Vererbung

Layout: Mario Hauke / Oliver Henkel

III. Vererbung

Inhalt

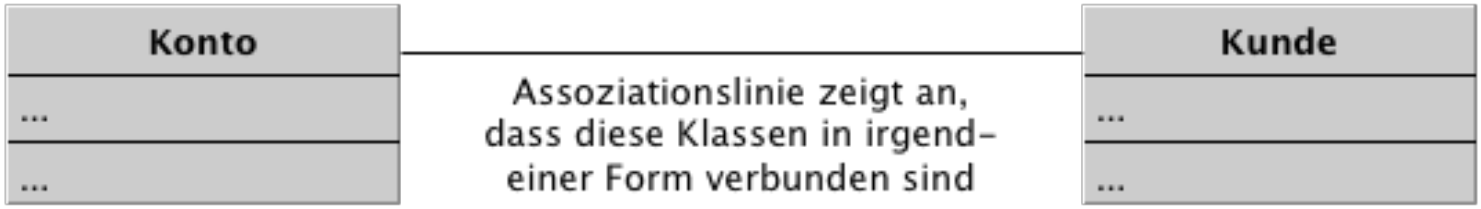
III.1 Vererbung

Inhalt

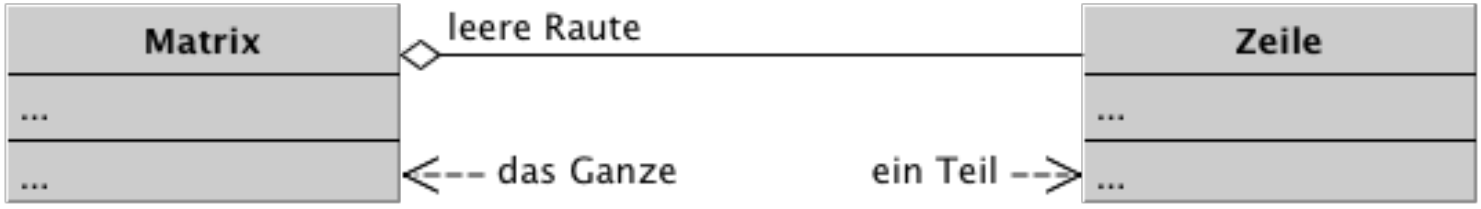
III.1 Objekte in Beziehung

Assoziation, Aggregation, Komposition

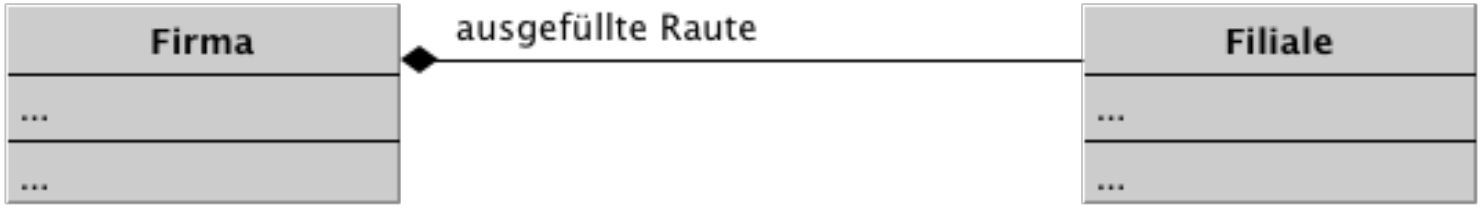
- Eine Beziehung zwischen Klassen nennt man Assoziation



- Eine Abhängigkeitsbeziehung des Ganzen zu seinen Teilen nennt man Aggregation



- Hängen Teile von der Existenz des Ganzen ab, spricht man von einer Komposition



III.1 Objekte in Beziehung

Delegation

☞ Objektbeziehungen_Quadrat_Rechteck

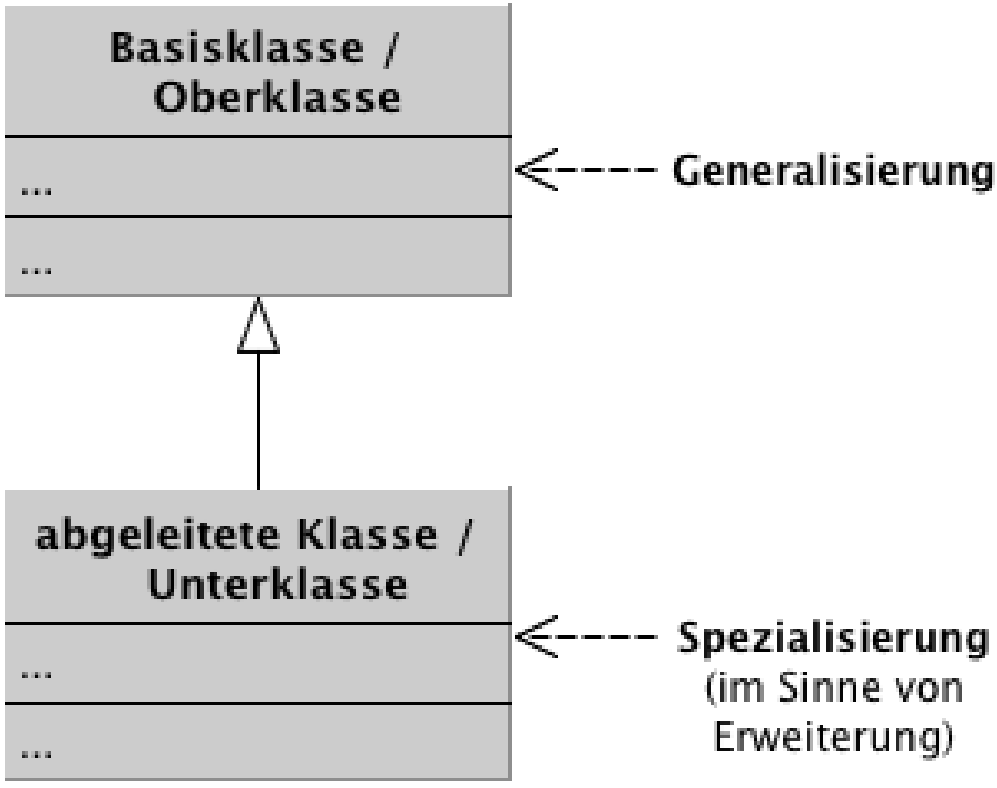
- Von einer Delegation spricht man, wenn eine Klasse Objekte einer anderen Klasse benutzt, indem Aufgaben mittels Methodenaufruf delegiert werden. Dies ist eine spezielle Form der Aggregation


```
class Rechteck {
 int hoehe, breite;
public:
 Rechteck(int h, int b)
 : hoehe(h), breite(b) {}
 void neueHoehe (int h) {  hoehe=h; } // koennen unabhaengig
 void neueBreite(int b) { breite=b; } // veraendert werden
};

class Quadrat {
 Rechteck r;                    // privates Attribut der Klasse
public:                         // zur Realisierung von Quadraten
 Quadrat(int seite)
 : r(seite,seite) {}           // Delegation an den Konstruktor
                               // der Klasse Rechteck
 void neueSeite (int neu) { // Delegation an Rechteck-Methoden
  r.neueHoehe(neu); r.neueBreite(neu);
 }
};
```

III.1 Objekte in Beziehung

Vererbung: Von der Generalisierung zur Spezialisierung

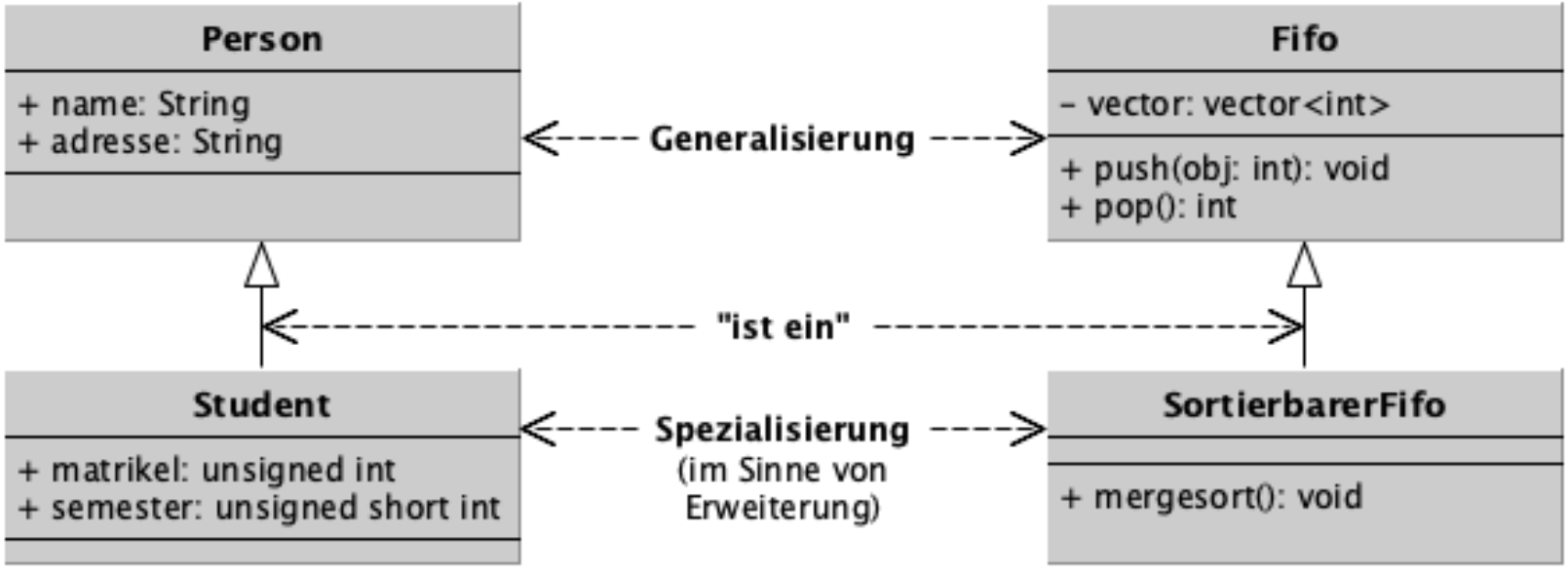
- Von Vererbung spricht man, wenn eine Klasse alle Daten und Methoden einer anderen Klasse übernimmt und um eigene Daten und Methoden erweitert



III.1 Objekte in Beziehung

Vererbung: Sprechweisen

- Die abgeleitete Klasse erbt alle Daten/Methoden der Basisklasse
- Die abgeleitete Klasse erweitert die Basisklasse durch weitere Daten/Methoden
- Die Basisklasse hat eine kleinere Schnittstelle als die abgeleitete Klasse. Jedes Objekt der abgeleiteten Klasse realisiert *mindestens* diese Schnittstelle und kann daher überall dort stehen, wo auch ein Objekt der Basisklasse stehen könnte („ist ein“-Beziehung)



III.1 Objekte in Beziehung

Vorsicht bei der Verwendung der Begriffe!

Warnung | Verwendung der Fachbegriffe

Die Begriffe "Generalisierung", "Spezialisierung" und "ist ein" werden im täglichen Sprachgebrauch anders verwendet! Ein Quadrat "ist ein" Rechteck

aber: Quadrat kann nicht von Rechteck erben, wenn die Klasse Rechteck Methoden zur separaten Änderung der Seitenlängen beinhaltet

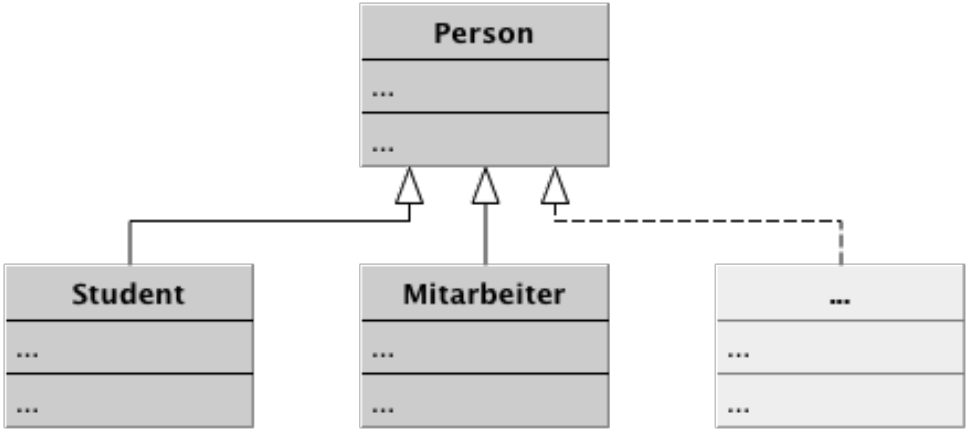
und: Rechteck erweitert auch nicht das Quadrat, da nicht überall dort, wo ein Quadrat stehen kann, auch ein Rechteck stehen darf!

III.1 Objekte in Beziehung

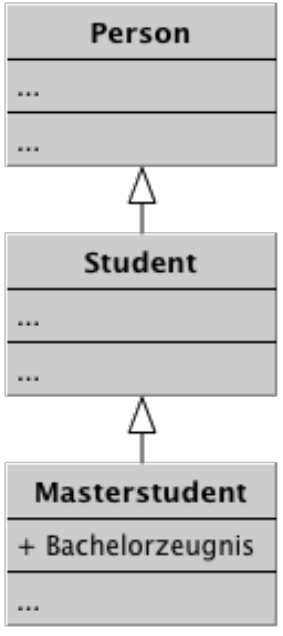
Ausprägungen der Vererbung

Vererbung ermöglicht die Erweiterung von Software um zusätzliche Funktionalitäten, ohne die bereits implementierten Operationen zu verändern

Es gibt hierbei verschiedene Ausprägungen:



Hinzufügen einer neuen Spezialisierung der Basisklasse



Hinzufügen weiterer Unterspezialisierungen

III.2 Vererbung

Inhalt

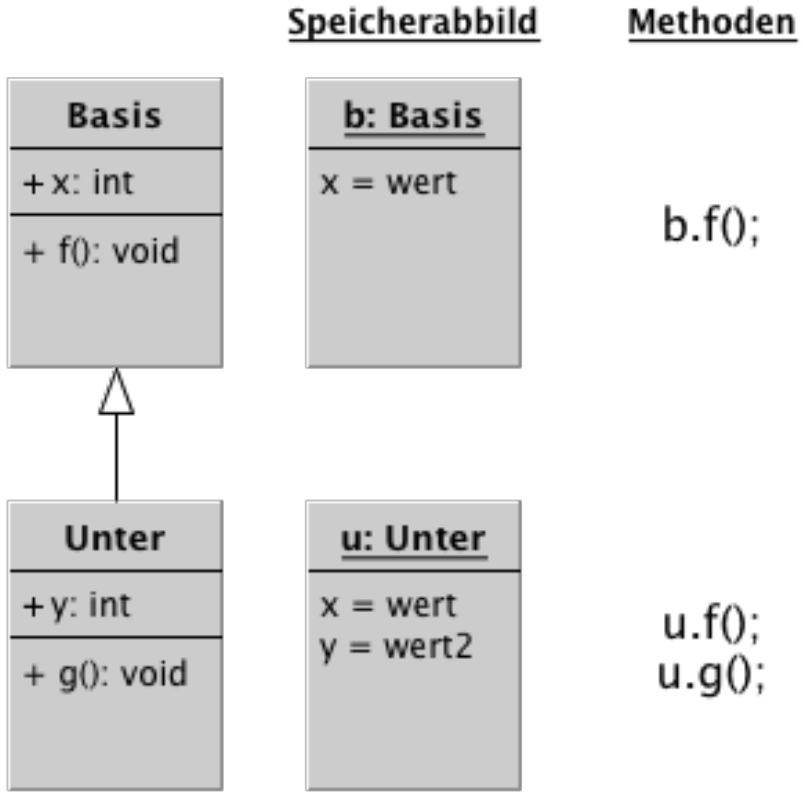
III.2a Vererbung

Inhalt

III.2a Umsetzung in C++

☞ VERERBUNG_C++

```
class Basis {
public:
 int x;
 void f() {}
};


class Unter: public Basis {
public:
 int y;
 void g() {}
};
```

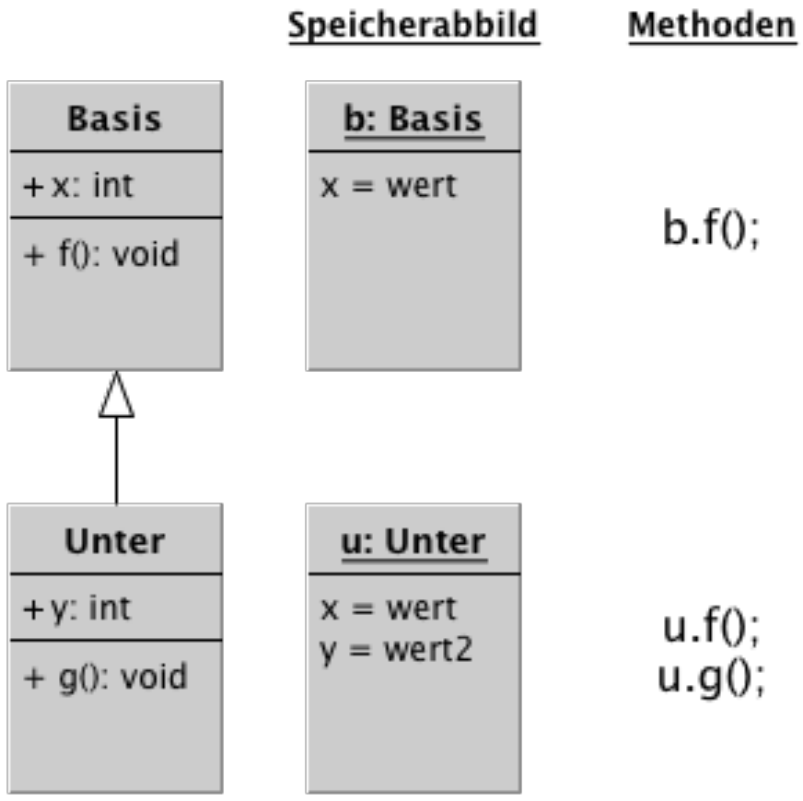



```
void func (Basis b) { b.x=5;          cout << "func (Basis)" << endl; }
void func (Unter u) { u.x=5; u.y=6; cout << "func (Unter)" << endl; }

func(b); // OK: Aufruf von func(Basis)
func(u); // OK: Aufruf von func(Unter),
         //      da Kandidat func(Unter) besser als func(Basis) passt
         //                  ^                          ^-- Kriterium-3
         //                  |                              Uebereinstimmung
         //                  -- Kriterium-1 Typ-Uebereinstimmung
```

III.2a Umsetzung in C++

☞ VERERBUNG_C++

```
class Basis {
public:
 int x;
 void f() {}
};


class Unter: public Basis {
public:
 int y;
 void g() {}
};
```




```
void funcB(Basis b) { b.x=5;          cout << "funcB(Basis)" << endl; }
void funcU(Unter u) { u.x=5; u.y=6; cout << "funcU(Unter)" << endl; }

funcB(b);      // OK
funcB(u);      // OK, denn "u ist auch ein b" (Upcast Unter --> Basis)
//funcU(b);    // nicht OK, da keine Typkonvertierung Basis->Unter vorhanden
funcU(u);      // OK
//
//
```

III.2b Vererbung

Inhalt

III.2b Auflösung von Namenskonflikten

Überdeckung vs Überladung

☞ WERTESEMANTIK_NAMENSKONFLIKTE

Die Vererbungshierarchie wird beginnend bei der Klasse des aktuellen Objektes von unten nach oben nach dem Komponentennamen abgesucht und die erste gefundene Komponente wird gewählt.

Bei Methoden kommt es dabei zur Überdeckung aller gleichnamigen Bezeichner in höheren Ebenen unabhängig(!) von der Signatur

```
class Basis {
public:
 int x;
 void f()       {}
 void g()       {}
 void g(int)    {}
 void g(double){}
};

class Unter: public Basis {
public:
 int y;
 void g(float) {}
 void h()       {}
};
```

```
Basis b;
b.x=1;
b.f();
b.g();
b.g(1);
b.g(1.5);

Unter u;
u.x=2;    // geerbt
u.y=3;
u.f();    // geerbt
//u.g(); // verdeckt!
u.g(1.5);// dasselbe wie:
u.g(static_cast<float>(1.5));
u.h();
```

III.2b Auflösung von Namenskonflikten

Überdeckung vs Überladung

☞ WERTESEMANTIK_NAMENSKONFLIKTE

Die Vererbungshierarchie wird beginnend bei der Klasse des aktuellen Objektes von unten nach oben nach dem Komponentennamen abgesucht und die erste gefundene Komponente wird gewählt.

Zugriff über die Vererbungshierarchie hinweg erfolgt durch explizite Namensbereichauflösung

```
class Basis {
public:
 int x;
 void f()       {}
 void g()       {}
 void g(int)    {}
 void g(double){}
};

class Unter: public Basis {
public:
 int y;
 void g(float) {}
 void h()       {}
};
```

```
Basis b;
b.x=1;
b.f();
b.g();
b.g(1);
b.g(1.5);

Unter u;
u.x=2;    // geerbt
u.y=3;
u.f();    // geerbt
u.Basis::g(); //! "super.g()"
u.g(1.5);// dasselbe wie:
u.g(static_cast<float>(1.5));
u.h();
```

III.2b Auflösung von Namenskonflikten

Überdeckung vs Überladung

☞ WERTESEMANTIK_NAMENSKONFLIKTE

Die Vererbungshierarchie wird beginnend bei der Klasse des aktuellen Objektes von unten nach oben nach dem Komponentennamen abgesucht und die erste gefundene Komponente wird gewählt.

Überladung über mehrere Ebenen der Vererbungshierarchie hinweg wird durch die **using**-Direktive ermöglicht, die Bezeichner bekannt macht

```
class Basis {
public:
 int x;
 void f()        {}
 void g()        {}
 void g(int)     {}
 void g(double){}
};

class Unter: public Basis {
public:
 int y;
 using Basis::g; //!
 void g(float) {}
 void h()        {}
};
```

```
Basis b;
b.x=1;
b.f();
b.g();
b.g(1);
b.g(1.5);

Unter  v;
v.g();       //! jetzt moeglich
v.g(1.5);    //! Basis::g(double)
v.g(static_cast<float>(1.5));
             //! Unter::g(float)
```

III.2c Vererbung

Inhalt

Da es in C++ keine Paket-Sichtbarkeit gibt, sind Objektvariablen in der Regel durch **private** geschützt. Abgeleitete Klassen haben Zugriff auf **protected**-Objektvariablen

```
class Basis {
private:
 int privat;
protected:
 int geschuetzt;
public:
 int offen;

 int gibPrivat() { return privat; }
 int gibGeschuetzt() { return geschuetzt; }
};

/// Zugriff von "aussen"
int i;
Basis b;
i = b.offen; i = b.gibGeschuetzt(); i = b.gibPrivat();
```

Da es in C++ keine Paket-Sichtbarkeit gibt, sind Objektvariablen in der Regel durch **private** geschützt. Abgeleitete Klassen haben Zugriff auf **protected**-Objektvariablen

```
/// Zugriff aus abgeleiteter Klasse heraus
class Unter: public Basis {
public:
 int erlaubt()     { return offen; }
 //int Verboten() { return privat; }
 int dasGeht()     { return geschuetzt; }

 int dasGehtAuch(Unter u) { return u.geschuetzt; }
  // Datenabstraktion:
  // Klasse hat Zugriff unabhaengig vom konkreten Objekt

 //int GehtNicht(Basis b) { return b.geschuetzt; }
  // b gehoert einer anderen Klasse an:
  // --> Zugriff von "aussen"
};

/// Zugriff von "aussen"
int i;
Unter u;
i = u.offen; i = u.gibGeschuetzt(); i = u.gibPrivat();
```

III.2c Zugriffsrechte

nicht-öffentliche Vererbung

C++ erlaubt öffentliche, geschützte und private Vererbung mittels:

```
class Unter: public Basis    {...}; //oeffentliche Vererbung
class Unter: protected Basis {...}; //unueblich
class Unter: private Basis   {...}; //Implementationsvererbung
```

Der Zugriff von aussen auf die geerbten Komponenten wird auf höchstens **public** / **protected** / **private** eingeschränkt.

Bei nicht-öffentlicher Vererbung gilt:
Nur die Klasse Unter kann auf die öffentlichen/geschützten Komponenten von Basis zugreifen, erbt also insbesondere die Implementationen der Methoden

Man sagt: Unter "hat ein" oder "ist implementiert mit" Basis
Alternative: Delegation statt privater Vererbung stets möglich

Warnung | Aufpassen bei der Typkonvertierung

Der Compiler kennt keine Typkonvertierung Unter → Basis bei privater Vererbung (da Basis in Unter privat ist)

III.2c Zugriffsrechte

nicht-öffentliche Vererbung

☞ ZUGRIFFSRECHTE_WARTESCHLANGEP/D

Implementationsvererbung

```
class Warteschlange
 : private list<int> {
public:
 // Abfragen
 using list<int>::empty;
 using list<int>::size;

 // am Ende einfuegen:
 void push(const int& x)
 { list<int>::push_back(x);}

 // am Anfang entnehmen:
 void pop()
 { list<int>::pop_front();}

 // am Anfang bzw. Ende lesen
 using list<int>::front;
 using list<int>::back;
};
```

Delegation

```
class Warteschlange {
private:
 list<int> liste;
public:
 // Abfragen
 bool empty( )
  { return liste.empty(); }
 size_t size()
  { return liste.size(); }

 // am Ende einfuegen:
 void push(const int& x)
  { liste.push_back(x);}

 // am Anfang entnehmen:
 void pop()
  { liste.pop_front();}

 // am Anfang bzw. Ende lesen
 const int& front()
  { return liste.front();}
 const int& back()
  { return liste.back();}
};
```

III.2d Vererbung

Inhalt

III.2d Standardmethoden

Konstruktoren & Destruktoren

Jede Klasse besitzt die Standardmethoden:
(überladener)Konstruktor, Destruktor, Zuweisungsoperator

- Konstruktoraufrufe (auch Kopierkonstruktor) in abgeleiteten Klassen:

  Es wird stets zunächst der parameterlose Konstruktor der Basisklasse aufgerufen, um die geerbten Attribute zu initialisieren, bevor der Konstruktor der abgeleiteten Klasse abgearbeitet wird.

  Oft ist das so nicht erwünscht. Um den passenden Konstruktor der Basisklasse aufzurufen, muss dieser in der Initialisierungsliste aufgerufen werden

- Destruktoren werden in der umgekehrten Reihenfolge aufgerufen

Default-Initialisierung

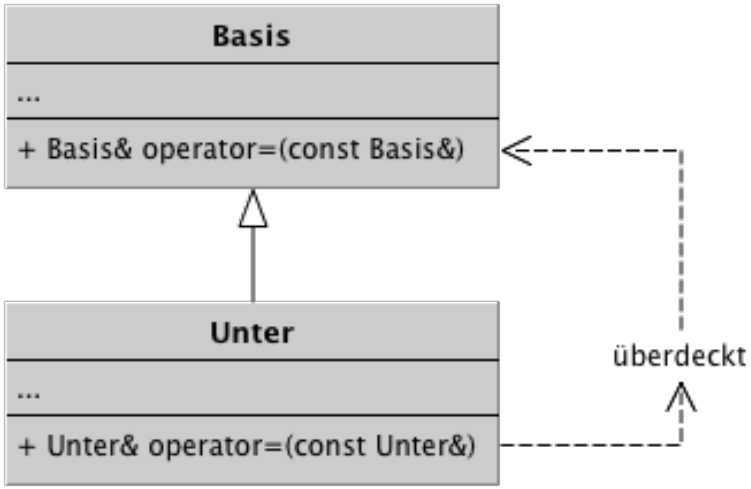
```
class Basis {
public:
 Basis()         {}
 Basis(int)      {}
 Basis(Basis&)   {}
 ~Basis()        {}
};

class Unter: public Basis {
public:
 Unter()         {}
 Unter(int)      {}
 Unter(Unter&)   {}
 ~Unter()        {}
};

Unter eins, zwei(0), drei(zwei);
//
// Aufrufe:
// Basis(), Unter()
// Basis(), Unter(int)
// Basis(), Unter(Unter&)
```

passende Initialisierung

```
class Basis {
public:
 Basis()         {}
 Basis(int)      {}
 Basis(Basis&)   {}
 ~Basis()        {}
};

class Unter: public Basis {
public:
 Unter()          : Basis()   {}
 Unter(int i)     : Basis(i)  {}
 Unter(Unter& u): Basis(u)  {}
 ~Unter()                    {}
};

Unter eins, zwei(0), drei(zwei);
//
// Aufrufe:
// Basis(), Unter()
// Basis(int), Unter(int)
// Basis(Basis&), Unter(Unter&)
```

III.2d Standardmethoden

Zuweisungsoperator



- Standard-Zuweisungsoperator in Unter ruft zunächst die Zuweisung in Basis auf, um die geerbten Komponenten zuzuweisen
- Bei eigenen Zuweisungsoperatoren in Unter muss Basis :: **operator**= explizit(!) aufgerufen werden

Mischfälle

- Basis :: **operator**= akzeptiert auch Unter-Objekte als Parameter
  $\longrightarrow$Zuweisung nur der Basis-Komponten möglich!
- Definition von Unter :: **operator**=(**const** Basis&) formal möglich
  $\longrightarrow$ohne weitere Überlegungen (später!) nicht sinnvoll, da als rechter Operand jedes Basis-kompatible Objekt zulässig

```
class Basis {
 int x;
public:
 Basis(int _x): x(_x) {}
 Basis& operator= (const Basis& b) {
  // ggf. Test Selbstzuweisung/Freigabe dynamischer Speicher...
  x=b.x;
  return *this;
 }
};

class Unter: public Basis {
 int y;
public:
 Unter(int _x, int _y): Basis(_x), y(_y) {}
                        //! ^-- x ist privat in Basis
 Unter& operator= (const Unter& u) {
  // ggf. Test Selbstzuweisung/Freigabe dynamischer Speicher...
  Basis::operator=(u); //! Zuweisung von privatem x
  y=u.y;               // Zuweisung von y
  return *this;
 }
};
```

III.3 Vererbung

Inhalt

III.3 Polymorphismus

Definition

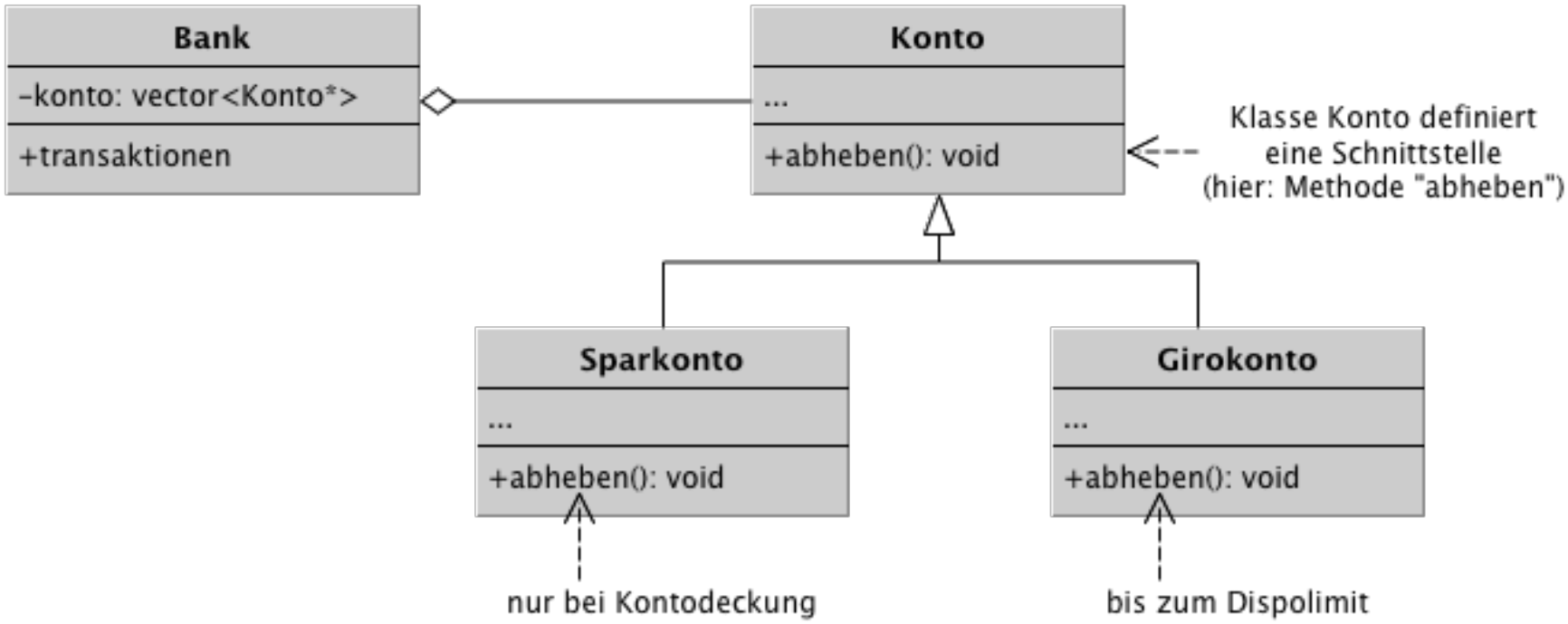
Polymorphismus ist die dynamische Bestimmung der auszuführenden Implementation einer Methode

- Man spricht von einer späten Bindung der Methode an das Objekt, da die Implementation der Methode erst zur Laufzeit ausgewählt wird und nicht vorab durch den Compiler/Quellcode festgelegt ist
- Von früher Bindung bzw. Compile-Zeit-Polymorphismus spricht man, wenn Funktionen/Methoden bereits durch den Compiler zugeordnet werden, z.B.
  - Überladungsauflösung
  - automatische Typzuordnung bei der Instanzierung von Klasssenschablonen

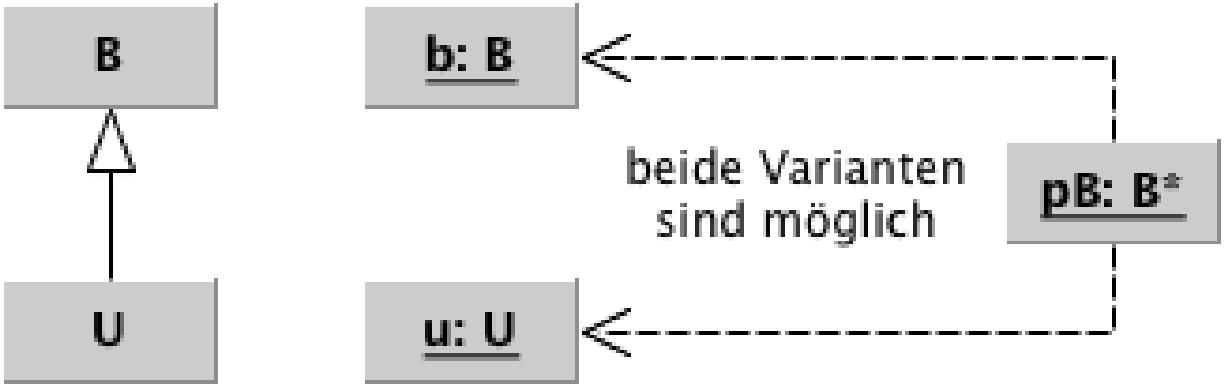
III.3 Polymorphismus

Beispiel: Heterogene Datenstrukturen in Containern

Kontenhaltung einer Bank



- Bank::konto enthält Basisklassenzeiger, damit unterschiedliche Ausprägungen der Klasse Konto darin gehalten werden können
- Die Methode abheben() kennzeichnet einen gleichartigen Dienst aller möglichen Bankkonten, dessen Implementation aber für jede Spezialisierung unterschiedlich ausfällt
- konto[347]->abheben() soll die passende Methode aufrufen



statischer Typ von pB: B

dynamischer Typ von pB: Typ des Zeigerziels (∗pB) → B oder U

Das Zeigerziel ist ggf. erst zur Laufzeit bekannt!

```
B b; U u; B* pB;
char eingabe; cin >> eingabe;
if (eingabe=='B') pB=&b;
else              pB=&u;
```

Der Compiler kennt nur den statischen Typ von pB und die im statischen Typ vorhandenen Daten/Methoden!

Der Compiler kennt nur den statischen Typ von pB und die im statischen Typ vorhandenen Daten/Methoden!

```
class B {
public:
 int x;
};

class U : public B {
public:
 int y;
};

void f(B*) {}
void g(U*) {}

// statischer Typ: B
B* pB;

// statischer Typ: U
U* pU;
```

```
///statischer Typ: B
pB = new B;// Zeigerziel: B-Objekt
pB->x=1;   // OK
f(pB);     // OK
//g(pB)    // nicht OK

pB = new U;// Zeigerziel: U-Objekt
pB->x=2;   // OK
//!pB->y=3;// nicht OK
f(pB);     // OK
//g(pB)    // nicht OK

///statischer Typ: U
pU = new U;// Zeigerziel: U-Objekt
pU->x=4;   // OK: x geerbt
pU->y=5;   // OK
f(pU);     // OK: U* "ist ein" B*
g(pU);     // OK
```

III.3a Umsetzung in C++

Teil II: virtuelle Methoden

☞ KONTO_SPAETE_BINDUNG.CPP

Damit eine Methode in den abgeleiteten Klassen polymorph überschrieben werden kann, muss sie in der Basisklasse als **virtual** markiert werden

```
class Bank {
  vector<Konto*> konto;   // heterogener Vektor-Container
};                        // (Basisklassenzeiger)
class Konto {
 virtual void abheben(); // Methode polymorph markiert
}
```

- konto[347]->abheben() sucht eine passende Implementation der Methode abheben() im dynamischen Typ von konto[347]
- Polymorphie benötigt also sowohl Referenzsemantik (Methodenaufruf über (Basisklassen-)Zeiger oder Referenzen), als auch (in der Basisklasse vereinbarte) virtuelle Methoden
- Bei Wertesemantik haben virtuelle Methoden bei konventionellen Methodenaufrufen aus einem Objekt heraus (anstelle über einen Zeiger/Referenz) **keine** Bedeutung ($\longrightarrow$Überdeckung)

ÜBUNG | Bestimmen Sie die Ausgabe

```
#include <iostream>
#include <string>
using namespace std;
#include "rgb.h"

class Window {
protected:
  int x0, y0, x1, y1;
public:  // ^-- Eckpunktkoordinaten
  Window(int x,int y, int u, int v): x0(x),y0(y), x1(u), y1(v) {}
};

class ColoredWindow: public Window {
protected:
  RGB_Pixel c;
public:  // ^-- Farbe
  ColoredWindow(int x,int y,int u,int v,RGB_Pixel _c): Window(x,y,u,v), c(_c) {}
  virtual string draw() const { return "Zeichne farbiges Fenster"; }
};

class FramedWindow: public ColoredWindow {
  RGB_Pixel f;
public:  // ^-- Rahmenfarbe
   FramedWindow(int x,int y, int u,int v,RGB_Pixel _c, RGB_Pixel _f)
   : ColoredWindow(x,y,u,v,_c), f(_f) {}
  virtual string draw() const { return "Zeichne gerahmtes farbiges Fenster"; }
};

void display(ColoredWindow cw) { cout << cw.draw() << endl; }
```

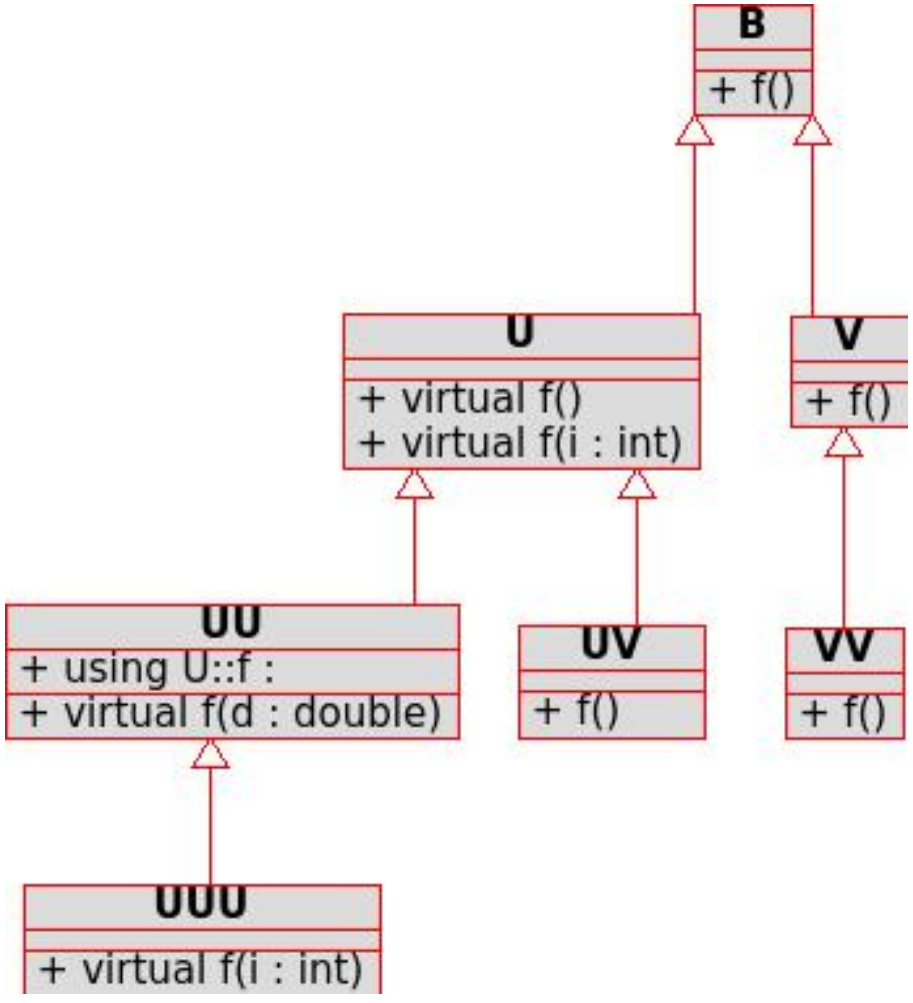
```
int main() {
  RGB_Pixel red(255,0,0), blue(0,255,0);
  ColoredWindow cw(0,0,100,50,blue);
  FramedWindow fw(0,0,100,50,red,blue);
  display(cw); display(fw);
  return 0;
}
```

III.3b Vererbung

Inhalt

III.3b Auswahlregeln bei virtuellen Methoden

beispielhafte Vererbungshierarchie

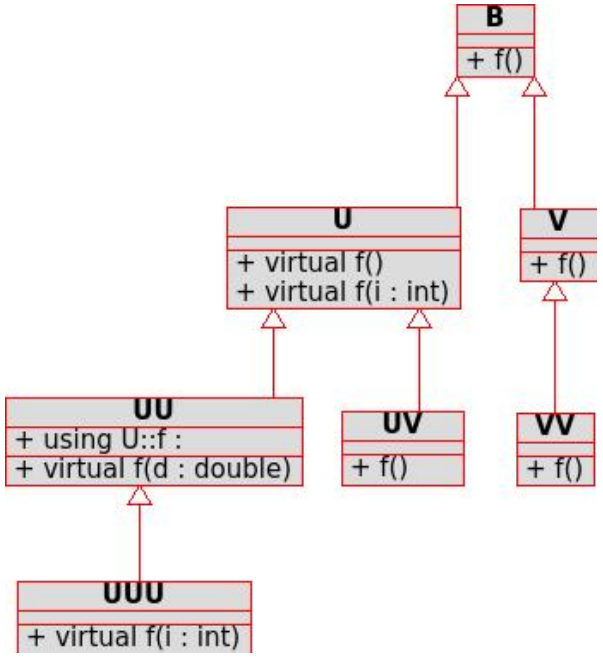


III.3b Auswahlregeln bei virtuellen Methoden

Wertesemantik: Methodenauswahl per **Objekt**

Auswahl einer Methode über ein Objekt:

- Die Methodenzuordnung erfolgt nach der Regel der Überdeckung durch den Methodennamen (unabhängig von den Parametern) von "unten nach oben"
- Überladungsauflösung erfolgt nur auf derselben Ebene bzw. durch Verwendung der **using**-Direkive in der betreffenden Klasse
- Das Schlüsselwort **virtual** hat keine Bedeutung

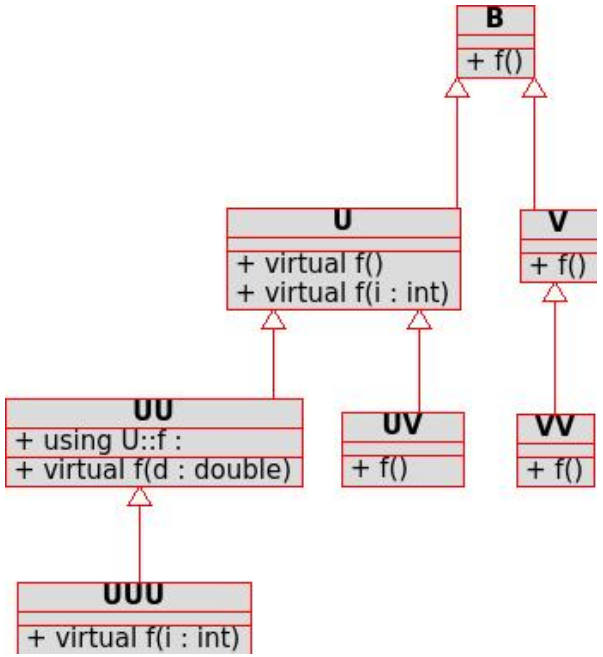


III.3b Auswahlregeln bei virtuellen Methoden

Referenzsemantik: Methodenauswahl per **Zeiger**

Auswahl einer Methode über einen Zeiger p:

- Der statische Typ entscheidet über die zur Verfügung stehenden Methoden nach denselben Regeln wie bei der Auswahl über ein Objekt
- Nach Überladungsauflösung wird die passende Signatur zugeordnet
- Ist die gewählte Methode virtuell, so wird im dynamischen Typ nach einer passenden Implementation[→a] gesucht und – falls vorhanden – ausgewählt
- Eine Methode gilt als virtuell, wenn sie im statischen Typ des Zeigers oder in einer Oberklasse als virtuell markiert wurde
- p->K::f() erzwingt stets den Aufruf von f in der Klasse K



[→a] Kriterien: gleiche Signatur und Rückgabetyp
(Ausnahme: Rückgabe von Zeigern/Referenzen auf abgeleitete Objekte)

III.3c Vererbung

Inhalt

III.3c Standardmethoden und Polymorphie

Konstruktor und Destruktor

- Konstruktoren sind nie virtuell, da sie durch den Klassennamen festgelegt sind
- Destruktoren müssen(!) hingegen stets virtuell sein, um vollständige Objektdestruktion zu ermöglichen:

```
Basis *pb=new Unter();
delete pb; // Problem: ruft nur Destruktor
           // des statischen Typs auf
```

Lösung:

```
class B {
 ...
 virtual ~B(); // virtueller Destruktor
}
```

III.3c Standardmethoden und Polymorphie

Kopie und Zuweisung — Kopie durch Erzeugung

Ziel

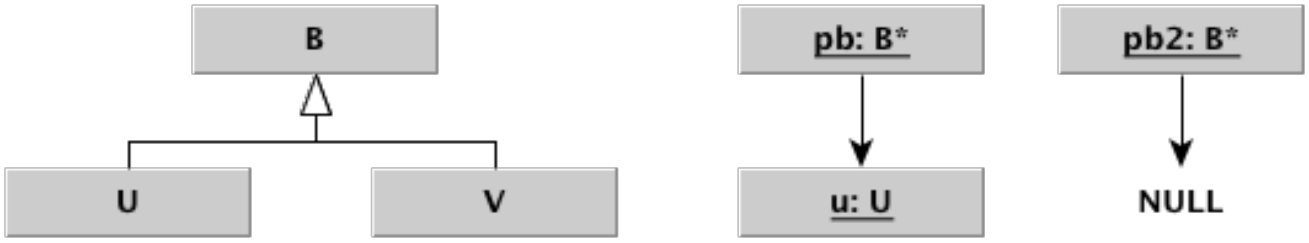
Es soll eine Kopie von ∗pb typgerecht angelegt werden und pb2 soll darauf zeigen

Kopie durch Erzeugung

- Problem
  - Kopierkonstruktoren sind nicht virtuell
  - Die Zielklasse von ∗pb ist erst zur Laufzeit bekannt, es kann somit kein Kopierkonstruktor-Aufruf benannt werden
- Lösung: Polymorphe clone()-Methode (virtuell, einheitlicher Name), die den klasseneigenen Kopierkonstruktor aufruft

III.3c Standardmethoden und Polymorphie

Kopie und Zuweisung — Kopie durch Erzeugung



Ziel

Es soll eine Kopie von ∗pb typgerecht angelegt werden und pb2 soll darauf zeigen

Implementation in jeder Klasse K (hier: B, U, V):

```
virtual K* clone() const { // Aufruf, wenn K=dynamischer Typ
 return new K(*this);      // Kopierkonstruktor der Klasse K
}   // (stets vorhanden, auch wenn nicht selbst geschrieben)
```

Aufruf durch: pb2 = pb−>clone();

III.3c Standardmethoden und Polymorphie

Kopie und Zuweisung — Kopie durch Zuweisung

Ziel

Es soll eine Kopie von $*pb$ typgerecht angelegt werden und pb2 soll darauf zeigen

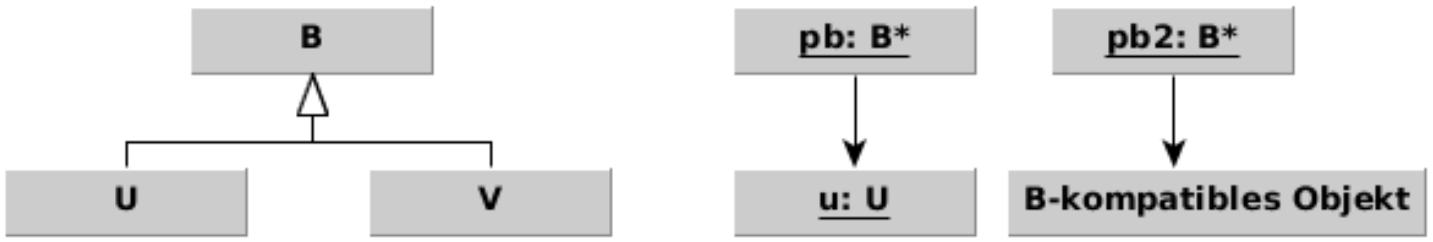
Kopie durch Zuweisung

- Voraussetzung: pb2 zeigt bereits auf ein B, U oder V-Objekt
- Problem: Alle Klassen K (B, U und V) besitzen bereits einen K& **operator**= (**const** K&), der aufgrund seiner festgelegten Signatur nicht polymorph ist
- Lösung: Jede Klasse implementiert zusätzlich eine virtuelle Methode K& assign(**const** B&), die vom Standard-Zuweisungsoperator aufgerufen wird und die Zuweisungen vornimmt.

  Die assign-Methode akzeptiert als Operanden jedes zur Klasse B kompatibles Objekt. Der Operator **dynamic_cast**<**const** K*>() übernimmt dabei die Typkonvertierung nach **const** K*, falls die übergebene Referenz kompatibel ist, und liefert ansonsten NULL zurück

III.3c Standardmethoden und Polymorphie

Kopie und Zuweisung — Kopie durch Zuweisung



Ziel

Es soll eine Kopie von ∗pb typgerecht angelegt werden und pb2 soll darauf zeigen

Implementation in jeder Klasse K (hier: B, U, V): ⌜Aufruf durch: ∗pb2 = ∗pb⌟

```
K& operator=(const K& rhs) {
  if (this==&rhs) return *this;
  return assign(rhs); // polymorpher Aufruf
}

virtual K& assign(const B& rhs) { // feste Signatur!
  const K* pK = dynamic_cast<const K*>(&rhs);
  if(!pk) /* Ausnahme behandeln */ return *this;

  B::assign(rhs); //Aufruf direkte Elternklasse
  /* lokale Komponentenzuweisungen */
  return *this;
}
```

III.3c Standardmethoden und Polymorphie

Beispielimplementation

☞ POLYMORPHE_STANDARDMETHODEN

```
class B {
 int x;
public:
 B(int _x): x(_x) {}
 virtual ~B() {}
 virtual B* clone()
 { return new B(*this); }
 B& operator=(const B& rhs) {
  if (this==&rhs) return *this;
  return assign(rhs);
 }
 virtual
 B& assign(const B& rhs) {
  x=rhs.x;
  return *this;
 }
};
```

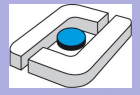
```
class U : public B {
 int y;
public:
 U(int _x, int _y)
 : B(_x), y(_y) {}

 virtual ~U() {}
 virtual U* clone()
 { return new U(*this); }
 U& operator=(const U& rhs) {
  if (this==&rhs) return *this;
  return assign(rhs);
 }
 virtual
 U& assign(const B& rhs) {
  const U* pU=
   dynamic_cast<const U*>(&rhs);
  if (!pU) return *this;
  B::assign(rhs);
  y = pU->y;
  return *this;
 }
};
```

III.3c Standardmethoden und Polymorphie

polymorpher Onlineshop und Favoritenliste

☞ POLYMORPHER_ONLINESHOP

☞ POLYMORPHE_FAVORITENLISTE

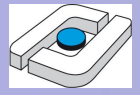


III.3c polymorpher Onlineshop und Favoritenliste

Artikel.h

```
/*
 * ------------------------------------------------------------------------------
 *
 * Klassenheader: Artikel.h
 *
 * Verwendung: In OnlineShop.cpp und Favoriten.cpp
 *
 * VL Programmierung 2, O. Henkel, HS Osnabrueck
 * ------------------------------------------------------------------------------
 */
#ifndef ARTIKEL_H
#define ARTIKEL_H

#include <string>
using namespace std;
typedef unsigned int uint;

class Artikel {
private:
 uint nr {0};
 string beschreibung {""};
 float preis {0.};
 uint lieferzeit {0};
public:
 Artikel() {}
 Artikel(uint _nr, const string& text, float _preis, uint _lieferzeit)
  : nr(_nr), beschreibung(text), preis(_preis), lieferzeit(_lieferzeit) {}

 unsigned int gibNr()                          { return nr; }
 void setzeNr(unsigned int val)                 { nr = val; }
 string gibBeschreibung()             { return beschreibung; }
 void setzeBeschreibung(string val)    { beschreibung = val; }
 float gibPreis()                            { return preis; }
 void setzePreis(float val)                  { preis = val; }
 unsigned int gibLieferzeit()           { return lieferzeit; }
 void setzeLieferzeit(unsigned int val) { lieferzeit = val; }

 const string toString();
};

#endif // ARTIKEL_H
```

```
/*
 * ------------------------------------------------------------------------------
 *
 * Klassenheader: Artikel.h
 *
 * Verwendung: In OnlineShop.cpp und Favoriten.cpp
 *
 * VL Programmierung 2, O. Henkel, HS Osnabrueck
 * ------------------------------------------------------------------------------
 */
#ifndef ARTIKEL_H
#define ARTIKEL_H

#include <string>
using namespace std;
typedef unsigned int uint;

class Artikel {
private:
 uint nr {0};
 string beschreibung {""};
 float preis {0.};
 uint lieferzeit {0};
public:
 Artikel() {}
 Artikel(uint _nr, const string& text, float _preis, uint _lieferzeit)
  : nr(_nr), beschreibung(text), preis(_preis), lieferzeit(_lieferzeit) {}

 virtual Artikel* clone() const { return new Artikel(*this); }
 Artikel& operator=(const Artikel& rhs)
 { if (this==&rhs) return *this; return assign(rhs); }
 virtual Artikel& assign(const Artikel& rhs);

 unsigned int gibNr()                          { return nr; }
 void setzeNr(unsigned int val)                 { nr = val; }
 string gibBeschreibung()             { return beschreibung; }
 void setzeBeschreibung(string val)    { beschreibung = val; }
 float gibPreis()                            { return preis; }
 void setzePreis(float val)                  { preis = val; }
 unsigned int gibLieferzeit()           { return lieferzeit; }
 void setzeLieferzeit(unsigned int val) { lieferzeit = val; }

 virtual const string toString();
 virtual ~Artikel() {}
};

#endif // ARTIKEL_H
```



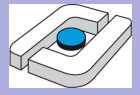
III.3c polymorpher Onlineshop und Favoritenliste

Artikel.cpp

```
/*
 * ------------------------------------------------------------------------------
 *
 * Implementationsdatei: Artikel.cpp
 *
 * Verwendung: In OnlineShop.cpp und Favoriten.cpp
 *
 * VL Programmierung 2, O. Henkel, HS Osnabrueck
 * ------------------------------------------------------------------------------
 */
#include <sstream>
#include "Artikel.h"

const string Artikel::toString() {
 ostringstream strom;
 strom << nr << ": " << beschreibung
       << ", " << preis << "Euro, Lieferzeit "
       << lieferzeit << "Tage";
 return strom.str();
}
```

1

```
/*
 * ------------------------------------------------------------------------------
 *
 * Implementationsdatei: Artikel.cpp
 *
 * Verwendung: In OnlineShop.cpp und Favoriten.cpp
 *
 * VL Programmierung 2, O. Henkel, HS Osnabrueck
 * ------------------------------------------------------------------------------
 */
#include <sstream>
#include "Artikel.h"

Artikel& Artikel::assign(const Artikel& rhs) {
 nr = rhs.nr;
 beschreibung = rhs.beschreibung;
 preis = rhs.preis;
 lieferzeit = rhs.lieferzeit;
 return *this;
}

const string Artikel::toString() {
 ostringstream strom;
 strom << nr << ": " << beschreibung
       << ", " << preis << "Euro, Lieferzeit "
       << lieferzeit << "Tage";
 return strom.str();
}
```



III.3c polymorpher Onlineshop und Favoritenliste

Sonderangebot.h

```
/*
 * ------------------------------------------------------------------------------
 *
 * Klassenheader: Sonderangebot.h
 *
 * Ableitung von Artikel mit zusaetzlichen Attributen
 * sonderpreis und angebotsdauer, sowie einer ueberladenen Methode
 * toString
 *
 * VL Programmierung 2, O. Henkel, HS Osnabrueck
 * ------------------------------------------------------------------------------
 */
#ifndef SONDERANGEBOT_H
#define SONDERANGEBOT_H

using namespace std;
#include "Artikel.h"

class Sonderangebot : public Artikel {
  float sonderpreis;
  uint angebotsdauer;
public:
  Sonderangebot();
  Sonderangebot(uint, const string&, float, uint, float, uint);
  Sonderangebot(const Sonderangebot& orig);
  float gibSonderpreis(){return sonderpreis;}
  uint gibAngebotsdauer(){return angebotsdauer;}
  void setzeSonderpreis(float _sonderpreis){sonderpreis=_sonderpreis;}
  void setzeAngebotsdauer(uint _angebotsdauer){angebotsdauer=_angebotsdauer;}
  const string toString();
  // Wertesemantik: Methode _ueberdeckt_ Artikel::toString()
};

#endif  /* SONDERANGEBOT_H */
```

```
/*
 * ------------------------------------------------------------------------------
 *
 * Klassenheader: Sonderangebot.h
 *
 * Ableitung von Artikel mit zusaetzlichen Attributen
 * sonderpreis und angebotsdauer, sowie einer ueberladenen Methode
 * toString
 *
 * VL Programmierung 2, O. Henkel, HS Osnabrueck
 * ------------------------------------------------------------------------------
 */
#ifndef SONDERANGEBOT_H
#define SONDERANGEBOT_H

using namespace std;
#include "Artikel.h"

class Sonderangebot : public Artikel {
  float sonderpreis;
  uint angebotsdauer;
public:
  Sonderangebot();
  Sonderangebot(uint, const string&, float, uint, float, uint);
  Sonderangebot(const Sonderangebot& orig);
  virtual Sonderangebot* clone() const { return new Sonderangebot(*this); }
  Sonderangebot& operator=(const Sonderangebot& rhs)
  { if (this==&rhs) return *this; return assign(rhs); }
  virtual Sonderangebot& assign(const Artikel& rhs);

  float gibSonderpreis(){return sonderpreis;}
  uint gibAngebotsdauer(){return angebotsdauer;}
  void setzeSonderpreis(float _sonderpreis){sonderpreis=_sonderpreis;}
  void setzeAngebotsdauer(uint _angebotsdauer){angebotsdauer=_angebotsdauer;}
  virtual const string toString();
  virtual ~Sonderangebot() {}
};

#endif  /* SONDERANGEBOT_H */
```

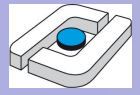
III.3c polymorpher Onlineshop und Favoritenliste

Sonderangebot.cpp

```
/*
 * ------------------------------------------------------------------------------
 *
 * Implementationsdatei: Sonderangebot.cpp
 *
 * VL Programmierung 2, O. Henkel, HS Osnabrueck
 * ------------------------------------------------------------------------------
 */
#include <sstream>
#include "Sonderangebot.h"

Sonderangebot::Sonderangebot()
: Artikel(), sonderpreis(0), angebotsdauer(0) {}

Sonderangebot::Sonderangebot(uint _nr, const string& _beschreibung,
                             float _preis, uint _lieferzeit,
                             float _sonderpreis, uint _angebotsdauer)
: Artikel(_nr, _beschreibung, _preis, _lieferzeit),
 sonderpreis(_sonderpreis), angebotsdauer(_angebotsdauer) {}

Sonderangebot::Sonderangebot(const Sonderangebot& orig)
: Artikel(orig),
 sonderpreis(orig.sonderpreis), angebotsdauer(orig.angebotsdauer)
{}

const string Sonderangebot::toString(){
  ostringstream strom;
  strom<< Artikel::toString();
  strom<<" Sonderangebot fuer "<< angebotsdauer
  <<" Tage, jetzt nur: "<< sonderpreis<<"!";

  return strom.str();

}
```

1

```
/*
 * ------------------------------------------------------------------------------
 *
 * Implementationsdatei: Sonderangebot.cpp
 *
 * VL Programmierung 2, O. Henkel, HS Osnabrueck
 * ------------------------------------------------------------------------------
 */
#include <sstream>
#include "Sonderangebot.h"

Sonderangebot::Sonderangebot()
: Artikel(), sonderpreis(0), angebotsdauer(0) {}

Sonderangebot::Sonderangebot(uint _nr, const string& _beschreibung,
                             float _preis, uint _lieferzeit,
                             float _sonderpreis, uint _angebotsdauer)
: Artikel(_nr, _beschreibung, _preis, _lieferzeit),
 sonderpreis(_sonderpreis), angebotsdauer(_angebotsdauer) {}

Sonderangebot::Sonderangebot(const Sonderangebot& orig)
: Artikel(orig),
 sonderpreis(orig.sonderpreis), angebotsdauer(orig.angebotsdauer)
{}

Sonderangebot& Sonderangebot::assign(const Artikel& rhs) {
  const Sonderangebot* pS = dynamic_cast<const Sonderangebot*>(&rhs);
  if (!pS) /* Warnmeldung/Ausnahme... */ return *this;
  Artikel::assign(rhs);
  sonderpreis = pS->sonderpreis;
  angebotsdauer = pS->angebotsdauer;
  return *this;
}

const string Sonderangebot::toString(){
  ostringstream strom;
  strom<< Artikel::toString();
  strom<<" Sonderangebot fuer "<< angebotsdauer
  <<" Tage, jetzt nur: "<< sonderpreis<<"!";

  return strom.str();

}
```



III.3c polymorpher Onlineshop und Favoritenliste

FavoritenListe.cpp

```
/*
 * ------------------------------------------------------------------------
 *
 * Implementationsdatei: FavoritenListe.cpp
 *
 * Verwendung: In Favoriten.cpp
 *
 * VL Programmierung 2, O. Henkel, HS Osnabrueck
 * ------------------------------------------------------------------------
 */
#include "FavoritenListe.h"

FavoritenListe::FavoritenListe() {}

FavoritenListe::~FavoritenListe() {  // Speicherfreigabe
 for (int i=0; i<favorit.size(); i++) delete favorit[i];
 favorit.clear();
}

FavoritenListe::FavoritenListe(const FavoritenListe& orig) {
 for (int i=0; i<orig.favorit.size(); i++) // tiefe Kopie
  favorit.push_back(new Artikel(*orig.favorit[i]));
}

FavoritenListe& FavoritenListe::operator=(const FavoritenListe& rhs)
{
 if (this == &rhs) return *this;       // Selbstzuweisung verhindern

 for (int i=0; i<favorit.size(); i++)  // alten Speicher freigeben
  delete favorit[i];
 favorit.clear();

 for (int i=0; i<rhs.favorit.size(); i++) // tiefe Kopie
  favorit.push_back(new Artikel(*rhs.favorit[i]));

 return *this;                         // dereferenzierter this-Zeiger
}

//
// Artikel zur Liste hinzufuegen
FavoritenListe& FavoritenListe::operator<< (Artikel* art)  {
 favorit.push_back(new Artikel(*art));  // tiefes Einspeichern

 return *this;                         // dereferenzierter this-Zeiger
}
```

1 →

```
/*
 * ------------------------------------------------------------------------
 *
 * Implementationsdatei: FavoritenListe.cpp
 *
 * Verwendung: In Favoriten.cpp
 *
 * VL Programmierung 2, O. Henkel, HS Osnabrueck
 * ------------------------------------------------------------------------
 */
#include "FavoritenListe.h"

FavoritenListe::FavoritenListe() {}

FavoritenListe::~FavoritenListe() {  // Speicherfreigabe
 for (int i=0; i<favorit.size(); i++) delete favorit[i];
 favorit.clear();
}

FavoritenListe::FavoritenListe(const FavoritenListe& orig) {
 for (int i=0; i<orig.favorit.size(); i++) // tiefe Kopie
  favorit.push_back(orig.favorit[i]->clone());
}

FavoritenListe& FavoritenListe::operator=(const FavoritenListe& rhs)
{
 if (this == &rhs) return *this;       // Selbstzuweisung verhindern

 for (int i=0; i<favorit.size(); i++)  // alten Speicher freigeben
  delete favorit[i];
 favorit.clear();

 for (int i=0; i<rhs.favorit.size(); i++) // tiefe Kopie
  favorit.push_back(rhs.favorit[i]->clone());

 return *this;                         // dereferenzierter this-Zeiger
}

//
// Artikel zur Liste hinzufuegen
FavoritenListe& FavoritenListe::operator<< (Artikel* art)  {
 favorit.push_back(art->clone());  // tiefes Einspeichern

 return *this;                         // dereferenzierter this-Zeiger
}
```

2 →

3 →

Übung | Bestimmen Sie die Ausgabe. Läuft alles wie gedacht?

```
1  #include<iostream>
2  using namespace std;
3
4  class K {
5  public:
6    K() { cout << "␣K" << endl; }
7   ~K() { cout << "~K" << endl; }
8  };
9
10 class B {
11 public:
12   B() { cout << "␣B" << endl; }
13  ~B() { cout << "~B" << endl; }
14 };
15
16 class U : public B {
17   K  k1;
18   K* k2;
19 public:
20   U(): k2(new K) { cout << "␣U" << endl; }
21  ~U() { cout << "~U" << endl; delete k2;  }
22 };
23
24 int main() {
25   B* pb = new U;
26   delete pb;
27
28   return 0;
29 }
```

III.4 Vererbung

Inhalt

III.4a Vererbung

Inhalt

III.4a Abstrakte Klassen

Merkmale

Abstrakte Klassen sind Klassen, von denen keine Objekte angelegt werden können. Sie treten ausschließlich als Basisklassen auf.

Beispiel | Klasse Konto

Die Klasse Konto sollte keine Objekte haben, da es nur spezielle Ausprägungen (Sparkonto, Girokonto etc.) gibt.

Merkmale abstrakter Klassen

- fassen allgemeine Eigenschaften ihrer abgeleiteten Klassen zusammen
- können als Zieltypen (statischer Typ) in Feldern/Vektoren von Basisklassenzeigern vorkommen und über den dynamischen Typ polymorph Ausprägungen der abgeleiteten Klassen repräsentieren
- stellen sicher, dass die abgeleiteten Klassen bestimmte Methoden anbieten (Schnittstellenanforderung)

III.4a Abstrakte Klassen

Implementation in C++

Implementation abstrakter Klassen in C++:

- durch Deklaration mindestens einer rein virtuellen Methode:
  virtual methode(Parameterliste)=0;
  - Methode kann, muss aber nicht implementiert werden
  - Eine Implementation könnte von den abgeleiteten Klassen mittels B::methode verwendet werden
- Die abgeleiteten Klassen müssen eine Implementation der rein virtuellen Methode enthalten (andernfalls ebenfalls abstrakt)
  - diese abgeleiteten Klassen heißen dann "konkrete Klassen"

```
// abstrakte Klasse
class B {
public:
 virtual void f()=0;// abstrakt durch rein virtuelle Methode
};              // (kann, muss aber nicht implementiert werden)


// konkrete Klasse  // muss die rein virtuellen Methoden
class U : public B {// implementieren
public:
 void f() { cout << "Durch diese Implementation"
                 << " bin ich eine konkrete Klasse"
                 << endl;
 }
};

//B b;              // geht nicht, da B abstrakt
B *pb;              // geht

//pb = new B;       // geht nicht, da B abstrakt
pb = new U;         // geht

pb->f();            // Aufruf von f im zugehoerigen
                    // dynamischen Typ U
```

III.4a Abstrakte Klassen

Implementation in C++

☞ KONTO_ABSTRAKT.CPP

```
class Konto {                          // abstrakte Klasse
public:
 virtual void abheben(int) = 0; // rein virtuelle Methode
 virtual ~Konto() {}                   // virtueller Destruktor
};
void Konto::abheben(int) {             // Standardimplementation
 cout <<"Konto abheben ";              // (muss es nicht geben)
}

class Sparkonto : public Konto {// konkrete Klasse
public:
 void abheben(int);                    // muss es geben, wenn Objekte
};                                     // angelegt werden sollen
void Sparkonto::abheben(int n) {// konkrete Implementation
 Konto::abheben(n);
 cout << "vom Sparkonto" << endl;
}

class Girokonto : public Konto {// konkrete Klasse
public:
 void abheben(int);                    // muss es geben, wenn Objekte
};                                     // angelegt werden sollen
void Girokonto::abheben(int n) {// konkrete Implementation
 Konto::abheben(n);
 cout << "vom Girokonto" << endl;
}
```

III.4b Vererbung

Inhalt

III.4b Interfaces

Interfaces sind Schnittstellen-Anforderungsklassen, deren Aufgabe darin besteht, sicherzustellen, dass die abgeleiteten Klassen die vorgegebene Schnittstelle implementieren. Ein virtueller Destruktor ist auch hier nötig, da ein Interface auch als statischer Typ eines Zeigers genutzt werden kann.

Implementation in C++

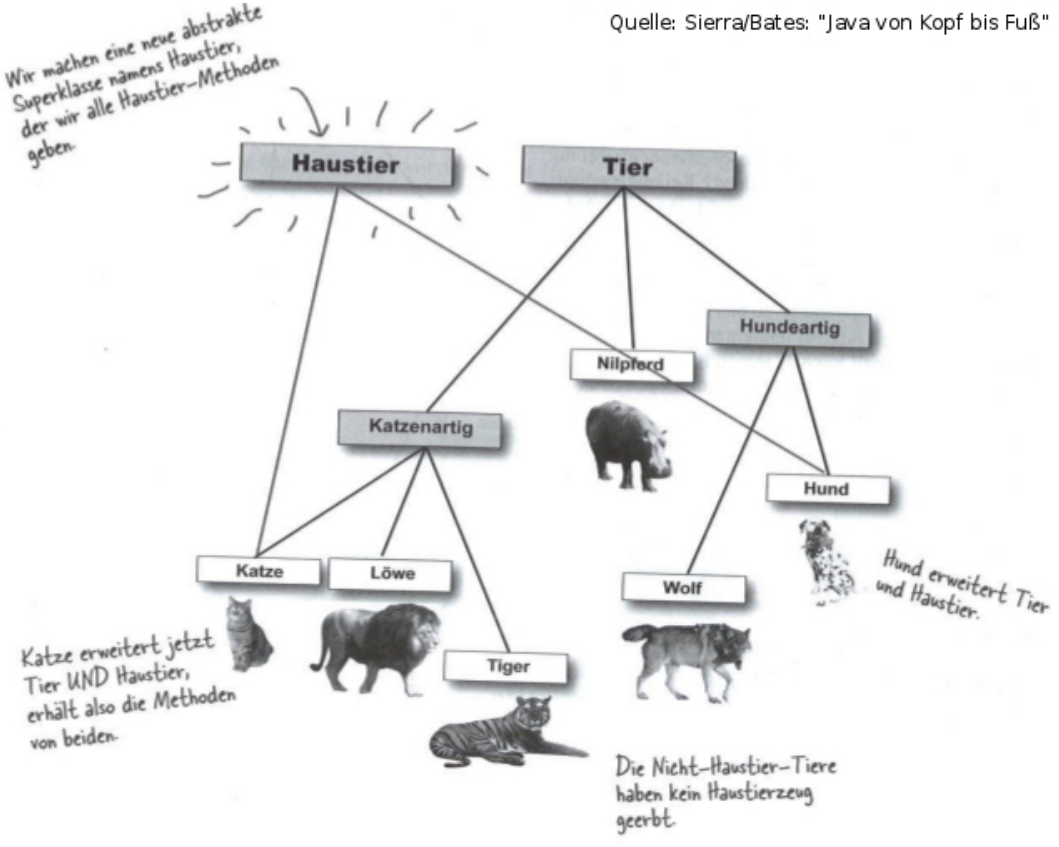
Abstrakte Klassen mit ausschließlich öffentlichen, rein virtuellen Methoden und ohne Daten

```
// Interface --- abstrakte Klasse mit 100% rein virtuellen
class IstVergleichbar {                  // Methoden ohne Daten
public:
 virtual bool operator<(const IstVergleichbar&) const = 0;
 virtual ~IstVergleichbar() = 0;
};

// Implementierende Klasse
class MeineDaten : public IstVergleichbar {
 /* private Daten und Methoden */
public:
 bool operator<(const IstVergleichbar& rhs) const {
  const MeineDaten* pMeineDaten=
    dynamic_cast<const MeineDaten*>(&rhs);
  if (!pMeineDaten) { /* Werfe Ausnahme */ }
  /* Jetzt den Vergleich implementieren */
 }
};
```

III.4b Interfaces

Definition von Rollen und Eigenschaften durch Interfaces

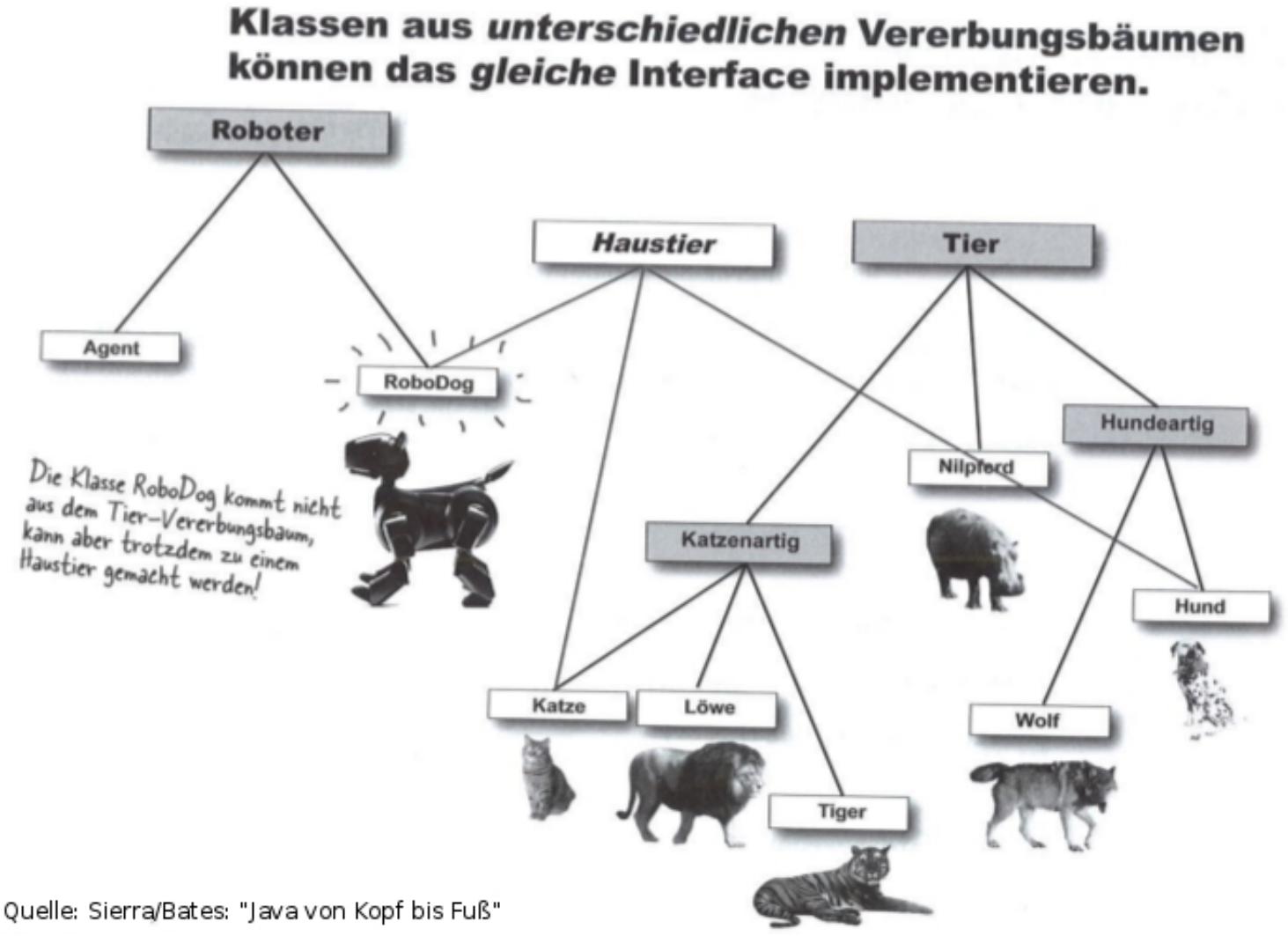
Interfaces definieren Rollen / Eigenschaften von Klassen in einer Vererbungshierarchie.



III.4b Interfaces

Mehrfachvererbung bei Interfaces

Klassen können mehrere Interfaces implementieren und ein Interface kann verschiedene Implementierungen haben. Dies funktioniert auch über verschiedene Vererbungsbäume hinweg



Quelle: Sierra/Bates: "Java von Kopf bis Fuß"

III.4c Vererbung

Inhalt

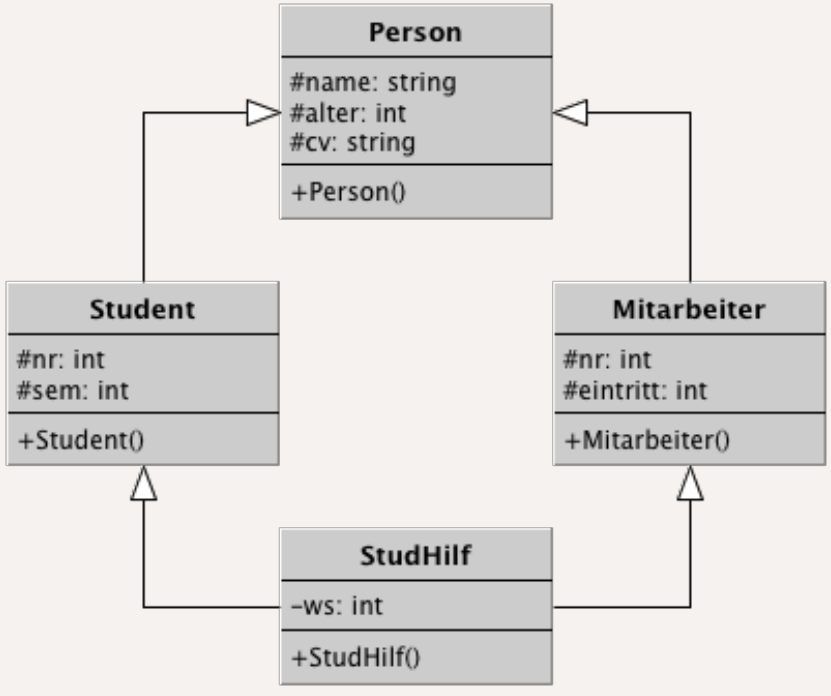
III.4c Mehrfachvererbung

C++ erlaubt grundsätzlich das Erben von mehreren Basisklassen:
class U: **public** B1, **public** B2 { ... };

Warnung | Technische Schwierigkeiten!

Die Klasse StudHilf erbt

- Die Attribute Student::nr und Mitarbeiter::nr
- Zweimal (!) die Attribute der Klasse Person:
  - Student::name, Student::alter, Student::CV
  - Mitarbeiter::name, Mitarbeiter::alter, Mitarbeiter::CV



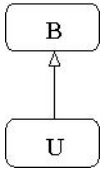
⇒ doppelte Datenhaltung!

「Workaround: „virtuelle“ Vererbung vererbt nur Zeiger auf Daten (unüblich)」

Handreichung: Vererbung

III.5 Handreichung: Vererbung

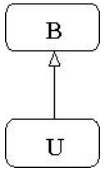
öffentliche Vererbung, private Vererbung und Delegation



- öffentliche Vererbung als "ist ein"-Beziehung
  - Jedes Objekt vom Typ U darf auch für ein Objekt vom Typ B stehen (insbesondere bei Parameterübergabe an Funktionen)
  - Die Unterklasse U erbt die Schnittstelle der Basisklasse und *erweitert* diese
- private Vererbung als "ist implementiert mit"-Beziehung
  - U kann nicht überall dort stehen, wo B erwartet wird
  - U erbt jedoch die Implementation der B-Methoden (Verwendung mittels using-Direktive oder Delegation)
- Delegation als "benutzt"-Beziehung
  - modelliert eine Klassenbeziehung, die nicht als öffentliche Vererbung im Sinne von Schnittstellenerweiterung besteht
  - Alternative zu privater Vererbung (durch "Benutzung"der benötigten Methoden)

III.5 Handreichung: Vererbung

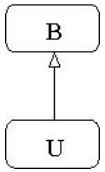
Komponentenselektion



- Objektzugriff
  - Komponenten*name* wird, beginnend beim aktuellen Objekt, von unten nach oben gesucht
  - Überdeckung aller gleichnamigen Komponenten der in der Hierarchie "darüberliegenden" Klassen, unabhängig von der Signatur
  - Überladung findet nur zwischen den gleichnamigen Methoden statt, die auf der "Ebene" liegen, in der die erste Namensübereinstimmung gefunden wurde
  - "Ebenenübergreifende" Überladung kann mittels der using-Direktive realisiert werden
- Zeiger-/Referenzzugriff
  - ermöglicht *generischen* Zugriff auf alle Objektinstanzen von B und seiner Unterklassen
  - Komponentenauswahl gemäß statischen Objekttyp
  - Auswahl der Methoden*implementation* gemäß dynamischen Objekttyp, wenn die Methode virtuell vereinbart wurde

III.5 Handreichung: Vererbung

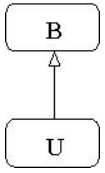
virtuelle Methoden



- gewöhnliche Methoden in B definieren eine *obligatorische Imlementation*, die in der Unterklassen nicht überschrieben werden soll
- virtuelle Methoden definieren eine *Schnittstelle und eine Standardimplementation*, die die Unterklasse erbt und überschrieben werden *kann*
- rein virtuelle Methoden definieren *Schnittstelle* (ggf. eine Teil-Implementation), die in der Unterklasse implementiert werden *muss*
- abstrakte Klassen sind Klassen, die mindestens eine rein virtuelle Methode haben. Von Ihnen können keine Objekte angelegt werden, sie dienen als Blaupause für davon abgeleitete konkrete Klassen, die die Implementation zur Verfügung stellen.
  Abstrakte Klassen sind allgemeine Konzepte, die jede andere (abgeleitete) Klasse der Klassenhierarchie konkretisiert
- Interfaces sind Klassen mit ausschließlich rein virtuellen Methoden ohne Daten. Sie fungieren als Anforderungen, Schnittstellen oder Eigenschaften für die von Ihnen abgeleiteten konkreten Klassen.
  Mehrfachvererbung von Schnittstellen ist kein Problem

III.5 Handreichung: Vererbung

Polymorphie der Standardmethoden



- Konstruktoren sind nie virtuell, da sie durch den Klassennamen stets eindeutig sind
- Destruktoren können (als Ausnahme von der Namensregel) und sollten virtuell sein, damit Objekte der Unterklassen korrekt zerstört werden können
- Kopierkonstruktoren sind nach Namensregel nicht virtuell. Eine polymorphe Objektkopie kann durch eine einheitliche virtuelle Methode (z.B. clone()) erreicht werden
- Der Zuweisungsoperator kann nicht vererbt werden, stattdessen überdeckt jeder Zuweisungsoperator der Unterklassen die "darüberliegenden" Versionen. Polymorphie wird ermöglicht durch Aufruf einer einheitlichen virtuellen Methode
  virtual U& assign(**const** B&) aus dem Standard-Zuweisungsoperator heraus und Nutzung von **dynamic_cast** zur Typprüfung des Operanden.