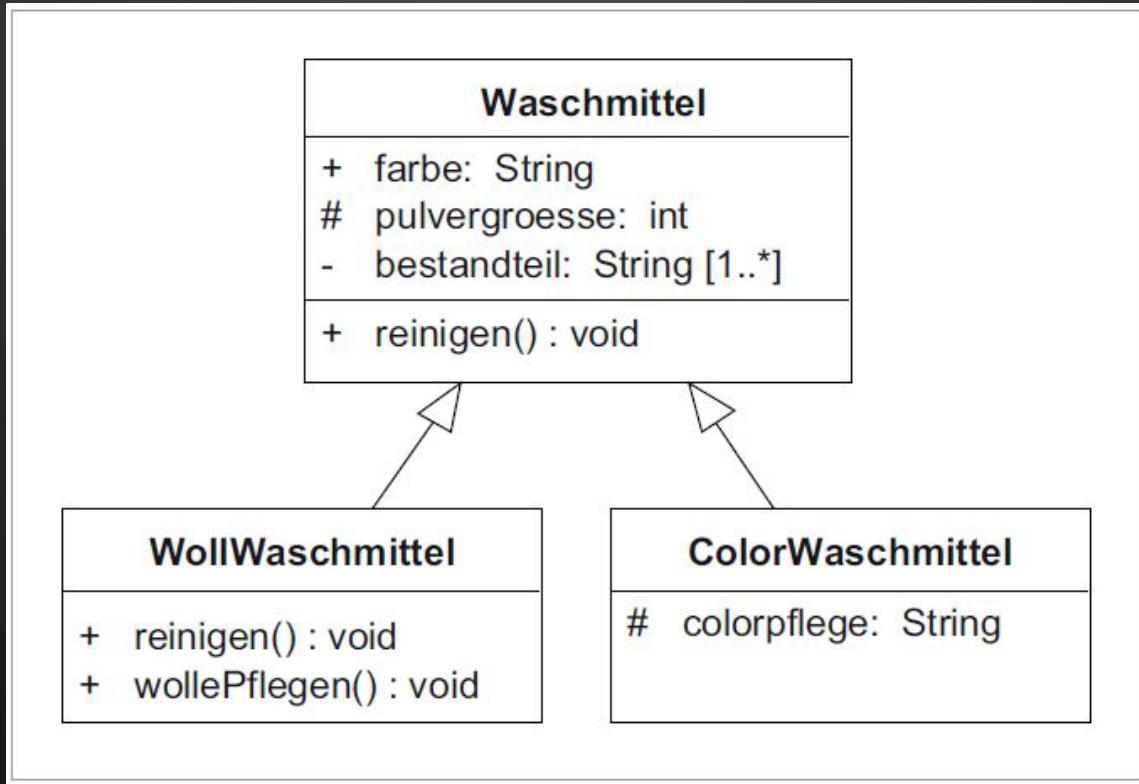
# {incomplete, disjoint}-Eigenschaft



# Mehrfachvererbung

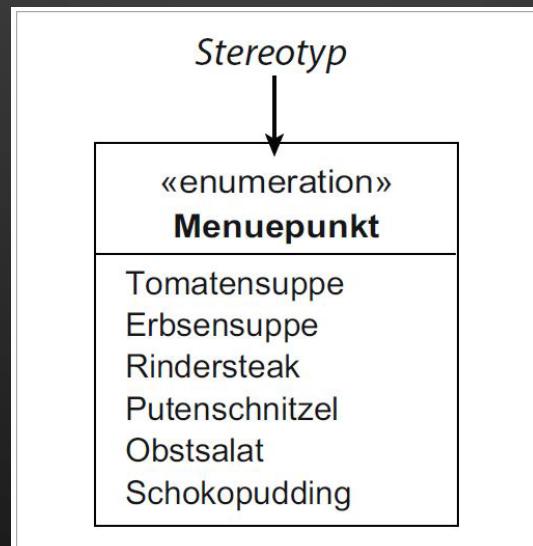


# Generalisierung Java



# Stereotyp

**Stereotypen** (engl. *Stereotypes*) können in der UML in allen Diagrammarten verwendet werden, da sie allgemein die **Art eines Notationselements** spezifizieren. Sie verändern nicht seine semantische Bedeutung, sondern geben eine **Auskunft über seinen Zweck oder seine Rolle** im Modell.



# Stereotyp Beispiel

«utility»  
**Mathematik**

- + PI: double = 3,14159265358 {readOnly}
- + e: double = 2,71828182846 {readOnly}
  
- + kreisFlaeche(radius :double) : double
- + kreisUmfang(radius :double) : double
- + wurzel(basis :double, exponent :double) : double
- + logarithmus(basis :double, numerus :double) : double

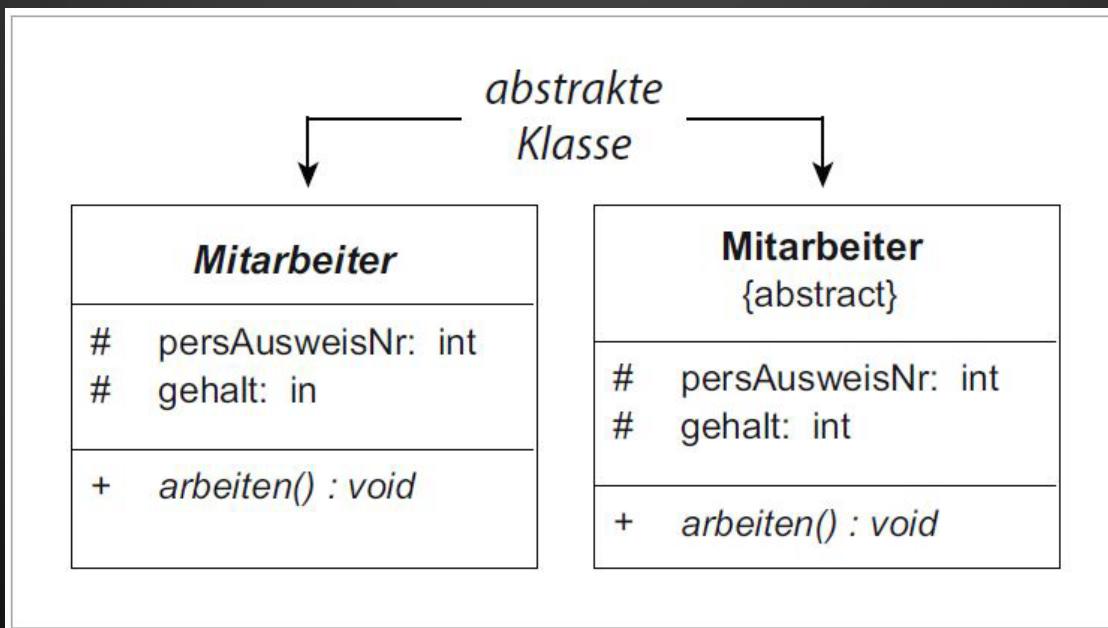


# ABSTRAKTE KLASSEN

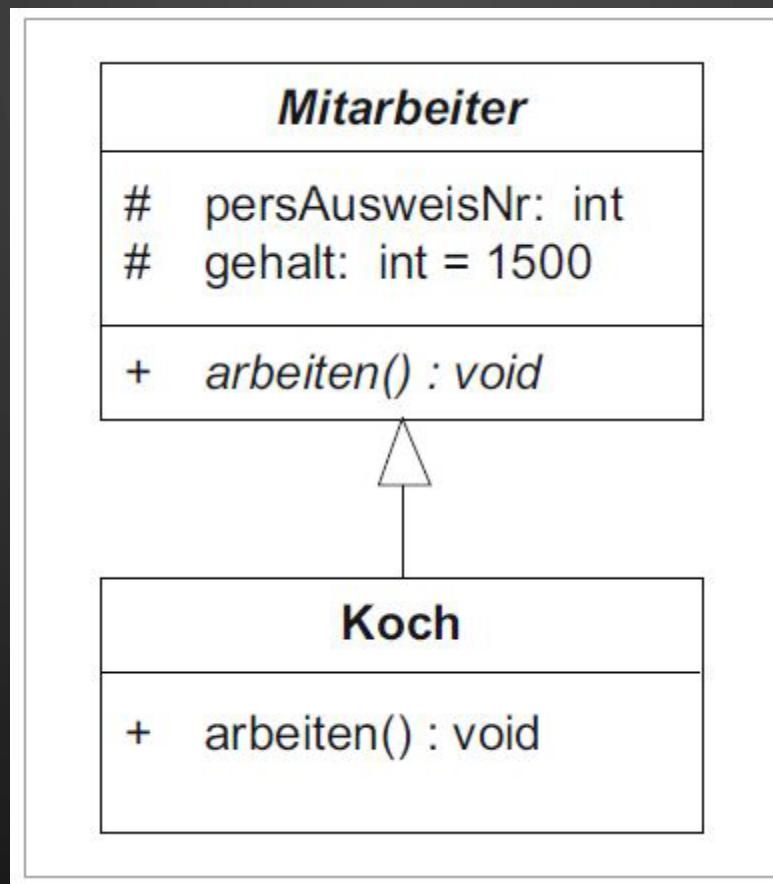
MARKUS SZYNSKA

# Abstrakte Klassen

Der **Bauplan**, den eine Klasse darstellt, kann bewusst **abstrakt** und **unvollständig** modelliert werden, sodass aus ihm **keine Objekte direkt instanziiert werden können**. Solche Klassen werden **abstrakte Klassen** (engl. *Abstract Classes*) genannt und werden in der UML durch eine kursive Schreibweise des Klassennamens oder mit dem Schlüsselwort `{abstract}` gekennzeichnet.



# Abstrakte Klasse in Java





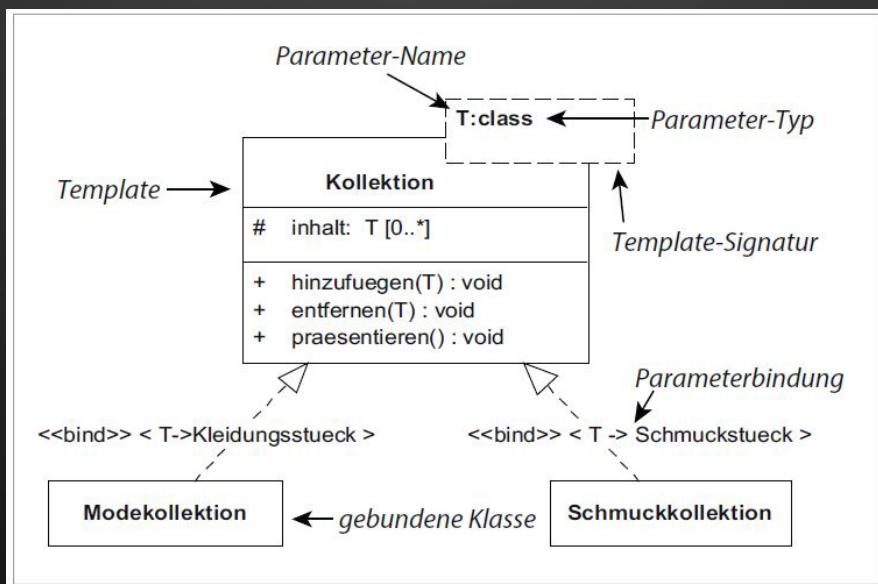
# TEMPLATE - GENERICS

MARKUS SZYNSKA

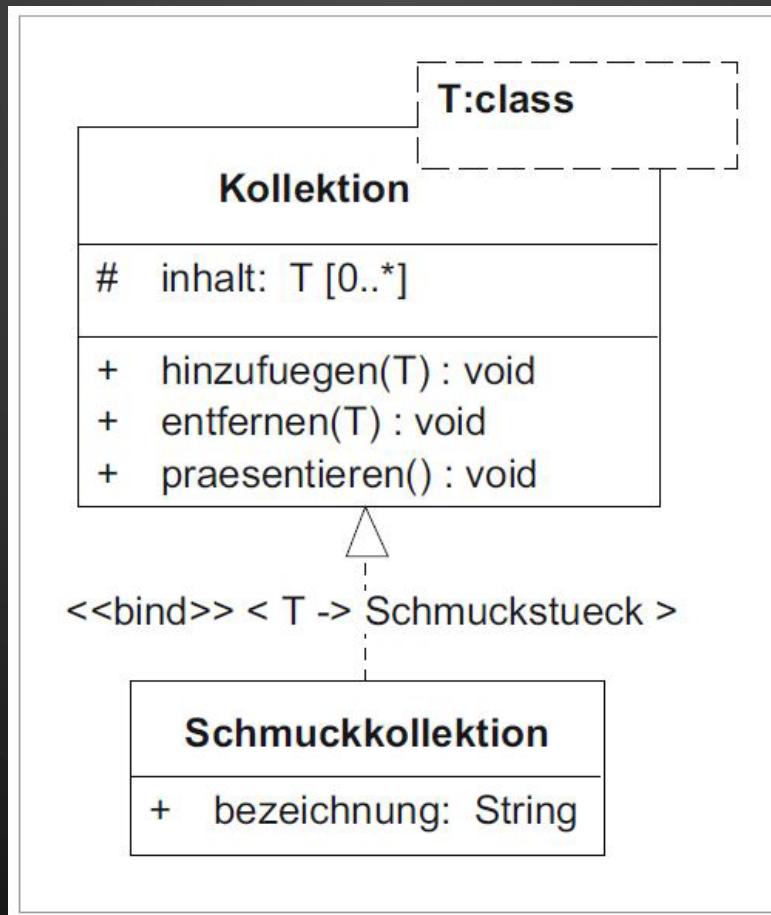
49

# Template - Generics

Ein **Template** (engl. *Template*) geht einen Schritt weiter und erlaubt, dass dieser Bauplan mit generischen Datentypen versehen wird. Sowohl bei Java als auch bei C# wird hierbei der Fachbegriff **Generics** verwendet.



# Template – Generics in Java



# Template – Generics in Java

```
A class Kollektion<T>
{
    B protected ArrayList<T> inhalt;

    C public Kollektion()
    {
        inhalt = new ArrayList<T>();
    }

    D public void hinzufuegen(T elem)
    {
        inhalt.add(elem);
    }

    E public void entfernen(T elem)
    {
        inhalt.remove(elem);
    }

    F public void praesentieren()
    {
        for (T temp : inhalt)
            System.out.println(temp);
    }
}
```

# Template – Generics in Java

```
class Schmuckstueck
{
    public String bezeichnung;
    public Schmuckstueck(String bez)
    {
        bezeichnung = bez;
    }

A  public String toString()
{
    return bezeichnung;
}

}
```

# Template – Generics in Java

```
public static void main(String[] args)
{
    A Schmuckstueck s1 = new Schmuckstueck("Ohrring");
    Schmuckstueck s2 = new Schmuckstueck("Halskette");
    Schmuckstueck s3 = new Schmuckstueck("Diamant-Ring");

    B Kollektion<Schmuckstueck> schmuckKollektion =
        new Kollektion<Schmuckstueck>();

    C schmuckKollektion.hinzufuegen(s1);
    schmuckKollektion.hinzufuegen(s2);
    schmuckKollektion.hinzufuegen(s3);

    D schmuckKollektion.praesentieren();
}
```

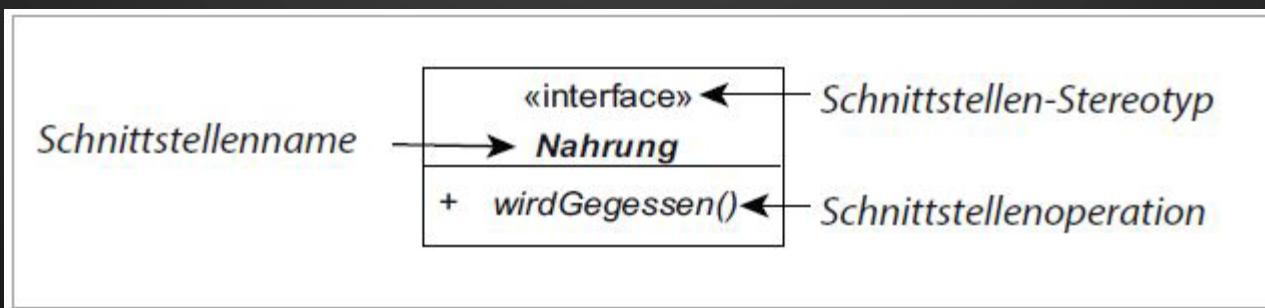


# INTERFACES

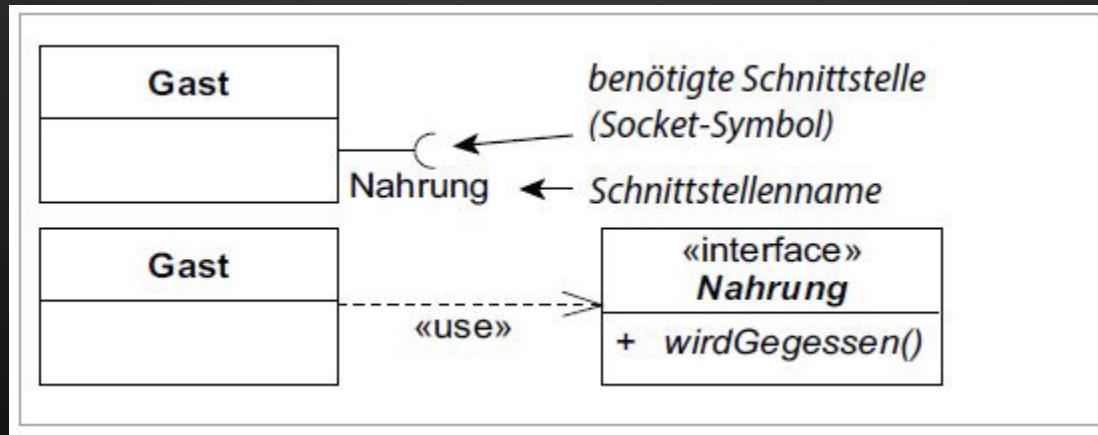
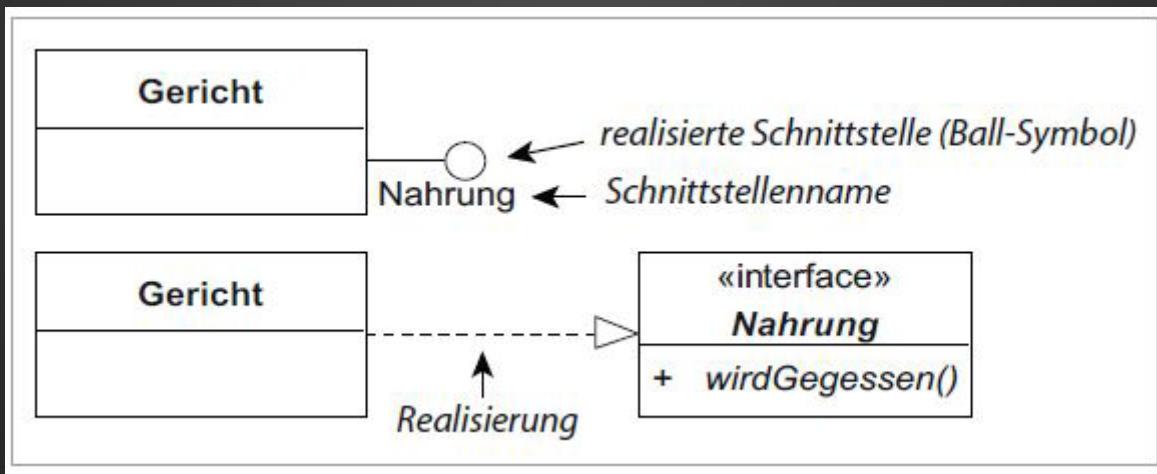
MARKUS SZYNSKA

# Interfaces

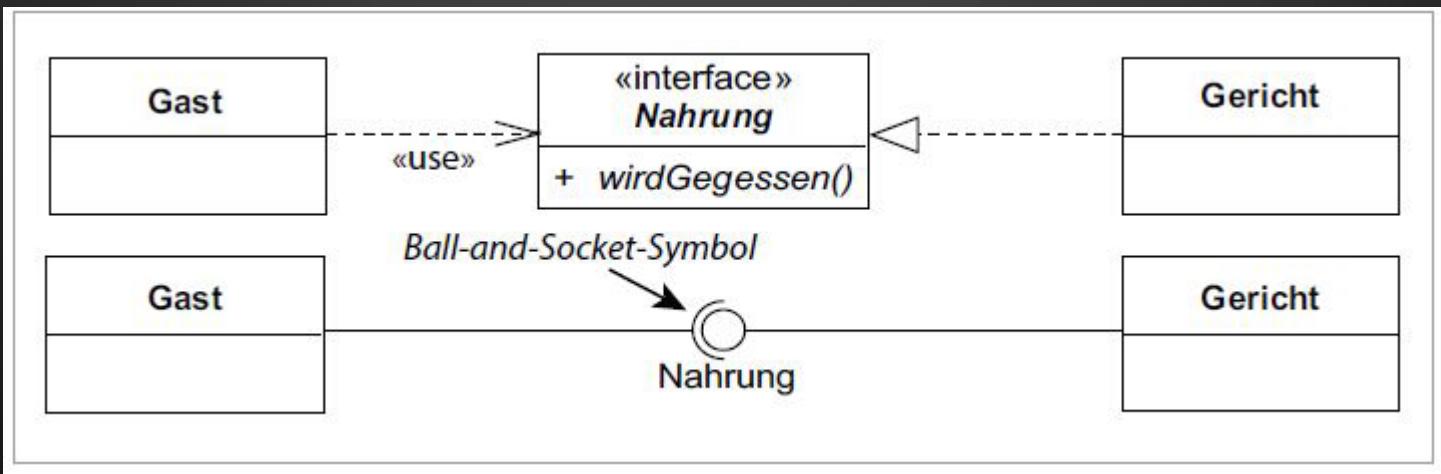
Eine **Schnittstelle** (engl. *Interface*) modelliert eine Art **Vertrag zwischen Klassen**, dessen Vertragsverpflichtungen dabei sowohl Attribute als auch Operationen umfassen können. Sie wird durch ein Klassensymbol mit dem Stereotyp <<interface>> notiert.



# Interfaces



# Interfaces



# Interfaces



# Interfaces Beispiel Java

```
A public interface Nahrung
```

```
{
```

```
B public void wirdGegessen();
```

```
}
```

```
A class Gericht implements Nahrung
```

```
{
```

```
    public String name;  
    public Gericht(String n)
```

```
{
```

```
    name = n;
```

```
}
```

```
B public void wirdGegessen()
```

```
{
```

```
    System.out.println("Gericht "+name+" wird gegessen");
```

```
}
```

```
}
```

# Interfaces Beispiel Java

```
class Gast
{
    A public Nahrung mittagessen;

    public Gast()
    {

        B mittagessen = new Gericht("Wiener-Schnitzel");

    }
}
```

```
public static void main(String[] args)
{
    A Gast gast = new Gast();
    gast.mittagessen.wirdGegessen();

}
```

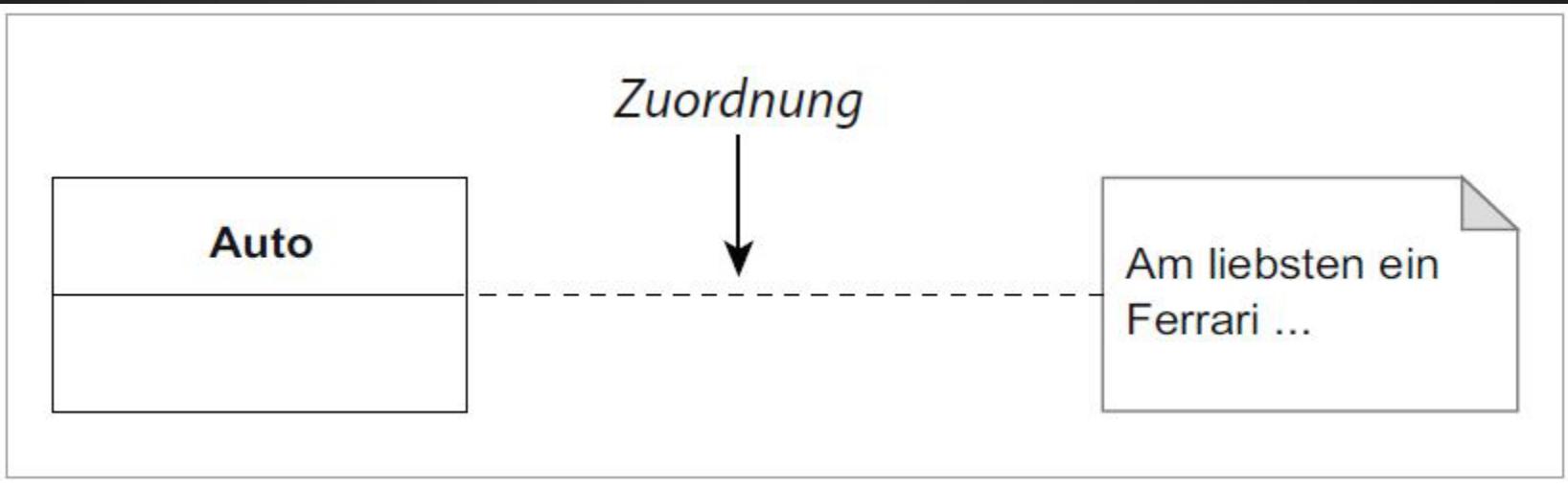


# ANMERKUNGEN

MARKUS SZYNSKA

62

# Anmerkungen

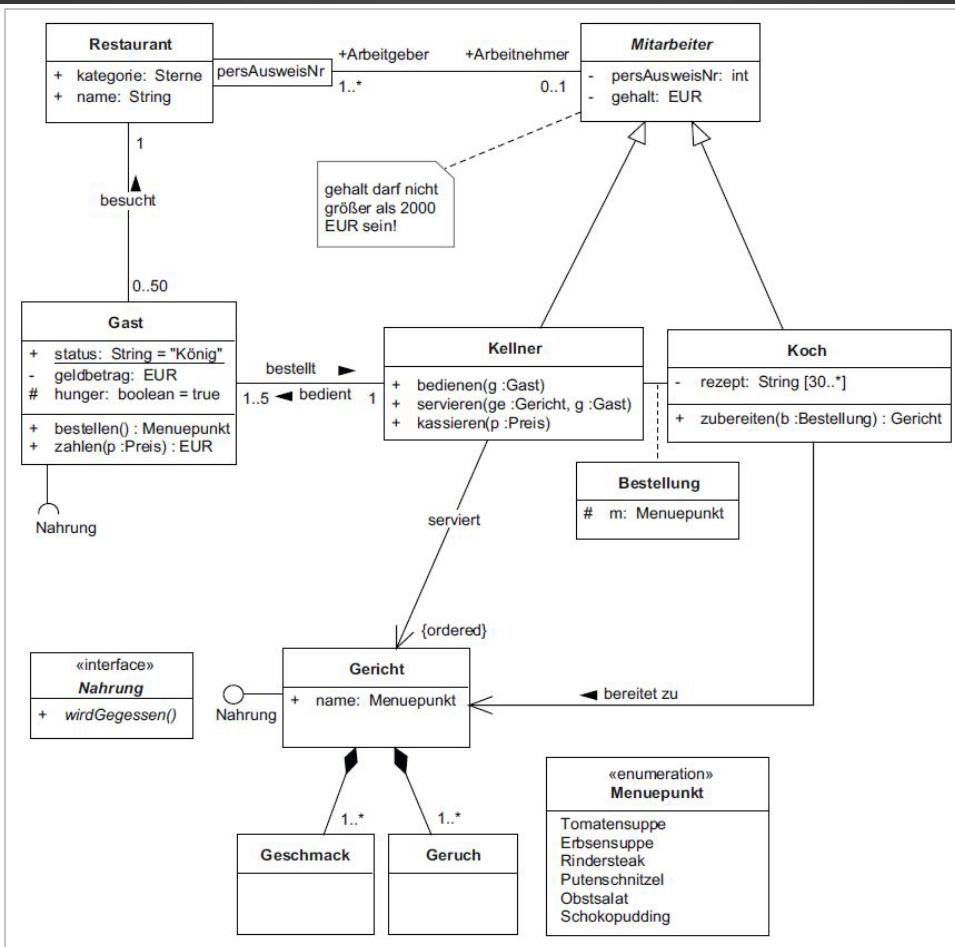




# LESEN EINES KLASSENDIAGRAMMS

MARKUS SZYNSKA

# Lesen eines Klassendiagramms

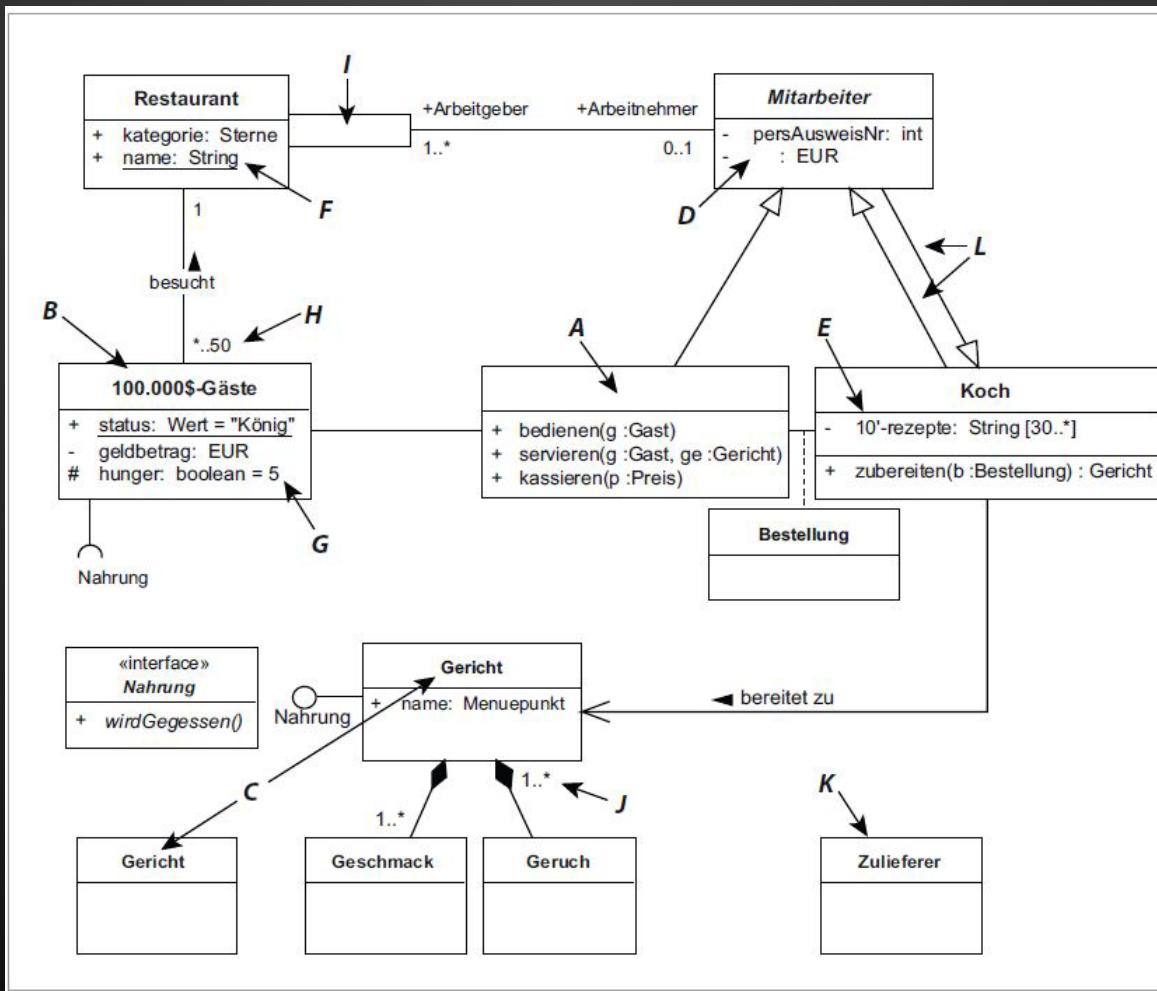




# IRRUNGEN UND WIRRUNGEN

MARKUS SZYNSKA

# Irrungen und Wirrungen





# ZUSAMMENFASSUNG

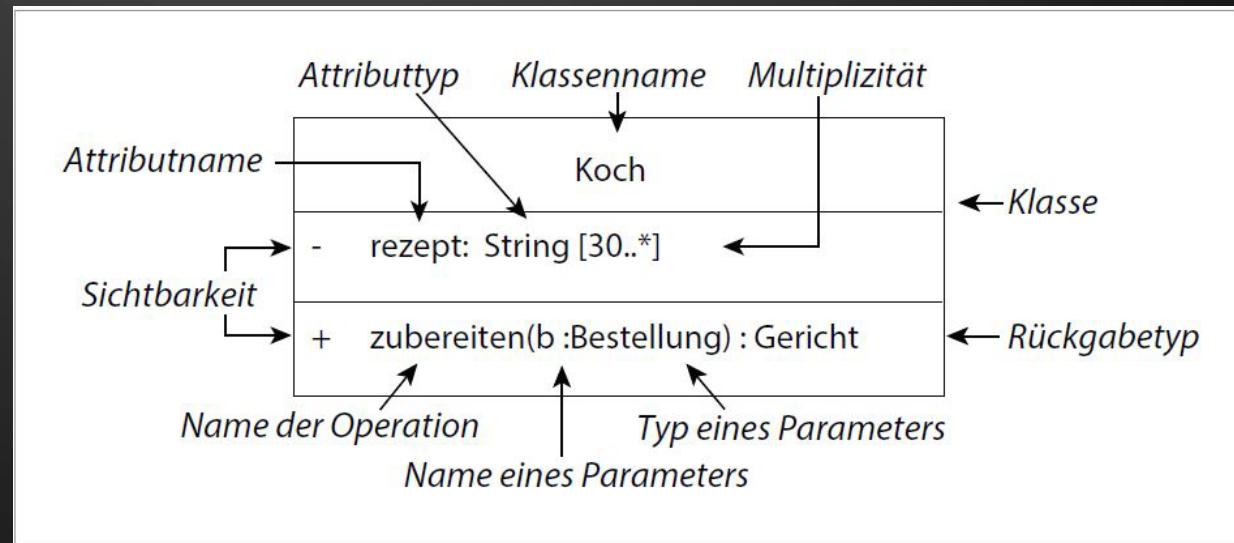
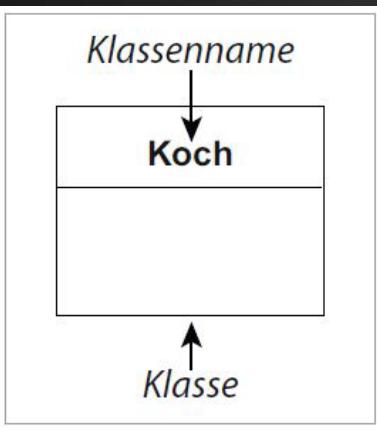
MARKUS SZYNSKA

68

# Zusammenfassung

**Klassen** repräsentieren Baupläne für Objekte.

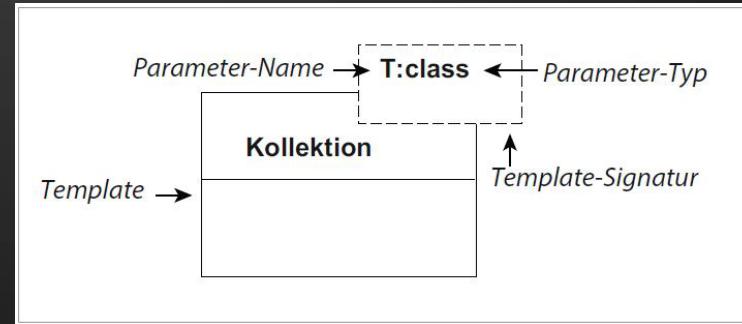
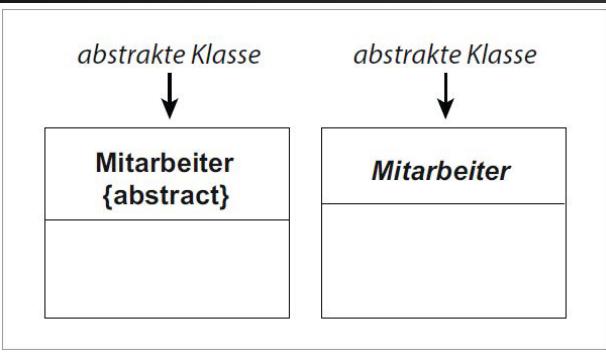
**Attribute** stellen die Eigenschaften der Klasse dar, **Operationen** ihre dynamischen Fähigkeiten



# Zusammenfassung

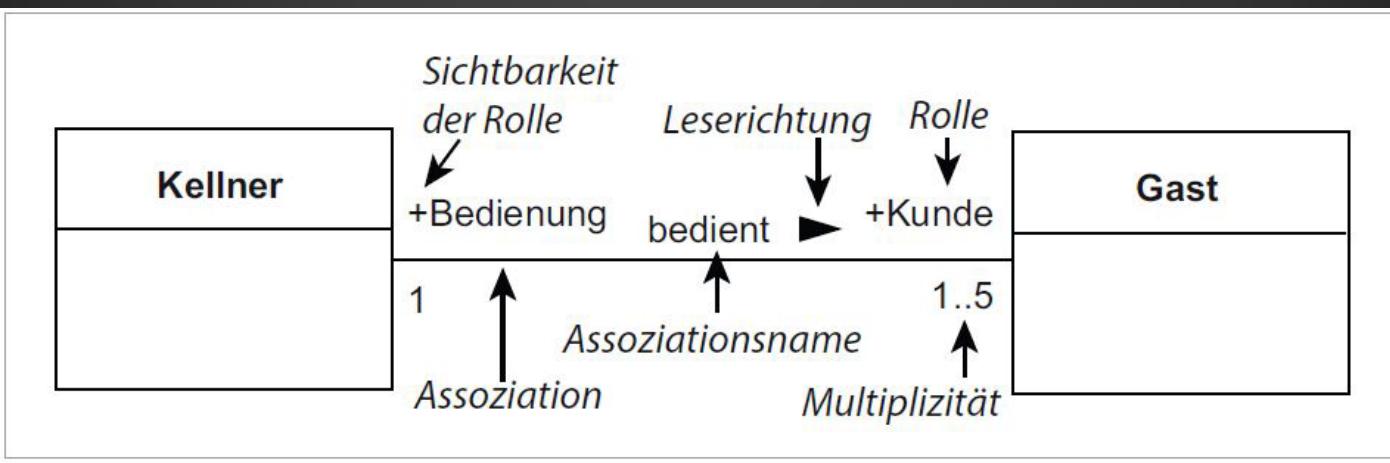
Eine **abstrakte Klasse** definiert einen (möglicherweise unvollständigen) Bauplan auf einer sehr abstrakten Ebene, auf der noch keine konkreten Objekte erzeugt werden können.

**Templates** sind Klassen, bei deren Attributen und Operationen vorab keine Datentypen festgelegt werden müssen. Weil man die Datentypen erst bei der Benutzung spezifiziert, können sie flexibel eingesetzt werden, ohne auf Typsicherheit verzichten zu müssen.



# Zusammenfassung

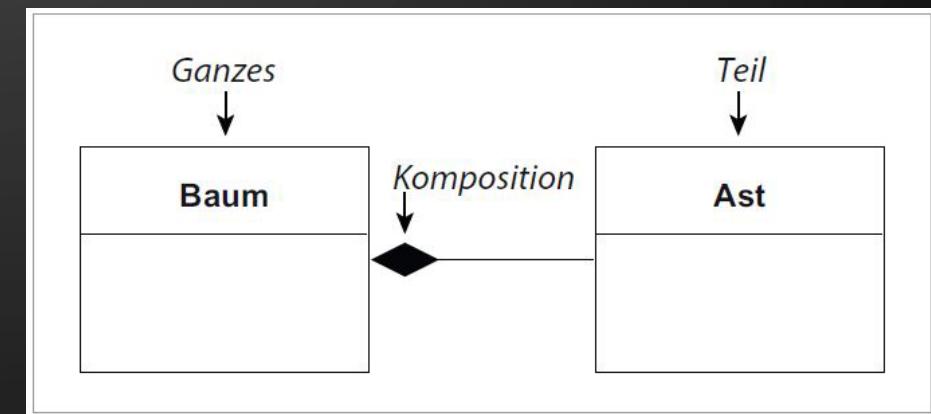
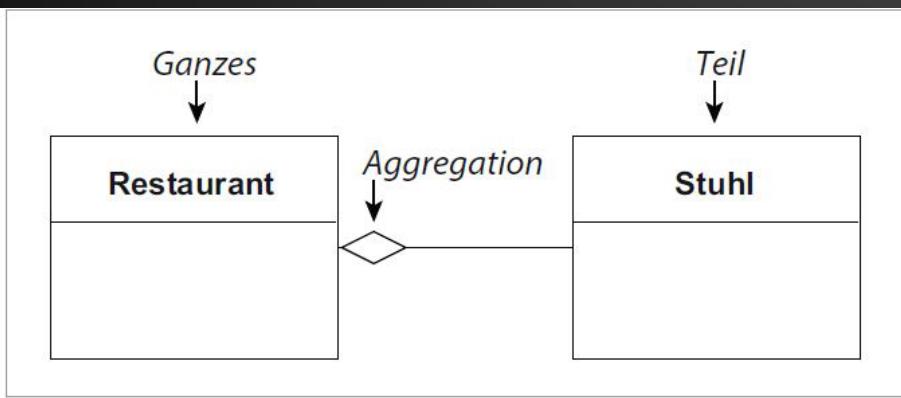
Assoziationen modellieren Beziehungen zwischen Klassen.



# Zusammenfassung

Eine **Aggregation** definiert eine lose Ganzes-Teile-Beziehung zwischen Klassen, bei der die Teile vom Ganzen getrennt werden können, ohne das Ganze zu zerstören.

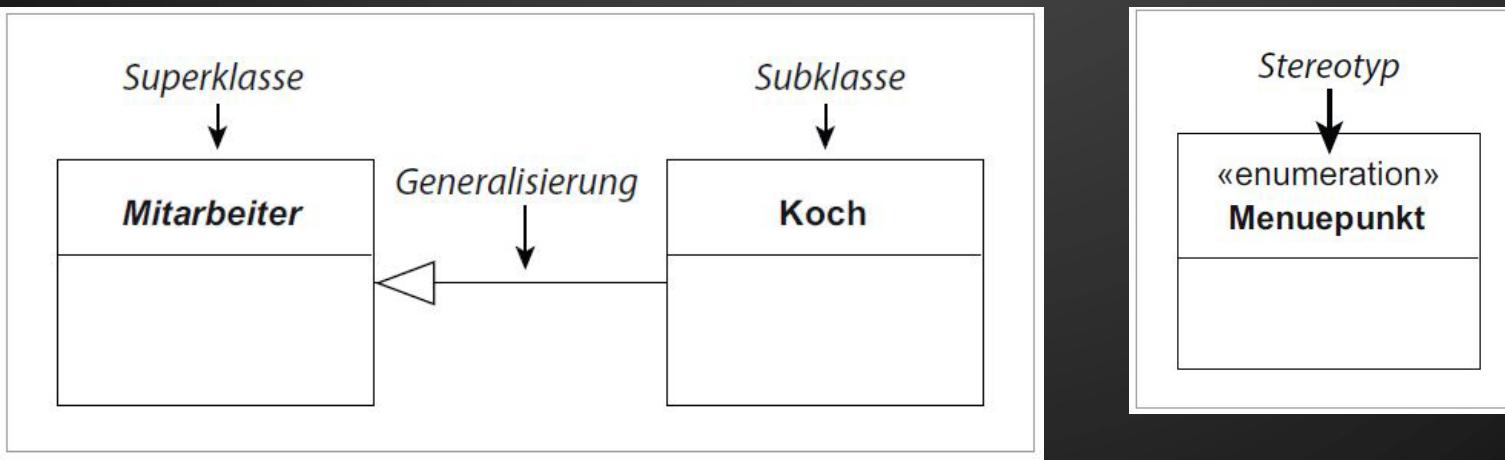
Eine **Komposition** stellt eine starke Form der Aggregation dar, bei der die Verbindung zwischen dem Ganzen und den Teilen als untrennbar angesehen wird.



# Zusammenfassung

Die **Generalisierung** modelliert eine Beziehung zwischen einer spezifischen Subklasse und einer allgemeinen Superklasse.

**Stereotypen** spezifizieren den Zweck oder die Art von Notationselementen.



# Zusammenfassung

Mit einer **Schnittstelle** wird eine Art **Vertrag zwischen Klassen** definiert, dessen Inhalt sowohl Attribute als auch Operationen umfassen kann.

In **Anmerkungen** können beliebige Kommentare platziert werden.

