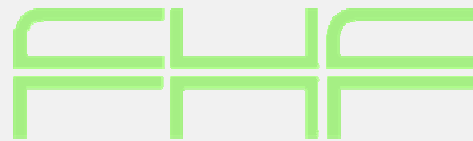


2. Strukturdiagramme

2.1 Das Klassendiagramm



Prof. Mario Jeckle

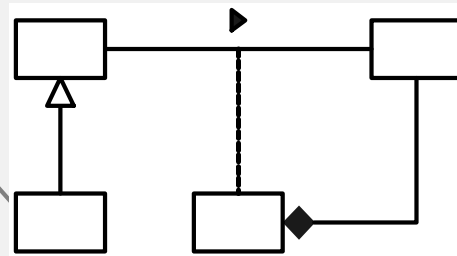
Fachhochschule Furtwangen

mario@jeckle.de

<http://www.jeckle.de>

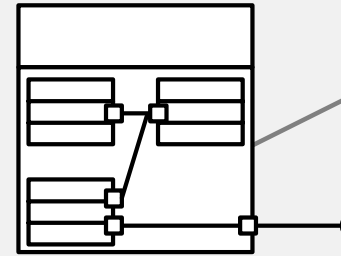
Strukturdiagramme

Aufbaustrukturmodellierung



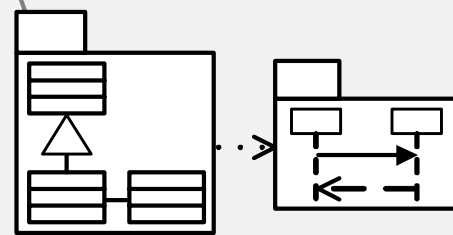
Klassendiagramm

Interaktionsstrukturmodellierung



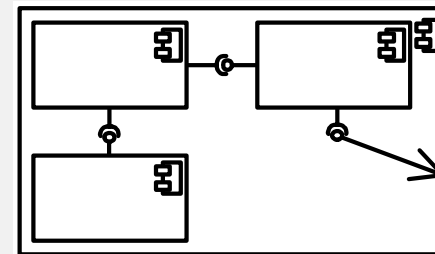
Kompositionsstrukturdiagramm

Ausprägungsmodellierung



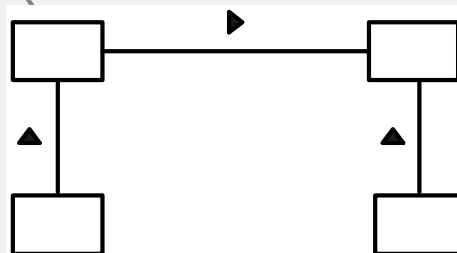
Paketdiagramm

Zusammenhangsstrukturmodellierung

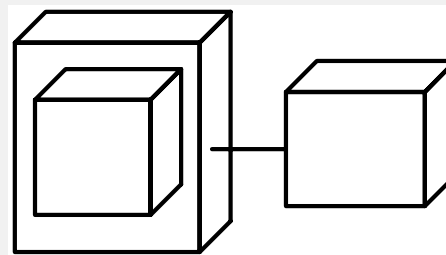


Komponentendiagramm

Einsatzstrukturmodellierung



Objektdiagramm



Verteilungsdiagramm

Was ist das?

Das Klassendiagramm wurde aus dem *Object Model* der Object Modeling Technique (OMT) entwickelt und dient:

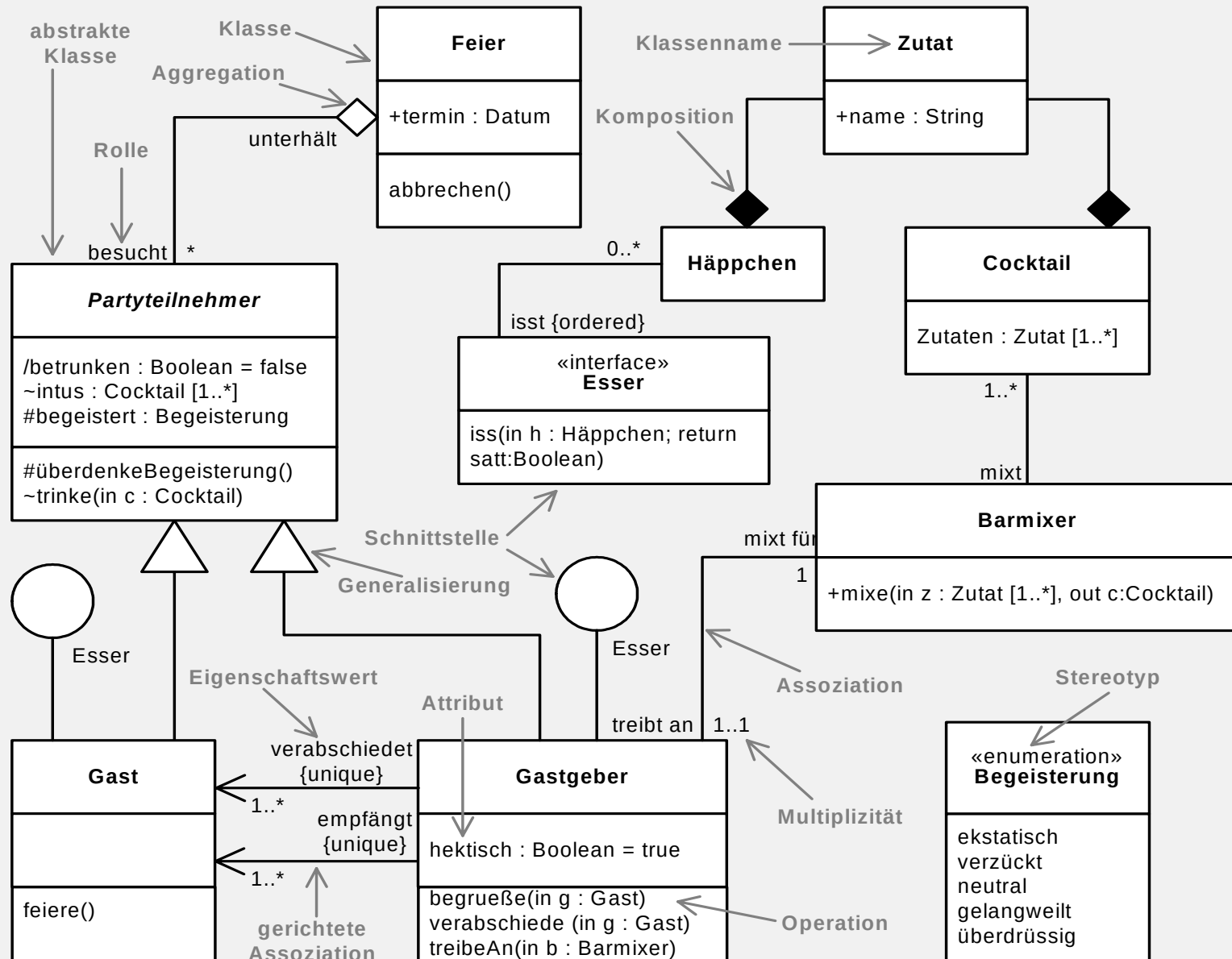
- Der Darstellung der Struktur eines Systems
- Dem Aufzeigen von statischen Eigenschaften und Beziehungen
- Der Sammlung grundlegender Modellierungskonstrukte
- Ist Antwort auf die Frage „Wie sind Daten und Verhalten meines Systems strukturiert?“

Elemente

Das Klassendiagramm besteht aus Elementen:

- Klassen
- Schnittstellen
- Attributen
- Operationen
- Assoziationen mit Sonderformen *Aggregation* und *Komposition*
- Generalisierungsbeziehungen
- Abhängigkeitsbeziehungen

Elemente im Überblick

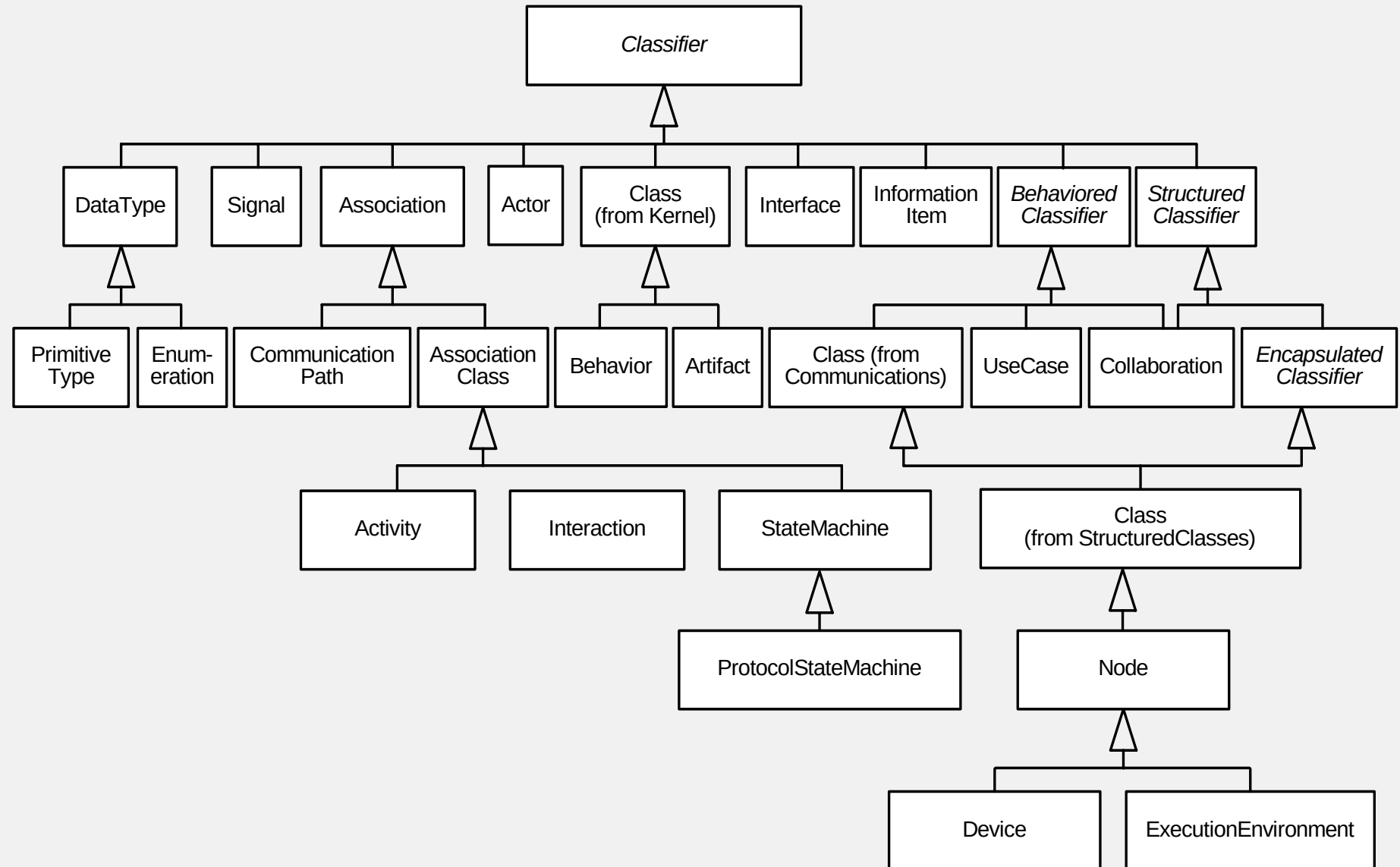


Classifier

Der *Classifier* ist zentraler Bestandteil des Klassendiagramms und anderer UML-Diagrammtypen:

- Zentraler Bestandteil der UML
- Abstraktion der *Klasse*
- Ist kein durch den Modellierer direkt nutzbares graphisches Konstrukt
- Kann einem anderen Classifier zugeordnet sein
- Ist die ordnende Instanz bei einer Reihe konkreter Ausprägungen
- Es existieren 27 Varianten

Classifier-Hierarchie



Anwendung Klassendiagramm

Die Anwendung des Klassendiagramms im Projekt:

- Ist in allen Projektphasen möglich
- Das Klassendiagramm ist hier das wichtigste Diagramm
- Von ersten Analyseschritten bis zur Codierung von Softwaresystemen
- Das Klassendiagramm ist zwar ein vordefinierter Diagrammtyp, jedoch in seiner Anwendung in zwei Einsatzfälle einteilbar
 - konzeptuell-analytisch
 - logisch

Anwendung

Konzeptuell–analytische Modellierung:

Ziel ist die korrekte Erfassung und Abbildung von vorgefundenen Zusammenhängen

WICHTIG: grundlegende Wesenseinheiten und Zusammenhänge eines Systems

UNWICHTIG: Technische Realisierungsdetails wie Attributdatentypen, technische Operationen, Schnittstellen

Anwendung

Logische Modellierung:

Ziel ist den Inhalt so zu erfassen, dass er alle Information für eine direkte Quellcode- Erstellung enthält, da konkrete Implementierungsvorschrift

WICHTIG: Detailgenauigkeit
(Voraussetzung der Quellcodeerzeugung)

Notation

Im Klassendiagramm finden graphische Primitive Anwendung:

- Klassen
- Attribute
- Operationen
- Schnittstellen
- Parametrisierte Klassen
- Generalisierung
- Assoziationen
- Assoziationsklassen
- Kommentar
- Stereotype
- Eigenschaftswerte

Klassendiagramm: Begriffe

- Klasse: beschreibt eine Menge von Instanzen die dieselben Eigenschaften, Einschränkungen und Semantik haben
- Objekt: Ausprägung ein oder mehrerer Klassen
- Attribut: Eigenschaften einer Klasse, die Charakteristika wie Auftretenshäufigkeit, Datentyp etc. beschreiben
- Wert: UML-Ausprägung als Wert eines Attributs oder Objektes
- Operation: Verhaltensmerkmal eines Classifiers, das Name, Typ, Parameter und Zusicherung für den Aufruf eines Verhaltens spezifiziert.
Ähnlich einer Signatur in einer Programmiersprache

Klassendiagramm: Begriffe

- Methode: Implementierung einer Operation
- Assoziation: beschreibt eine Menge von Tupeln von getypte Instanzen
- Link: Ausprägung einer Assoziation
- Sichtbarkeit: definiert, welche anderen Systemkomponenten den Wert eines Attributs lesen und schreiben bzw. eine Methode ausführen dürfen
- Assoziationsklasse: vereinigt Eigenschaften einer Klasse und einer Assoziation
- Multiplizität: Definition eines Intervalls der erlaubten Kardinalitäten eines Elements

Klassen

Klassenname in Rechteck fett und mittig dargestellt

Klassenname

- Zentrales Element
- Beschreibt Menge von Objekten mit gemeinsamen Werten
- Sichtbare Darstellung von Attributen und/oder Operationen

Klassenname

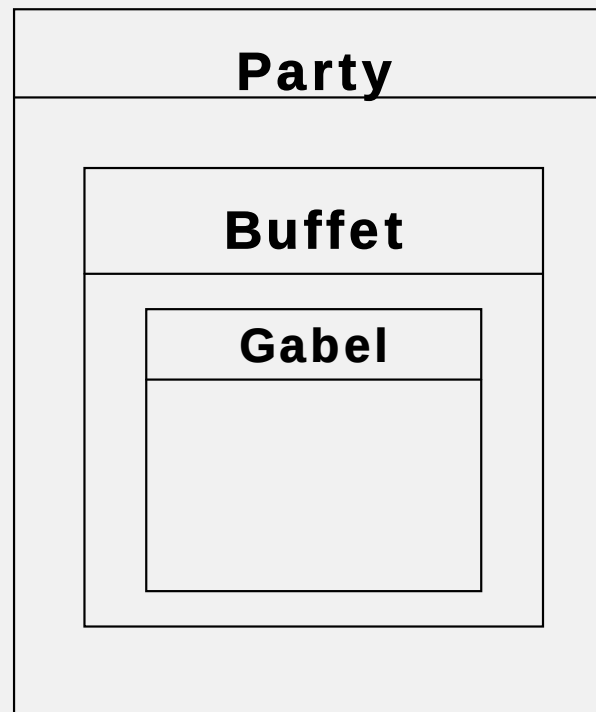
attribut

operation ()

Klassen

Die Schachtelung von Klassen ist beliebig möglich

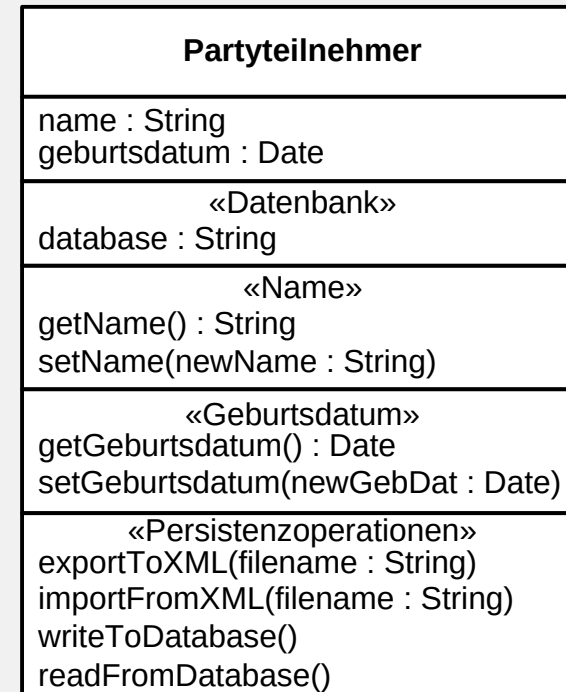
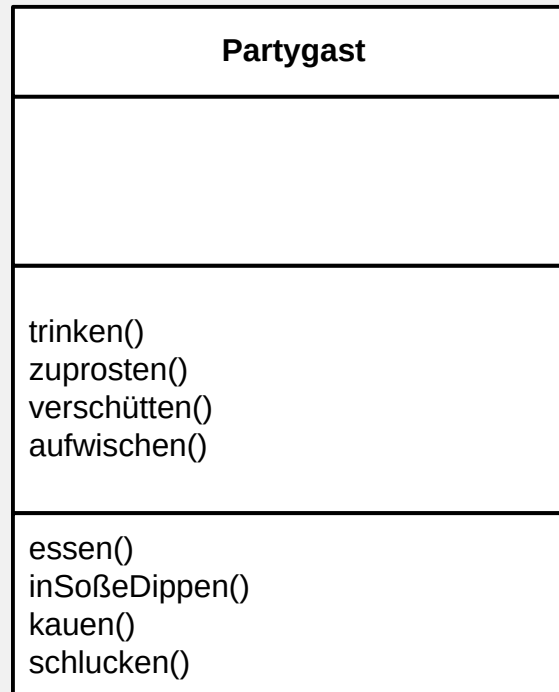
➡ Eingebettete Klasse: hat keine Außenbeziehung



hier: Gabel nur
Beziehung zu
Buffet

Klassen

- Anzahl und Inhalt der Rechteckelemente NICHT beschränkt
- Operationen und Attribute können gruppiert und in eigenen Darstellungselementen untergebracht werden



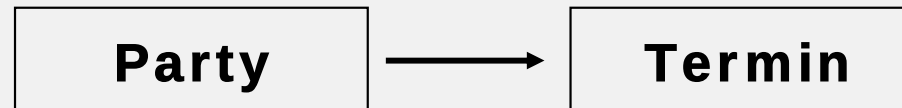
Attribut

Notationsmöglichkeiten: **implizit**



Das Attribut
linksbündig in
Rechteckselement

explizit



Das Attribut
als eigenständige
assoziierte Klasse

Attribut

Das Attribut wird durch einen klassenweit eindeutigen Namen spezifiziert!

Allgemeine Syntax der Attributdeklaration:



Attribut

- Jedes Attribut kann vorgabegemäß genau einen Wert aufnehmen
- Unterstützung fehlender Werte (Verwaltung von NULL-Werten) muss durch Multiplizität [0..] gefordert werden
- Für jedes Attribut wird Speicherplatz in der Klassenausprägung (d.h. im Objekt) vorgesehen.
Ausnahmen hiervon sind *statische Attribute*, deren Wertbelegung in der Klasse selbst abgelegt wird.

Attribut

Bedeutung der Komponenten:

/ = abgeleitetes Attribut. Muss nicht gespeichert werden, da es zur Laufzeit aus anderen Daten berechnet werden kann

Name = Attributname, frei wählbar aus Zeichensatz

Typ = Datentyp des Attributs

Multiplizität = legt Unter- und Obergrenze der Anzahl der Ausprägungen fest, die unter einem Attributnamen abgelegt werden können

Attribut

Zugelassene Attributspezifikationen sind:

Attributdeklaration	Anmerkung
public zähler : int	Umlaut erlaubt
/alter	Datentyp nicht zwingend
private adressen: String [1...*]	Menge von Strings
protected bruder: Person	Datentyp kann andere Klasse, Schnittstelle oder selbstdefinierter Typ sein

Attribut

Nicht zugelassene Attributspezifikationen sind:

Attributdeklaration	Fehlerursache
String	Attributname fehlt
private, public name: String	Mehr als eine Sichtbarkeitsdefinition
public / int	Attributname fehlt
private dateiname: String=„temp“ lock=exclusive	Eigenschaftswert nicht in { }
public int zähler	Name und Typ vertauscht

Attribut

Die UML sieht symbolische Schreibweisen für die Sichtbarkeit von Attributen und Operationen vor:

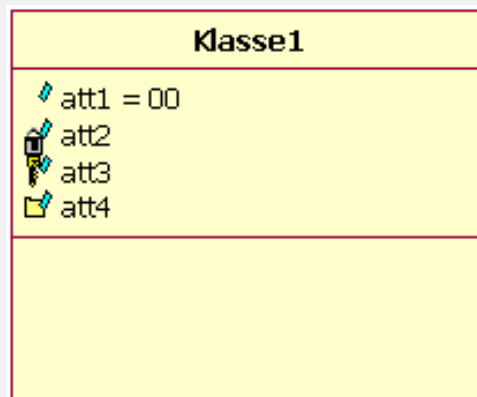
- **public**
Jede andere Systemkomponente hat uneingeschränkten Zugriff
- **private**
Nur Ausprägungen der das Attribut oder die Operation beherbergenden Klasse dürfen zugreifen
- **protected**
Das Attribut, die Operation, ist nur in der definierenden und denjenigen Klassen sicht- und zugreifbar, die als Spezialisierungen von ihr definiert sind
- **package**
Das Attribut, die Operation, ist für alle Klassen zugreifbar, die sich im selben Paket finden, wie die definierende Klasse

Attribut

Die UML sieht symbolische Schreibweisen für die Sichtbarkeit von Attributen und Operationen vor:

- + public
- private
- # protected
- ~ package

Sichtbarkeitseinschränkungen können auch werkzeugspezifisch (d.h. unabhängig vom Standard) durch Symbole dargestellt werden:



Attribut

Umsetzungsbeispiele

AttMix

```
att1 : int
att6 : KlasseB [0..1] {composite}
att7 : String [0..*] {ordered}
/ att8
public:
    att2 : int
    pi : double = 3.1415
private:
    att3 : boolean
protected:
    att4 : short
package:
    att5 : String = "Test" {readOnly}
```

Attribut Umsetzung in C++

AttMix
att1 : int att6 : KlasseB [0..1] {composite} att7 : String [0..*] {sequence} / att8 public: att2 : int <u>pi : double = 3.1415</u> private: att3 : boolean protected: att4 : short package: att5 : String = "Test" {readOnly}

```
class AttMix {  
    int att1;  
    KlasseB *att6;  
    vector<string> att7;  
    public:  
        int att2;  
        static double pi;  
    static const string att5;  
    private:  
        bool att3;  
    protected:  
        short att4;  
};  
const string AttMix::att5("Test");  
double AttMix::pi=3.1415;
```

Attribut Umsetzung in Java

AttMix

```
att1 : int
att6 : KlasseB [0..1] {composite}
att7 : String [0..*] {sequence}
/ att8
public:
  att2 : int
  pi : double = 3.1415
private:
  att3 : boolean
protected:
  att4 : short
package:
  att5 : String = "Test" {readOnly}
```

```
class AttMix {
    int att1;
    public int att2;
    public static double pi=3.1415;
    private boolean att3;
    protected short att4;
    final String att5 = "Test";
    KlasseB att6;
    java.util.Vector<String> att7;

    Object getAtt8() {
        //Berechne Wert für att8
        return wert;
    }
}
```

Attribut Umsetzung in C#

AttMix
att1 : int att6 : KlasseB [0..1] {composite} att7 : String [0..*] {sequence} / att8 public: att2 : int pi : double = 3.14 private: att3 : boolean protected: att4 : short package: att5 : String = "Te

```
using System;
using System.Collections.Specialized;

class AttMix {
    int att1;
    public int att2;
    public static double pi=3.1415;
    private bool att3;
    protected short att4;
    internal const string att5 = "Test";
    KlasseB att6;
    System.Collections.Specialized.StringCollection att7;
    Object att8 {
        //Berechne Wert für att8
        get{return wert;}
    }
}
```

Operation

Operationen einer Klasse werden mindestens durch Namen und wahlfreie weitere Angaben dargestellt:



Abstrakte Attributspezifikation:

[Sichtbarkeit] Name (Parameterliste) : Rückgabetyp

[{Eigenschaftswert}]

Parameterliste:

[Übergaberichtung] Name : Typ ['[Multiplizität]']

[= Vorgabewert] ['{Eigenschaftswert}']

Operation

Bedeutung der Komponenten:

Sichtbarkeit = Regelt Sicht- und Zugreifbarkeit auf Methoden

Name = Klassenweit eindeutige Benennung der Operation

Parameter = Aufzählung verarbeiteter Parameter
(Hierunter fallen: Übergabe und Rückgabeparameter)

Operation

Übergaberichtung	Parameter wird ausgelesen, schreibend verwendet oder gelesen, verarbeitet und neu geschrieben
Name	Operationsweit eindeutig
Typ	Datentyp des Parameters
Multiplizität	Wie viele Inhalte umfasst der Parameter
Rückgabetyt	Datentyp oder Klasse, der nach Operationsausführung zurück geliefert wird
Eigenschaftswert	Besondere Charakteristika des Attributs

Operation Umsetzungsbeispiele

OpMix
<u>+ op1()</u> - op2(in param1 : int=5) : int {readOnly} # op3(inout param2 : KlasseC) ~ op4(out param3 : String[1..*] {sequence}) : KlasseB

Operation Umsetzung in C++

+ op
- op
op
~ op

```
class OpMix {
    KlasseB op4(vector<string> param3) {
        //Implementierung
        return wert;
    }
public:
    static void op1() {
        //Implementierung
    }
private:
    int op2(const int param1=5) const {
        //Implementierung
        return wert;
    }
protected:
    void op3(KlasseC* param2) {
        //Implementierung
    }
};
```

Operation Umsetzung in Java

+ op
- op
op
~ op

```
public class OpMix {
    public static void op1() {
        //Implementierung
    }
    private int op2(final int param1) {
        //Implementierung
        return wert;
    }
    protected void op3(KlasseC param3) {
        //Implementierung
    }
    KlasseB op4(String param3[]) {
        //Implementierung
        return wert;
    }
}
```

Operation Umsetzung in C#

```
class OpMix {  
    public static void op1() {  
        //Implementierung  
    }  
    private int op21(int param1) {  
        //Implementierung  
        return wert;  
    }  
    private int op22(params object[] list) {  
        //Implementierung  
        return wert;  
    }  
    protected void op3(ref ClassC param3) {  
        //Implementierung  
    }  
    internal int op4(out StringCollection param3) {  
        //Implementierung  
        return wert;  
    }  
}
```

Stereotypen

- Kennzeichnung von benutzerspezifischer Terminologie oder Notation
- Klasse innerhalb des Metamodells, die andere Klassen durch Erweiterung näher spezifizieren kann

Beispiele für UML-Standard-Stereotypen:

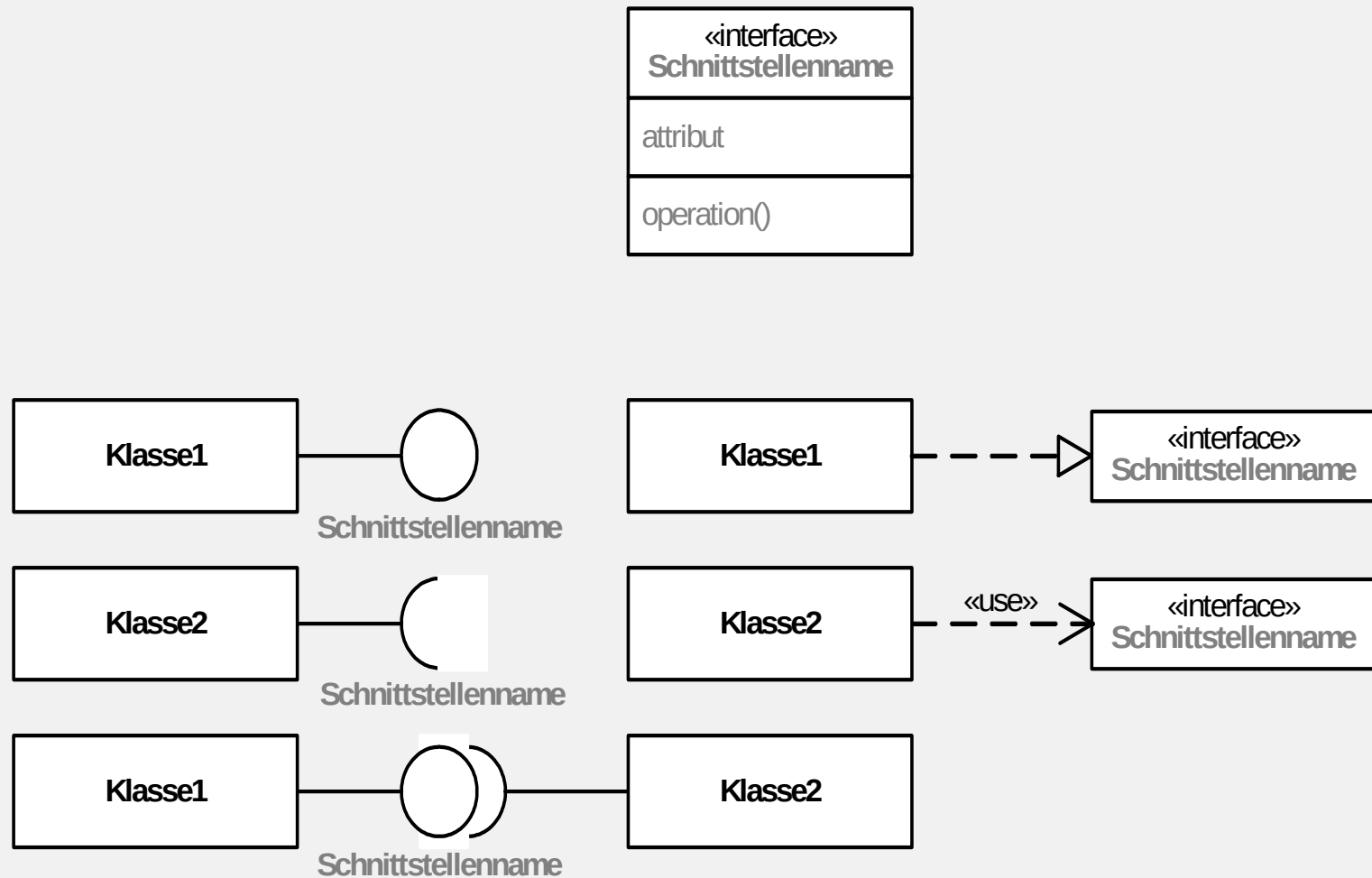
- *auxiliary* Klasse implementiert sekundäre Logik oder sekundären Kontrollfluss
- *call* besitzt Operationen oder Klassen mit Operationen auf die sich Assoziation bezieht
- *derive* Modellelemente werden voneinander abgeleitet und sind vom selben Typ
- ...

Schnittstelle

- Darstellung durch das Klassensymbol mit Zusatz «interface»
- Schnittstelle kann weitere Schnittstellen
 - beinhalten
 - spezialisieren
 - generalisieren

NICHT möglich: Implementationsbeziehung zwischen zwei Schnittstellen

Schnittstelle



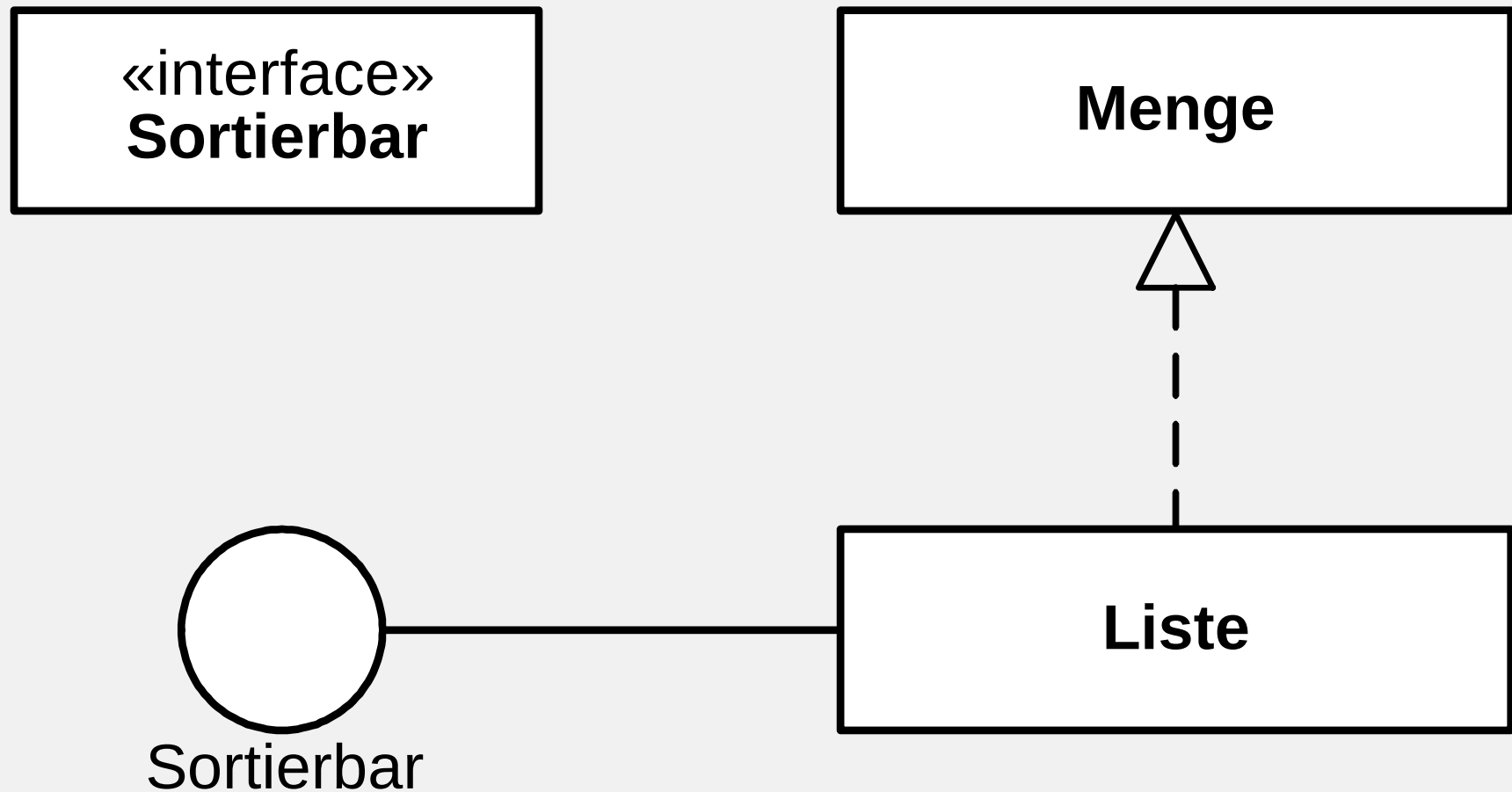
Schnittstelle

Die Schnittstelle wird nicht instanziiert, sondern durch Classifier realisiert:

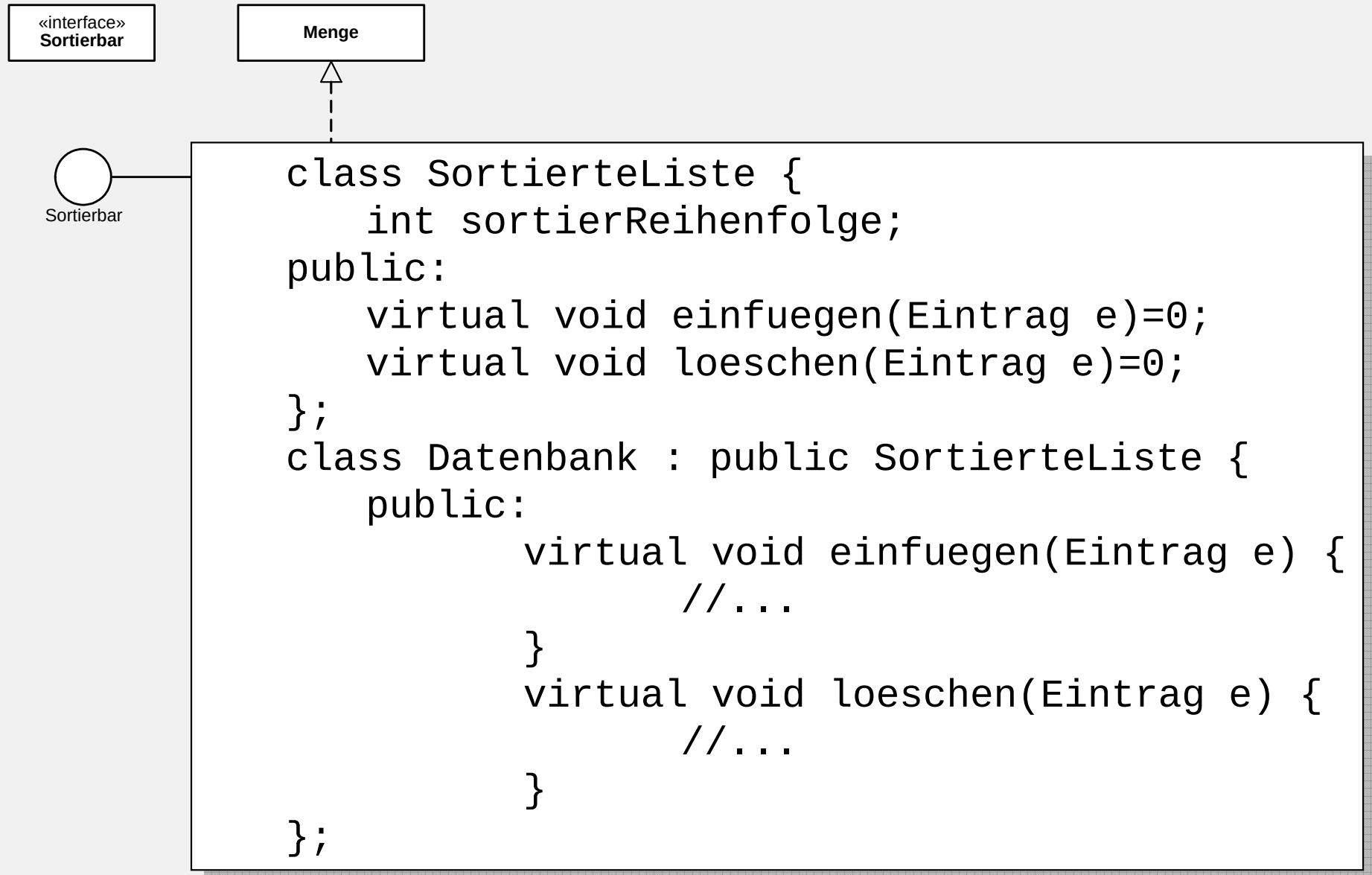
- Zeichnet Classifier mit
 - Operationen
 - öffentlichen Merkmalen
 - Verpflichtungen aus
- Hierbei ist der Classifier konform zur Schnittstelle
- Classifier setzt Operationen in Methoden um
- Eine Schnittstelle kann von mehreren Classifiern umgesetzt werden
- Ein Classifier kann mehrere Schnittstellen umsetzen

Schnittstelle

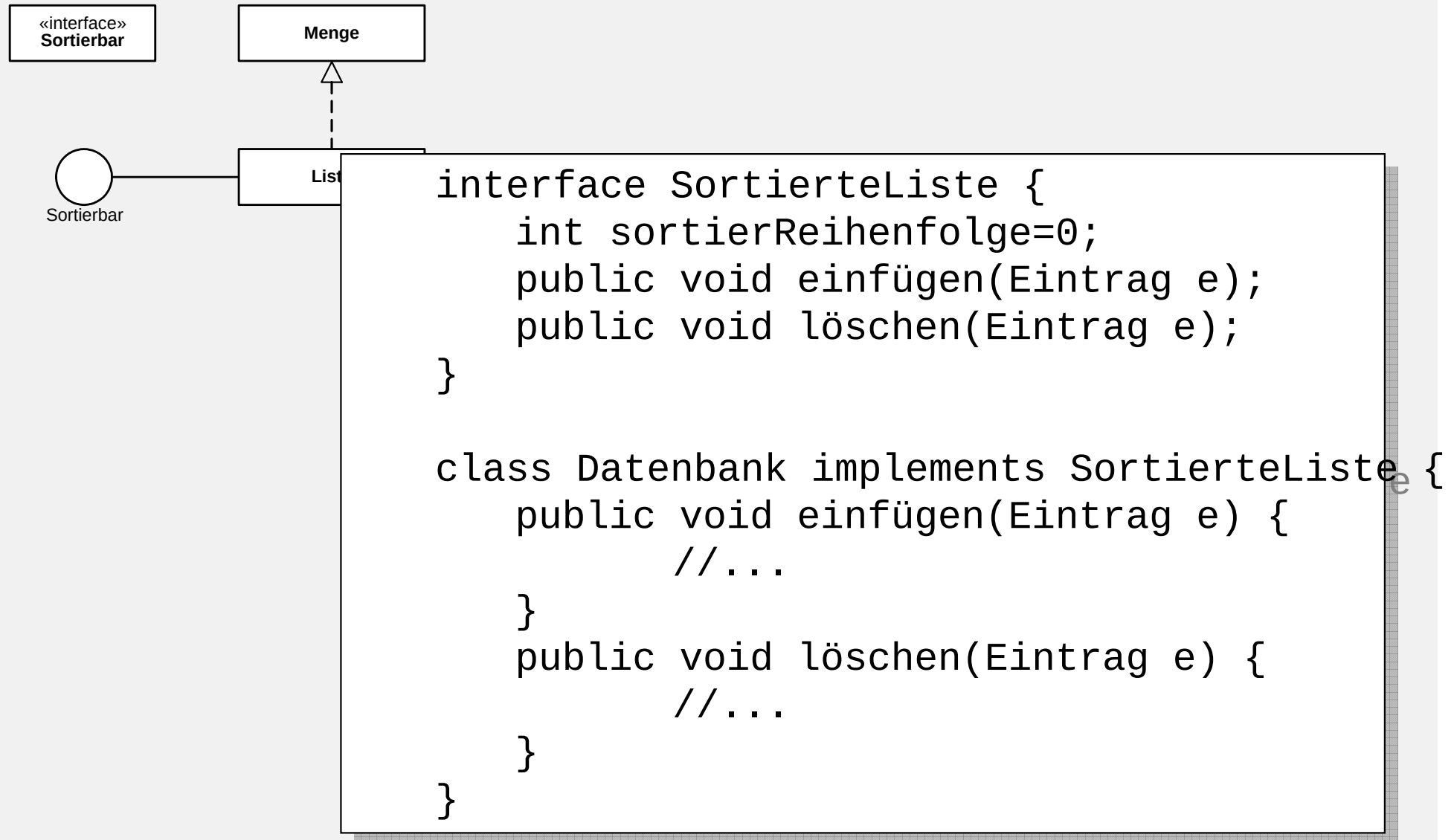
Umsetzungsbeispiel



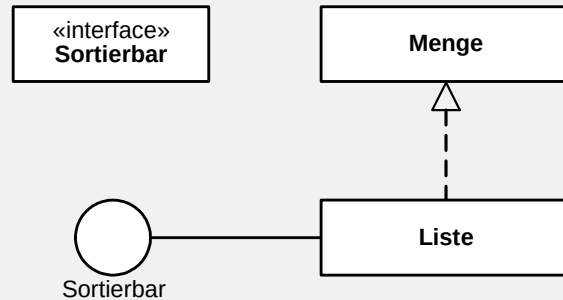
Schnittstelle Umsetzung in C++



Schnittstelle Umsetzung in Java



Schnittstelle Umsetzung in C#

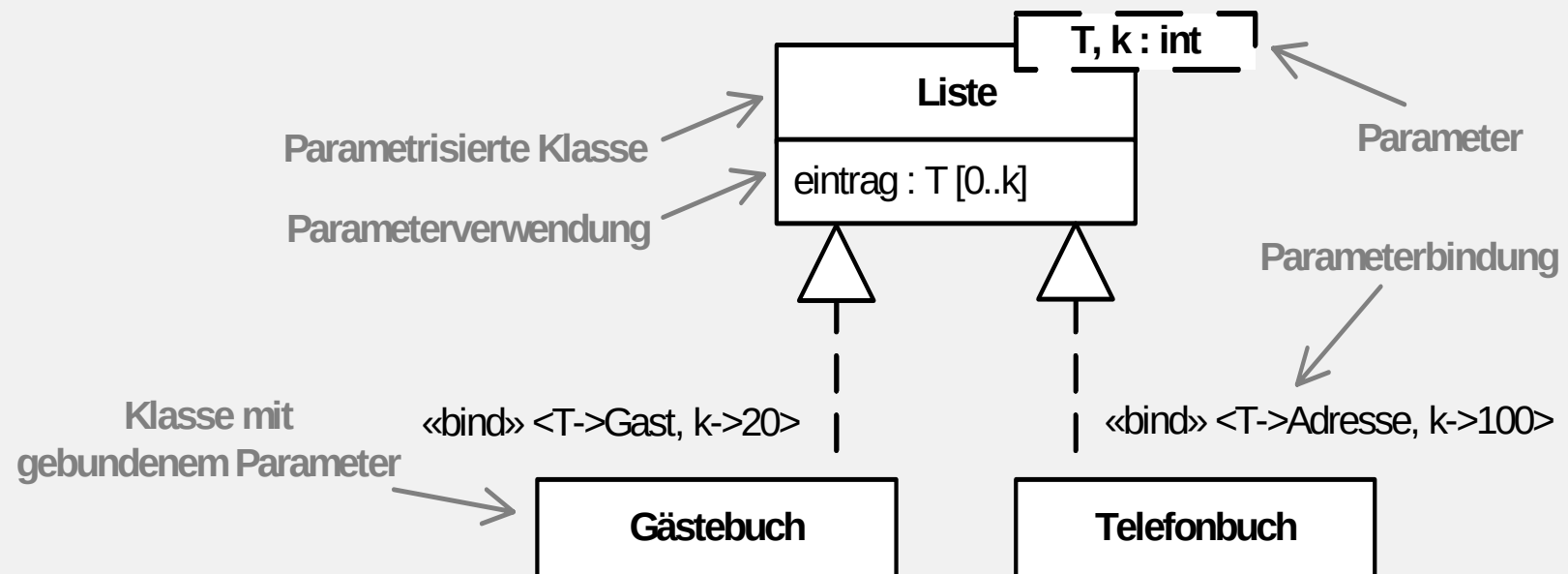


```
interface SortierteListe {
    void einfuegen(Eintrag e);
    void loeschen(Eintrag e);
}

class Datenbank : SortierteListe {
    public void einfuegen(Eintrag e) {}
    public void loeschen(Eintrag e) {}
}
```

Parametrisierte Klasse

Darstellung durch übliches Klassensymbol mit zusätzlicher Rechtecksbox mit Schablonenparameter der Klasse:



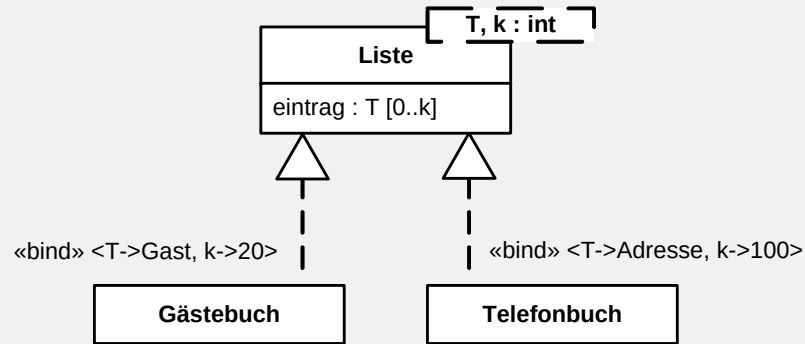
Parametrisierte Klasse

Verwendung:

- Belegung der Schablonen-Parameter mit konkreten Werten (Schablone (ein Muster, engl. Template) der Funktion, aus der beliebig viele strukturell identische Implementierungen erzeugen werden können.)
- Vorsehen von Spezifikationslücken zur Auffüllung durch Bindungsprozess
- Wenn Attribut- oder Operationstyp bis zur Laufzeit offen gehalten werden sollen
- Allgemeine Fassung parametrisierter Strukturen, um Verwendung für viele verschiedene Inhaltstypen zu gewährleisten

Parametrisierte Klasse

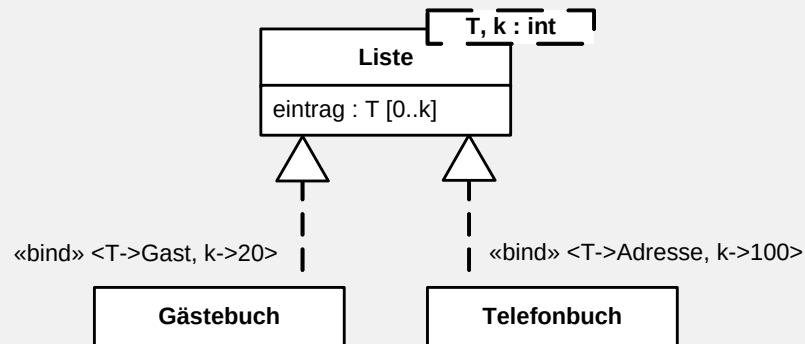
Umsetzung in C++



```

template <class T, int k>
class Liste {
    T elements[k];
};
...
Liste<Gast, 20> Gaestebuch;
Liste<Adresse, 100> Telefonbuch;
    
```

Parametrisierte Klasse Umsetzung in Java



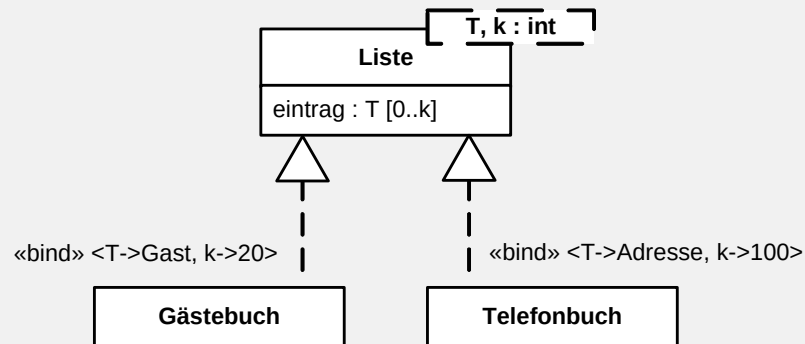
```

class Liste<T> {
    Vector<T> elements;
    public Liste(int k) {
        elements = new Vector<T>(k);
    }
}

...
Liste<Gast> Gästebuch=new Liste<Gast>(20);
Liste<Adresse> Telefonbuch=new Liste<Adresse>(100);
    
```

Parametrisierte Klasse

Umsetzung in C#



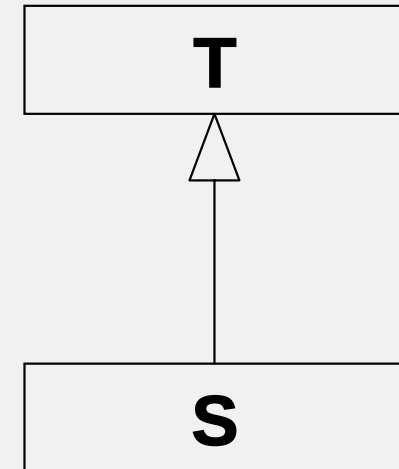
```

class Liste<T> {
    T[] eintrag;
    public Liste(int k) {
        eintrag = new T[k];
    }
}
...
Liste<Gast> Gästebuch=new Liste<Gast>(20);
Liste<Adresse> Telefonbuch=new Liste<Adresse>(100);
    
```


Generalisierung

Darstellung durch gerichtete Kante mit leerer Spitze am Pfeilende

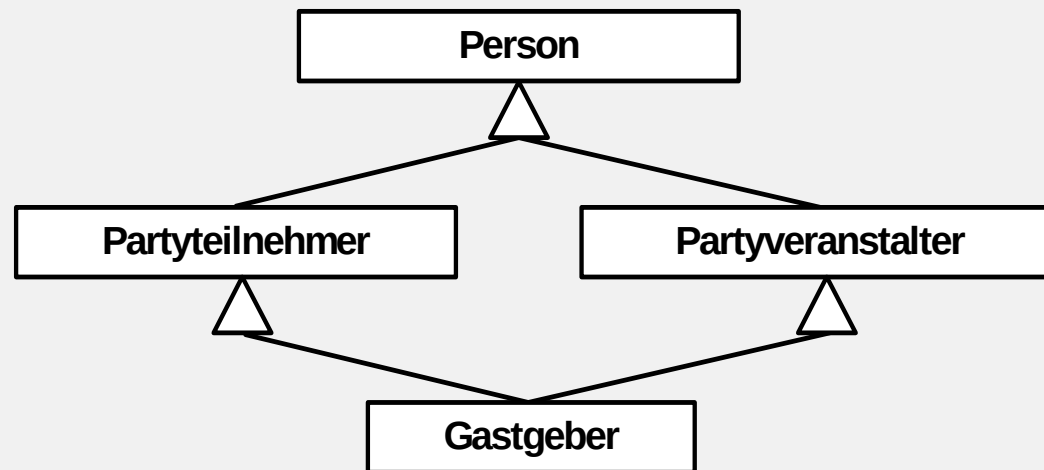
- Semantik ist unabhängig von konkreter Implementierungssprache definiert, berücksichtigt jedoch zentrale Gesichtspunkte
 - Vererbung
 - Substitution



Generalisierung

Diamantförmige Generalisierung

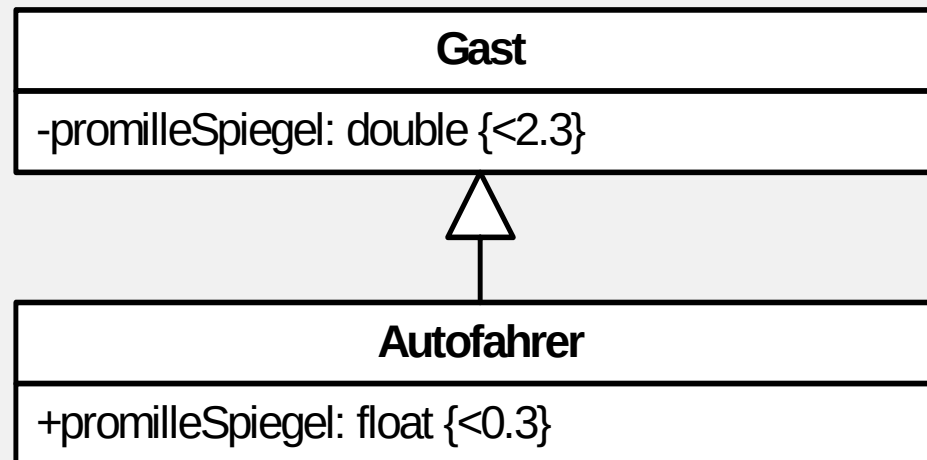
- Syntaktisch möglich
- In verschiedenen Programmiersprachen problematisch
- In einigen Programmiersprachen nicht umsetzbar



Generalisierung

Generalisierung

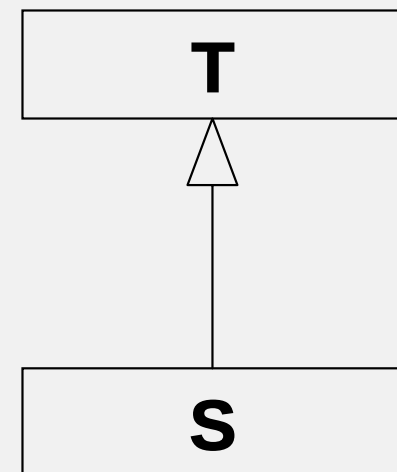
- Kovariante Vererbung
- Kontravariante Substitution



Generalisierung

1988: Liskovsches Substitutionsprinzip
(Liskov Substitution Principle, LSP)

„[...] substitution property: If for each object o_1 of type S there is an object of type T such that for all programs P defined in terms of T , the behavior of P is unchanged when o_1 is substituted for o_2 then S is a subtype of T .“



Generalisierung

LSP bedeutet verdeutlicht des Zusammenhang zwischen Vererbung und Substitution:

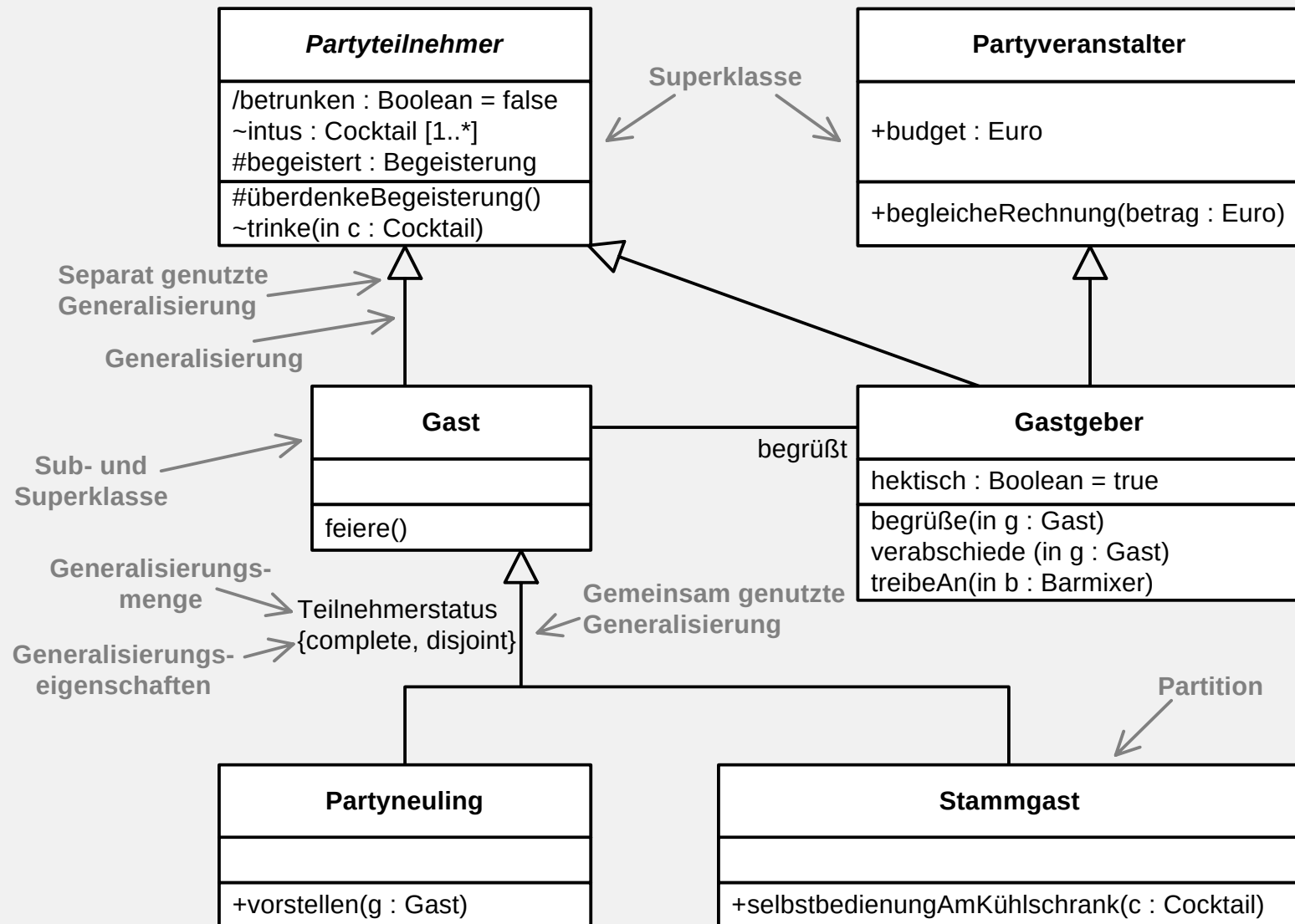
- Forderung an alle Unterklassen (S) einer Klasse(T) dieselbe Semantik für eine Schnittstelle anzubieten
- Sicherstellung, dass sich Programm bei Verwendung einer Schnittstelle gleich verhält, egal ob Instanz einer Klasse oder der Unterklasse verwendet wird
- Besonders wichtig bei Redefinition von Methoden

Generalisierung

Generalisierungsbeziehung zwischen Classifiern:

- ist möglich an allen Classifiern
- nicht auf Klassen beschränkt
- auch *Partition* genannt

Generalisierung



Generalisierung

Generalisierungseigenschaften: in UML durch vier Schlüsselworte dargestellt

<i>complete</i>	Vereinigung aller Partitionen enthält alle sinnvollen Spezialisierungen eines Typs
<i>incomplete</i>	notierte Subtypen beinhalten nicht alle sinnvollen Spezialisierungen eines Supertypen
<i>disjoint</i>	Subtypen können nicht die selbe Instanz enthalten
<i>overlapping</i>	Subtypen können selbe Instanz enthalten

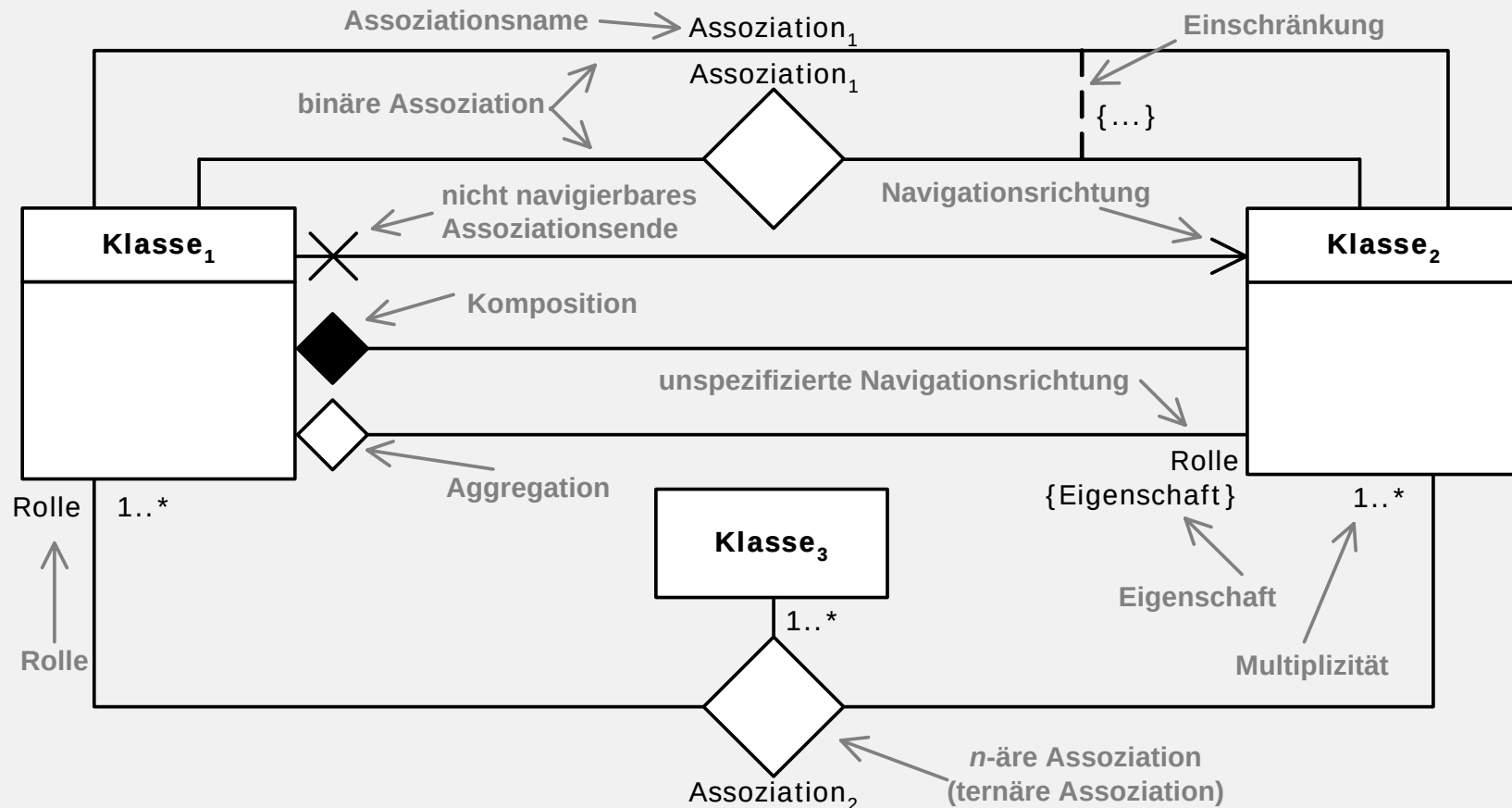
Generalisierung

Nutzen der Generalisierung:

- Codeabstraktion
- Wiederverwendung
- Hierarchische Strukturierung des Entwurfs
- Einsatz bei Implementierungsvererbung und Substituierbarkeit
- Vermeidung von Coderedundanz

Assoziation

Assoziation = Menge semantisch gleichartiger Beziehungen zwischen Classifiern



Assoziation

Ausprägung einer Assoziation ist eine Beziehungsinstanz (Link)

Verschiedene Ausprägungen:

- Binäre / zweiwertige Assoziation: hat zwei Enden
- n-äre / höherwertige Assoziation: mehr als zwei Enden
- Zirkulär-reflexive Assoziation: kehrt Ausgangsclassifizierung zurück
- Abgeleitete Assoziation: gekennzeichnet durch „/“ und kann aus anderen Zusammenhängen abgeleitet werden

Assoziation

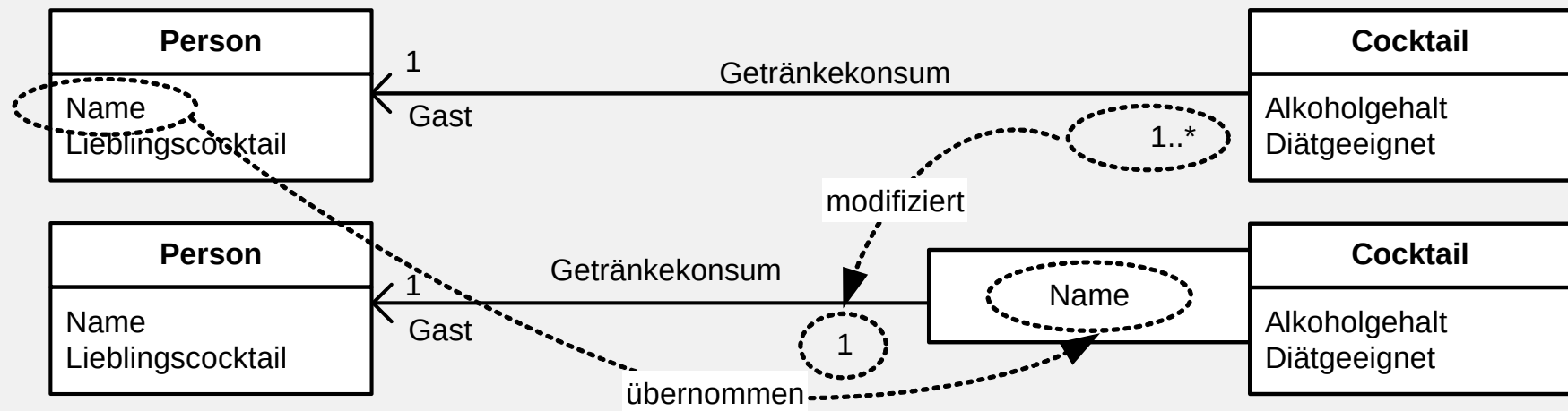
Qualifizierte Assoziation:

- Herabsetzen der Multiplizität
- Aufteilung der Assoziationsmenge
- Vollständig
- Überlappungsfrei
- Jedes referenzierte Objekt ist nur genau einer Partition zugeordnet
- Kein Objekt wird bei Zuordnung übergangen

ABER:

- Kann nur für binäre Assoziationen verwendet werden
- Sollte als navigierbar spezifiziert sein

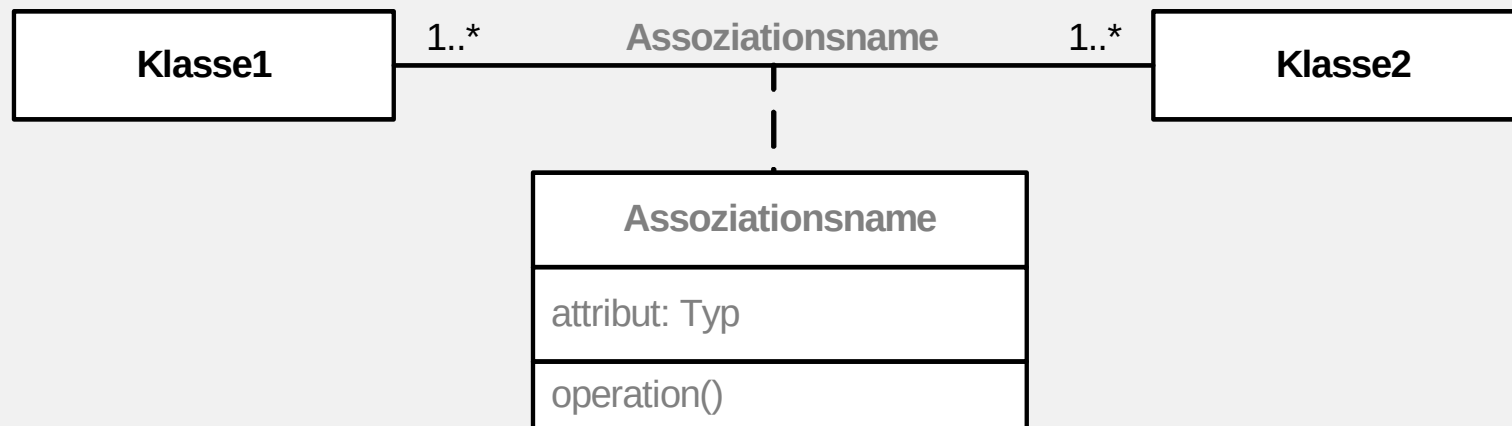
Assoziation



Die beiden Diagramme sind nicht äquivalent

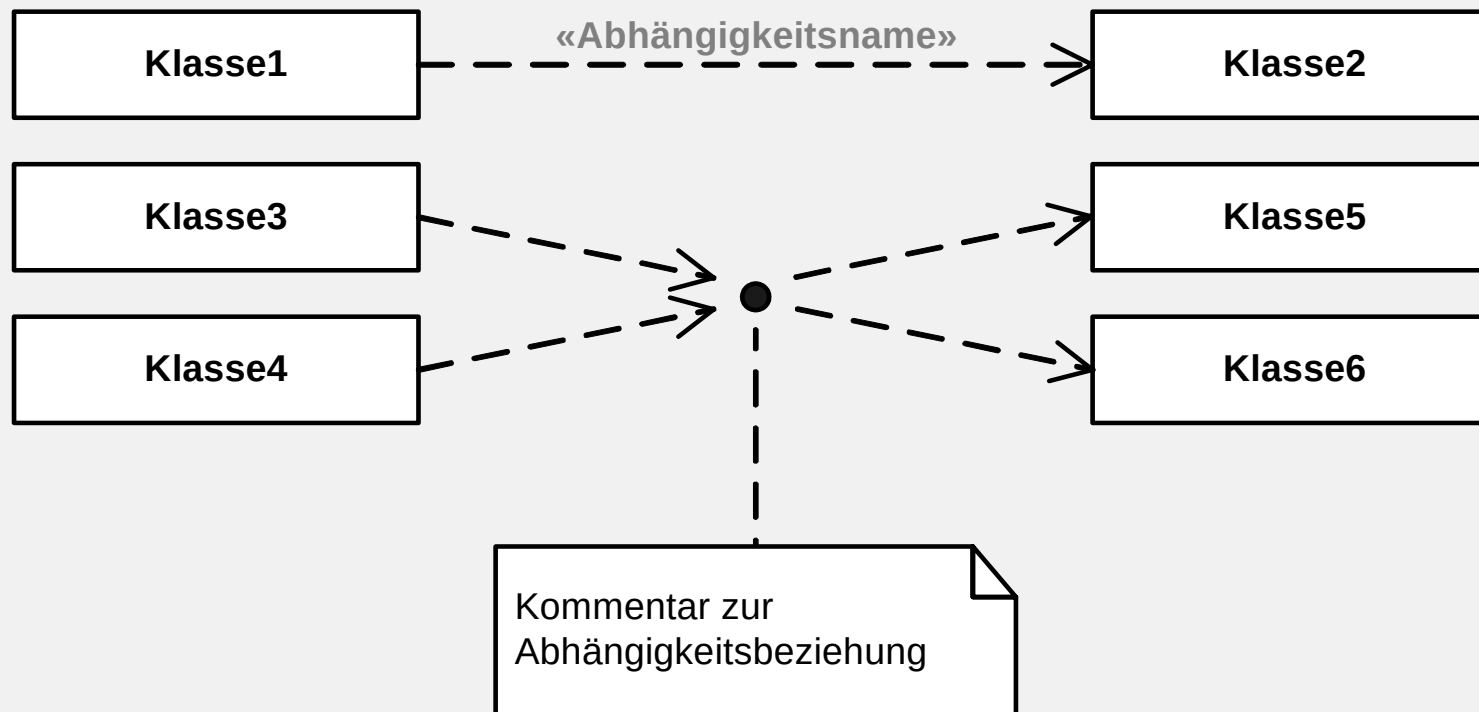
Assoziationsklasse

Stellt Abhängigkeitsbeziehung dar



Kommentar

Kommentar annotiert eine Klasse und wird mit „Eselsohr“ gekennzeichnet:



Eigenschaftswerte

- Spezifizieren Charakteristika der Elemente
- Werden durch { } gekennzeichnet und durch Kommata getrennt
 - abstract Classifler ist nicht instanziiierbar
 - derived Ein Element ist von einem anderen abgeleitet
 - ordered Eine Menge von Elementen ist geordnet