

Objektorientierte Software-Entwicklung

Prof. Dr. Rolf Hennicker

25.10.2007

Kapitel 2

Objektorientierte Modellierungstechniken

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), aktuelle Version 2.0

2.1 Statisches Modell

2.1.1 Klassen und Objekte

Klassen

Allgemeine Form:

Klassenname
attribut attribut: Typ attribut: Typ = Defaultwert
operation operation(Argumentenliste) operation(Argumentenliste): Typ

Beispiel:

Kunde
name: String adresse: String umsatz: Real
Kunde(name: String) setName(name: String) getName(): String umsatzErhoehen(n: Real)

Kurzform:

Klassenname

Objekte

Allgemeine Form:

<u>Objektname</u> :Klassenname
attribut = Wert attribut: Typ = Wert

Beispiel:

<u>:Kunde</u>
name = "Fritz Meier" adresse = "München" umsatz = 4590,30

Kurzformen:

<u>objektname</u>

<u>:Klassenname</u>

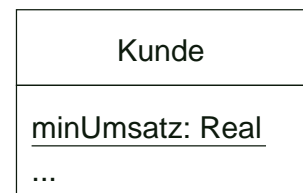
Bemerkungen

- Als Typen verwenden wir Standarddatentypen (Boolean, Integer, Real, String) oder Klassennamen.
- In der Analysephase 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.
- Operationen ändern i.a. den Zustand eines Objekts.
- Operationen, die den Zustand nicht ändern, nennt man "Queries".

Weiterführende Begriffe und Notationen

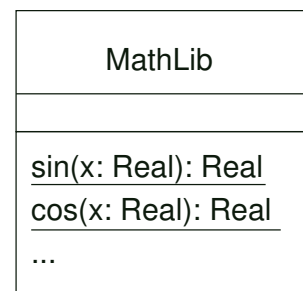
- Ein Attribut, das bei jedem existierenden Objekt der Klasse denselben Wert hat, heißt *Klassenattribut* und wird in UML unterstrichen.

Beispiel:



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

Beispiel:

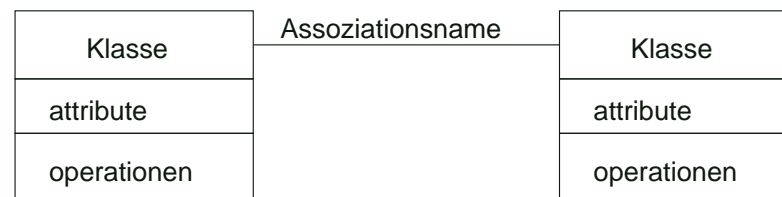


2.1.2 Assoziationen und Objektbeziehungen

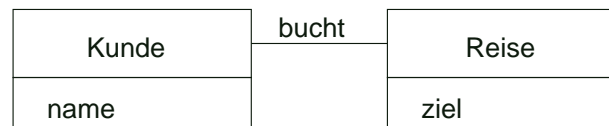
- Eine *Objektbeziehung (Link)* ist eine semantische (physikalische oder konzeptionelle) Verbindung zwischen Objekten.
- Eine *Assoziation* beschreibt eine Menge gleichartiger Beziehungen zwischen Objekten bestimmter Klassen.

Assoziationen

Allgemeine Form:

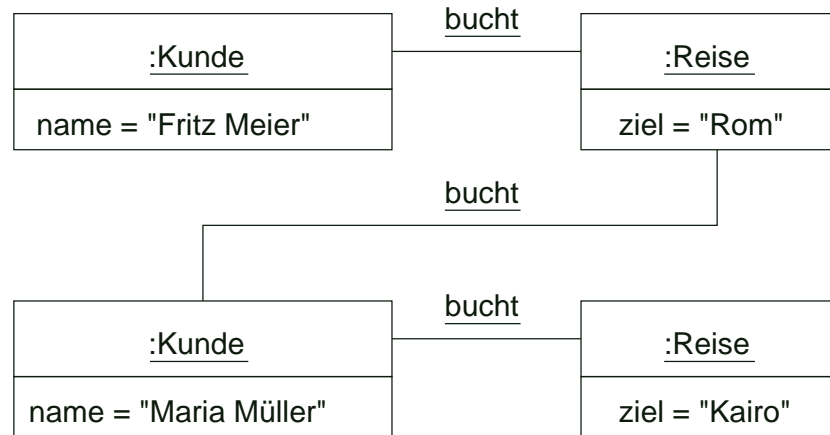


Beispiel:



Objektbeziehungen

Beispiel:



- Ein UML-Diagramm mit Klassen und Assoziationen (und Vererbung) heißt *Klassendiagramm*.
- Ein UML-Diagramm mit Objekten und Objektbeziehungen heißt *Objektdiagramm* (oder *Instanzendigramm*). 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.

Notation:

Bedeutung:



Jedes Objekt von A steht in Beziehung mit

genau einem Objekt von B



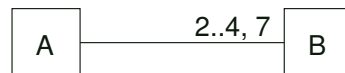
keinem oder einem Objekt von B



einem oder mehreren Objekten von B



beliebig vielen Objekten von B (auch keinem)



2 bis 4 oder 7 Objekten von B

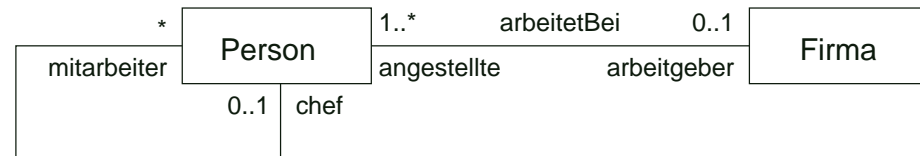
Beispiel:



Assoziationsrollen

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

Beispiel:



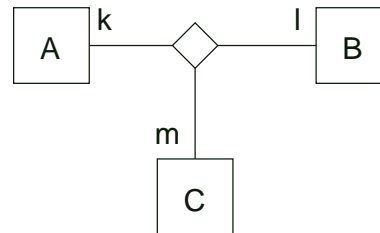
Bemerkung:

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

Mehrstellige Assoziationen

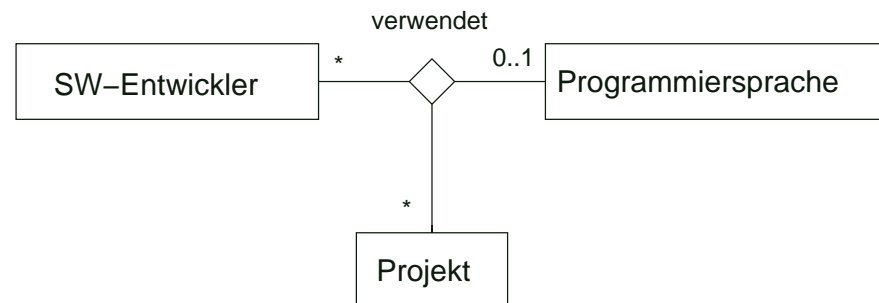
Verbinden drei oder mehr Klassen.

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.

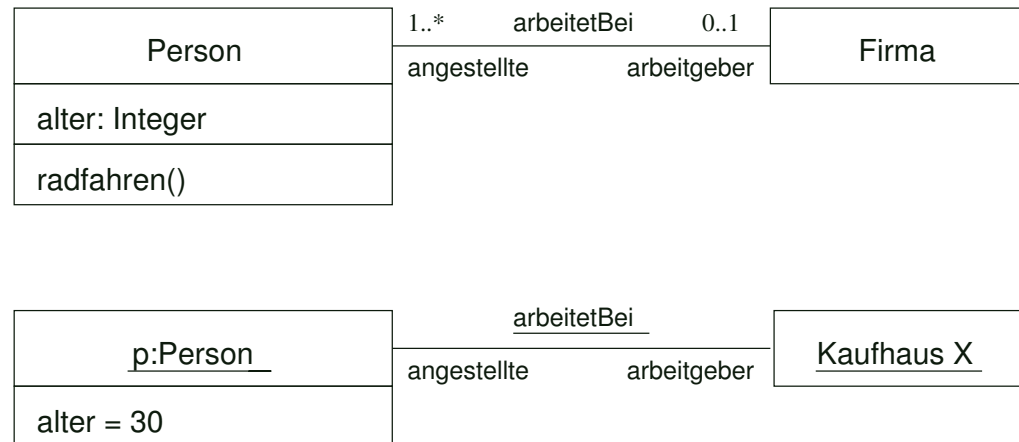
Beispiel:



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" für das Objekt p
  
```

Aggregation

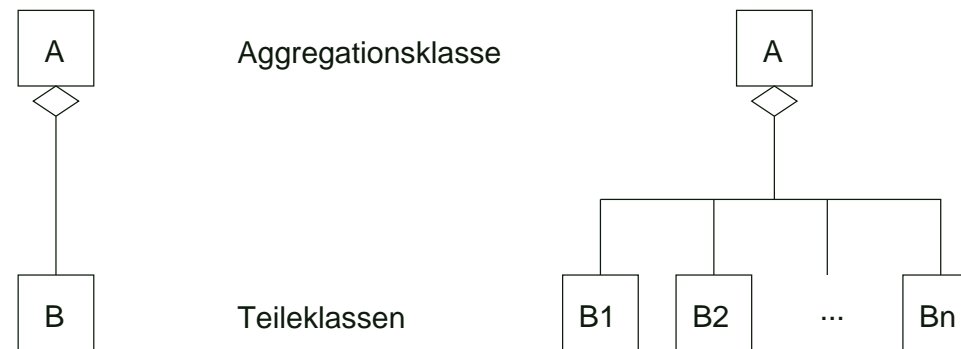
Spezielle Form der Assoziation, die eine "Gesamtheit-Teil"-Beziehung ausdrückt.

z.B.

ICE-Lok besitzt 6 Motoren (physikalisch),

Stundenplan umfasst mehrere Vorlesungen (konzeptionell)

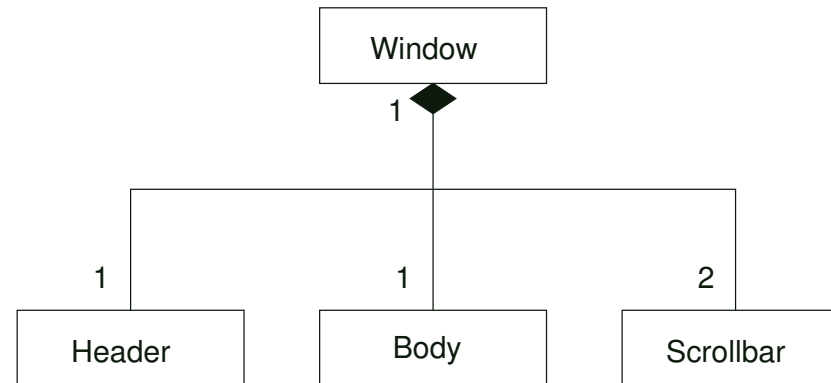
Darstellung:



Komposition

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

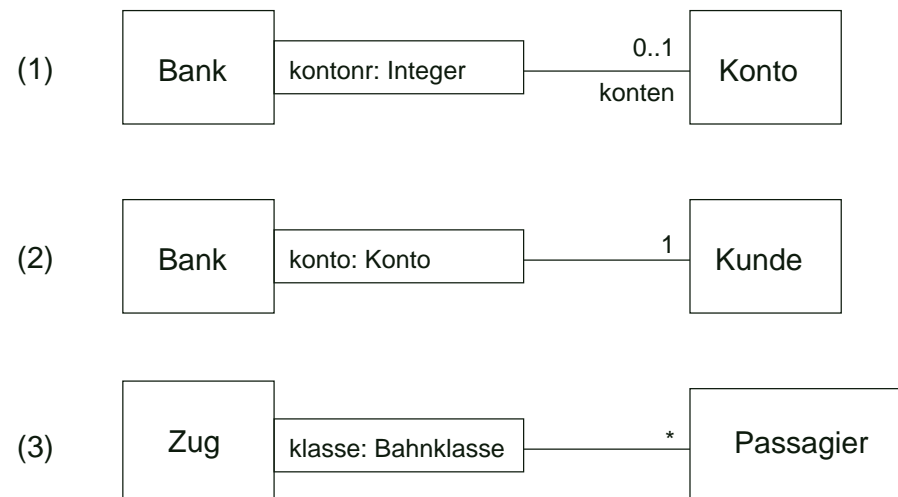
Darstellung (Beispiel):



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.

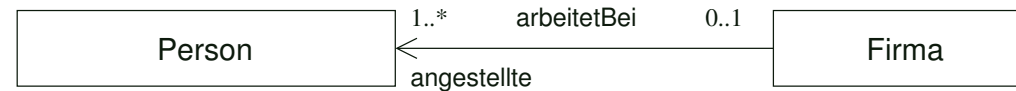
Beispiele:



Gerichtete Assoziation

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

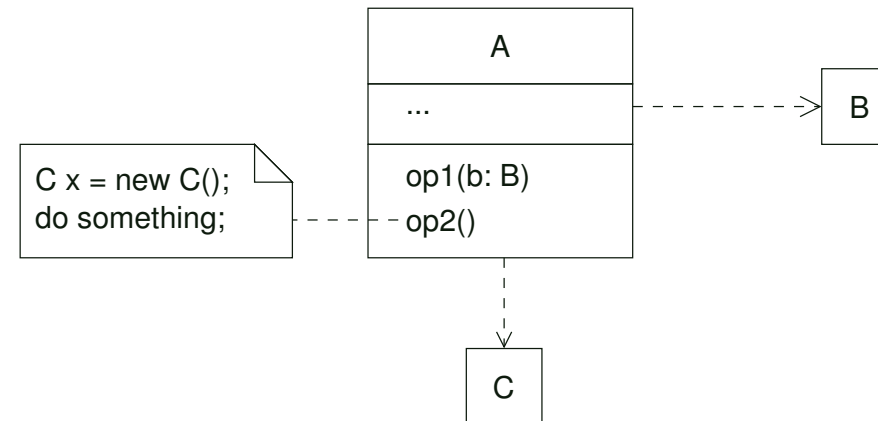
Beispiel:



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:



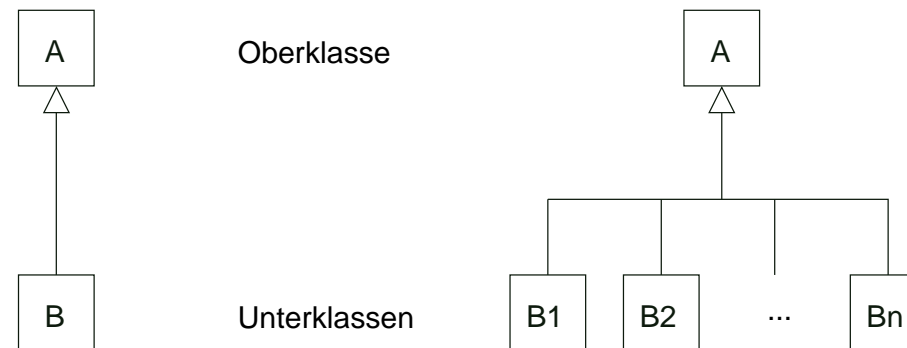
2.1.4 Vererbung

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

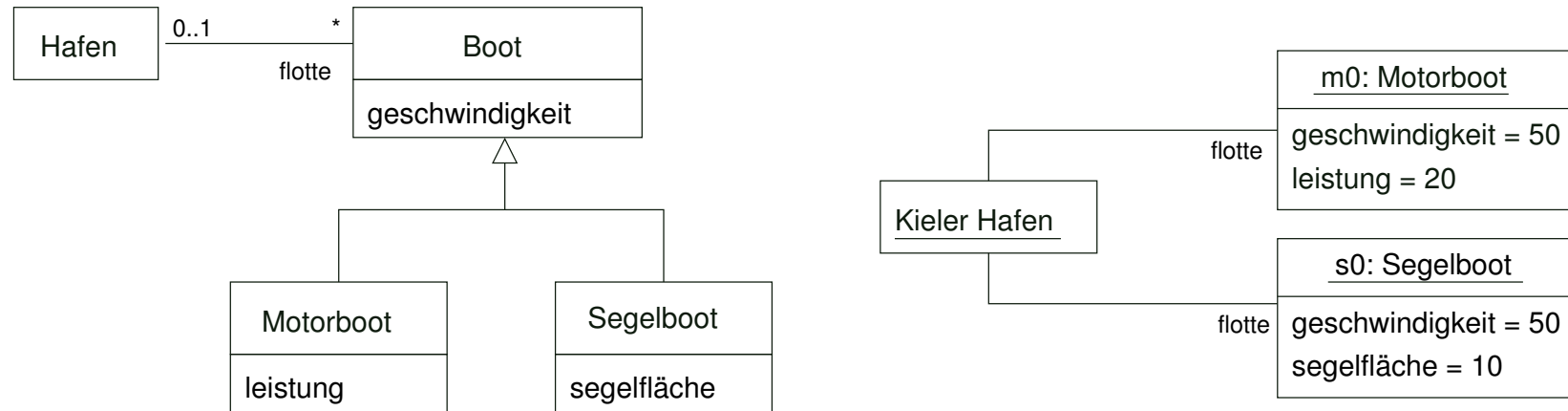
Darstellung:



- A ist eine *Generalisierung* von B. Wichtig in der Analyse
- B ist eine *Spezialisierung* von A. Wichtig im Entwurf und bei Wiederverwendung

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 von A eingesetzt werden.

Abstrakte Klasse

Klasse, von der keine Instanzen erzeugt werden können.

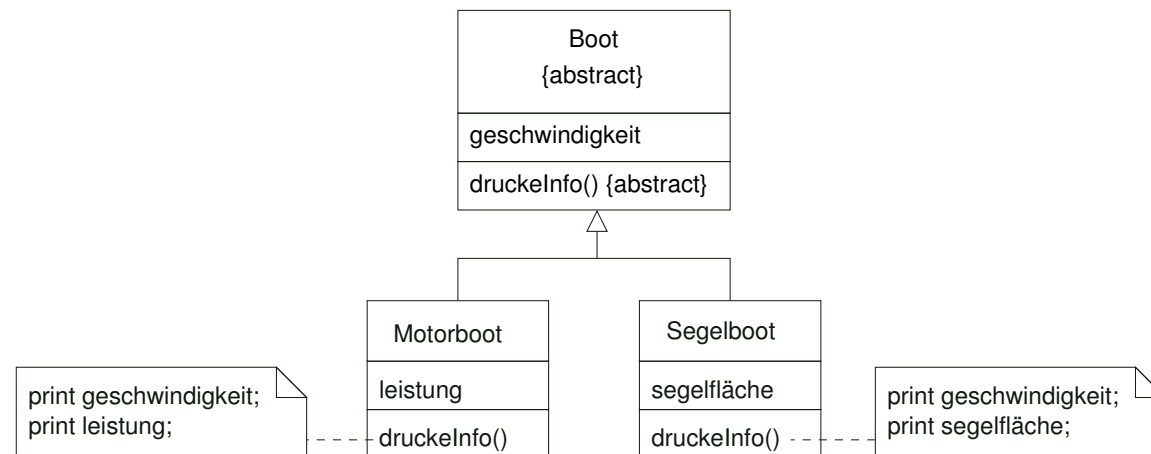
Abstrakte Operation

Operation ohne Implementierung.

Beachte:

Eine Klasse mit mindestens einer abstrakten Operation ist abstrakt.

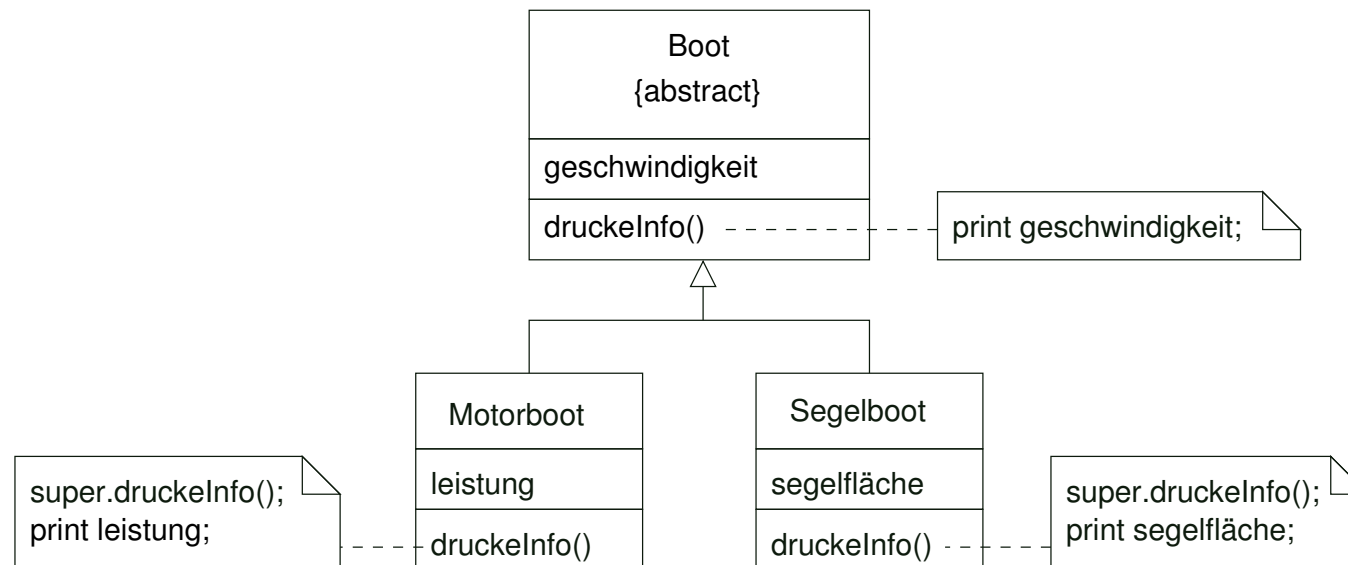
Beispiel:



Überschreiben

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

Beispiel:



Bemerkung:

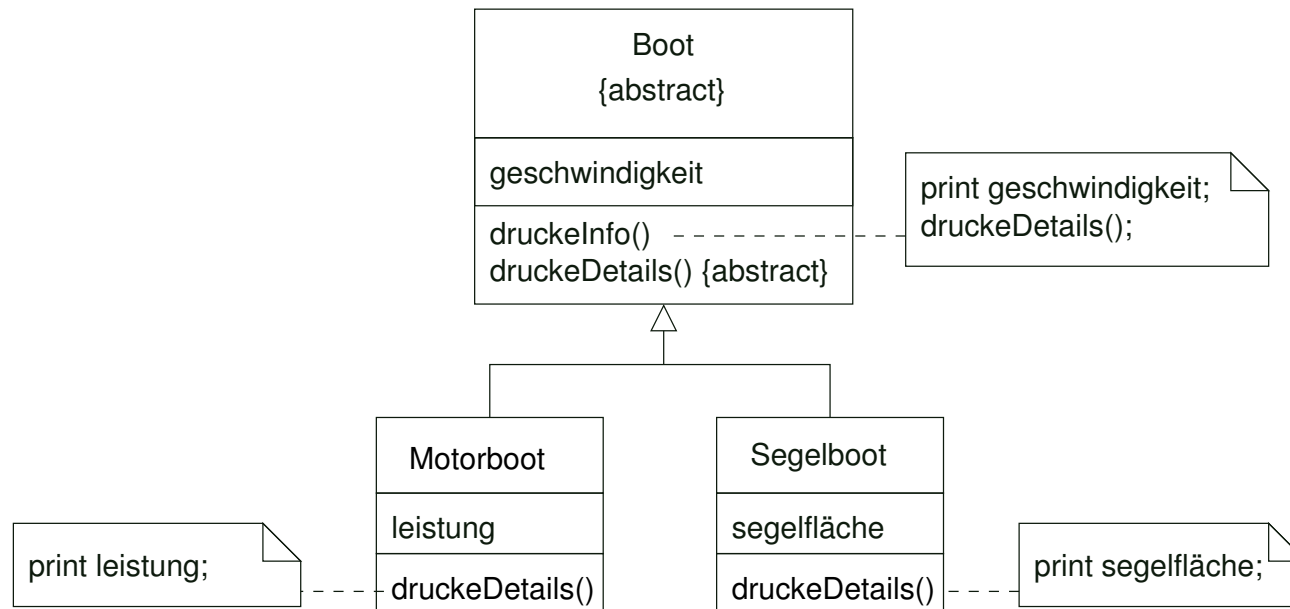
Durch Überschreiben soll die Semantik einer Operation nicht verändert werden.

Gefährlich! Nach Möglichkeit Überschreiben vermeiden (besser Template Operationen)

Template Operation

Operation, deren Implementierung eine oder mehrere abstrakte Operationen aufruft.

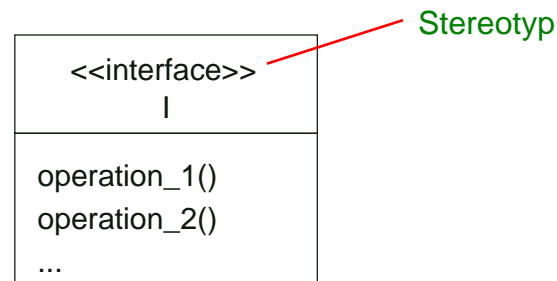
Beispiel:



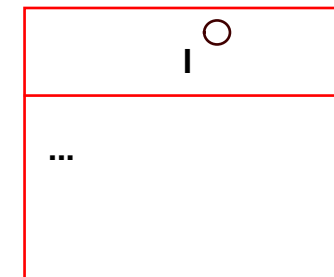
Schnittstellen

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

Darstellung:



Notation mit Icon:



keine Attribute:

=> keine Assoziationen vom Interfaces aus (andersum natürlich schon)
=> alles muss mit Funktionen ausgedrückt werden

(Konstanten {frozen} sind erlaubt, aber Darstellung nicht klar)

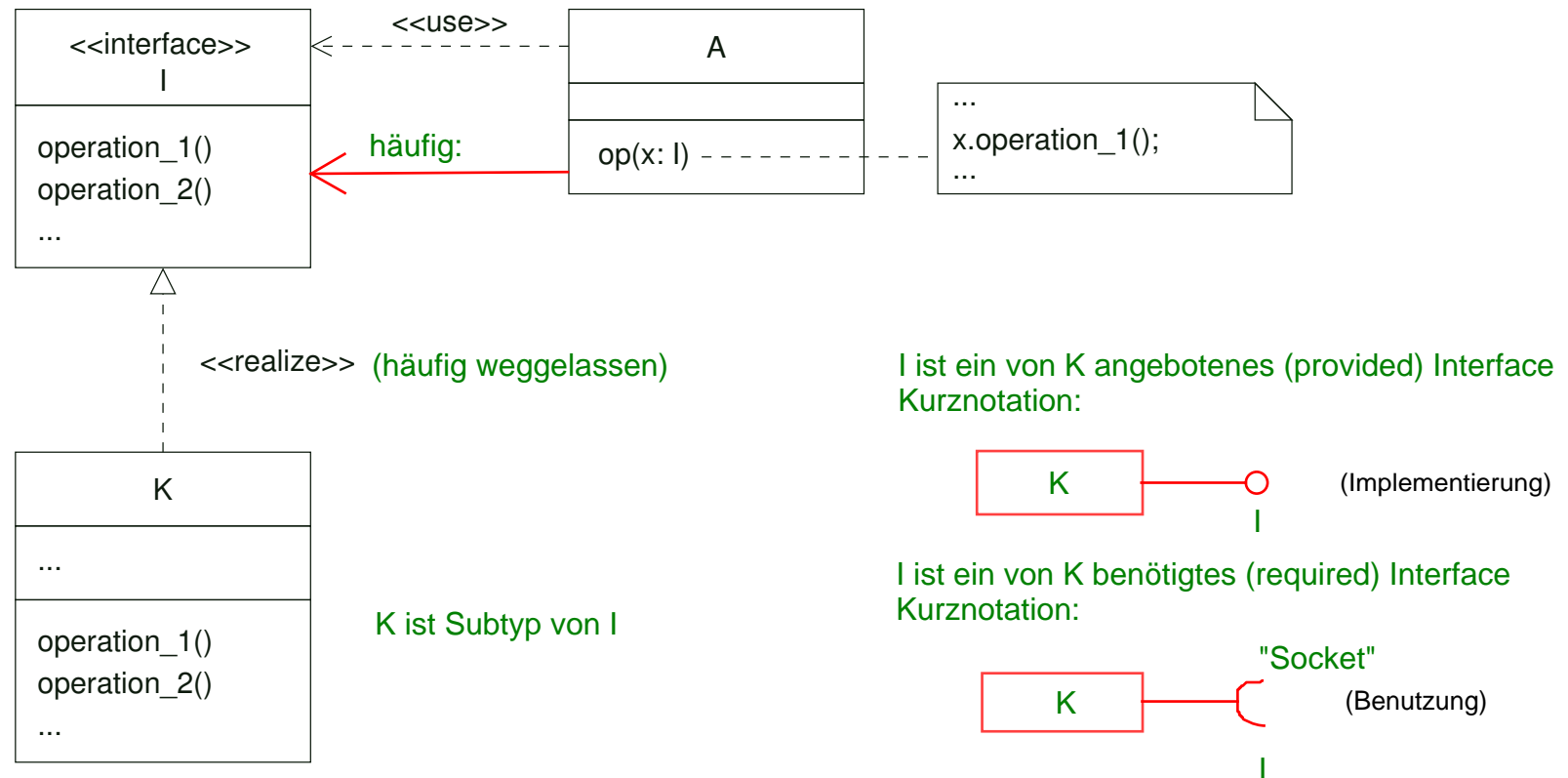
Beachte:

Namen von Schnittstellen, abstrakten Klassen und abstrakten Operationen werden häufig kursiv geschrieben.

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.

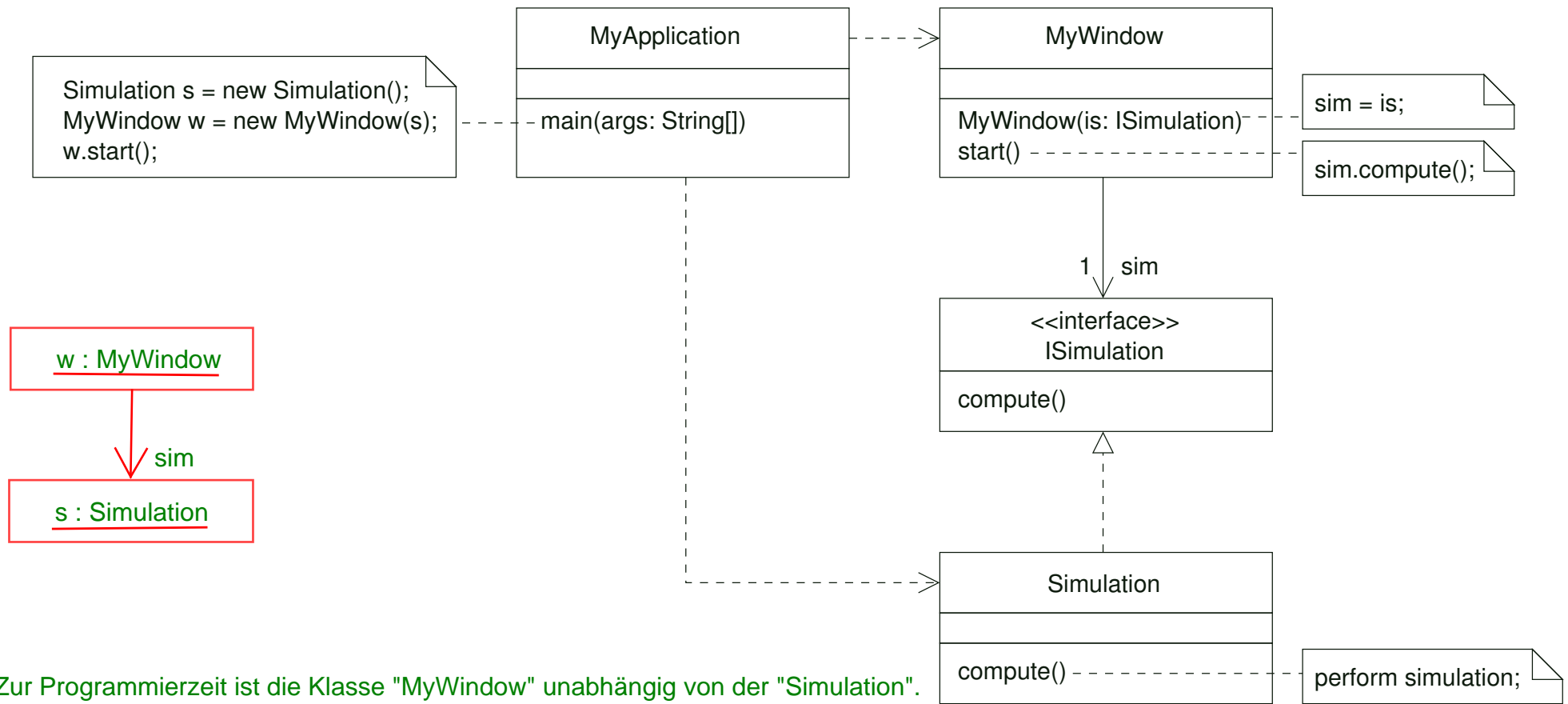
Beispiel:



Bemerkungen

- Überall, wo ein Bezeichner mit einem Interface-Typ verwendet wird, kann ein Objekt einer realisierenden Klasse eingesetzt werden.
- 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.

Beispiel:

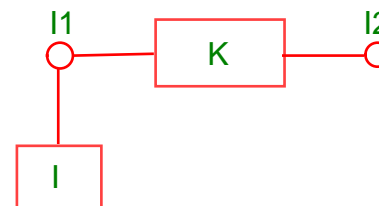


Zur Programmierzeit ist die Klasse "MyWindow" unabhängig von der "Simulation".

Die Klasse "Simulation" kann durch eine andere Klasse ersetzt werden (die "ISimulation" implementiert), ohne Änderung von "MyWindow"

Schnittstellen können untereinander in Vererbungsbeziehung stehen.

Eine Klasse K kann mehrere Schnittstellen implementieren und umgekehrt kann eine Schnittstelle von mehreren Klassen implementiert werden

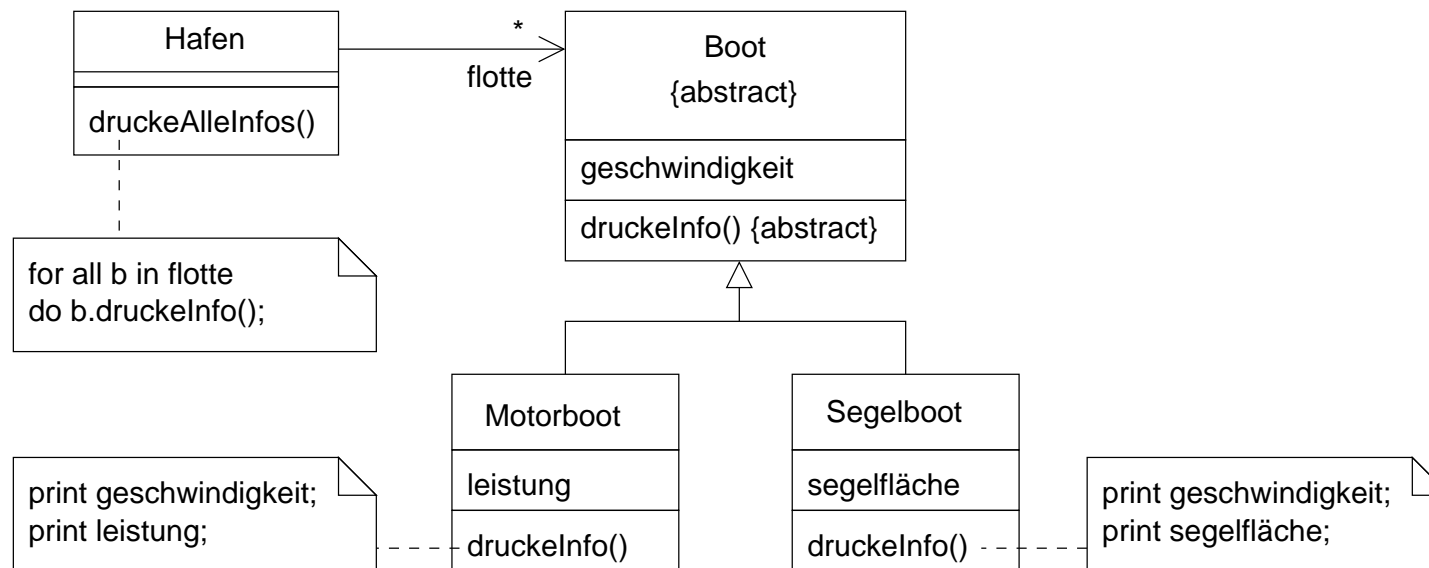


WICHTIGE FOLIE
 (erklärt viel, typische Aufgabe: wie bekomme ich Abhängigkeit zwischen Klassen unter Verwendung von Interfaces weg?)

(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.

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();
```

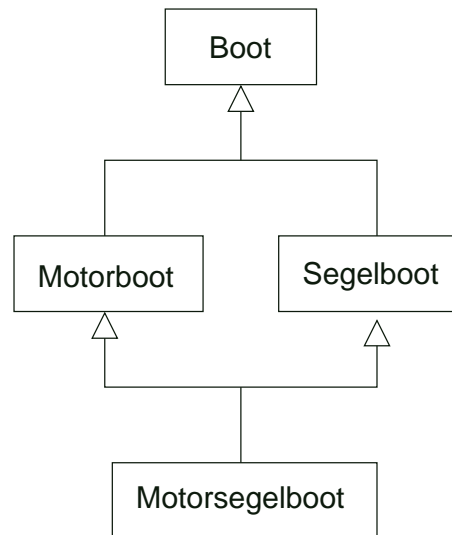
Zur **Programmierzeit** kann nicht festgestellt werden, welche Implementierung von "druckeInfo()" an der Stelle (*) gültig ist.

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

Mehrfachvererbung

Eine Subklasse hat mehr als eine (direkte) Oberklasse.

Beispiel:

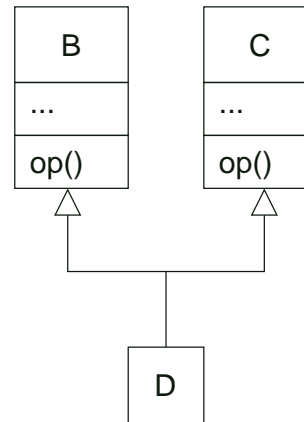


Vorteil:

Zusammenfügen von Informationen aus mehreren Quellen.

Problem:

Mögliche Konflikte



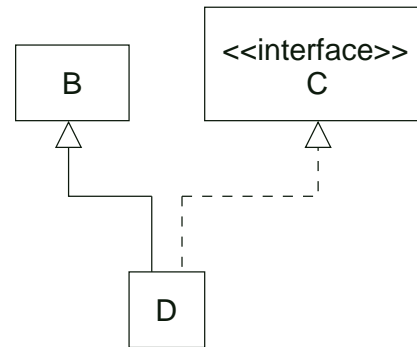
Welche Implementierung von op gilt für D-Objekte?

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.



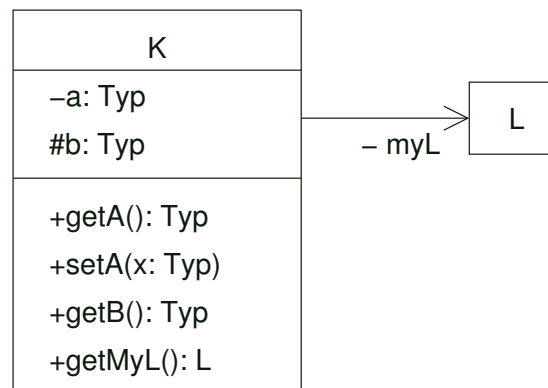
- Mehrfachvererbung von Klassen muss beim Entwurf aufgelöst werden, wenn die Zielsprache keine Mehrfachvererbung unterstützt.

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
- 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:
Elemente einer öffentlichen Klasse, die (nur) innerhalb eines Pakets sichtbar sein sollen werden mit "~" markiert.

Zusammenfassung von Abschnitt 2.1

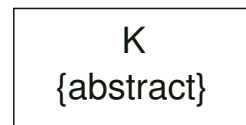
- Klassendiagramme werden aus Klassen (einschl. Schnittstellen), Assoziationen, Abhängigkeiten und Vererbungsbeziehungen gebildet.
- Objektdiagramme werden aus Objekten und Objektbeziehungen gebildet.
- Mit Hilfe des Vererbungskonzepts kann spezialisiert und generalisiert werden.
- Es gilt das Substitutionsprinzip.
- 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

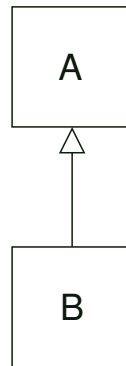


Java

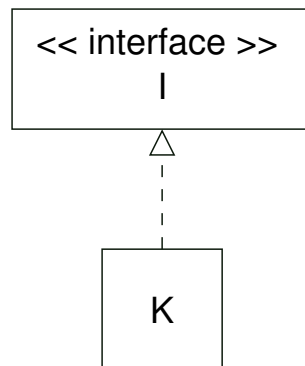
```
class K {...}
```

```
abstract class K {...}
```

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

UML**Java**

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



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

2.2.2 Attribute deklarieren

UML

attribut:Typ

Java

JavaTyp attribut;

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

UML

Boolean

Integer

Real

String

Java

boolean

int

float oder double

String

2.2.3 Methodenköpfe deklarieren

UML

op(x: Typ)

op(x: Typ): ResTyp

op() {abstract}

K(x: Typ)

Java

```
void op(JavaTyp x) {...};
```

```
JavaResTyp op(JavaTyp x) {...};
```

```
abstract void op();
```

```
K(JavaTyp x) {...}; //Konstruktor
```

2.2.4 Zugriffsrechte bestimmen

UML

Java

-

private

#

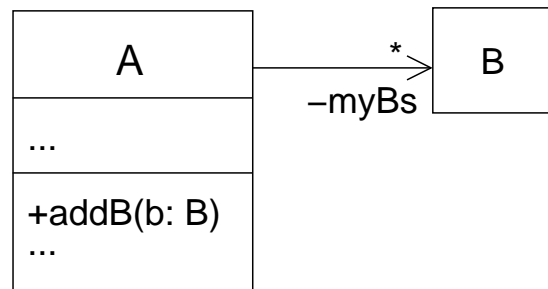
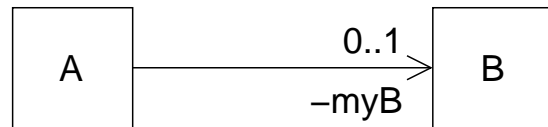
protected //in Subklassen und im selben Paket sichtbar

+

public

2.2.5 Assoziationen darstellen

UML



Java

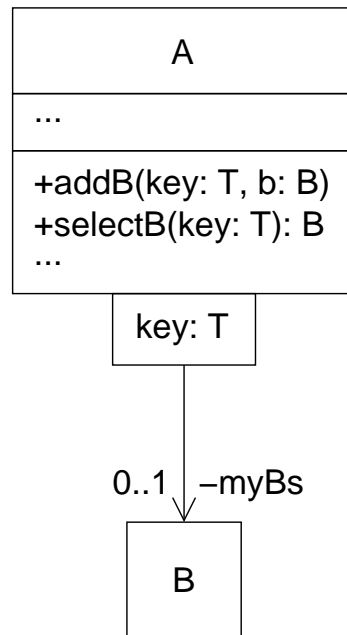
```

class A {
    private B myB; //Referenzattribut
    ...
}
  
```

```

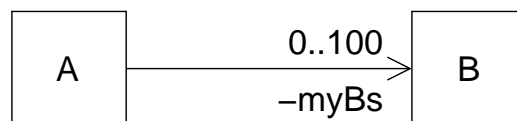
//Set = Interface für Mengen
//HashSet = Implementierung v. Set
import java.util.*;

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

UML*Java*

```

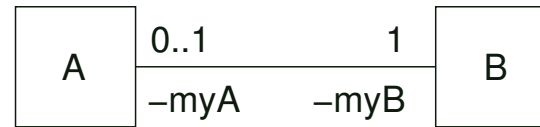
//Map = Interface f. Schlüssel/Element-Paare
import java.util.*;
class A {
    private Map<T,B> myBs = new HashMap<T,B>()
    ...
    public void addB(T key, B b) {
        myBs.put(key, b);
    }
    public B selectB(T key) {
        return myBs.get(key);
    }
    ...
}
  
```



```

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

Bidirektionale Assoziationen implementieren



```
class A {
    private B myB;

    public A(B b) {
        myB = b;
        myB.setMyA(this);
    }

    public B getMyB() {
        return myB;
    }

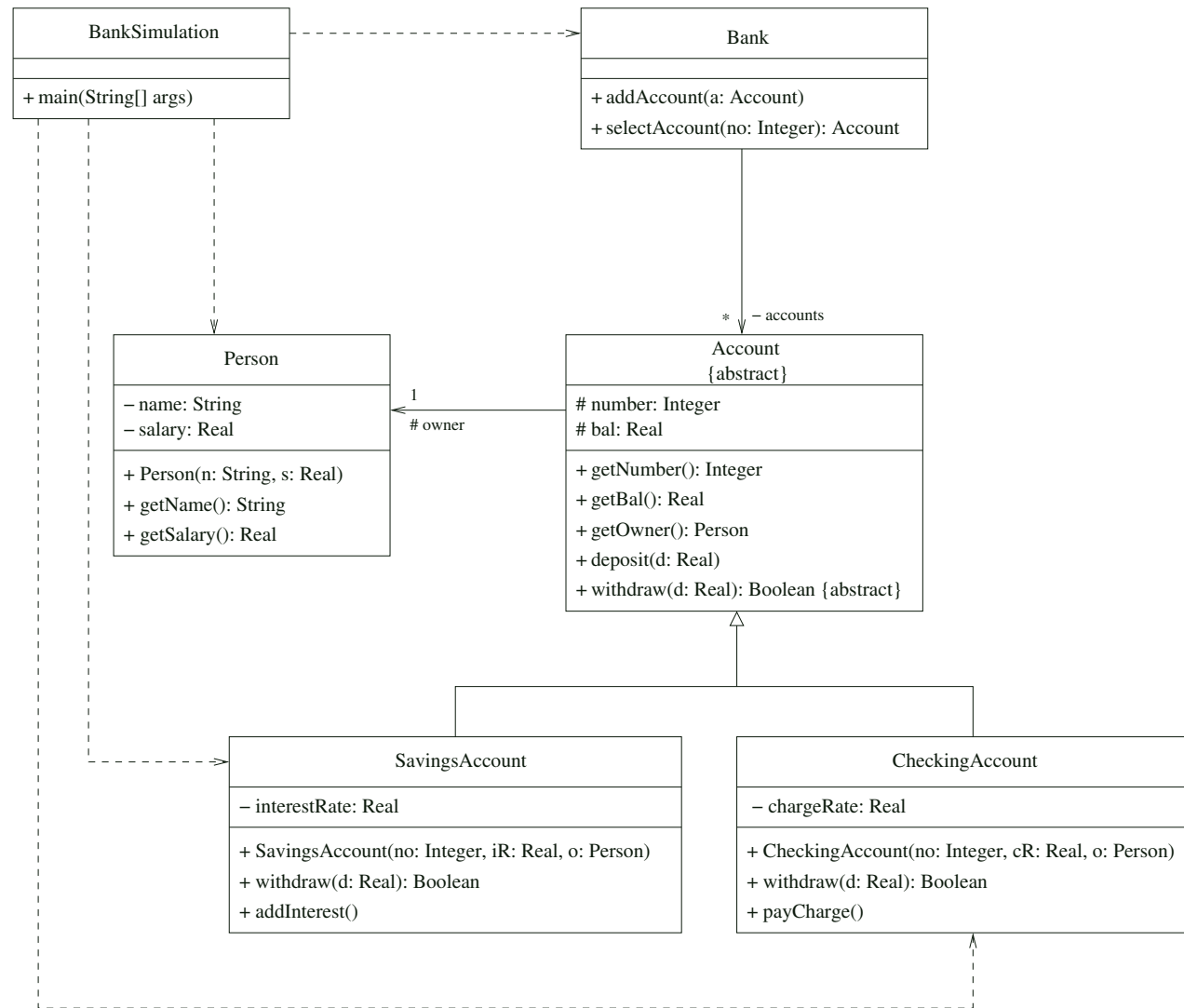
    public void relate(B b) {
        myB = b;
        myB.setMyA(this);
    }
}
```

```
class B {
    private A myA;

    public A getMyA() {
        return myA;
    }

    void setMyA(A a) {
        myA = a;
    }
}
```

Beispiel (Klassendiagramm)



Beispiel (Codegerüst)

```
class BankSimulation {  
  
    public static void main(String[] args) {  
        //Rumpf einfügen  
    }  
}  
  
import java.util.*;  
class Bank {  
  
    private Set accounts = new HashSet();  
  
    public void addAccount(Account a) {  
        //Rumpf einfügen  
    }  
    public Account selectAccount(int no) {  
        //Rumpf einfügen  
    }  
}
```

```
class Person {  
  
    private String name;  
    private double salary;  
  
    public Person(String n, double s) {  
        //Rumpf einfügen  
    }  
    public String getName() {  
        //Rumpf einfügen  
    }  
    public double getSalary() {  
        //Rumpf einfügen  
    }  
}
```

```
abstract class Account {  
  
    protected int number;  
    protected double bal;  
    protected Person owner;  
  
    public int getNumber() {  
        //Rumpf einfügen  
    }  
    public double getBal() {  
        //Rumpf einfügen  
    }  
    public Person getOwner() {  
        //Rumpf einfügen  
    }  
    public void deposit(double d) {  
        //Rumpf einfügen  
    }  
    public abstract boolean withdraw(double d);  
}
```

```
class SavingsAccount extends Account {  
  
    private double interestRate;  
  
    public SavingsAccount(int no, double iR, Person o) {  
        //Rumpf einfügen  
    }  
    public boolean withdraw(double d) {  
        //Rumpf einfügen  
    }  
    public void addInterest() {  
        //Rumpf einfügen  
    }  
}
```

```
class CheckingAccount extends Account {  
  
    private double chargeRate;  
  
    public CheckingAccount(int no, double cR, Person o) {  
        //Rumpf einfügen  
    }  
    public boolean withdraw(double d) {  
        //Rumpf einfügen  
    }  
    public void payCharge() {  
        //Rumpf einfügen  
    }  
}
```