Kapitel 2

Objektorientierte Modellierungtechniken

Prof. Dr. Rolf Hennicker

26.10.2010

Ziele

- Klassendiagramme in UML erstellen können.
- Objektdiagramme in UML erstellen können.
- Das Vererbungs- und Subtypprinzip verstehen, insbesondere
  - abstrakte Klassen und Operationen
  - Schnittstellen
  - dynamische Bindung
- Klassendiagramme in Java implementieren können.
- (Flache und hierarchische) Zustandsdiagramme in UML erstellen können.
- Zustände, Ereignisse und Transitionen verstehen.
- Aktivitätsdiagramme in UML erstellen können.
- Das Prinzip der Metamodellierung verstehen.

Grundidee

Modellierung dient dazu, ein System zu verstehen, bevor es gebaut wird.

Wesentliches Prinzip

Abstraktion (auf wesentliche Aspekte konzentrieren, keine Details)

Es werden i.a. verschiedene Sichten auf ein System modelliert:

- *Statisches Modell*: beschreibt strukturelle und datenbezogene Eigenschaften
- *Dynamisches Modell*: beschreibt das Verhalten der Objekte, deren Zustandsänderungen und Interaktionen.

Notation

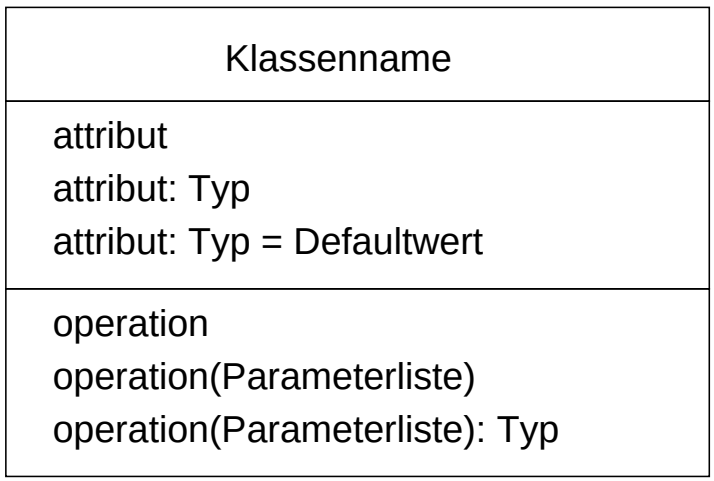
UML (Unified Modeling Language), 1997 Version 1.0 (Booch, Rumbaugh, Jacobson), aktuell UML 2

2.1 Modellierung statischer Systemeigenschaften

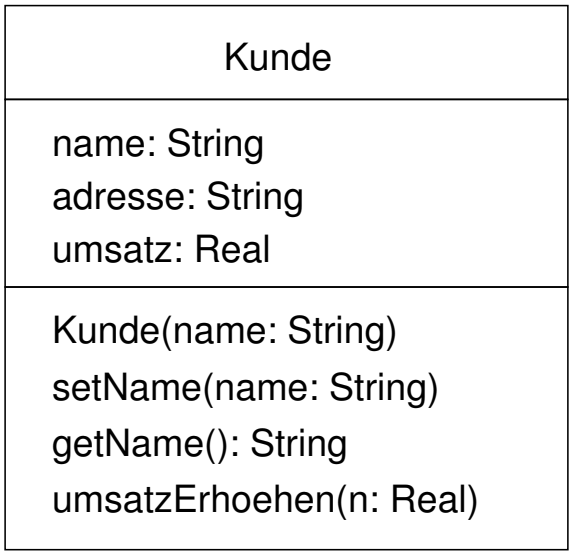
2.1.1 Klassen und Objekte

Klassen

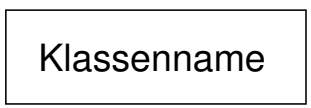
Allgemeine Form:



Beispiel:

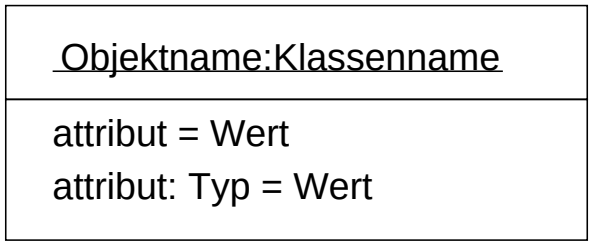


Kurzform:

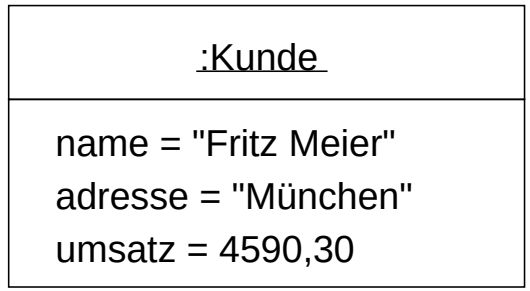


Objekte

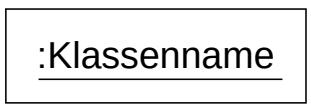
Allgemeine Form:



Beispiel:



Kurzformen:



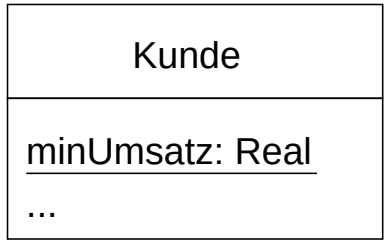
Bemerkungen

- Als Typen verwenden wir Standarddatentypen (Boolean, Integer, Real, String) oder Klassennamen.
- In der Analysephase (und möglichst auch im Entwurf) sollten als Typen von Attributen nur Standarddatentypen verwendet werden.
- Operationen mit Ergebnistyp liefern nach Ausführung einen Wert dieses Typs. Der Rückgabewert kann auch ein Objekt sein.
- Ebenso können aktuelle Parameter von Operationen Objekte sein, wenn der formale Parameter einen Klassentyp hat.
- Operationen ändern i.a. den Zustand eines Objekts.
- Operationen, die den Zustand nicht ändern, nennt man "Queries".

Weiterführende Begriffe und Notationen

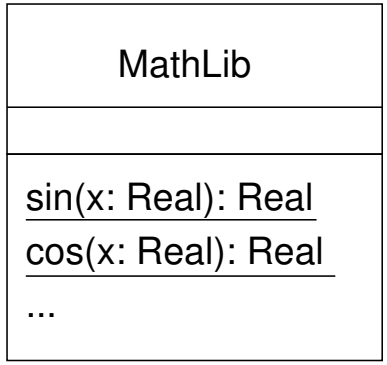
Ein Attribut, das bei jedem existierenden Objekt der Klasse denselben Wert hat, heißt *Klassenattribut* und wird in UML unterstrichen.

Beispiel:



Eine Operation, die vom Zustand konkreter Objekte unabhängig ist, heißt *Klassenmethode* und wird in UML unterstrichen.

Beispiel:

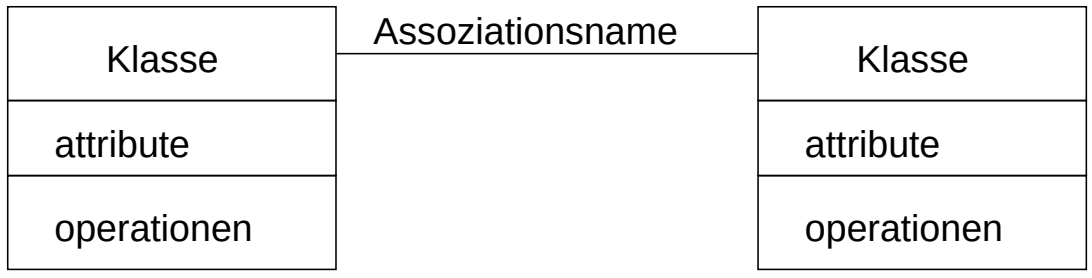


2.1.2 Assoziationen und Objektbeziehungen

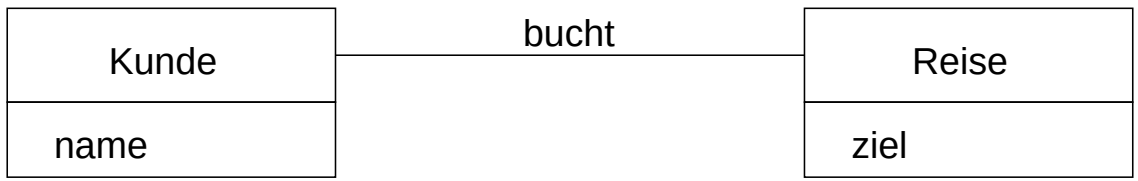
- Eine *Objektbeziehung (Link)* stellt eine semantische (physikalische oder konzeptionelle) Verbindung zwischen Objekten dar.
- Eine *Assoziation* beschreibt eine Menge gleichartiger Beziehungen zwischen Objekten bestimmter Klassen.

Assoziationen

Allgemeine Form:

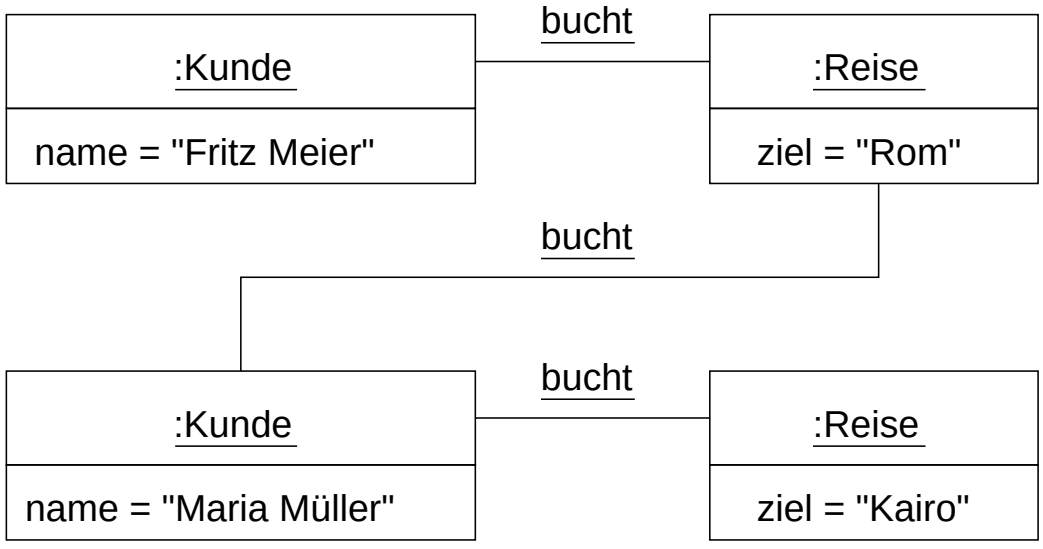


Beispiel:



Objektbeziehungen

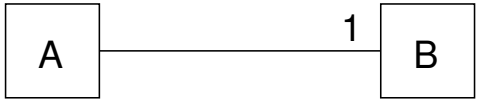
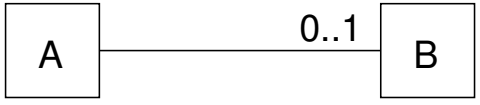
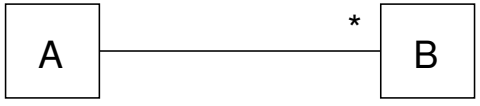
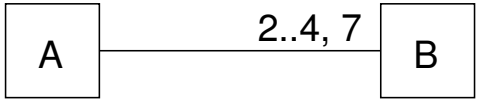
Beispiel:



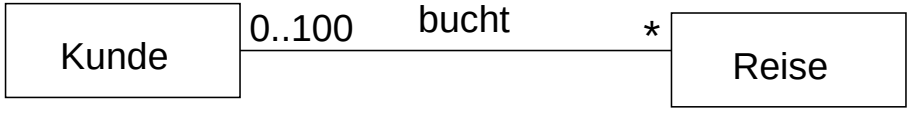
- Ein UML-Diagramm mit Klassen und Assoziationen (und Vererbung) heißt *Klassendiagramm*.
- Ein UML-Diagramm mit Objekten und Objektbeziehungen heißt *Objektdiagramm* (oder *Instanzendiagramm*). Es stellt einen augenblicklichen Systemzustand ("Snapshot") dar.

Multiplizitäten

Geben an, wieviele Objekte einer Klasse mit wievielen Objekten einer (meist anderen) Klasse gemäß einer Assoziation in Beziehung stehen können.

*N*otation:	Bedeutung:
	Jedes Objekt von A steht in Beziehung mit
	genau einem Objekt von B
	keinem oder einem Objekt von B
	einem oder mehreren Objekten von B
	beliebig vielen Objekten von B (auch keinem)
	2 bis 4 oder 7 Objekten von B

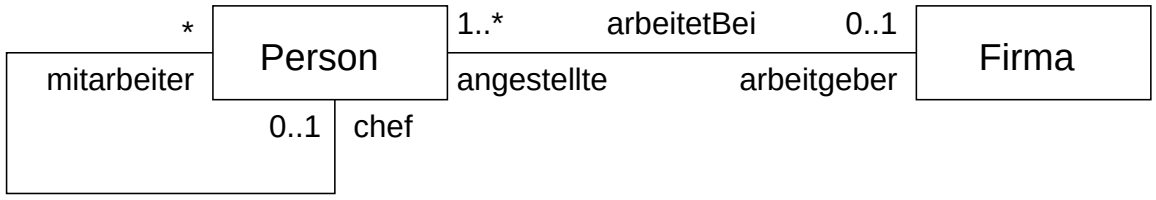
Beispiel:



Assoziationsrollen

Beschreiben, welche Rolle die Objekte einer Klasse in einer Assoziation einnehmen.

Beispiel:



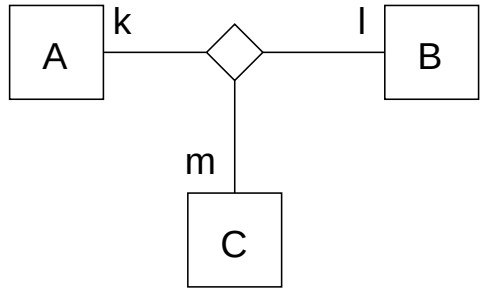
Bemerkung:

Assoziationsnamen und Rollennamen können weggelassen werden. Rollennamen sind dann implizit durch die kleingeschriebenen Klassennamen (evtl. im Plural) gegeben.

Mehrstellige Assoziationen

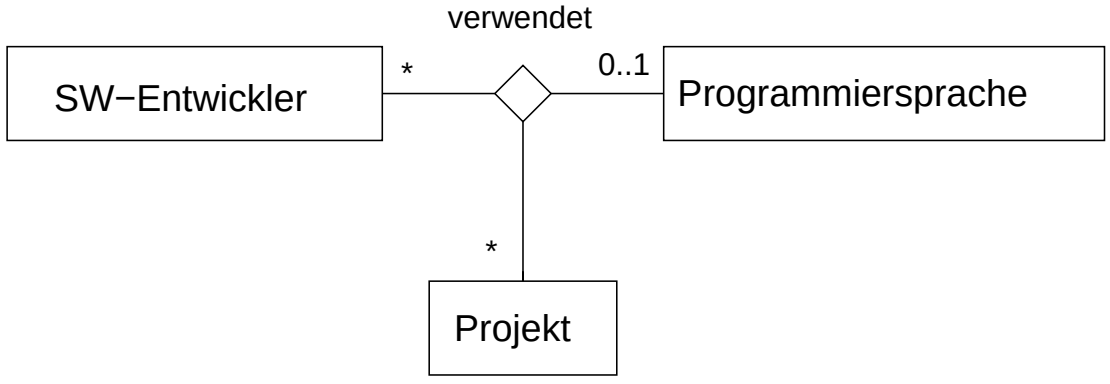
Verbinden drei oder mehr Klassen.

Darstellung:



Die Multiplizität einer Rolle in einer n-stelligen Assoziation spezifiziert, wieviele Objekte mit dieser Rolle mit fest gegebenen (fixierten) n-1 Objekten der anderen Klassen in Beziehung stehen können.

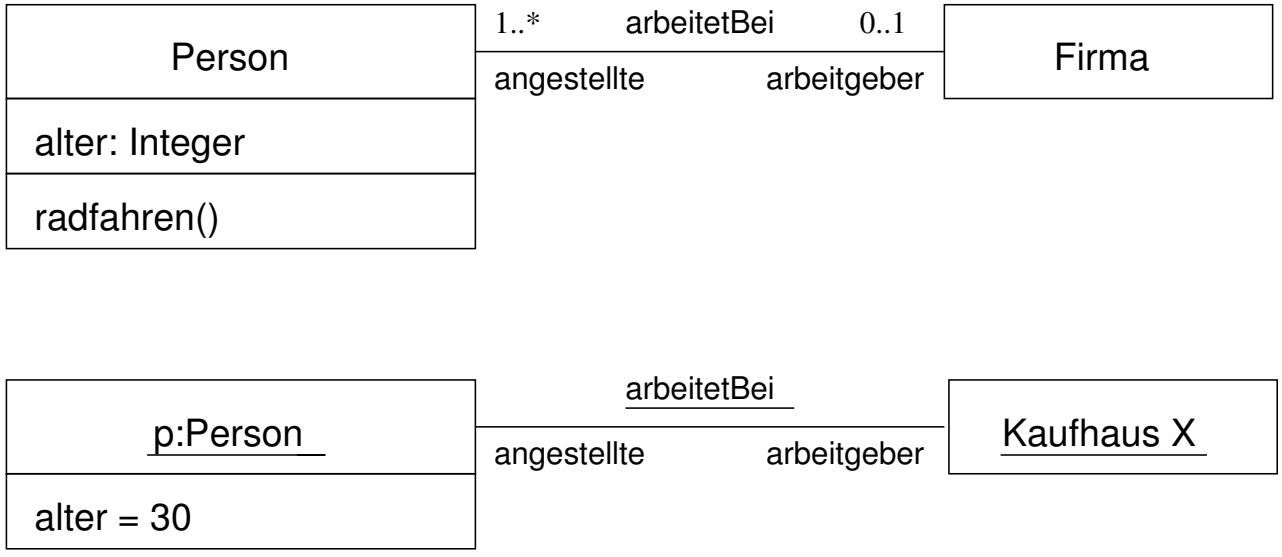
Beispiel:



Zugriff auf die Merkmale eines Objekts

Wird durch die "."-Notation ausgedrückt.

Beispiel:



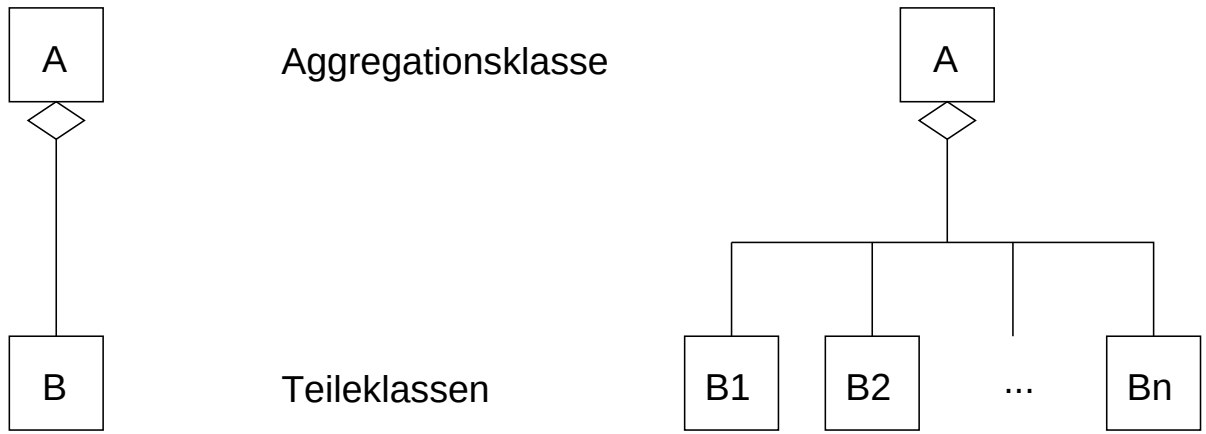

```
p.alter = 30;
p.arbeitgeber = Kaufhaus X;
p.radfahren(); //Aufruf der Operation "radfahren" fuer das Objekt p
```

Aggregation

Spezielle Form der Assoziation, die eine "Gesamtheit-Teil"-Beziehung ausdrückt.

z.B.
ICE-Lok besitzt 6 Motoren (physikalisch),
Stundenplan umfasst mehrere Vorlesungen (konzeptionell)

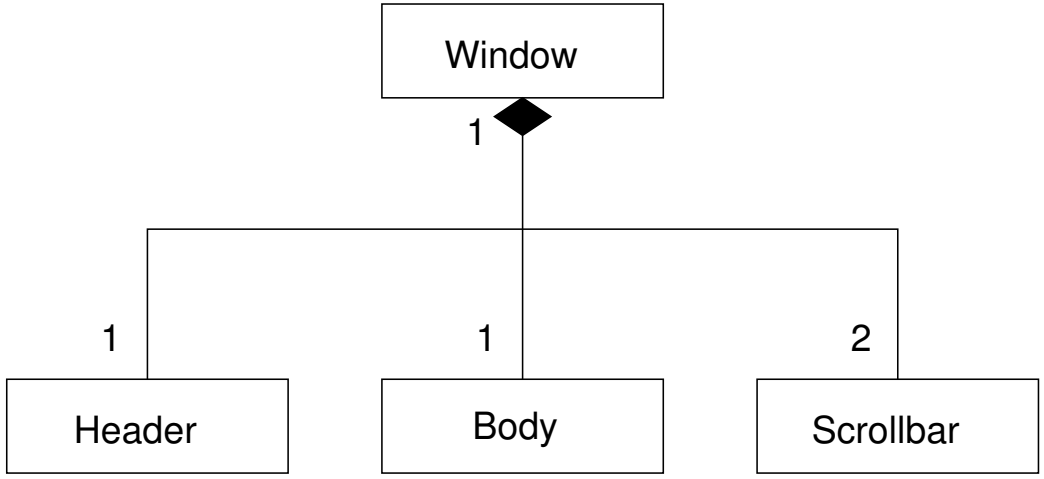
Darstellung:



Komposition

Spezielle Form der Assoziation: das Teil ist existenzabhängig vom Ganzen.

Darstellung (Beispiel):

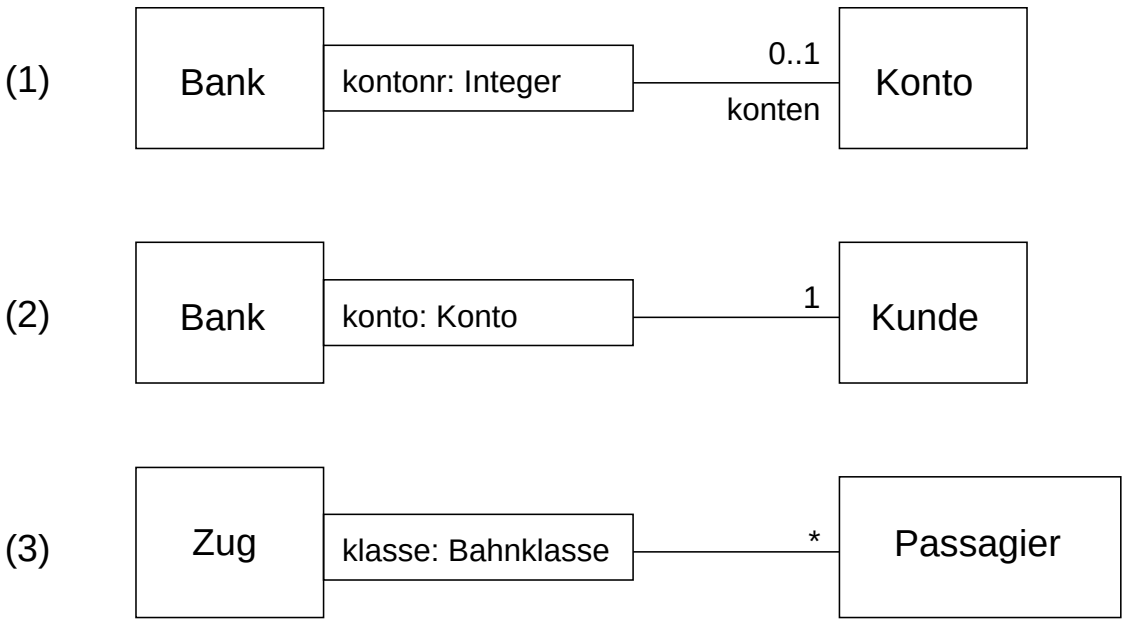


Das Teil ist vollständig im Besitz des Ganzen [OMG 2004]. Das Teil kann erst bei (oder nach) der Erzeugung des Ganzen erzeugt werden und wird mit dem Ganzen gelöscht.

Qualifizierte Assoziation

- Eine qualifizierte Assoziation unterteilt die Menge der Objekte auf einer Seite der Assoziation in Partitionen.
- Qualifizierer haben (wie Attribute) einen Typ, der auch eine Klasse sein kann.

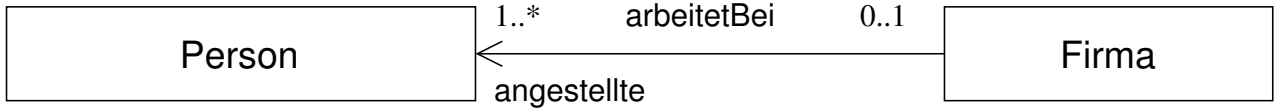
Beispiele:



Gerichtete Assoziation

Falls eine Assoziation nur in einer Richtung durchlaufen wird ("unidirektional"), wird das entsprechende Assoziationsende mit einer Pfeilspitze markiert.

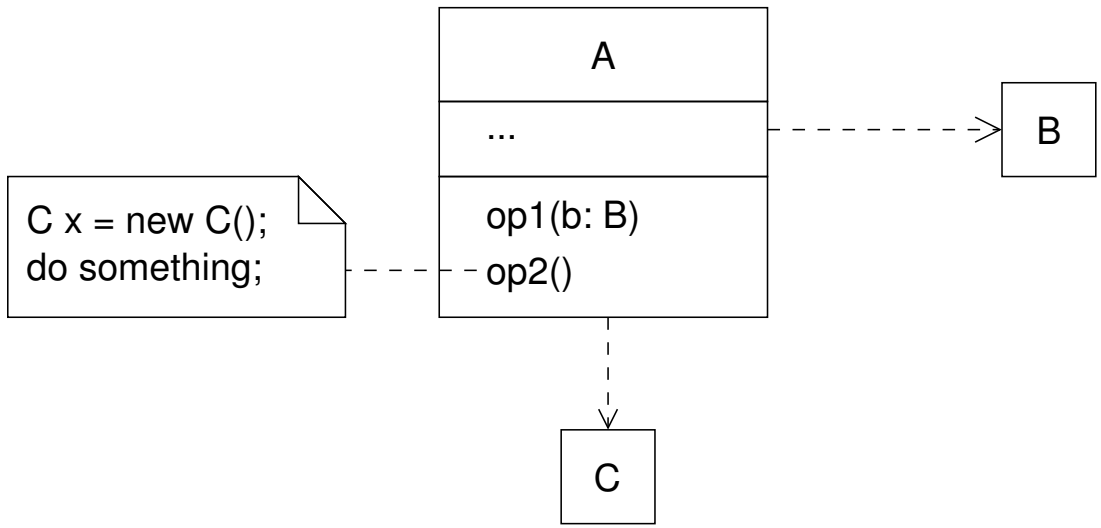
Beispiel:



2.1.3 Abhängigkeiten (Dependencies)

Eine Abhängigkeit ist eine gerichtete Beziehung zwischen Modellelementen, die besagt, dass Änderungen im Zielelement möglicherweise Änderungen im davon abhängigen Element nach sich ziehen.

Beispiel:

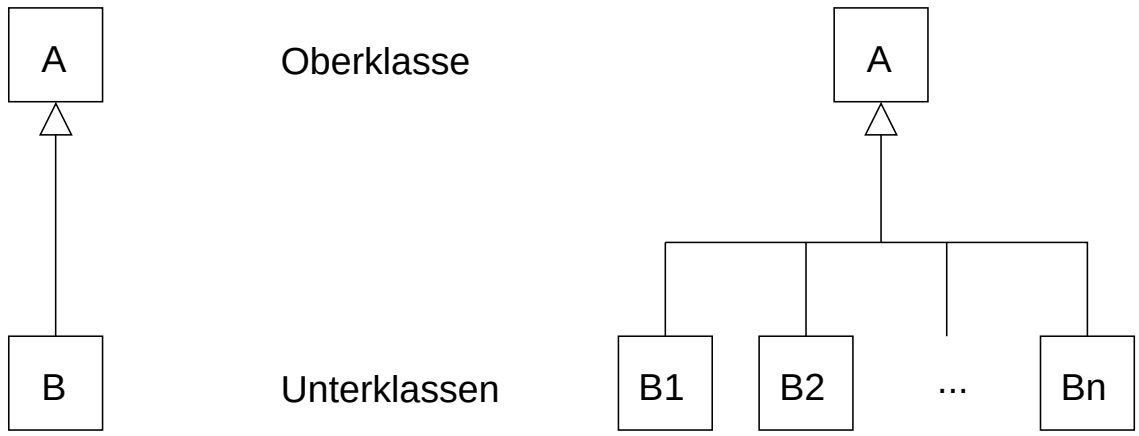


2.1.4 Vererbung

Relation zwischen einer "allgemeineren" Klasse (Ober- bzw. Superklasse) und einer "spezielleren" Klasse (Unter- bzw. Subklasse). Jedes Objekt der Subklasse ist auch ein Objekt der Oberklasse.

Beispiel:
Stoppuhr, Standuhr, Armbanduhr, Digitaluhr sind Uhren

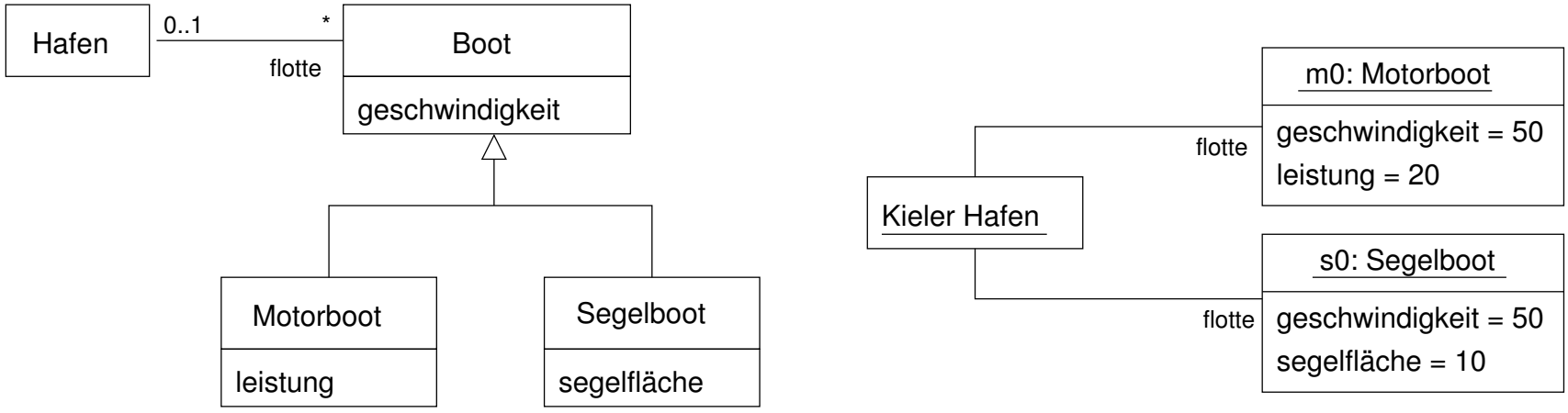
Darstellung:



- A ist eine *Generalisierung* von B.
- B ist eine *Spezialisierung* von A.

Beachte: Die Vererbungsbeziehung ist transitiv.

Beispiel:



Jede Unterklasse besitzt (erbt) alle Attribute, Assoziationen und Operationen der Oberklasse und kann eigene hinzufügen.

Substitutionsprinzip

Immer wenn ein Objekt einer Oberklasse A erwartet wird, kann ein Objekt einer Unterklasse B von A eingesetzt werden.

Beachte:

Klassennamen sind Typen. Ist A ein Klassenname und B der Name einer Unterklasse von A, dann ist B ein *Subtyp* von A.

Abstrakte Klasse

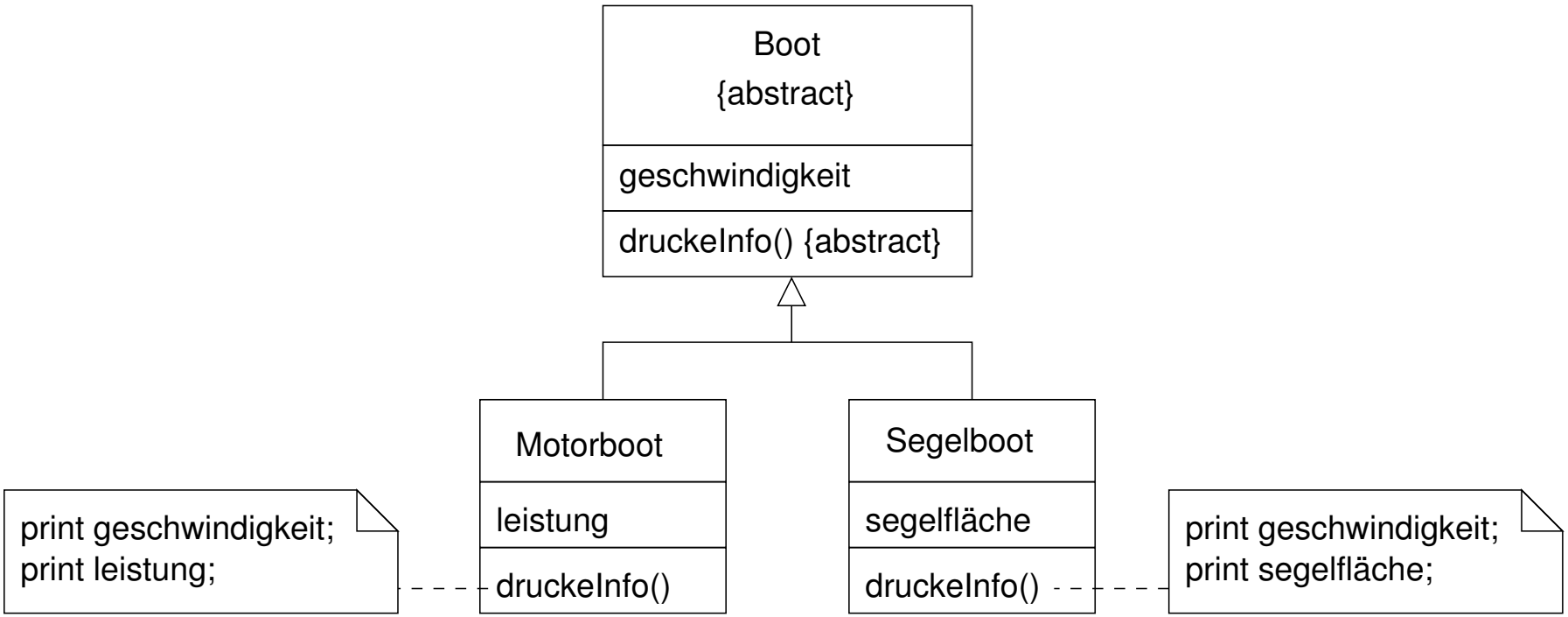
Klasse, von der keine Instanzen erzeugt werden können.

Abstrakte Operation

Operation ohne Implementierung.

Beachte:
Eine Klasse mit mindestens einer abstrakten Operation ist abstrakt.

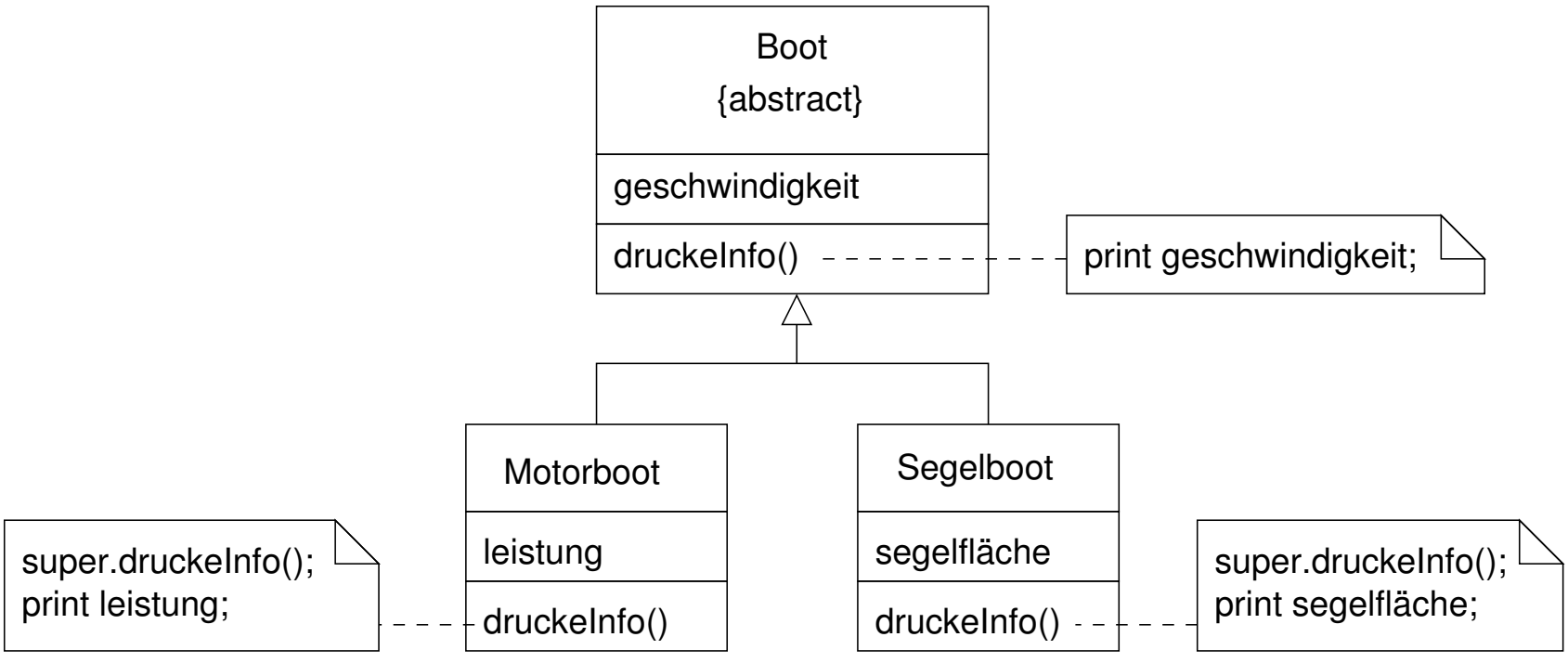
Beispiel:



Überschreiben

Eine in einer Oberklasse implementierte Operation wird in einer Unterklasse neu implementiert (redefiniert).

Beispiel:

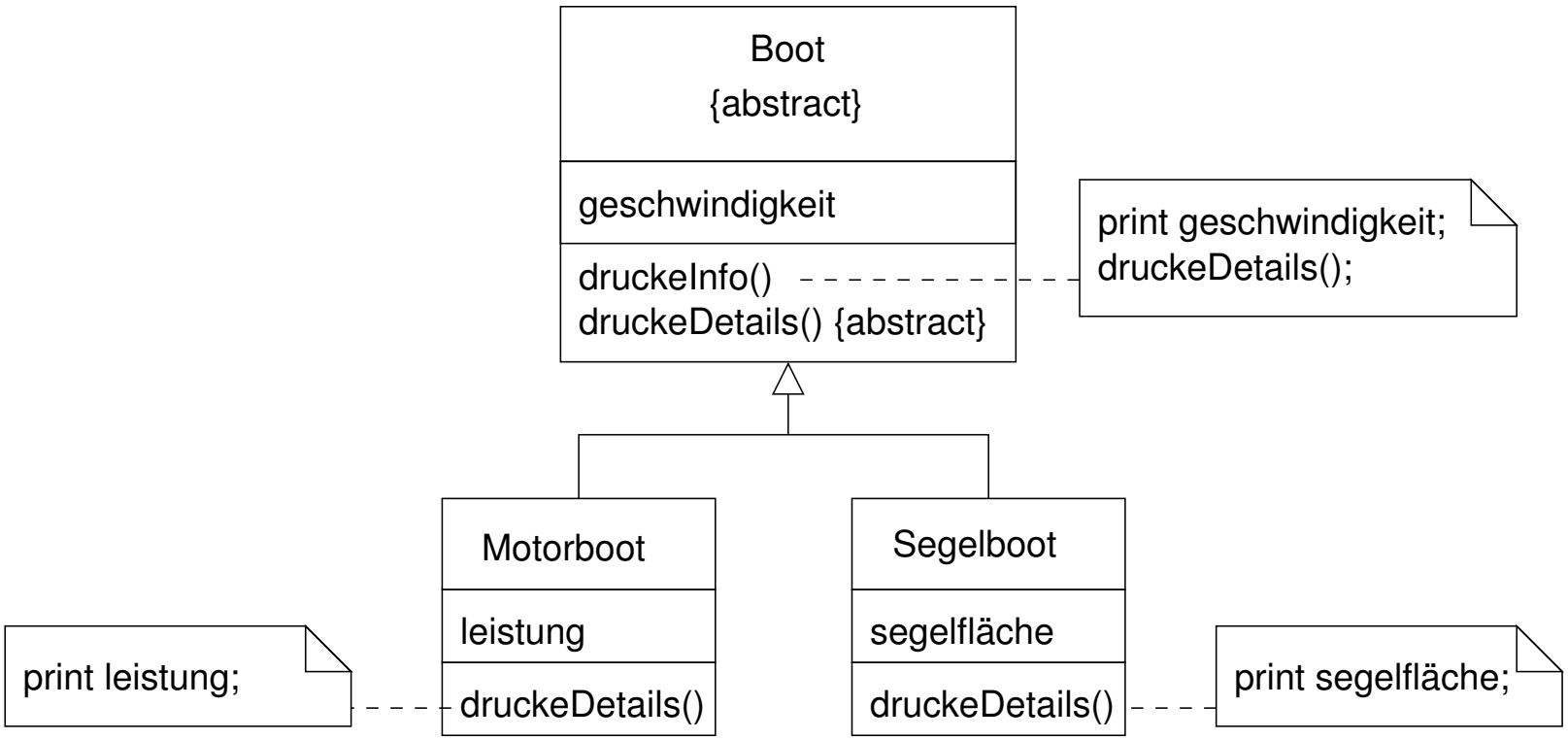


Bemerkung:
Durch Überschreiben soll die Semantik einer Operation nicht verändert werden.

Template Operation

Operation, deren Implementierung eine oder mehrere abstrakte Operationen aufruft.

Beispiel:

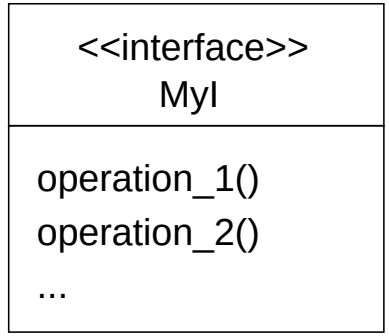


Schnittstellen

Sind abstrakte Klassen, deren sämtliche Operationen abstrakt sind und die keine Attribute und keine bidirektionalen oder wegführenden Assoziationen besitzen.

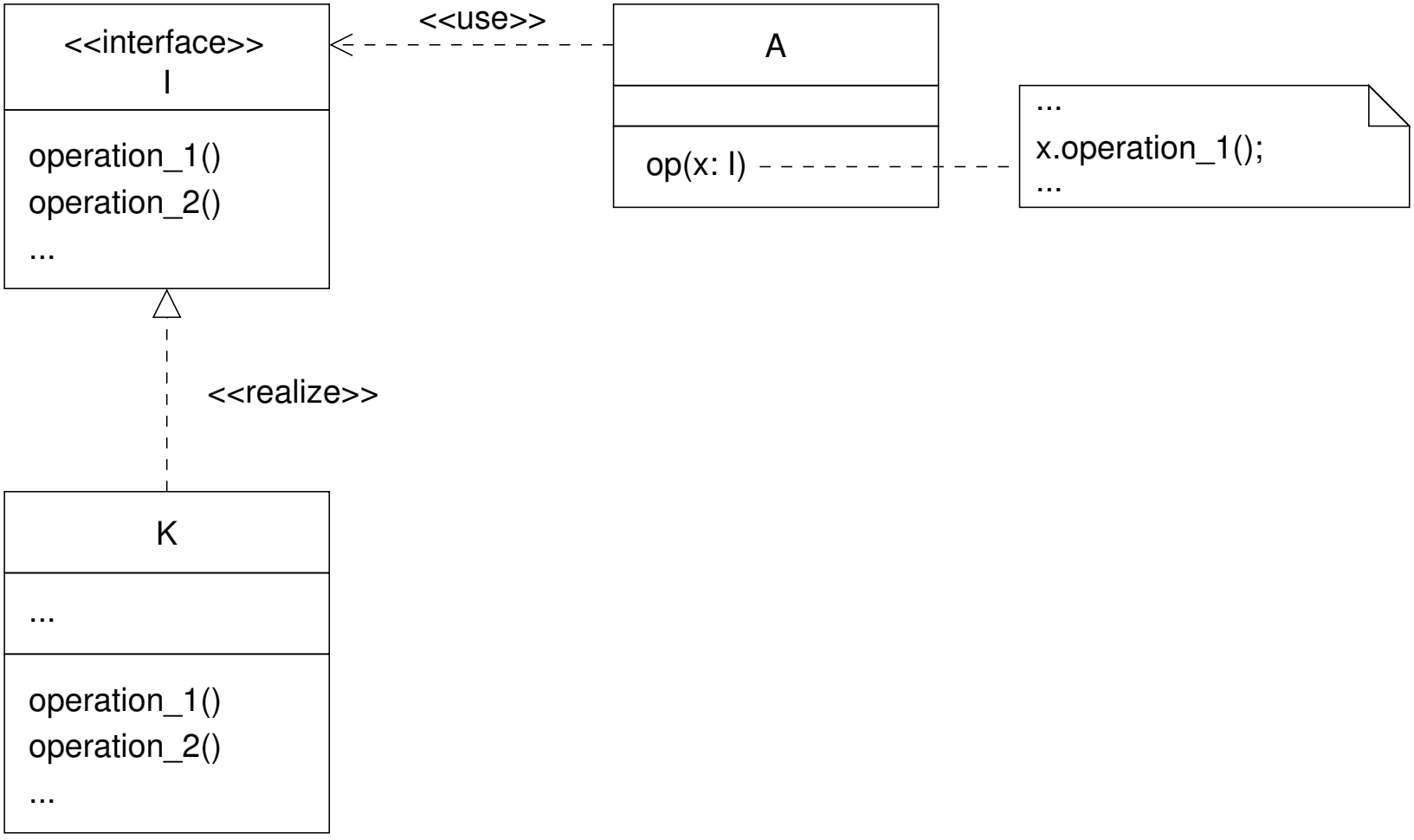
Folglich sind Interface-Namen auch Typen.

Darstellung:

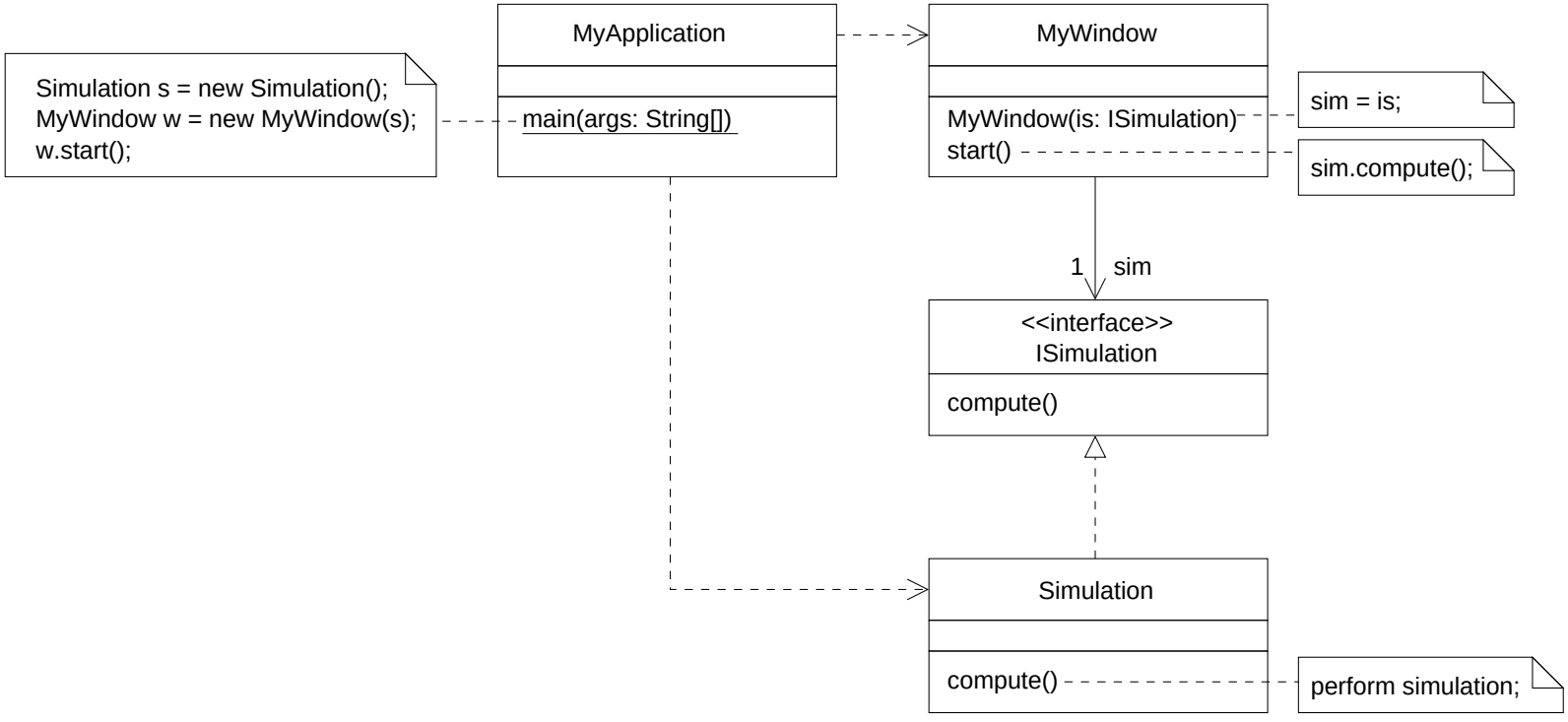


Realisierung und Benutzung von Schnittstellen

Schnittstellen bieten Dienste an, die von verschiedenen Klassen realisiert (implementiert) werden können und die von anderen Klassen genutzt werden können.



Beispiel:


MyApplication
MyWindow
Simulation s = new Simulation();
MyWindow w = new MyWindow(s);
w.start();
main(args: String[])
MyWindow(is: ISimulation)
start()
sim = is;
sim.compute();
1
sim
<<interface>>
ISimulation
compute()
Simulation
compute()
perform simulation;

Bemerkungen

- Überall, wo ein Bezeichner mit einem Interface-Typ verwendet wird, kann ein Objekt einer realisierenden Klasse eingesetzt werden.
- Ist I ein Interface-Name und K der Name einer realisierenden Klasse von I, dann ist K ein Subtyp von I.
- Durch Verwendung von Schnittstellen (oder von Oberklassen) können Objekte bestimmter Klassen zur Laufzeit in Beziehung stehen, obwohl die betreffenden Klassen zur Programmierzeit voneinander unabhängig sind.
- Implementierungen von Schnittstellen können ausgetauscht werden ohne Änderungen am Nutzer (Client) vornehmen zu müssen.
- Schnittstellen sind ein wichtiges Strukturierungsmittel für flexible Software-Architekturen.
- Häufig gibt es statt der Benutzungs-Abhängigkeit eine gerichtete Assoziation vom “Client” zum Interface.

Dynamisches Binden

Beispiel:

```
Boot b;
Motorboot m = new Motorboot();
Segelboot s = new Segelboot();
int x;
... "x einlesen" ...
if (x > 0) b = m;
else b = s;
(*) b.druckeInfo();
```

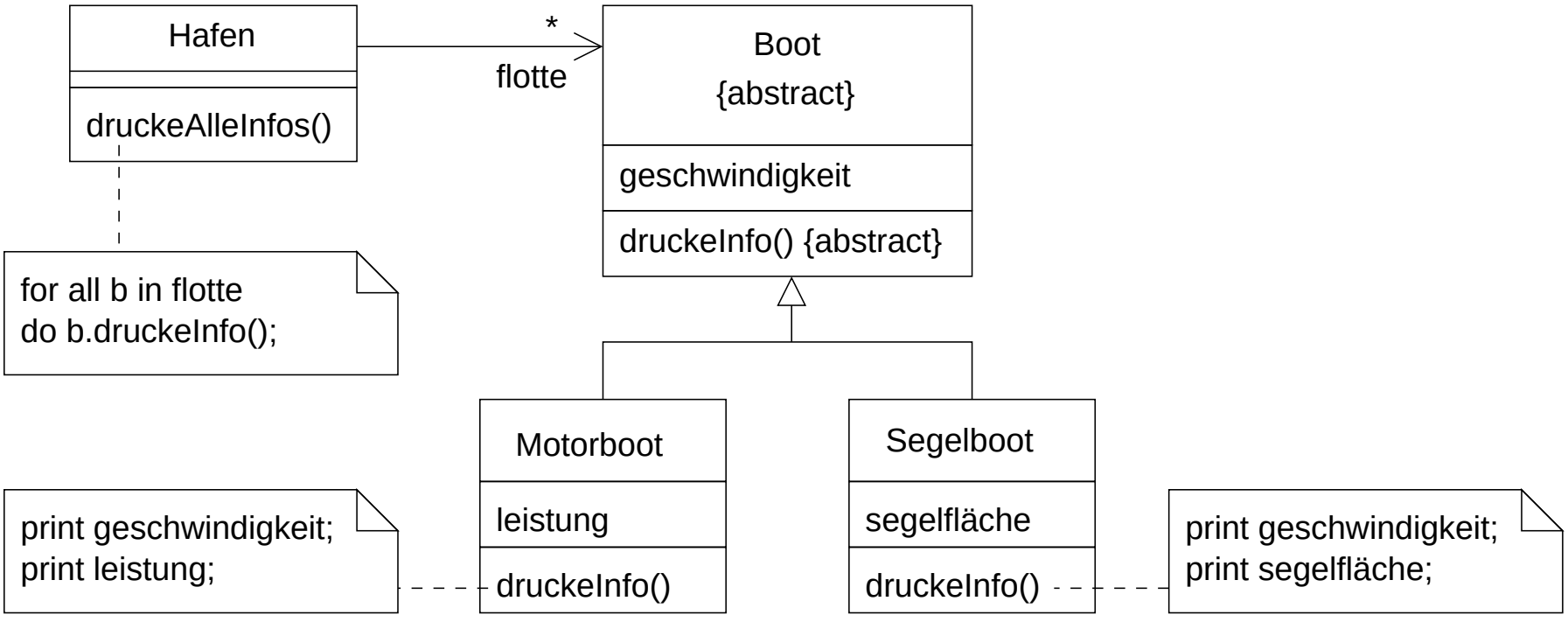
Zur Programmierzeit kann nicht festgestellt werden, welche Implementierung von "druckeInfo()"an der Stelle (*) gültig ist.

Zur Laufzeit wird festgestellt, welchen Typ das mit b bezeichnete Objekt hat. Der in der entsprechenden Klasse implementierte Code wird dann ausgeführt.

Subtyp-Polymorphismus

Eine Operation einer Oberklasse (bzw. einer Schnittstelle) kann für alle Objekte von Unterklassen (bzw. realisierenden Klassen) aufgerufen werden.

Beispiel:

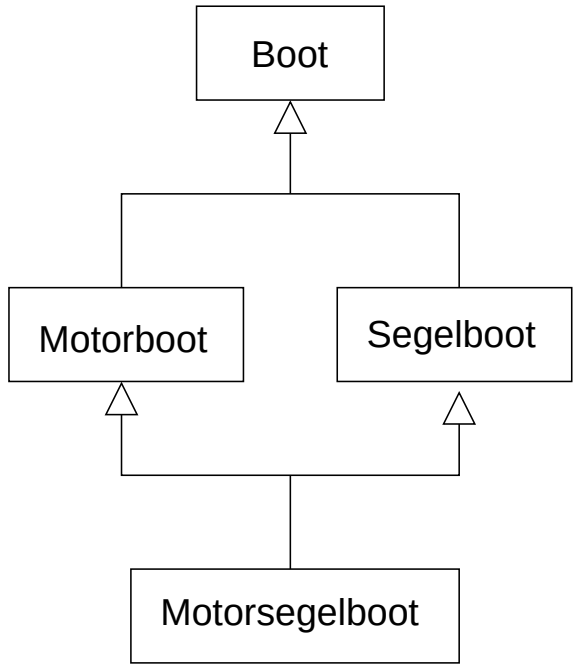


Vorteil:
Einfache Erweiterbarkeit durch Hinzunahme neuer Subklassen.

Mehrfachvererbung

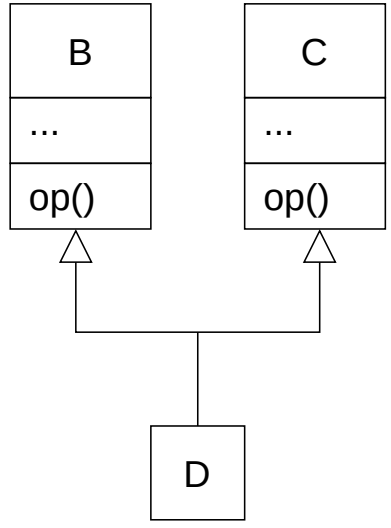
Eine Subklasse hat mehr als eine Oberklasse.

Beispiel:



Vorteil:
Zusammenfügen von Informationen aus mehreren Quellen.

Problem:
Mögliche Konflikte

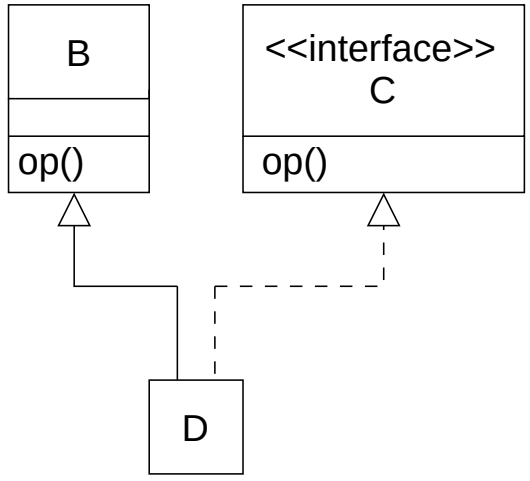


Welche Implementierung von op gilt für D-Objekte?

Konfliktauflösung:
D implementiert op neu durch Überschreiben oder op ist in B oder in C abstrakt.

Bemerkungen

- In Java ist Mehrfachvererbung nur bei Verwendung von Schnittstellen möglich.



- Mehrfachvererbung von Klassen muss beim Entwurf aufgelöst werden, wenn die Zielsprache keine Mehrfachverebung unterstützt.

Vorteile des Vererbungsprinzips

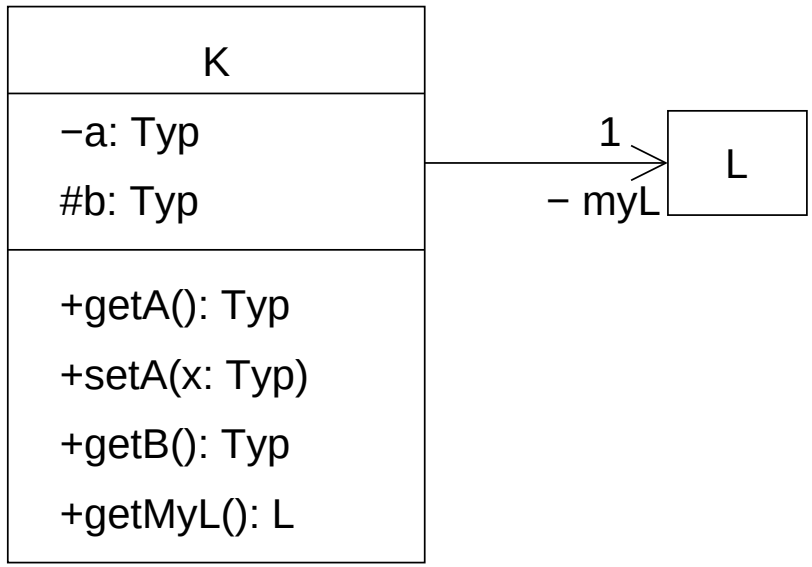
- Konzeptionelle Vereinfachung durch Zusammenfassen gemeinsamer Merkmale verwandter Klassen in einer Oberklasse (*Generalisierung*)
- Wiederverwendung bereits vorhandener Klassen durch Subklassenbildung (*Spezialisierung*)
- Einfache Erweiterbarkeit von Vererbungshierarchien durch Hinzunahme von Subklassen

Vorteile von Schnittstellen

- Keine Abhängigkeit von konkreten Implementierungen.
- Einfache Austauschbarkeit von Realisierungen von Schnittstellen.

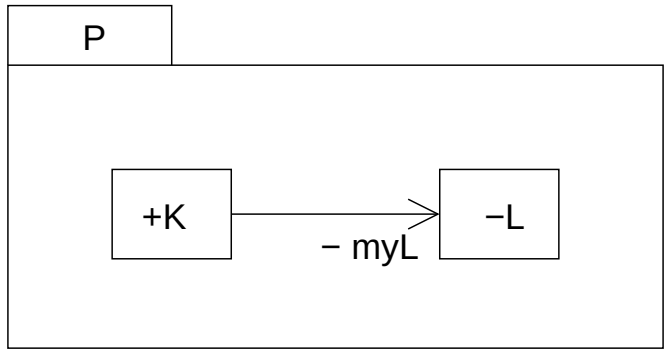
2.1.5 Zugriffsrechte (Sichtbarkeiten)

- Häufig sollen nur bestimmte Merkmale der Objekte einer Klasse von außen zugreifbar sein ("*Kapselungsprinzip*").
- Zur Zugriffskontrolle verwendet man *Sichtbarkeitsmarkierungen* für Attribute, Rollennamen und Operationen:
  - `+name` d.h. öffentlich zugreifbar ("public")
  - `-name` d.h. nur innerhalb der Klasse verwendbar ("private")
  - `#name` d.h. nur innerhalb der Klasse und in allen Subklassen verwendbar ("protected")
- **Regel:** Attribute und Rollennamen sollten nicht öffentlich zugreifbar sein!



Bemerkung

Innerhalb von *Paketen* (vgl. später) können auch Klassen und Interfaces mit Sichtbarkeiten "+"oder "-"versehen werden.



Wirkung:

- Die Klasse K und deren öffentliche Elemente sind auch außerhalb des Pakets P sichtbar.
- Die geschützten Elemente von K sind auch in Subklassen außerhalb des Pakets P sichtbar.
- Die Klasse L und deren öffentliche Elemente sind nur innerhalb des Pakets P sichtbar.
- Die geschützten Elemente von L sind nur in Subklassen innerhalb des Pakets P sichtbar.

Bemerkung

Attribute, Rollennamen und Operationen, die nur innerhalb eines Pakets sichtbar sein sollen, werden mit "∼"markiert ("komponenten-privat").

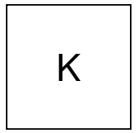
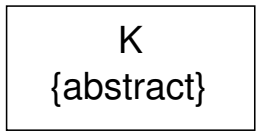
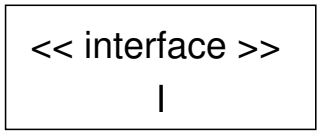
Zusammenfassung von Abschnitt 2.1

- Klassendiagramme werden aus Klassen (einschl. Schnittstellen), Assoziationen, Abhängigkeiten, Vererbungs- und Realisierungsbeziehungen gebildet.
- Objektdiagramme werden aus Objekten und Objektbeziehungen gebildet.
- Mit Hilfe des Vererbungskonzepts kann spezialisiert und generalisiert werden.
- Es gilt das Substitutionsprinzip (bzgl. der Subtypbeziehung).
- Durch dynamische Bindung wird zur Laufzeit festgestellt, welchen Typ ein Objekt hat und der dementsprechende Code ausgeführt.
- Schnittstellen bieten ein wichtiges Strukturierungsmittel für flexible Software-Architekturen.
- Zugriffsrechte können mit Hilfe von Sichtbarkeitsmarkierungen spezifiziert werden.

2.2 Implementierung von Klassendiagrammen in Java

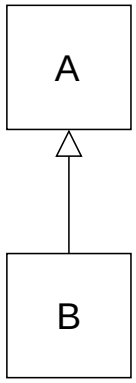
- Die statischen Informationen eines Klassendiagramms können direkt nach Java übersetzt werden.
- Das entstehende Codegerüst enthält noch keine Methodenimplementierungen.

2.2.1 Klassen und Schnittstellen deklarieren

UML	Java
	`class K {...}`
	`abstract class K {...}`
	`interface I {...}`

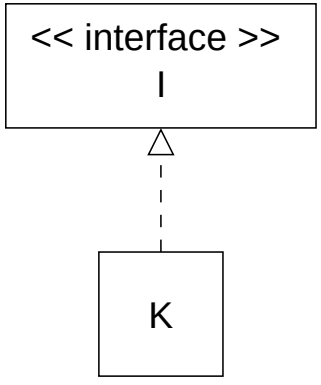
UML

Java




```
class A {...}
```

```
class B extends A {...}
```




```
interface I {...}
```

```
class K implements I {...}
```

2.2.2 Attribute deklarieren

UML	Java
attribut:Typ	JavaTyp attribut;

wobei die verwendeten Standarddatentypen folgendermaßen in Java-Typen übersetzt werden:

UML	Java
Boolean	boolean
Integer	int
Real	float oder double
String	String

2.2.3 Methodenköpfe deklarieren

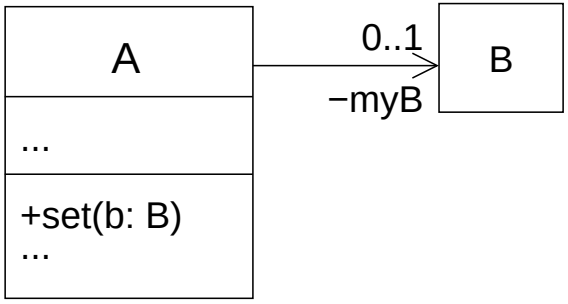
UML	Java
op(x: Typ)	void op(JavaTyp x) {...}
op(x: Typ): ResTyp	JavaResTyp op(JavaTyp x) {...}
op() {abstract}	abstract void op();
K(x: Typ)	K(JavaTyp x) {...} //Konstruktor

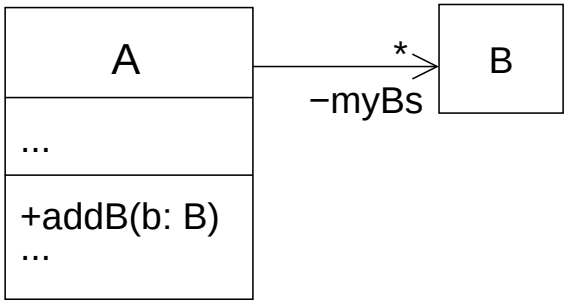
2.2.4 Zugriffsrechte bestimmen

UML	Java	
-	private	//in derselben Klasse sichtbar
$\sharp$	protected	//in Subklassen und im selben Paket sichtbar
+	public	//außerhalb der Klasse sichtbar
$\sim$		//Java Default-Visibility: im selben Paket sichtbar

2.2.5 Gerichtete Assoziationen darstellen

UML





Java

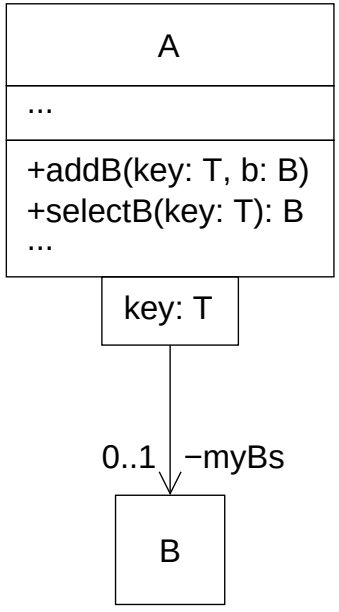
```
class A {
  private B myB; //Referenzattribut
  ...
  public void set(B b) {
    myB = b;
  }
  ...
}
```

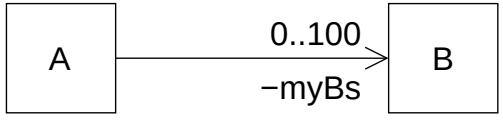
```
//Set = Interface für Mengen
//HashSet = Implementierung v. Set

import java.util.*;

class A {
  private Set<B> myBs = new HashSet<B>();
  ...
  public void addB(B b) {
    myBs.add(b);
  }
  ...
}
```

UML



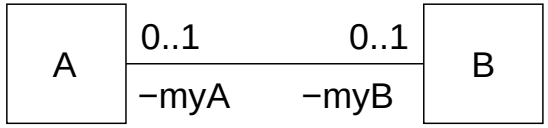


Java

```
//Map = Interface für Schlüssel/Element-Paare
import java.util.*;
class A {
  private Map<T,B> myBs = new HashMap<T,B>();
  ...
  public void addB(T key, B b) {
    myBs.put(key, b);
  }
  public B selectB(T key) {
    return myBs.get(key);
  }
  ...
}
```

```
class A {
  private B[] myBs = new B[100];
  ...
}
```

2.2.6 Bidirektionale Assoziationen implementieren



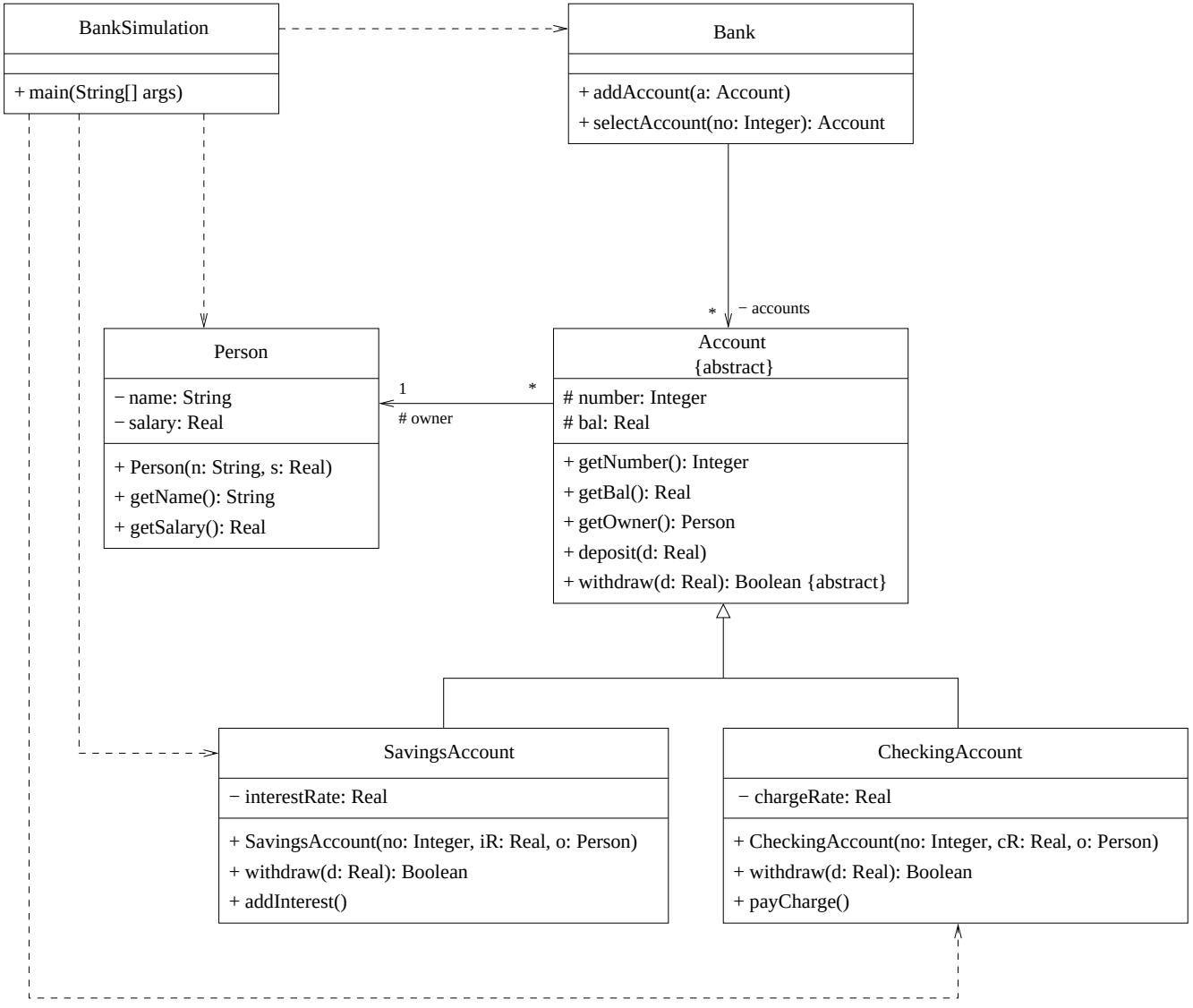

```
class A {
  private B myB;

  public B getMyB() {
    return myB;
  }
  public void relate(B b) {
    myB = b;
    myB.setMyA(this);
  }
  public void unrelate() {
    myB.unset();
    myB = null;
  }
}
```

```
class B {
  private A myA;

  public A getMyA() {
    return myA;
  }
  void setMyA(A a) {
    myA = a;
  }
  void unsetMyA() {
    myA = null;
  }
}
```

2.2.7 Beispiel (Klassendiagramm)



Beispiel (Codegerüst)


```
class BankSimulation {
    public static void main(String[] args) {
        //Rumpf einfuegen
    }
}
import java.util.*;
class Bank {
    private Set<Account> accounts = new HashSet<Account>();
    public void addAccount(Account a) {
        //Rumpf einfuegen
    }
    public Account selectAccount(int no) {
        //Rumpf einfuegen
    }
}
class Person {
    private String name;
    private double salary;
    public Person(String n, double s) {
        //Rumpf einfuegen
    }
    public String getName() {
        //Rumpf einfuegen
    }
    public double getSalary() {
        //Rumpf einfuegen
    }
}
```

```
abstract class Account {
    protected int number;
    protected double bal;
    protected Person owner;

    public int getNumber() {
        //Rumpf einfuegen
    }
    public double getBal() {
        //Rumpf einfuegen
    }
    public Person getOwner() {
        //Rumpf einfuegen
    }
    public void deposit(double d) {
        //Rumpf einfuegen
    }
    public abstract boolean withdraw(double d);
}
class SavingsAccount extends Account {
    private double interestRate;
    public SavingsAccount(int no,double iR,Person o) {
        //Rumpf einfuegen  }
    public boolean withdraw(double d) {
        //Rumpf einfuegen  }
    public void addInterest() {
        //Rumpf einfuegen  }
}
class CheckingAccount extends Account {...}
```

2.3 Modellierung des dynamischen Verhaltens

Techniken

- *Interaktionsdiagramme:*
  Beschreiben die Kommunikation und die Zusammenarbeit von *mehreren* Objekten.
- *Zustandsdiagramme:*
  Beschreiben das Verhalten *eines* Objekts einer bestimmten Klasse während seiner Lebenszeit.
- *Aktivitätsdiagramme:*
  Beschreiben die (evtl. parallelen) Abläufe von Aktivitäten.

2.3.1 Zustände und Ereignisse

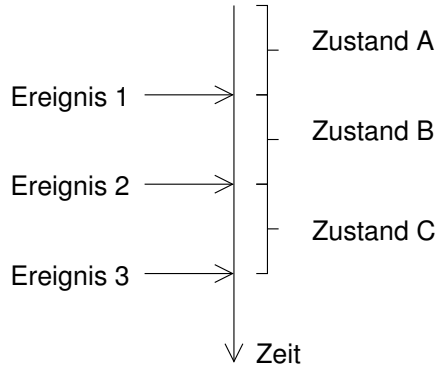
Zustände

- Die aktuellen Attributwerte und Beziehungen eines Objekts zu einem Zeitpunkt bestimmen den *aktuellen Objektzustand*.
- Objektzustände, in denen Objekte qualitativ die gleichen Reaktionen auf eintreffende Ereignisse haben, sind äquivalent. Sie werden zu *einem* (abstrakten) Zustand zusammengefasst.

Beispiel: 4 Briefzustände

In jedem einzelnen Zustand wird dieselbe Briefmarke aufgeklebt.

Ereignis *(event)* = Vorfall, der zu einem bestimmten Zeitpunkt stattfindet.



Arten von Ereignissen

- Empfang eines Signals: *signal event* (z.B. Button klicken, Telefonhörer abheben)
- Aufruf einer Operation: *call event* (z.B. `myKonto.einzahlen(1000)`)
- Eine Bedingung wird wahr: *change event* (z.B. **when** (temperature < 0))
- Ablauf einer Zeit: *time event* (z.B. **after**(5 sec))
- Beendigung einer Aktivität: *completion event* (z.B. WWW-Seite ist geladen)

Beachte:
Ereignisse haben keine Dauer (im Gegensatz zu Zuständen)!

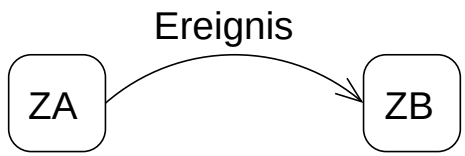
2.3.2 Flache Zustandsdiagramme

Gerichteter Graph mit

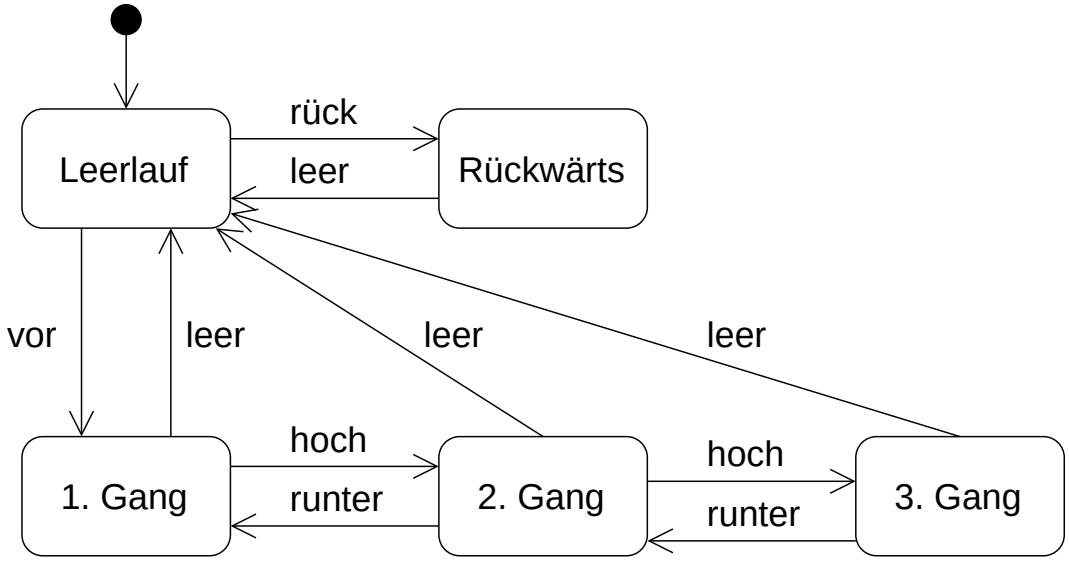
- Knoten = Zustände
- Kanten = Transitionen

Transition

Beschreibt den durch ein Ereignis verursachten Übergang von einem "alten" in einen "neuen" Zustand.



Beispiel: Automatikgetriebe



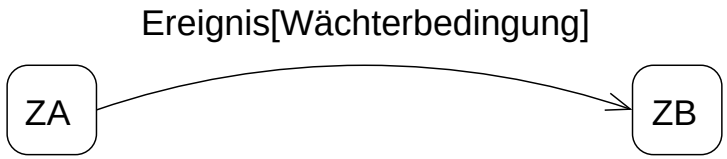
Bemerkungen

- Ein Ereignis, für das es von einem Zustand aus keine Transition gibt, wird in diesem Zustand "ignoriert".
- Das Symbol • bezeichnet den Anfangszustand ("Pseudozustand").
- Das Symbol ⊙ bezeichnet einen Endzustand (Destruktion des Objekts oder Beendigung einer Aktivität).

Wächter *(guards)*

- Eine Bedingung (boolescher Ausdruck) kann als Wächter für eine Transition verwendet werden.
- Die Transition feuert, wenn das Ereignis eintritt *und* die Bedingung erfüllt ist.

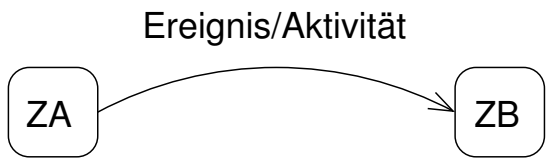
Syntax:



Aktivität

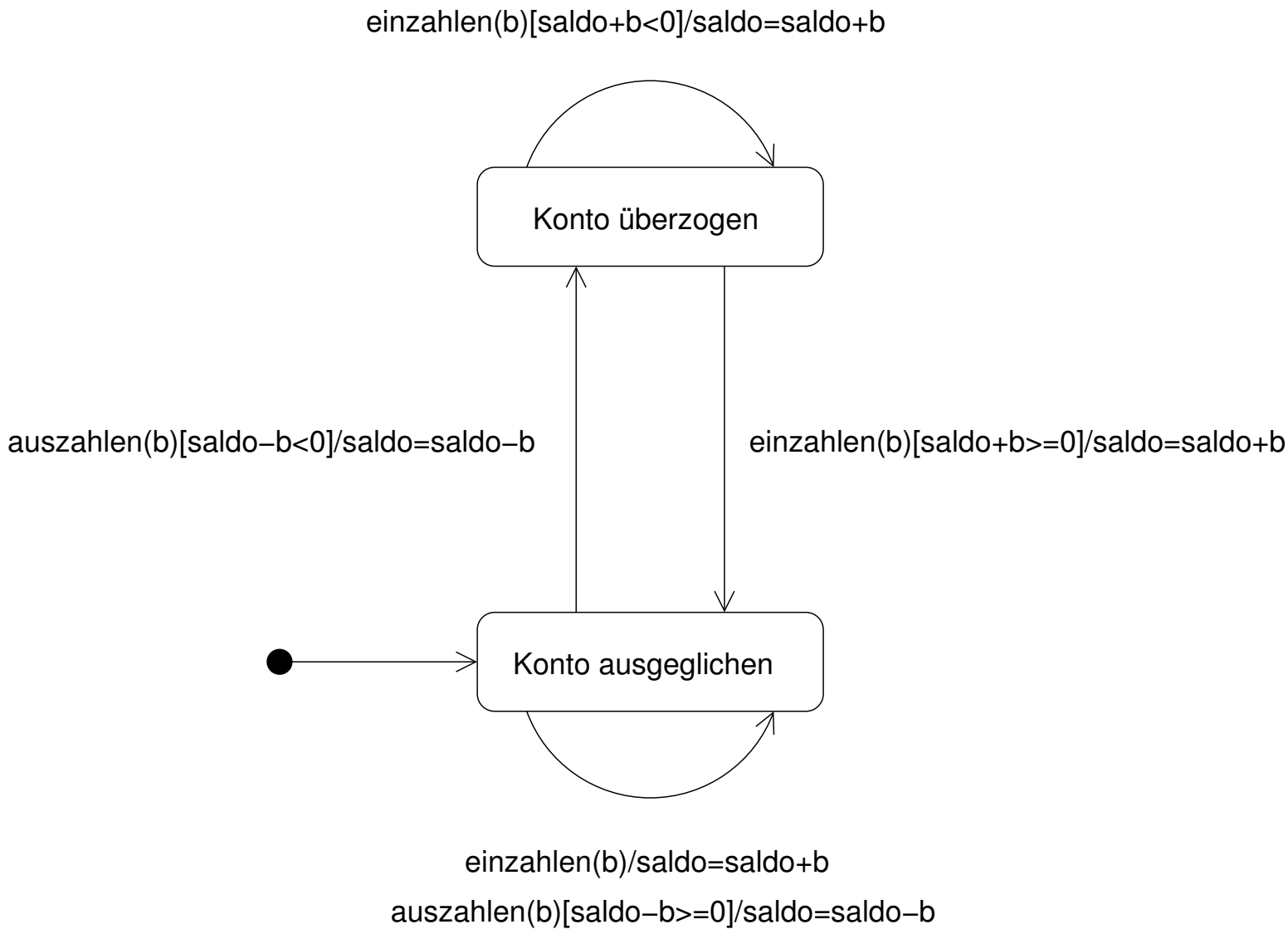
- Tätigkeit (die Zeit in Anspruch nehmen kann).
- Kann als Reaktion auf ein Ereignis erfolgen.

Syntax:

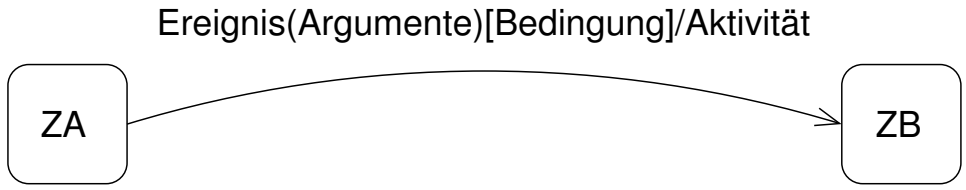


Beachte:
Eine auf einer Transition vorkommende Aktivität kann nicht durch ein Ereignis unterbrochen werden.

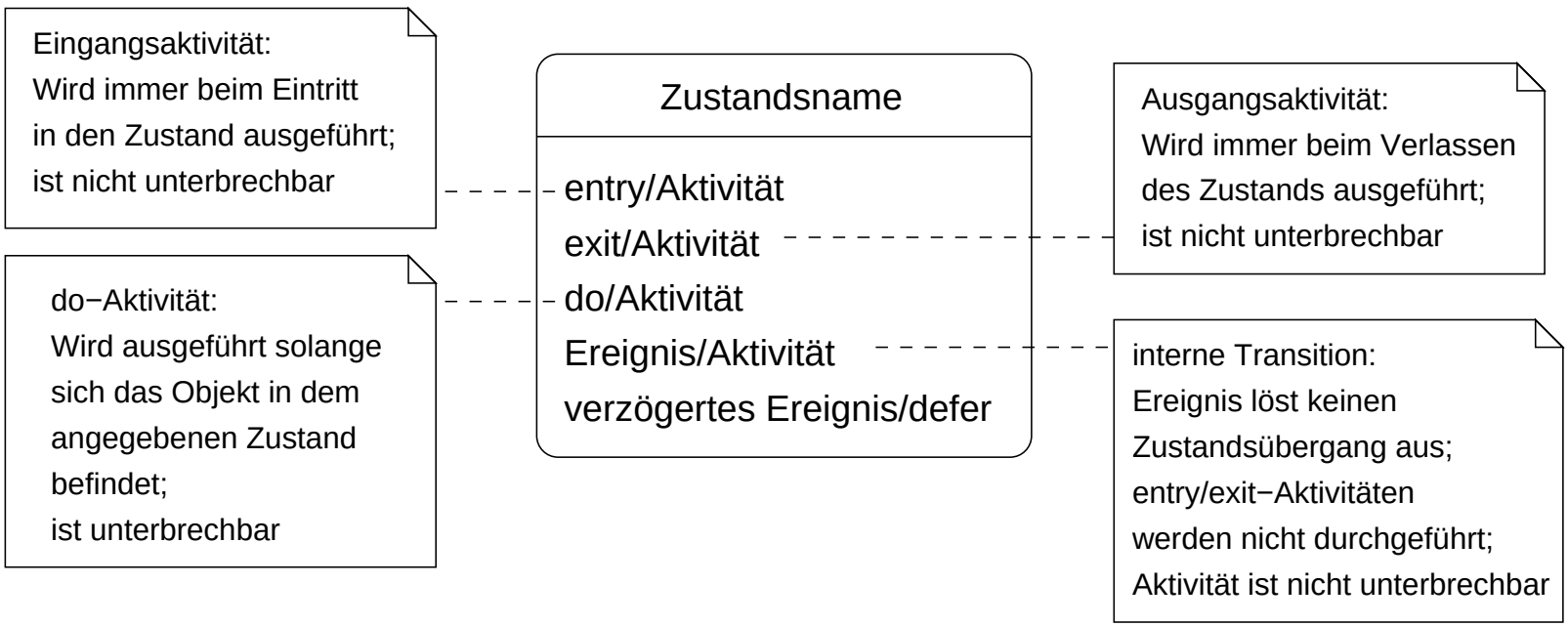
Beispiel:


einzahlen(b)[saldo+b<0]/saldo=saldo+b
Konto überzogen
auszahlen(b)[saldo−b<0]/saldo=saldo−b
einzahlen(b)[saldo+b>=0]/saldo=saldo+b
Konto ausgeglichen
einzahlen(b)/saldo=saldo+b
auszahlen(b)[saldo−b>=0]/saldo=saldo−b

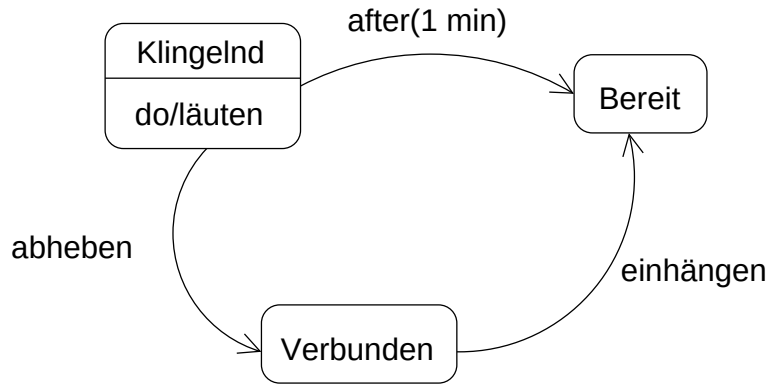
Allgemeine Syntax von Transitionen



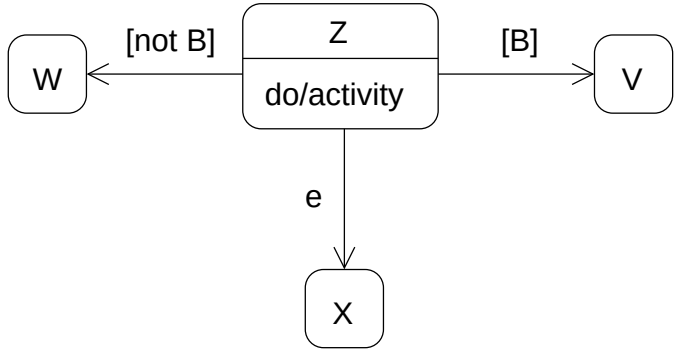
Allgemeine Syntax von Zuständen



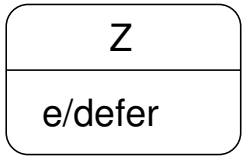
Beispiel: Telefon



Beachte:
Wird die do-Aktivität von selbst beendet, dann erfolgt ein (evtl. bedingtes) Completion Event.

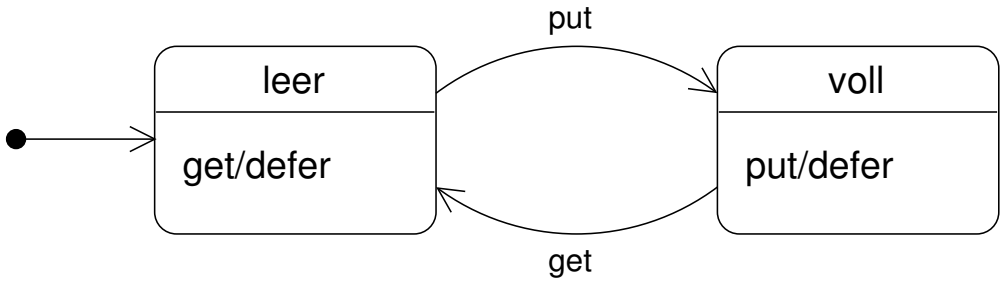


Verzögertes Ereignis



Erfolgt das Ereignis e im Zustand Z und gibt es in Z keine ausgehende Transition mit dem Ereignis e, dann wird das Ereignis aufbewahrt und erst dann verarbeitet, wenn sich das Objekt in einem (anderen) Zustand befindet, in dem das Ereignis verarbeitet werden kann.

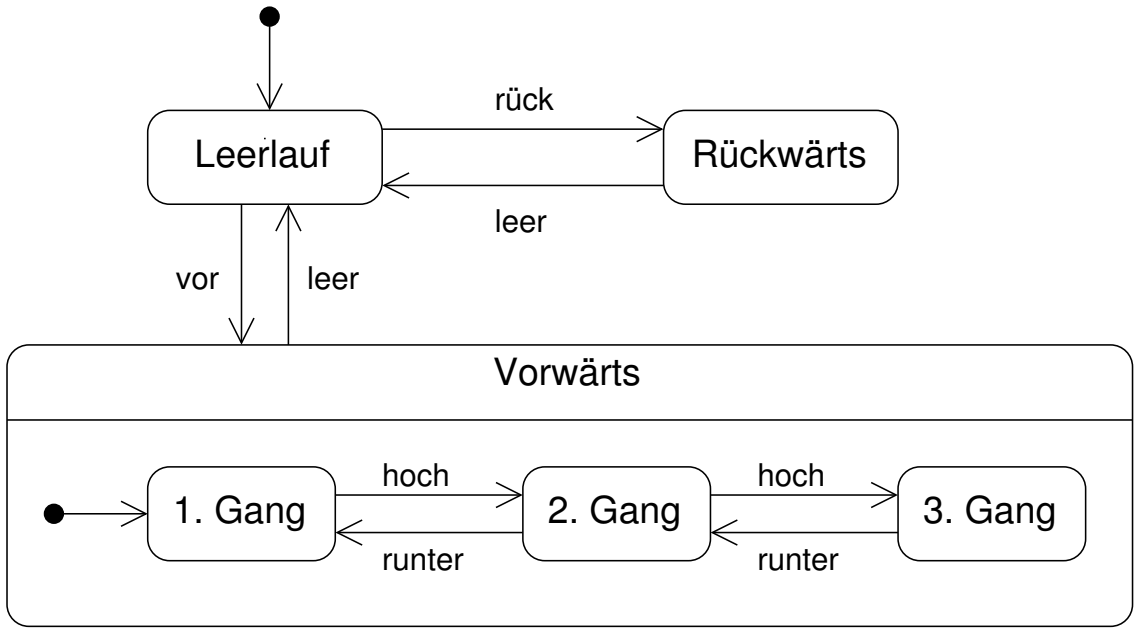
Beispiel: Einelementiger Puffer



2.3.3 Hierarchische Zustandsdiagramme

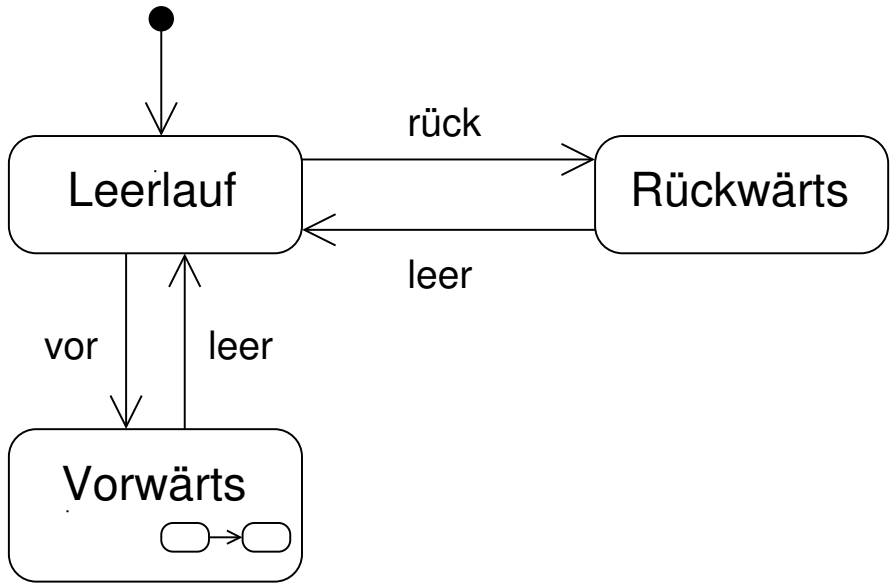
Ein Zustand kann in Unterzustände verfeinert werden.

1. Sequentielle Unterzustände



- Transition in einen Oberzustand (hier: Vorwärts) bedeutet Transition in den Anfangszustand des geschachtelten Diagramms (hier: 1. Gang).
- Transition aus einem Oberzustand bedeutet Transition ausgehend von jedem Unterzustand.

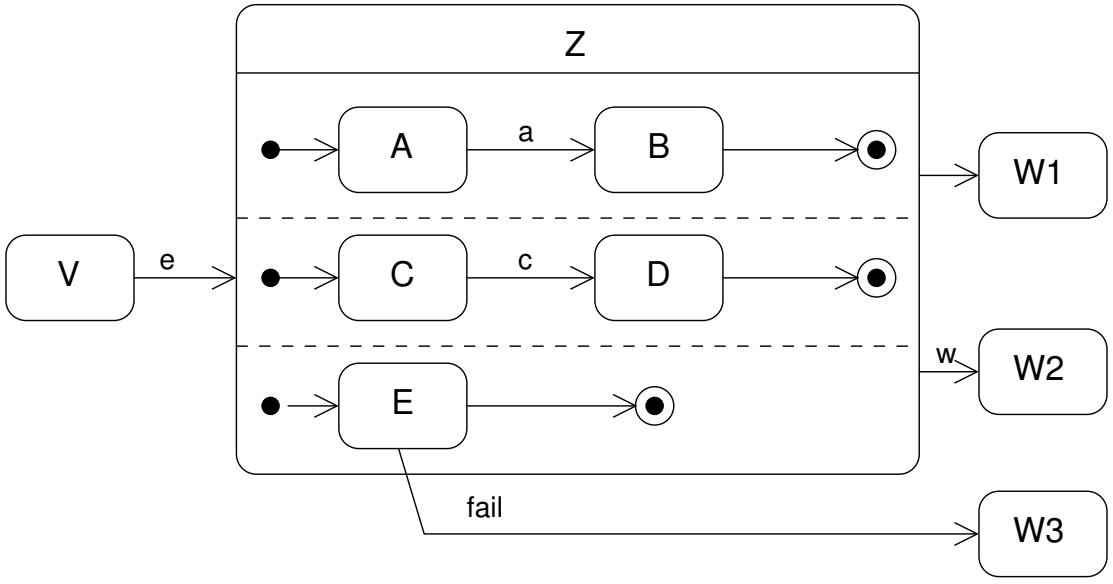
Abstrakte Darstellung eines komplexen Zustands:



Bemerkung

Komplexe Zustände können mit *Einstiegs-* und *Ausstiegspunkten (entry, exit points)* versehen werden.

2. Parallele Unterzustände



- Ein Objekt befindet sich gleichzeitig in mehreren Unterzuständen. Beim Eintritt in den Oberzustand befindet sich das Objekt gleichzeitig in den Anfangszuständen der einzelnen Regionen.
- Der Oberzustand wird verlassen, wenn in jeder Region ein Endzustand erreicht ist oder wenn eine Transition von einem Unterzustand aus direkt nach außen führt oder wenn eine Transition den Oberzustand (wegen eines expliziten Ereignisses) verlässt.

2.3.4 Aktivitätsdiagramme

Können zur Beschreibung der Abläufe von

- Geschäftsprozessen in Unternehmen
- Anwendungsfällen (Use Cases) oder von
- Operationen und Prozessen

verwendet werden.

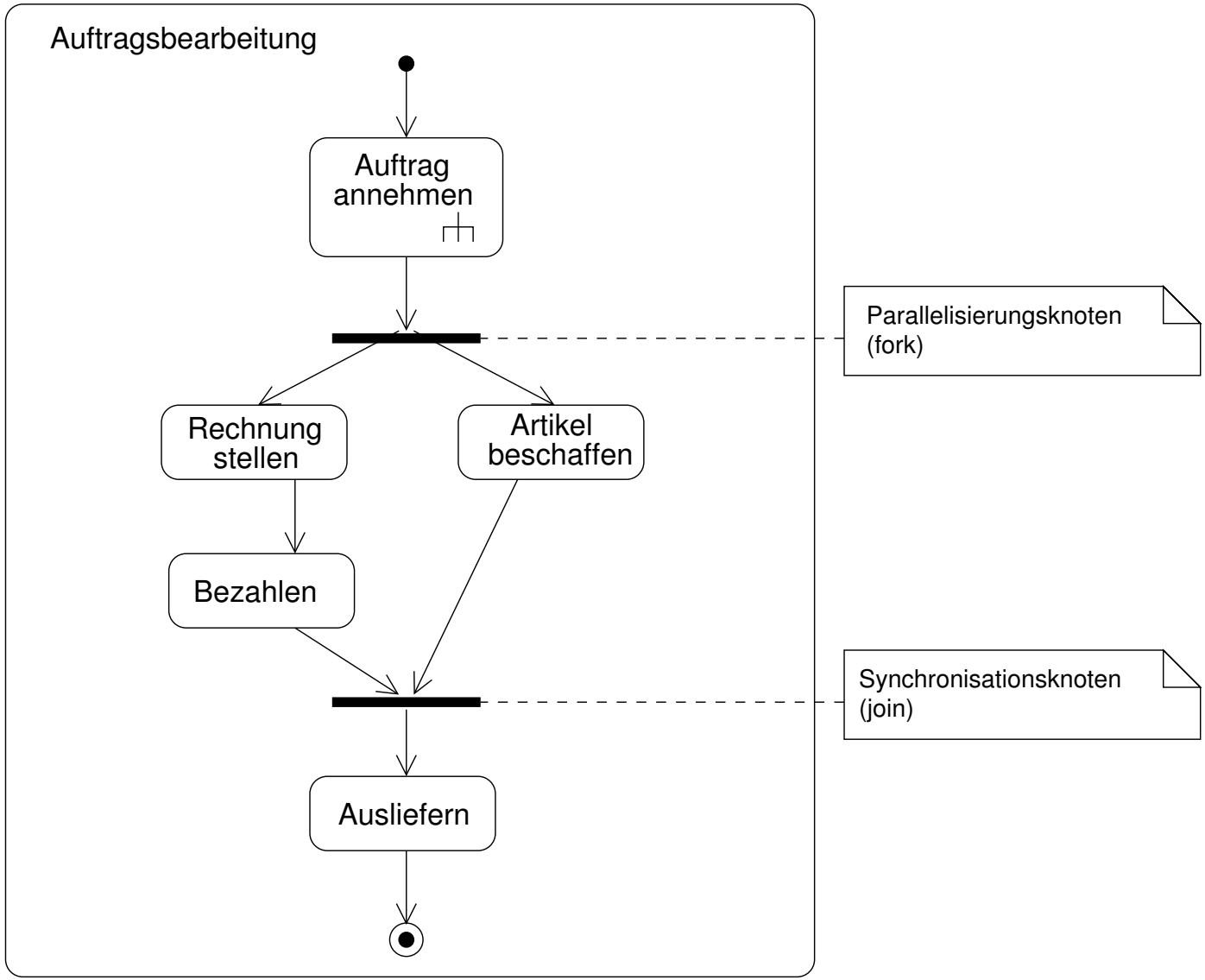
Ein Aktivitätsdiagramm ist ein gerichteter Graph, bestehend aus

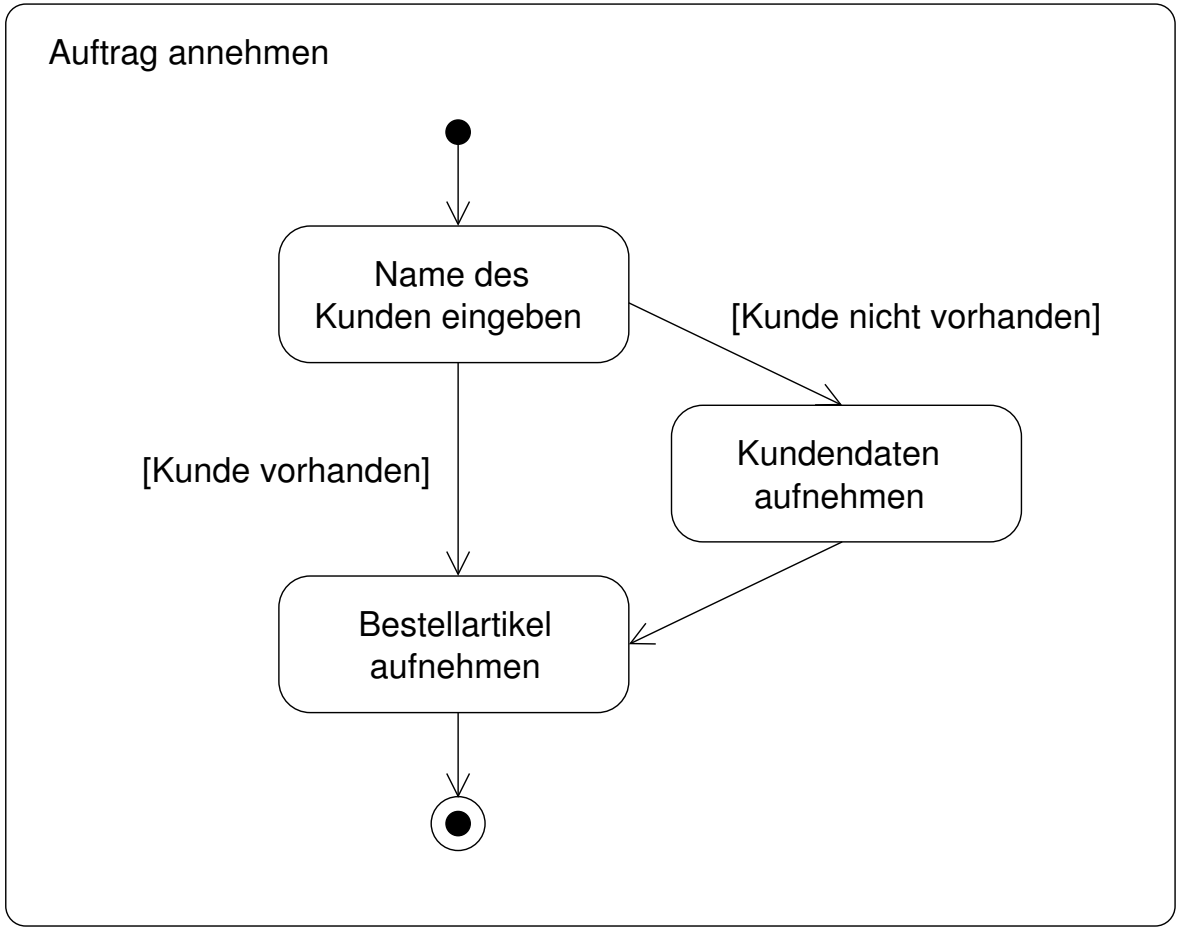
- *Aktivitätsknoten* zur Darstellung von Aktionen, Kontrollstrukturen und Daten,
- *Aktivitätskanten*, die die Aktivitätsknoten verbinden und damit die möglichen Abläufe einer Aktivität beschreiben.

Bemerkung

Beispielsweise können Aktivitäten, die in Form von entry-, exit-. oder do-Aktivitäten an Zustände gebunden sind, durch Aktivitätsdiagramme genauer beschrieben werden.

Beispiel: Geschäftsprozess “Auftragsbearbeitung”





Bemerkung

Fallunterscheidungen können auch durch Verwendung von *Entscheidungsknoten* dargestellt werden.

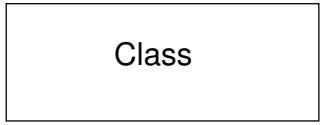
Zusammenfassung von Abschnitt 2.3

- Zustandsdiagramme beschreiben das Verhalten jedes Objekts einer bestimmten Klasse während seiner Lebenszeit.
- Eine Transition beschreibt den durch ein Ereignis verursachten Zustandsübergang.
- Ereignisse sind im Gegensatz zu Zuständen (und Aktivitäten) zeitlos.
- Wir unterscheiden fünf verschiedene Arten von Ereignissen.
- Ereignisse können bewacht sein und auf ein Ereignis kann eine Aktivität folgen.
- Zustandsdiagramme können hierarchisch strukturiert werden.
  Wir unterscheiden
  - sequentielle Unterzustände
  - parallele Unterzustände
- Aktivitätsdiagramme beschreiben Abläufe von Aktivitäten.

2.4 Metamodellierung

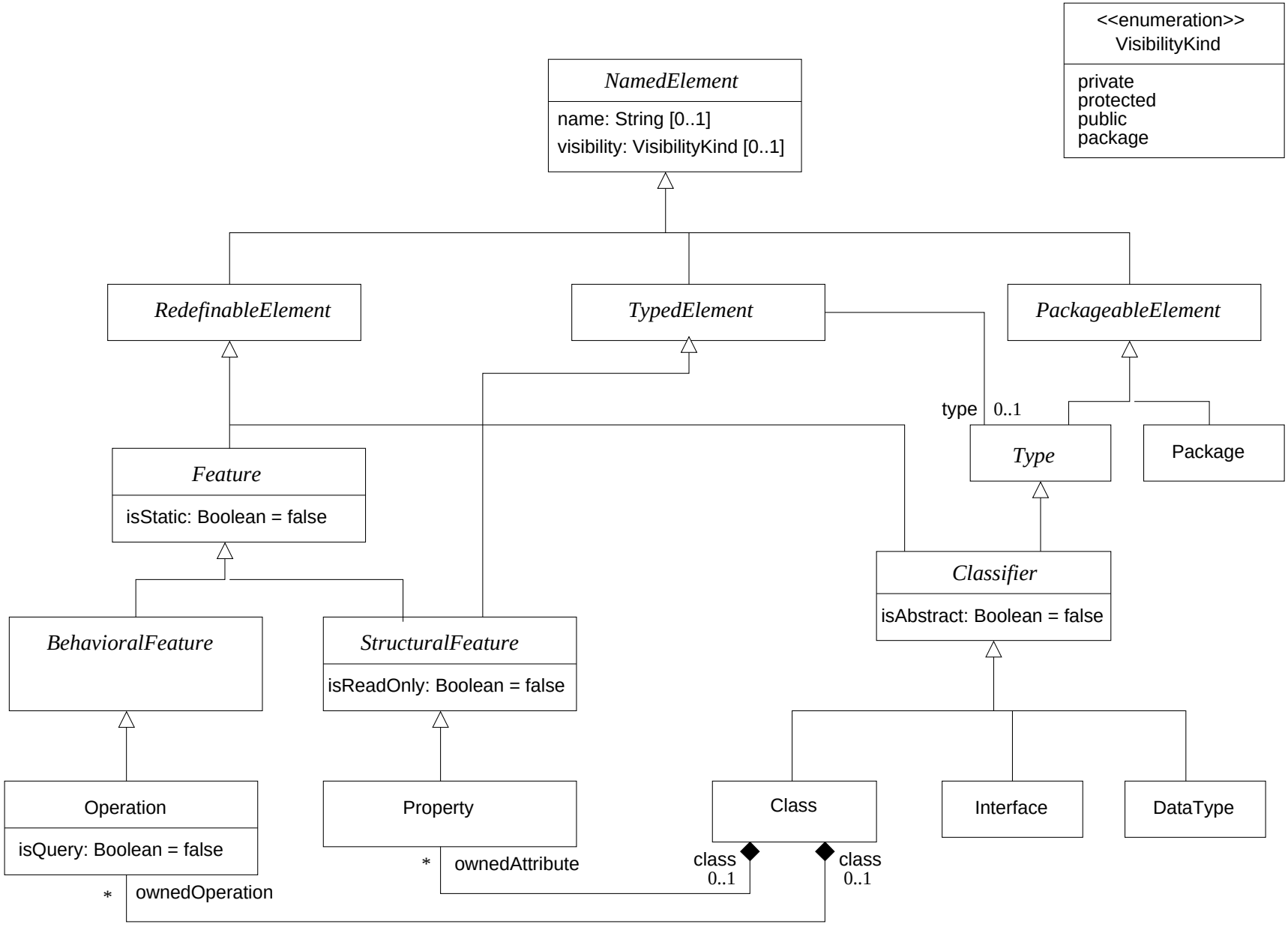
Alle in einem UML-Modell benutzten Konzepte (wie z.B. Klasse, Operation, Zustand, Aktivität, ...) werden selbst durch ein Klassenmodell beschrieben.

Beispiel: Metaklasse Class hat als Instanzen Klassen

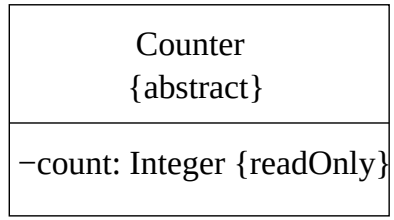


- Das Metamodell spezifiziert alle zulässigen UML-Modelle, die Instanzen des Metamodells sein müssen.
- Damit ist
  - die (syntaktische) Korrektheit von UML-Modellen überprüfbar
  - die Basis für ein standardisiertes Austauschformat geschaffen (XMI)
- Das Metamodell kann für die Modellierung bestimmter Anwendungsbereiche (Business Modeling, Web Engineering, ...) erweitert werden.

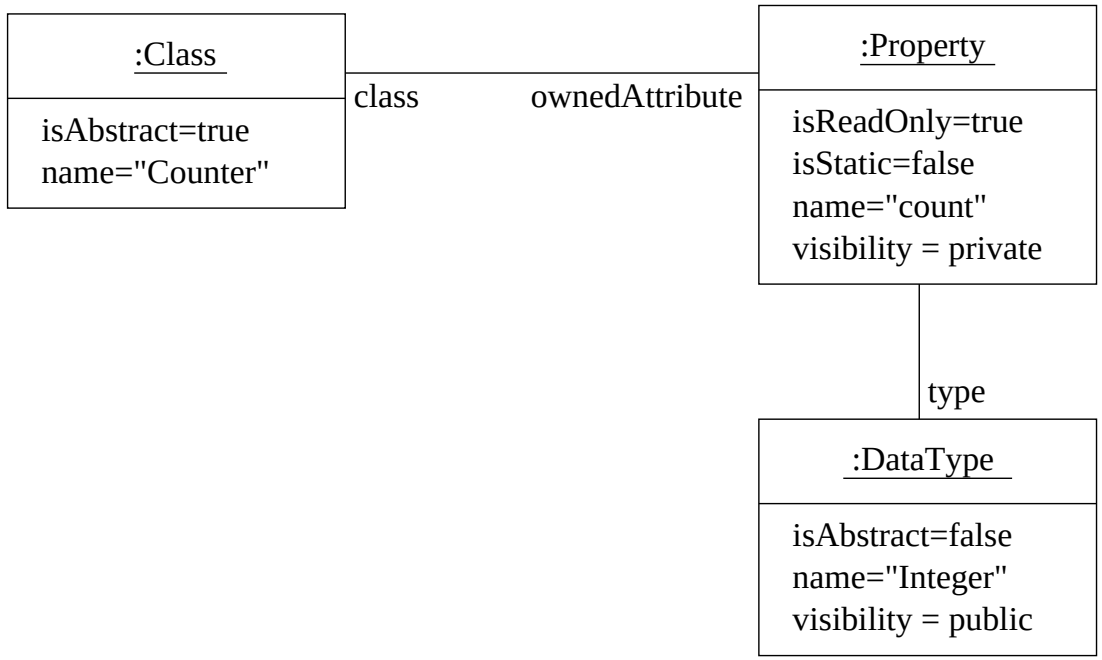
Ausschnitt aus dem UML-Metamodell



Beispiel für die Anwendung des Metamodells



Darstellung der Klasse Counter als Instanzendiagramm des Metamodells



Metamodell für Zustandsdiagramme

(aus der UML 2.2 Superstructure Specification)

Package BehaviorStateMachines