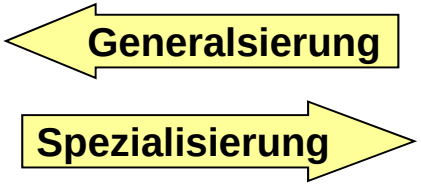
Programmieren - Vererbung & Polymorphie

Reiner Nitsch
✉ r.nitsch@fbi.h-da.de

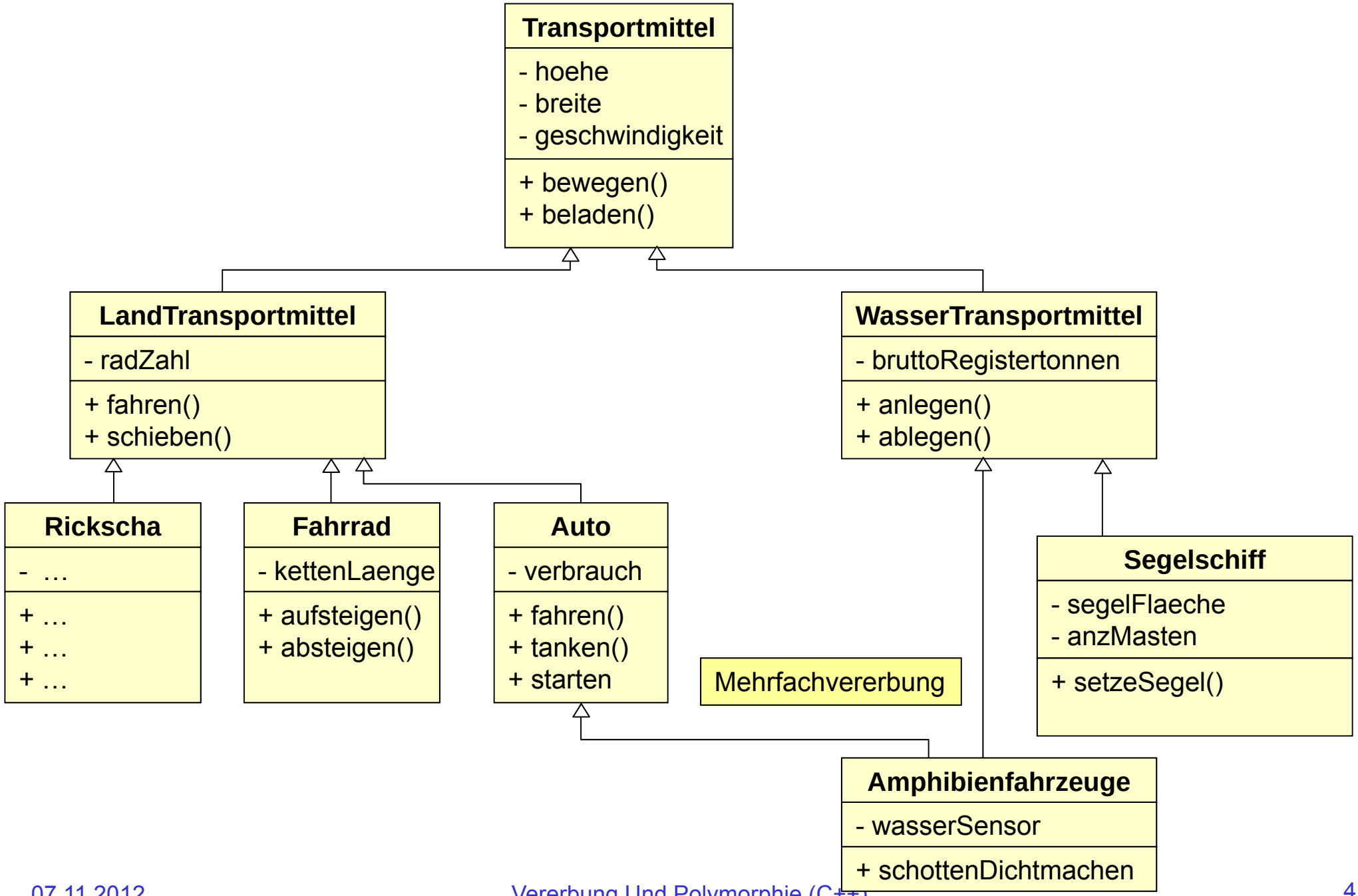
Vererbung - Was ist das?

- Alle Objekte derselben Klasse haben die gleichen Eigenschaften, beschrieben durch ihre Membervariablen, und das gleiche Verhalten, beschrieben durch die Methoden ihrer Klasse.
- Als Abbildungen von Dingen des realen Lebens sind viele Objekte in OO-Anwendungen einander nur ähnlich (z.B. Giro-, Spar-, Festgeldkonten).
  - Ihre Eigenschaften sind teilweise gleich (z.B. Kontonummer, Kontostand, Kunde). es gibt aber auch Unterschiede (Dispositionskredit, Laufzeit, Zinssatz)
  - Ihr Verhalten ist teilweise gleich (Eröffnen, einzahlen, auszahlen). Es gibt aber auch Unterschiede (Bei Festgeldkonten einzahlen und auszahlen nur am Beginn und Ende der Laufzeit, bei Sparkonten auszahlen unter Abzug der Vorschusszinsen, bei Girokonten Berücksichtigung des Dispolimits beim auszahlen).
- Konsequenzen:
  - Diese Gemeinsamkeiten sollten aus Gründen der Effizienz auch gemeinsam behandelt werden (in OOP in der Oberklasse)
  - Die Unterschiede werden separat behandelt (in OOP in den Unterklassen)
  - Ein Vererbungsmechanismus stellt sicher, dass die in der Oberklasse gebündelten Gemeinsamkeiten auch den Unterklassen zur Verfügung stehen, d.h. vererbt werden
- das Herausfiltern der Gemeinsamkeiten nennt man Generalisierung (Oberklasse = Generalisierung der Unterklassen)
- Das Herausstellen der Unterschiede nennt man Spezialisierung (Unterklassen =Spezialisierungen der Oberklasse). Da die Unterklassen von der Oberklasse erben, müssen diese nur noch die Unterschiede definieren.
- Die Vererbung ist hierarchisch organisiert. Klassen mit Unterklasse aber ohne Oberklasse nennt man Basisklasse.
- Vererbung ist ein wichtiges Konzept zur Unterstützung der Wiederverwendbarkeit, wenn auch nicht das Wichtigste.

Ober-klasse	Unterklassen
Konto	Girokonto Sparkonto Festgeldkonto
Studien-gang	Bachelor KoSI Master
Flaeche	Kreis Dreieck Rechteck



Vererbung in UML am Beispiel Transportmittel



Vererbung in C++


```
class Transportmittel {
    double hoehe, breite, geschwindigkeit;
public:
    void bewegen();
    void beladen();
};
```

erben

```
class LandTransportmittel : public
Transportmittel {
    int radZahl;
public:
    void bremsen();
    void fahren();
};
```

```
class Auto : public LandTransportmittel {
    double verbrauch;
public:
    void fahren();
    void tanken(double);
    void starten();
};
```

Redefiniert Transportmittel::fahren()

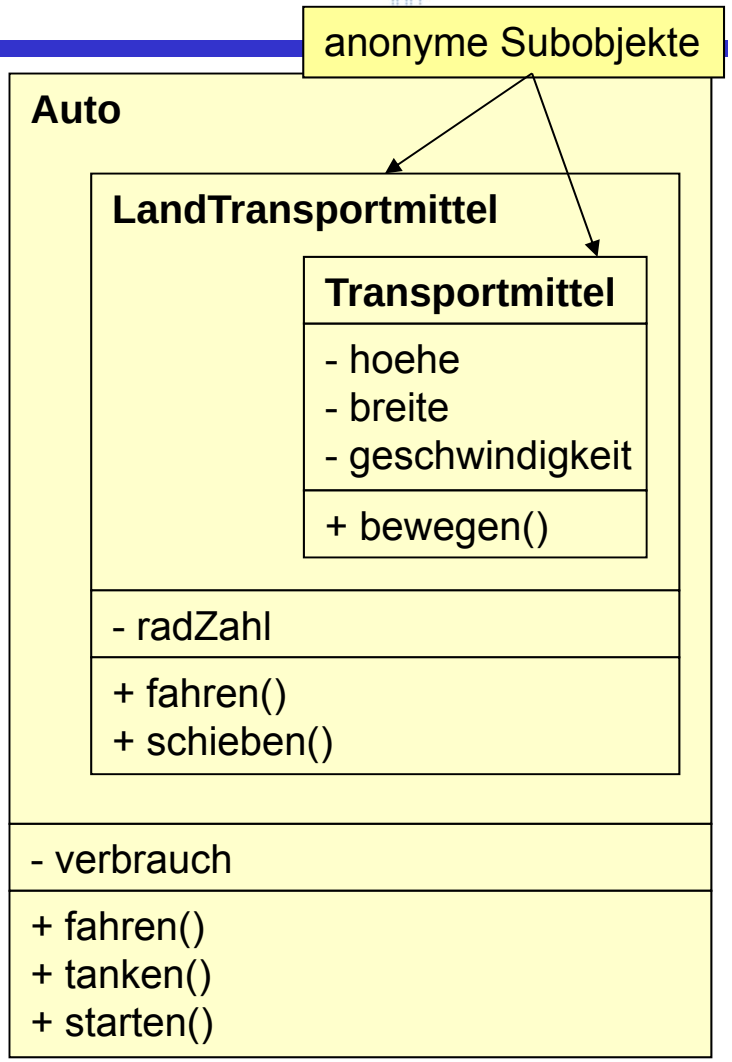
```
class WasserTransportmittel : public Transportmittel {
    int bruttoRegistertonnen;
public:
    void anlegen();
    void ablegen();
};
```

```
class Amphibienfahrzeug
: public LandTransportmittel, public Wassertransportmittel
 {
    char* wasserSensor;
public:
    void schottenDichtmachen();
};
```

Mehrfachvererbung

Vererben - Was ist damit gemeint?

- **Jedes Objekt `objAbgeleitet` vom Typ `Abgeleitet` enthält ein (anonymes) Objekt vom Typ der Oberklasse (Subobjekt).** Durch diesen Mechanismus hat ein `Auto`-Objekt die Attribute `verbrauch`, `radZahl`, `hoehe`, `breite` und `geschwindigkeit`. Der Standard-Konstruktor des Subobjekts wird implizit vor dem Konstruktor von `Abgeleitet` aufgerufen.
- **Jede `public`-Memberfunktion der Oberklasse(n) kann von `objAbgeleitet` aufgerufen werden.** `Auto`- und `Landtransportmittel`-Objekte können die Methode `bewegen()` aufrufen. Beim Aufruf ist nicht zu erkennen, in welcher (Ober)Klasse die Methode implementiert ist.
- **Klasse `Abgeleitet` kann zusätzliche Attribute und Methoden enthalten, die für die Oberklasse keine Bedeutung haben** (z.B. `verbrauch` und `tanken()` bei `Auto`)



Vererben - Was ist damit gemeint?

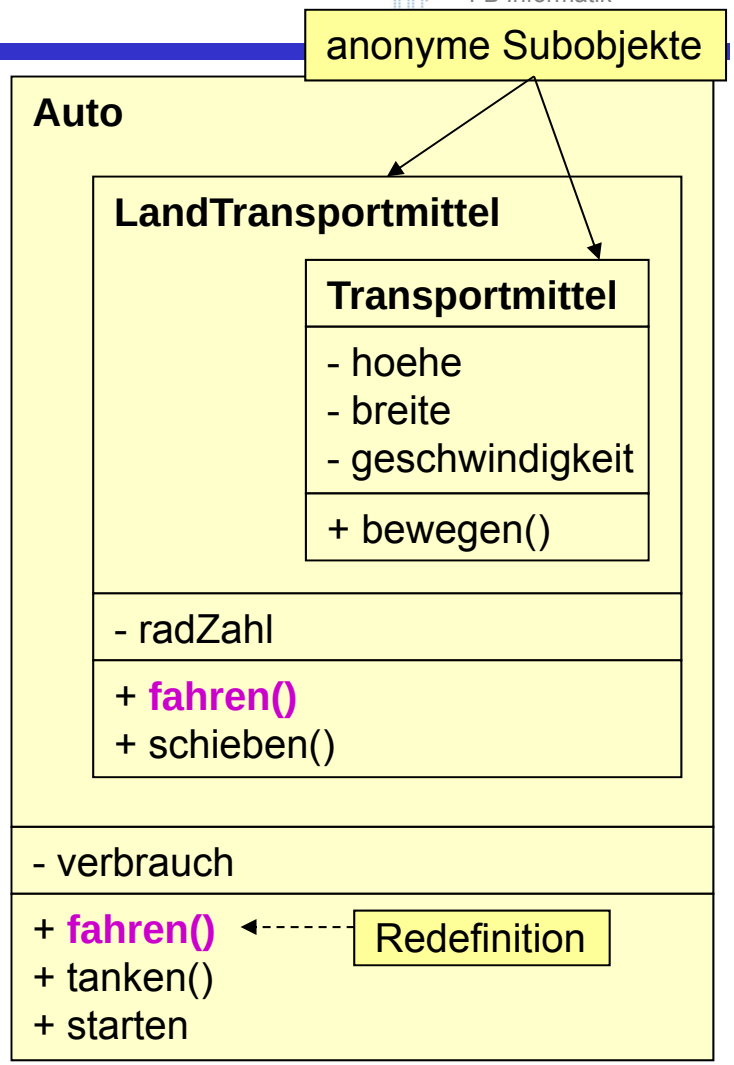
- **Wenn eine geerbte Methode in `Abgeleitet` geändert oder verfeinert werden muss, kann dies durch Redefinition, d.h. durch die Deklaration und Implementation einer Methode gleicher Signatur in `Abgeleitet`** (Beispiel: `Auto::fahren`) **geschehen**. Es ist durchaus üblich, dass dabei die geerbte Methode (Beispiel: `Landtransportmittel::fahren`) wiederverwendet wird.
- **Die Zuweisung**

```
objOberklasse = objAbgeleitet
```

ist zulässig. Kein Typkonflikt, weil `objOberklasse` als Subobjekt in `objAbgeleitet` enthalten ist (`objAbgeleitet` ***ist*** auch ein `objOberklasse`) .
Die Attribute des Subobjekts `objAbgeleitet` werden in die des `objOberklasse` kopiert.
ABER: Die Umkehrung

```
objAbgeleitet = objOberklasse
```

ist nicht erlaubt, weil z.B. zusätzliche Attribute von `Abgeleitet` undefiniert bleiben würden.




```
Landtransportmittel ltm(4);
Auto vw(4,8.2);
ltm = vw;      ------- OK!
vw = ltm;      ------- Fehler!
```

Vererbung - Begriffe

Durch die Deklaration einer neuen Klasse unter Bezugnahme auf eine Oberklasse

- übernimmt die neue Klasse (= abgeleitete Klasse) die Attribute und Methoden der Oberklasse ("Attribute/Verhaltensweisen werden geerbt")

  Implementationsvererbung (= Wiederverwendung von Code)

- kann die abgeleitete Klassen zusätzliche Attribute haben ("**hat mehr**")

  ⇒ Spezialisierung

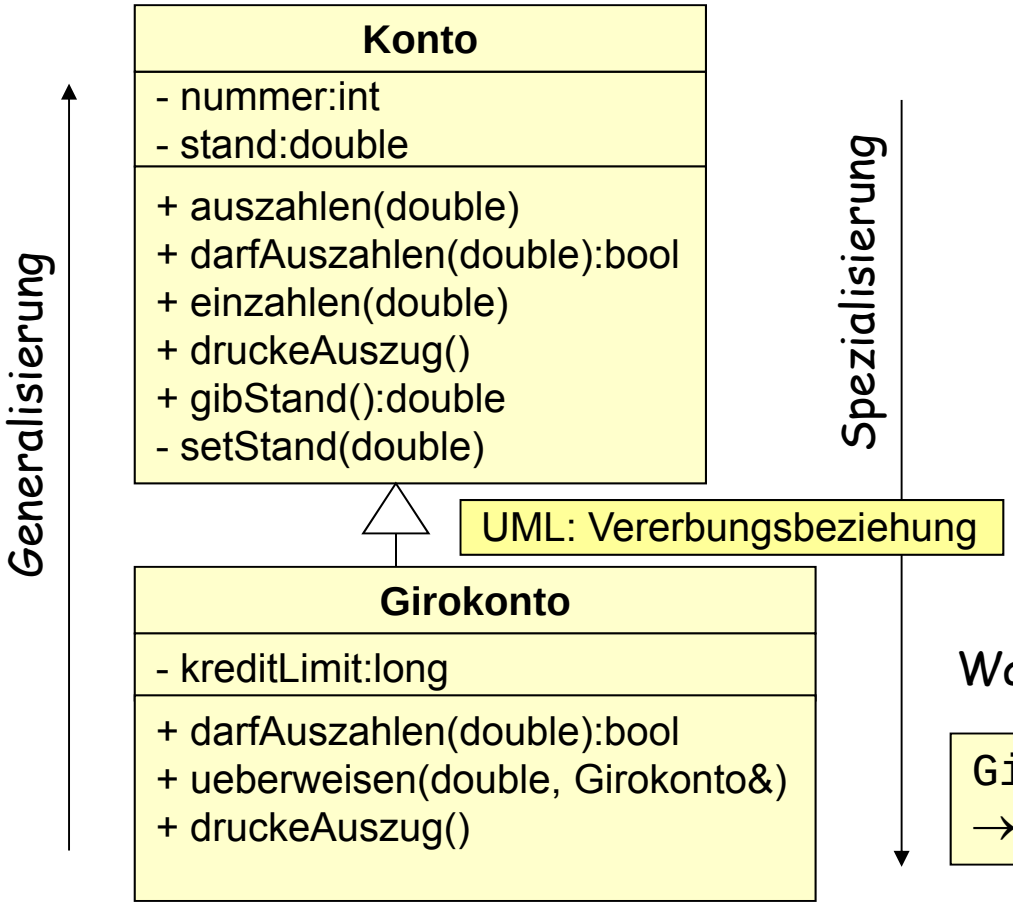
- kann die abgeleitete Klasse können ihr Verhalten durch **Redefinieren** der ererbten und/oder zusätzliche Methoden erweitern ("**kann mehr**")

  ⇒ Spezialisierung

- wird speziell bei `public`-Ableitung eine "**is a**"-Beziehung zwischen der Oberklasse und der abgeleiteten Klasse hergestellt mit Auswirkungen auf die Typkontrolle.
  Zur Erinnerung: Assoziation, Aggregation und Komposition beschreiben eine **know-a**- bzw. **has-a**-Beziehung

  public-Ableitung ⇒ Schnittstellenvererbung

Ein ausführliches Beispiel



Warum nur 1-mal `nummer`, `stand`, `einzahlen()`?

`Girokonto` erbt alle Attribute u. Methoden von `Konto`
→ Alles was `Konto` hat, hat `Girokonto` auch

Warum 2mal `druckeAuszug()` ?

`Konto::druckeAuszug()` gibt `nummer` und `stand` aus.
`Girokonto::druckeAuszug()` gibt zusätzlich `kreditLimit` aus.

→ `Girokonto::druckeAuszug()` redefiniert `Konto::druckeAuszug()`
Typ des Objekts, des Pointers oder der Referenz zur Compilierzeit entscheidet, welches `druckeAuszug()` gemeint ist (***statisches Binden***, ***early binding***).

Vererbung in C++ - Basisklasse Konto


```
class Konto {
  int nummer;
  double stand;
  void setStand(double neuerStand) { stand = neuerStand; }
public:
  Konto() { nummer=0; stand=0.0; }
  Konto( int nr, double anfangsStand )
 { nummer = nr; stand = anfangsStand; }

  void einzahlen(double betrag){
    if (betrag<0.0) throw ...;
    stand += betrag;
  }
  bool darfAuszahlen(double betrag) const { return betrag<=stand; }

  void auszahlen(double betrag) {
    if (betrag<0.) throw ...;
    if(this->darfAuszahlen(betrag)) stand -=betrag; else throw ...
  }
  void druckeAuszug() const {
    cout <<"Kontonummer: "<<nummer<<endl<<"Kontostand:  "<<stand<< endl;
  }
  double gibStand() const { return stand; }
};
```

Vererbung in C++ - abgeleitete Klasse Girokonto

Ableitungsspezifizierer

abgeleitete Klasse

Basisklasse

```
class Girokonto : public Konto
{
private:
  long kreditLimit;
public:
  Girokonto(int nr, double anfangsStand, long limit)
  ~Girokonto() { }
void druckeAuszug()const;
void ueberweisen(double betrag, Girokonto& k);
bool darfAuszahlen(double betrag)const;
};
```

Girokonto wird von Konto **public** abgeleitet:

→ Girokonto ist auch ein Konto (*is-a-Beziehung*)

→ Girokonto erbt von Konto

- **die Schnittstelle**
- alle Attribute
- alle Methoden

hat mehr als Konto → **Spezialisierung**

muss hier spezieller sein → **Redefinition** von `Konto::druckeAuszug()`

kann mehr als Konto → **Spezialisierung**

Redefinition

Unterschied: Redefinieren ↔ Überladen

- Überladene (redefinierte) Methode gehört zu gleichem (anderem) Namensbereich.
- redefinierte Methoden MÜSSEN die gleiche Signatur **und** Rückgabetyp, wie die gleichnamige Methode in der Oberklasse haben.
- Redefinierte Methoden werden vom Compiler auch in den Oberklassen gesucht, wenn sie in einer Unterklasse nicht enthalten ist.

Vererbung und Konstruktoren

```
//Konto-Konstruktor **********************
Konto::Konto(int nr,double anfangsStand):nummer(nr),stand(anfangsStand){  }
//Girokonto-Konstruktor ******************
Girokonto::Girokonto(int nr, double anfangsStand, long limit)
  :nummer(nr),stand(anfangsStand),kreditLimit(limit) { }    // Fehler
```

Vorinitialisierungsliste darf enthalten:
- Attribute der Klasse selbst aber **keine geerbten Attribute**,
- Kontruktoraufrufe der Oberklassen

```
Girokonto::Girokonto(int nr, double anfangsStand, long limit)
  /* hier automatischer Aufruf der Oberklassen-Standard-Konstruktoren
     beginnend bei der Basisklasse.
     Destruktion in umgekehrter Reihenfolge */ {
  nummer = nr;
  stand = anfangsStand;
  kreditLimit = limit;
}
```

Fehler: Zugriffschutz!
Effizienz schlecht: `nummer` und `stand` wurden schon vom Std-Konstr. initialisiert.
Besser: Initialisierung in ***einem*** Schritt durch expliziten Aufruf des Oberklassen-Konstruktors

```
Girokonto::Girokonto(  int nr, double anfangsStand, long limit)
: Konto(nr, anfangsStand) {
  kreditLimit = limit;
}
```

Vorinitialisierung

Vererbung in C++ - Zugriffschutz

```
void Girokonto::druckeAuszug() {
  cout << nummer << endl;
  cout << stand  << endl;
  cout << kreditLimit << endl;
}
```

Redefinition von `Konto::druckeAuszug`

geht nicht! Warum? Zugriffschutz (private!)

hier spezieller

Abhilfe:

1. Weniger Zugriffsschutz durch **`protected`** oder besser
2. `public`-Zugriffsfunktionen in Klasse `Konto` oder am besten
3. geerbte Methode `druckeAuszug()` der Klasse `Konto` aufrufen

zu 1. Übersicht zum Zugriffschutz

Level	Attribute und Methoden der Klasse
public	unterliegen keiner Zugriffsbeschränkung
private	sind nur im Namensbereich der Klasse und für friend-Klassen/Funktionen sichtbar
protected	sind nur im Namensbereich der Klasse und aller von ihr **public** abgeleiteten Klassen sichtbar

zu 1. Aufweichung des Zugriffsschutzes für abgeleitete Klassen

```
class Konto {
private:
    //...
protected:
    int      nummer;
    double stand;
    void setStand(double);
public:
    //...
}
```

Zugriff nur für Objekte derselben Klasse bzw. von friend-Funktionen und -Klassen

Zugriff für Objekte derselben Klasse, `public`-abgeleiteter Klassen bzw. von `friend`-Funktionen und -Klassen

```
void Girokonto::druckeAuszug() {
   cout << nummer << endl;
   cout << stand  << endl;
   cout << kreditLimit << endl;
}
```

geht jetzt, weil `protected`

Nachteil von protected:

- Grösserer Sichtbarkeitsbereich der Attribute
- Mehraufwand bei Fehlersuche und Änderungen der internen Datenstruktur

⇒ deshalb vermeiden!

Guter Programmierstil: `protected` möglichst **nicht** verwenden!

Zu 3. Redefinierte Funktionen der Basisklasse wiederverwenden

Falsch!

```
void Girokonto::druckeAuszug() const {
  cout << nummer << endl;
  cout << stand  << endl;
  druckeAuszug();
  cout << kreditLimit << endl;
}
```

Nummer und Stand wieder private Attribute!

Absturz! Warum?

Hier wird Girokonto::druckeAuszug() rekursiv aufgerufen

Erklärung: Die Klasse GiroKonto hat nach der Redefinition immer noch die geerbte Methode `Konto::druckeAuszug()`. Durch die Redefinition kam eine weitere Methode `GiroKonto::druckeAuszug()` hinzu. Bei Aufruf ohne Namensbereichsangabe bezieht sich der Aufruf jedoch immer auf den aktuellen Namensbereich (hier: GiroKonto). Anders ausgedrückt: Die unmittelbare Sichtbarkeit der Methode `Konto::druckeAuszug()` wurde durch die Redefinition für die Klasse GiroKonto (und alle von ihr abgeleiteten Klassen) aufgehoben. Durch Angabe des Namensbereichs beim Aufruf kann sie aber wieder sichtbar gemacht werden.

Richtig!

```
void Girokonto::druckeAuszug() const {
  cout << nummer << endl;
  cout << stand  << endl;
  Konto::druckeAuszug();
  cout << kreditLimit << endl;
}
```

☺ Wiederverwendung von Code

Vererbung in C++

```
bool Girokonto::darfAuszahlen(double betrag) const {
  return betrag <= gibStand() + kreditLimit;
}
```

Redefinition

Zugriffsfunktion statt `protected`-Attribute!

```
void main()
{
  Girokonto gk(1,0,1000);
  gk.einzahlen(1000);
  gk.druckeAuszug();
  gk.auszahlen(100);
}
```

Objektversion

von `Konto` geerbt → Wiederverwendung von Code

in `Girokonto` redef.

von `Konto` geerbt

Zur Erinnerung: Beim Aufruf ist nicht zu erkennen, in welcher (Ober)Klasse die Methode implementiert ist.

Zur Erinnerung: Typ des Objekts, des Pointers oder der Referenz zur Compilierzeit (***statisches Binden***, ***early binding***) entscheidet, welches druckeAuszug() gemeint ist

```
void main()
{
  Girokonto *pgk =
    new Girokonto(1,0,1000);
  pgk->einzahlen(1000);
  pgk->druckeAuszug();
  pgk->auszahlen(100);
}
```

Pointerversion

```
void main()
{
  Girokonto& gk = *(new Girokonto(1,0,1000));
  gk.einzahlen(1000);
  gk.druckeAuszug();
  gk.auszahlen(100);
  delete &gk;
}
```

Referenz-Version

Identische Ergebnisse bei allen Anwendungs-Versionen

Probleme mit der Wiederverwendung von vererbtem Code

- `darfAuszahlen` wurde von Girokonto redefiniert (Kreditlimit wird berücksichtigt)
- `Konto::auszahlen` wurde nicht redefiniert, weil unterstellt wurde, dass in `Konto::auszahlen` schon das richtige `darfAuszahlen` benutzt wird.

Irrtum!!

Obwohl `Konto::auszahlen` an Girokonto vererbt wurde, wird stets nur `Konto::darfAuszahlen` benutzt, wenn `auszahlen` für ein Girokonto-Objekt aufgerufen wird. D.h. das KreditLimit wird bei Auszahlung von einem Girokonto nicht geprüft ☹

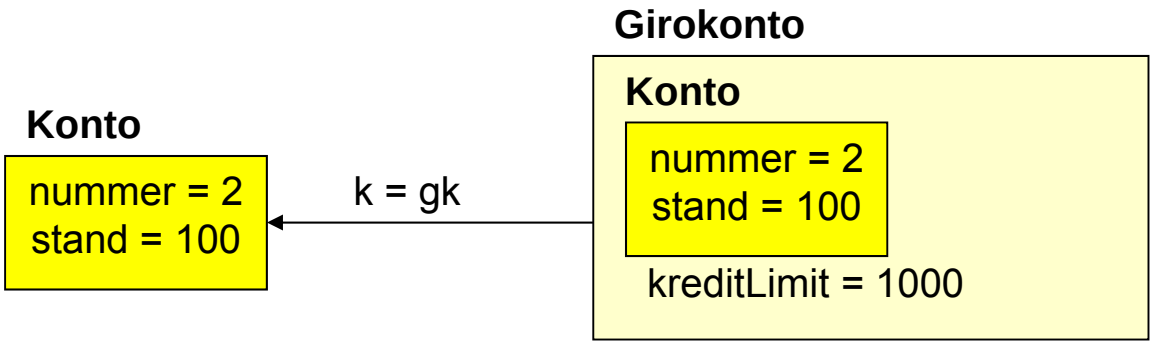
Grund: Der Compiler entscheidet, an welchen Typ ein Aufruf gebunden ist. Der Compiler zur Compilezeit aber nicht wissen, auf welchen Typ eine Konto-Referenz oder ein Konto-Zeiger (z.B. `this`) zur Laufzeit zeigen wird. Solange er keine anderen Instruktionen erhält, hat für ihn ein aufrufendes Objekt stets den Typ, der sich aus dem aktuellen Namensbereich ergibt. Wenn in `Konto::auszahlen` der Aufruf `this->darfAuszahlen` übersetzt, ist `this` vom Typ `Konto*`, d.h. `Konto::darfAuszahlen` wird aufgerufen. Diese Art der Bindung an den Typ bezeichnet man als "early binding" oder auch "static binding".

Vererbung und Zuweisung

Zur Erinnerung: Die Zuweisung objOberklasse = objAbgeleitet ist zulässig. Kein Typkonflikt, weil objOberklasse als Subobjekt in objAbgeleitet enthalten ist (objAbgeleitet ***ist*** auch ein objOberklasse) . Die Attribute von Subobjekt werden in die von objOberklasse kopiert

```
void main() {
  Konto k(1,0);
  Girokonto gk(2, 100, 1000);
  // Zuweisung an Basisklassenobjekt
  k = gk;
```

Normalerweise Typkonflikt! Aber: hier OK, weil Zuweisung an Basisklasse



- **Regel:** Überall, wo ein Objekt, Zeiger oder Referenz der Basisklasse erwartet wird, darf auch ein entsprechendes Element einer abgeleiteten Klasse stehen ☺.

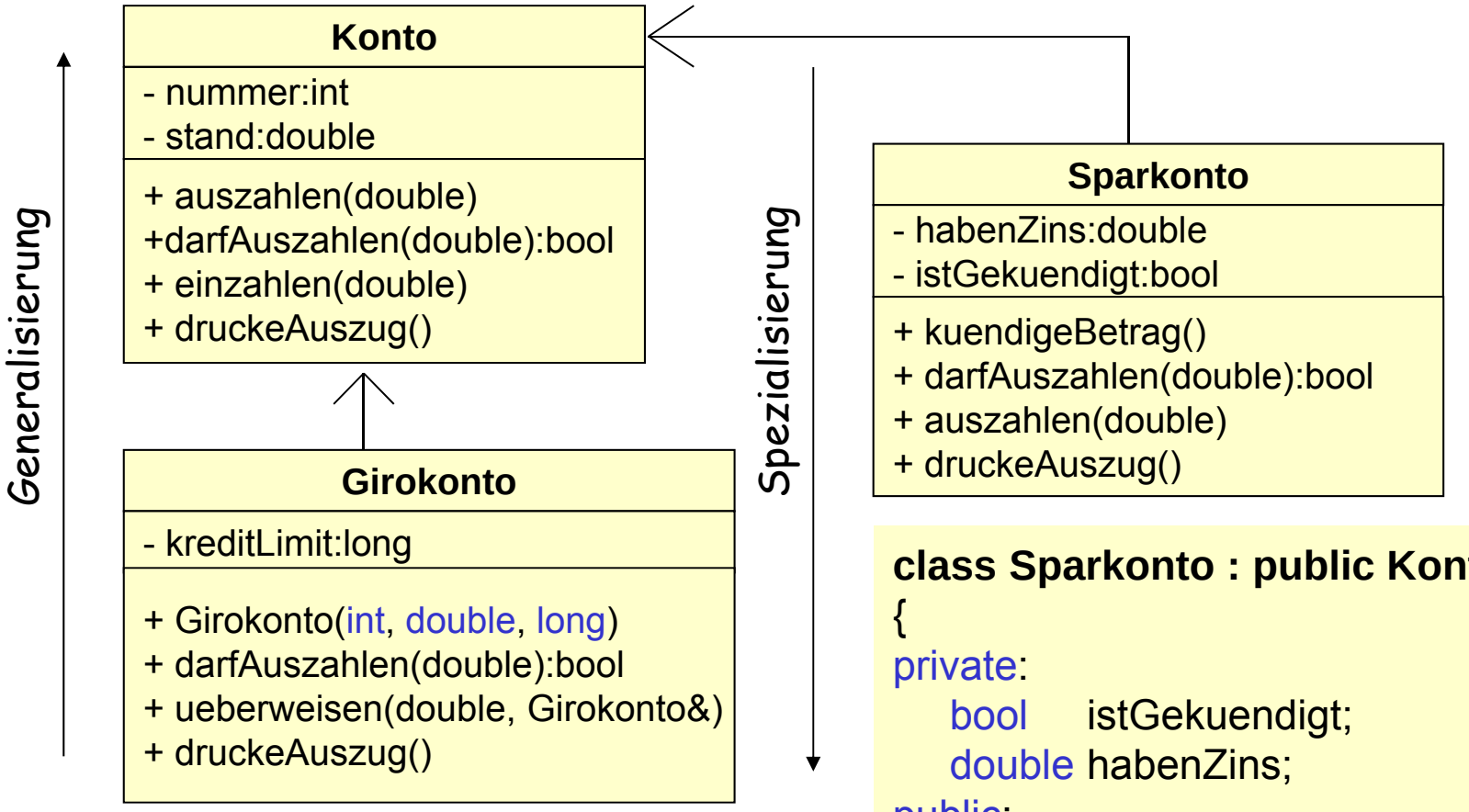
`Girokonto` ist auch ein `Konto` (**is-a**-Beziehung)

☹
- Der Compiler macht hier eine implizite Typumwandlung.
- Danach steht k nur noch die Schnittstelle und Implementierung von `Konto` zur Verfügung, d.h. der Aufruf k.getKreditLimit() könnte nicht übersetzt werden.

- **`gk = k;`**

Umgekehrt geht's nicht, weil `Konto` natürlich kein `Girokonto` ist.
Ihm fehlt z.B. das Attribut `kreditLimit` und die Methode `ueberweisen`.

Noch eine Spezialisierung: Sparkonto




```
class Sparkonto : public Konto
{
private:
    bool    istGekuendigt;
    double habenZins;
public:
    Sparkonto( int nr, double  stand, double zins);
    ~Sparkonto();
    void druckeAuszug() const;
    bool darfAuszahlen( double betrag) const;
    void auszahlen( double betrag );
    void kuendigeBetrag();
    };
```

Redefinition:
Sparkonto ist hier spezieller

kann mehr

Sparkonto - Implementationsdetails

```
bool Sparkonto::darfAuszahlen( double betrag ) const
// gibt true zurück, wenn gekuendigt und Guthaben ausreicht
{
   if(istGekuendigt)
      return this->gibStand() >= betrag;
   else {
      cout << "Zuerst kuendigen!" << endl;
      return false;
   }
}
```

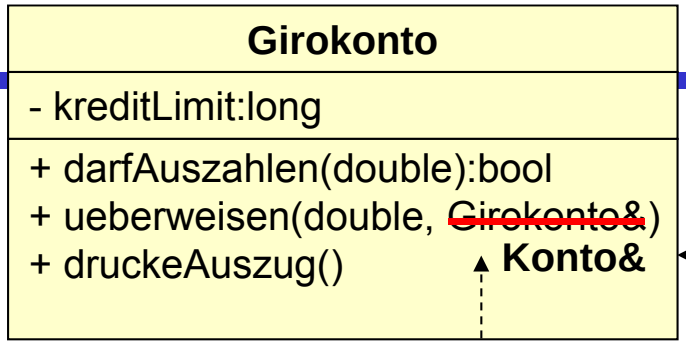
Aufruf der geerbten Methode `Konto::gibStand()`

```
void Sparkonto::auszahlen( double betrag )
//Prüft Auszahlungsbedingungen und verwaltet istGekuendigt
{
     Konto::auszahlen(betrag);
     istGekuendigt = false;
}
```

Wiederverwendung von Methode `Konto::auszahlen(double)`

Flag **`istGekuendigt`** wird zurück gesetzt.

Anwendung der Konto-Hierarchie




```
void main() {
   Girokonto gk(1,1000,1000);
   Sparkonto sk(2,0,3);
   gk.ueberweisen(1500, sk);
   sk. druckeAuszug();
}
```

Typkonflikt: hier wird ein Girokonto-Objekt erwartet

```
void Girokonto::ueberweisen(double betrag, ~~Girokonto&~~ Konto& k) {
    this->auszahlen(betrag);
    k.einzahlen(betrag);
}
```

Abhilfe:
"is a"-Beziehung ausnutzen und Typ des 2. Parameters in `Konto&` ändern

Einem Zeiger-Typ `Girokonto*` steht die `Girokonto`-SS sowie die SS aller Oberklassen zur Verfügung. Da SS `Girokonto` kein `auszahlen` kennt sucht der Compiler `auszahlen` in den Oberklassen und findet `Konto::auszahlen`

vorher:
Typ `Girokonto& Girokonto::einzahlen` gibt es nicht!
Compiler sucht `einzahlen` in Oberklassen und findet `Konto::einzahlen`
nachher:
k ist vom Typ `Konto&`. Weil k nur die `Konto`-SS zur Verfügung steht sucht der Compiler `einzahlen` nur in `Konto`
.

Neue Probleme (s. nächste Folie)

Neue Probleme

```
void Girokonto::ueberweisen( double betrag, Konto& k) {

   this->auszahlen(betrag);
   k.einzahlen(betrag);
}
```

nicht redefiniert: also `Konto::auszahlen`!

Aufrufer ist ein Konto-Objekt: also `Konto::einzahlen()`

Neues Problem:

- Nach der impliziten Typumwandlung von `sk` in die Basisklassenreferenz `k` (oder -Objekt) steht dieser nur noch die Schnittstelle und Implementierung der Basisklasse zur Verfügung, d.h. auch dann, wenn `einzahlen` von `Sparkonto` redefiniert würde, würde `k.einzahlen(betrag)` stets nur `Konto::einzahlen(betrag)`aufrufen.

Grund: Der Compiler entscheidet standardmäßig stets nach dem zur Compilezeit bekannten Typ (hier: `Konto&`).

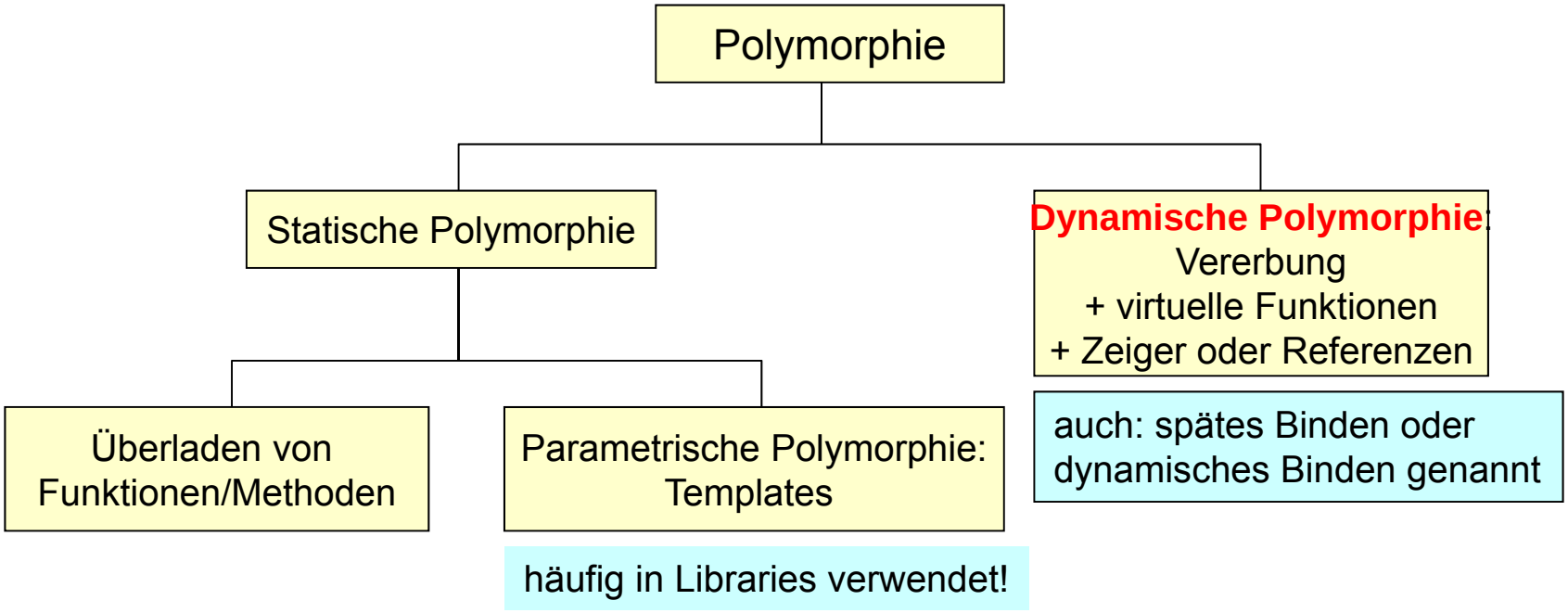
frühes Binden des Methodenaufrufs an den Typ (statisches Binden)

Vererbung für sich allein ist wenig hilfreich!

Objektorientierte Lösung: Dynamische Polymorphie mit Hilfe von virtuellen Funktionen

- Mit **Polymorphie** (Vielgestaltigkeit) ist die Fähigkeit einer **Basisklassen-Objektreferenz** (Zeiger oder Referenz) gemeint, zur Laufzeit eines Programms auf Objekte verschiedenen Typs zu verweisen, und dennoch zur Laufzeit die zum tatsächlichen Objekt-Typ passende Realisierung einer Methode zu bestimmen.
- Eine zur Laufzeit ausgewählte Methode heißt "***virtuelle Funktion***". Den Vorgang der Auswahl zur Laufzeit bezeichnet man als "***späte Bindung*** (***late binding***)"
- Späte Bindung funktioniert in C++ **nur** in einer Vererbungshierarchie **mit public-Ableitung**

Formen der Polymorphie (Griech.: Vielgestaltigkeit)



Merkmale virtueller Methoden

- Virtuelle Methoden **und nur dort**
  - **enthalten Schlüsselwort 'virtual' in Deklaration**
  - **werden in Basisklasse deklariert**
  - werden von abgeleiteter Klasse geerbt
  - können in abgeleiteter Klasse redefiniert werden, müssen dann aber
    - gleiche Signatur und
    - gleichen Ergebnistyp

    wie in der Basisklasse haben und sind automatisch wieder virtuell. Das Schlüsselwort 'virtual' ist hier zwar nicht nötig, sollte zu Dokumentationszwecken aber trotzdem eingesetzt werden.
- Virtuellen Methoden wird indirekt (vom Compiler) die Info über den Objekt**typ** mitgegeben (Erklärung folgt)
- Der Aufruf einer nicht-virtuellen Methode **hängt vom Typ des aufrufenden Zeigers oder der aufrufenden Referenz ab**, den diese(r) zur Compilezeit hat (Early binding).
- Der Aufruf einer **virtuellen** Methode **hängt vom Typ des Objekts ab**, auf das der Zeiger oder die Referenz verweist. Wird eine virtuelle Methode über eine Objektreferenz (Zeiger oder Referenz) angesprochen, ruft das Laufzeitsystem die zum Objekttyp passende Methode auf.

```
class BasisKlasse {
public:
   void g();
   void f();
   virtual void vf1();
   virtual void vf2();
   virtual BasisKlasse* vf3();
   virtual BasisKlasse& vf4();
};

class AbgeleiteteKlasse
    : public BasisKlasse
{
public:
   void f();          // Redefinition
   void vf1();
   char vf2();        // Fehler:
                      //   falscher Ergebnistyp
   BasisKlasse* vf3();
   BasisKlasse& vf4();
   void fa();
};
```

// C++-Standard läßt hier auch `AbgeleiteteKlasse` als Ergebnistyp zu.

Beispiel zu virtuellen Methoden

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
FB Informatik
Prof. Dr. R.Nitsch

```
class BasisKlasse {
public:
  void g();
  void f();
  virtual void vf1();
  virtual void vf2();
  virtual BasisKlasse* vf3();
  virtual BasisKlasse& vf4();
};

class AbgeleiteteKlasse
   : public BasisKlasse
{
public:
  void f();
  virtual void vf1();
  virtual BasisKlasse* vf3();
  virtual BasisKlasse& vf4();
  void fa();
};
```

```
void main () {
   AbgeleiteteKlasse a;
   BasisKlasse b;
   BasisKlasse* pba = &a;
   BasisKlasse* pbb = &b;
   b.g();          //BasisKlasse::g
   a.g();          //BasisKlasse::g (geerbt)
   b.f();          //BasisKlasse::f
   a.f();          //AbgeleiteteKlasse::f (redef.)
   b.vf1();        //BasisKlasse::vf1
   a.vf1();        //AbgeleiteteKlasse::vf1 (redef.)
   pbb->g();       //BasisKlasse::g (nicht virtuell)
   pba->g();       //BasisKlasse::g (nicht virtuell)
   pbb->f();       //BasisKlasse::f (nicht virtuell)
   pba->f();       //BasisKlasse::f (nicht virtuell)
   pbb->vf1();     // BasisKlasse::vf1 (weil virtuell und pbb)
   pba->vf1();     // AbgeleiteteKlasse::vf1 (weil virtuell und pba)
   pbb->vf2();     //BasisKlasse::vf2 (weil virtuell  und  pbb)
   pba->vf2();     //BasisKlasse::vf2 (geerbt und nicht redefiniert)
   pbb->vf3();     // BasisKlasse::vf3 (weil virtuell und pbb)
   pba->vf3();     // AbgeleiteteKlasse::vf3 (weil virtuell und pba)
   pbb->vf4();     // BasisKlasse::vf4 (weil virtuell und pbb)
   pba->vf4();     // AbgeleiteteKlasse::vf4 (weil virtuell und pba)

   AbgeleiteteKlasse* pa = a.vf3();
   pa = static_cast<AbgeleiteteKlasse*>(a.vf3());
}
```

early binding weil Aufruf durch Objekte

late binding nur wenn virtual

//**Typkonflikt**: Ergebnistyp BasisKlasse*

//ok!

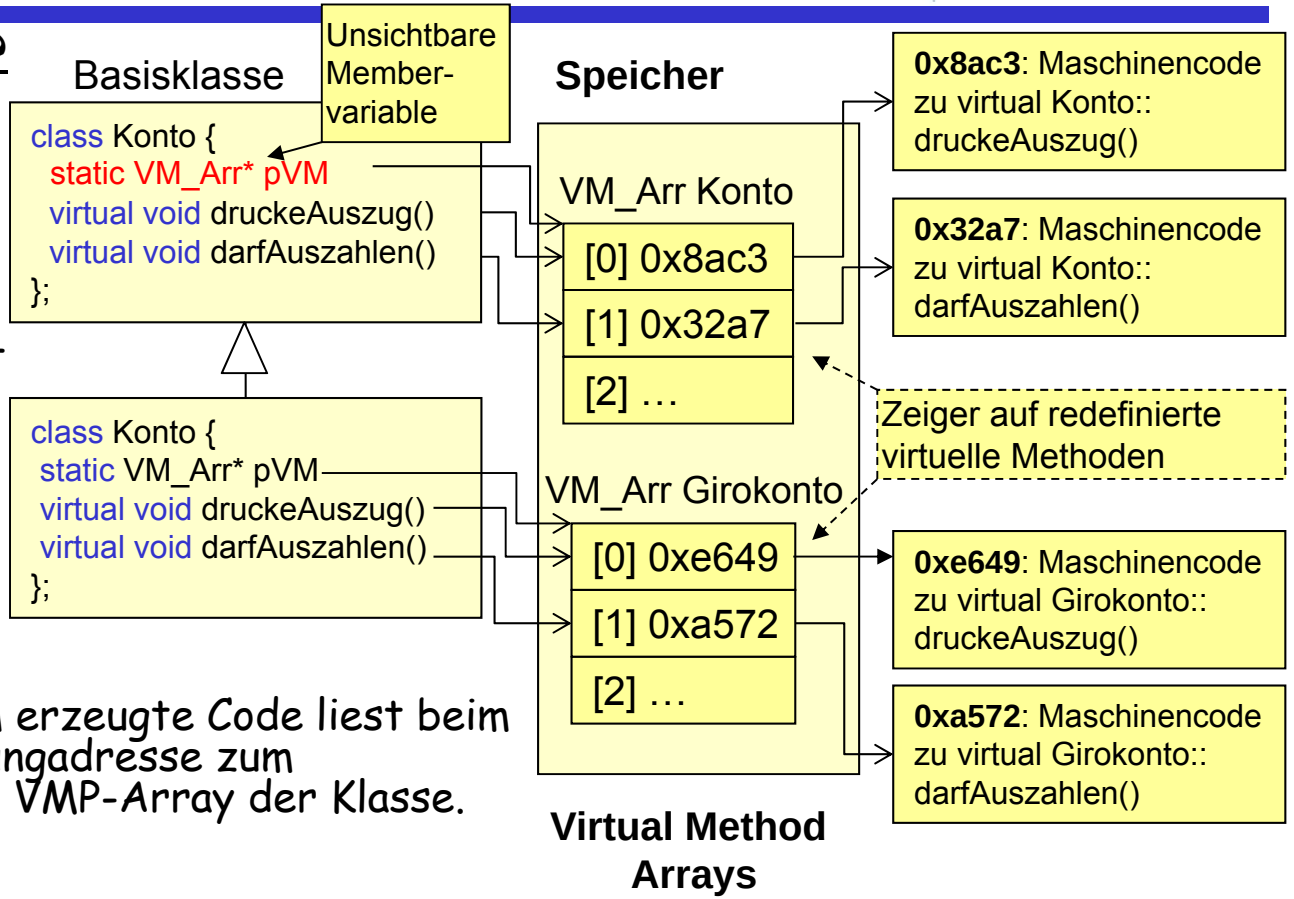
//AbgeleiteteKlasse::vf3 (redef.)

Realisierung der dynamischen Polymorphie (spätes Binden) in C++

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
FB Informatik
Prof. Dr. R.Nitsch

Was bewirkt das Schlüsselwort 'virtual'?

- Für jede Basisklasse bekommt das Laufzeitsystem vom Compiler ein Array von Pointern auf seine virtuellen Methoden VMArr (VMP-Array). Jede dieser Methoden bekommt einen bestimmten Index.
- Jedes Objekt der Klasse bekommt automatisch als erstes (verstecktes) Attribut einen Zeiger auf das VMP-Array (VMPA-Pointer) seiner Klasse
- Für jede abgeleitete Klasse kopiert der Compiler das VMP-Array der Superklasse und modifiziert die Adressen für redefinierte virtuelle Methoden



- Der vom Compiler für das Laufzeitsystem erzeugte Code liest beim Aufruf einer virtuellen Methode die Sprungadresse zum Maschinencode über ihren Index aus dem VMP-Array der Klasse.

Nachteile:

- VMP-Arrays belegen Speicherplatz
- geringer Laufzeitverlust wegen indirekter Adressierung

Vorteile:

- Leichte Erweiterbarkeit der Klassenhierarchie
- Alter Code kann neuen Code nutzen, ohne dass der alte Code geändert werden muss.

→ **Beispiel**: `Konto::auszahlen` nutzt `Sparkonto::darfAuszahlen` ohne dass der Quellcode von Konto geändert wird.

Beispiel **Konto** mit virtuellen Funktionen

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
FB Informatik
Prof. Dr. R.Nitsch

```
class Konto {
  int nummer;
  double stand;
public:
   Konto( int nr=0, double anfangsStand=0.0 )
  { nummer = nr; stand = anfangsStand; }

   virtual ~Konto()
  void einzahlen(double betrag){
    assert( betrag >= 0. ); stand += betrag; }

  virtual bool darfAuszahlen(double betrag) { return betrag<=stand; }
  void auszahlen(double betrag) {
    assert(betrag >= 0.);
    if( darfAuszahlen(betrag) ) stand -=betrag;
  }
  virtual void druckeAuszug()    {
    cout << "Kontonummer: " << nummer << endl;
    cout << "Kontostand:  " << stand  << endl;
   }

   double gibStand() const { return stand; }
};
```

Was muss geändert werden?

Virtuelle Destruktoren sollten immer dann verwendet werden, wenn nicht auszuschliessen ist, dass von der Klasse durch Ableitung neue Klassen gebildet werden.

Beispiel Konto mit virtuellen Funktionen

```
void main() {
   // Teil 1: mit konkreten Objekten
   Konto k(0,0);
   Girokonto gk(1,0,10000);
   Sparkonto sk(2,0,3);
   k=gk;
   k.einzahlen(1000);
   k.auszahlen(2000);
   k.ueberweisen(2000, sk);
   k.druckeAuszug();
   // Teil 2: mit Zeigern
   Konto* pK[3]={ 0 };

   pK[0] = new Girokonto(1,0,10000);
   pK[1] = new Sparkonto(2,0,3);
   pK[2] = new Konto(3,0);
   pK[0]->einzahlen(1000);
   pK[0]->auszahlen(2000);
   pk[0]->ueberweisen(2000, sk);
   // Alle Kontoauszüge drucken
   for(int i=0; i<3;i++)
      pK[i]->druckeAuszug();
```

→ statisches Binden → kein polymorphes Verhalten!

early binding

// Konto::einzahlen
// Konto::auszahlen, virtual Konto::darfAuszahlen() ☹
// virtual Konto::darfAuszahlen, Konto::auszahlen ☹, Konto::einzahlen
// Konto::druckeAuszug(): ohne `Kreditlimit`
// Konto::druckeAuszug(): weiss nicht von `Kreditlimit` obwohl virtuell ☹
→ dynamisches Binden → polymorphes Verhalten!

// Konto::einzahlen
// Konto::auszahlen (nicht virtuell), virtual Girokonto::darfAuszahlen

```
KtoNr:      1
  Stand:      -1000
  Kreditlimit: 10000
KtoNr:      2
  Stand:      0
  Habenzins:   3 %
KtoNr:      3
  Stand:      0
```

// virtual Girokonto::druckeAuszug()

// virtual Sparkonto::druckeAuszug()

// virtual Konto::druckeAuszug()

Es werden auch dann die richtigen Auszüge gedruckt, wenn später neue Klassen abgeleitet werden, ohne die `for`-Anweisung zu ändern!

Beispiel Konto mit virtuellen Funktionen


```
// Fortsetzung main()
pK[0]->ueberweisen(2000,*pK[1]);
pK[1]->kuendigeBetrag();
```

// Fehler! Fkt. `ueberweisen` nicht definiert in Basisklasse

// error C2039: '`kuendigeBetrag`' : Ist kein Element von '`Konto`'

Beachte: Aufrufer hat den Typ `Konto*`. Ihm steht daher auch nur die SS von `Konto` zur Verfügung: `überweisen` gehört nicht dazu.
Auch wenn ein Basisklassen-Zeiger auf ein Objekt einer abgeleiteten Klasse verweist, kann er nur die Basisklassen-SS nutzen, d.h. mit ihm können nur solche Methoden aufgerufen werden, die die Basisklasse 'versteht'.

Das gleiche gilt für `kuendigeBetrag`.

Polymorphie beginnt in der Basisklasse.

Beispiel Konto mit virtuellen Funktionen


```
class Konto {
public: /* … */
  virtual void ueberweisen(double betrag, Konto& zielKto);
  virtual void kuendigeBetrag();
}
```

In der Implementation darf `virtual` nicht verwendet werden!

```
void Konto::ueberweisen(double betrag, Konto& zielKto) {
   this->auszahlen(betrag);
   zielKto.einzahlen(betrag);
}
```

darfAuszahlen wird auch dann spät gebunden, wenn überweisen nicht virtual ist

Der Bindungstyp der aufgerufenen Methode ist maßgebend für den Compiler, nicht der Bindungstyp der aufrufenden Methode.

Diskussion:

- Muss `Konto::ueberweisen` virtuell sein? Nur wenn es in abgeleiteten Klassen redefiniert werden soll
- Welche weiteren Methoden müssen polymorphes Verhalten zeigen und müssen daher in der Basisklasse deklariert werden? `Sparkonto::kuendigeBetrag()`

```
void Konto::kuendigeBetrag() {
  throw("ERROR: Konto::kuendigeBetrag für Konto-Objekt nicht definiert!");
}
```

Vorläufige Lösung!

`Sparkonto::kuendigeBetrag()` bleibt unverändert, sollte allerdings durch vorangestelltes 'virtual' in der Deklaration sein nunmehr polymorphes Verhalten dokumentieren.

Beispiel Konto mit virtuellen Funktionen


```
  // Fortsetzung main()

  pK[0]->ueberweisen(2000, *pK[1]);

  sk.kuendigeBetrag();

  pK[1]->kuendigeBetrag();
}
```

jetzt ok!

Zur Erinnerung: In main steht auch

```
Sparkonto sk(2,0,3);
pK[1] = new Sparkonto(2,0,3);
```

Statisches Binden an `Sparkonto::kuendigeBetrag` weil Aufruf durch Objekt. Indirekter Aufruf über VMT nur mit Zeiger oder Referenz!

Dynamisches Binden an `kuendigeBetrag`, weil virtual und Aufruf über Zeiger, d.h. indirekter Aufruf der passenden Methode über VMP-Array!

Abstrakte Klassen

- Klassen, von denen Objekte erzeugt werden können, nennt man **konkrete Klassen**.
- In vielen Fällen **sollte** die **Basisklasse** einer Hierarchie **sehr allgemein sein** und Code enthalten, der aller Voraussicht nach nicht verändert werden muß.
- Oft ist es **nicht** notwendig oder erwünscht, dass Objekte dieser Klassen angelegt werden.
- Klassen, von denen keine Objekte angelegt werden können, nennt man **abstrakte Klassen**. Diese abstrakten Klassen dienen ausschliesslich als Basisklassen. Objekte werden ausschliesslich von den abgeleiteten Klassen erzeugt, die dann jeweils ein Subobjekt vom Typ der abstrakten Basisklasse enthalten.
- In C++ werden Klassen automatisch zu abstrakten Klassen, wenn sie mindestens eine **rein virtuelle Methode** enthalten. Diese haben **keinen Implementationsteil**. Die Deklaration rein virtueller Methoden wird um ein '**=0**' ergänzt. Von diesen rein abstrakten Klassen können keine Objekte erzeugt werden.

```
virtual int reinVirtuelleMethode(int) = 0;
```

- Von abstrakten Klassen abgeleitete Klassen erben die rein virtuellen Methoden und sind zunächst selbst abstrakt. Sie werden konkrete Klassen, indem die rein virtuellen Methoden einen Implementationsteil erhalten. Daraus folgt:
  - **Die Basisklasse (auch abstrakte) definieren eine öffentliche Schnittstelle**, die an alle abgeleiteten Klassen vererbt wird (→ **Schnittstellenlieferant**).
  - Abstrakte Basisklassen zwingen den Systemprogrammierer zur Implementation der rein virtuellen Methoden (→ **Schnittstellengarant**).

Abstrakte Klasse und Beispiel Konto

Alle Konten sind entweder Girokonten, Sparkonten, … . Konto Objekte werden nicht benötigt. Basisklasse Konto dient der Implementationsvererbung und der Schnittstellenvererbung

```
class Konto {
  int nummer;
  double stand;
public:
   Konto(int nr=0, double anfangsStand=0.)
  {  nummer = nr; stand = anfangsStand; }

  virtual ~Konto()

  virtual void kuendigeBetrag() =0;
  virtual bool darfAuszahlen(double betrag)const =0;

  virtual void auszahlen(double betrag);

  virtual void einzahlen(double betrag)  {
    assert(betrag >= 0.);
    stand += betrag;
  }

  virtual void ueberweisen(double betrag, Konto& zielKto);

  virtual void druckeAuszug() {
    cout << "Kontonummer: " << nummer << endl;
    cout << "Kontostand:  " << stand  << endl;
  }
  double gibStand () const { return stand; }
protected:
  void setStand( double neuerStand ) { stand = neuerStand; }
};
```

Hier keine sinnvolle Implementation möglich

würde nur für Kontoobjekte benötigt, die aber nicht erzeugt werden können, weil abstrakte Klasse

Redefinition in Sparkonto (Status `istGekuendigt` verwalten)

Wahrscheinlich kein Änderungsbedarf in abgeleiteten Klassen

gibt Subobjekt-Attribute aus

Zugriff auf Subobjekt-Attribute

Abstrakte Klasse und Beispiel Konto


```
class Girokonto
{ /* … */
public:
  virtual ~Girokonto();
  virtual void kuendigeBetrag()
  { cout << "Girokonto::kuendigeBetrag() nicht erforderlich!\n"; }
  virtual bool darfAuszahlen(double betrag);
};
```

Girokonto muß `kuendigeBetrag` implementieren, um eine konkrete Klasse zu werden.

Girokonto muß `darfAuszahlen` implementieren, um eine konkrete Klasse zu werden.

`einzahlen`, `auszahlen`, `ueberweisen`, `setStand`, `gibStand` werden von `Konto` geerbt (Schnittstellen- und Implementationsvererbung)

Abstrakte Klasse und Beispiel Konto


```
class Sparkonto : public Konto
{
  double habenZins;
  bool istGekuendigt;
public:
  virtual ~Sparkonto();
  virtual bool darfAuszahlen(double betrag);
  virtual void auszahlen(double betrag);
  virtual void kuendigeBetrag();
}
```

einzahlen, ueberweisen, setStand, gibStand werden von Konto geerbt (Schnittstellen- und Implementationsvererbung)

Sparkonto muß darfAuszahlen implementieren, um eine konkrete Klasse zu werden.

```
void Sparkonto::auszahlen( double betrag ) {
  Konto::auszahlen( betrag );
  istGekuendigt = false;
}
```

hier ist Sparkonto spezieller

```
bool Sparkonto::darfAuszahlen( double betrag ) const
// gibt true zurück, wenn gekuendigt und Guthaben ausreicht
{
   if(istGekuendigt)
      return getStand() >= betrag;
   else
      return false;
}
```

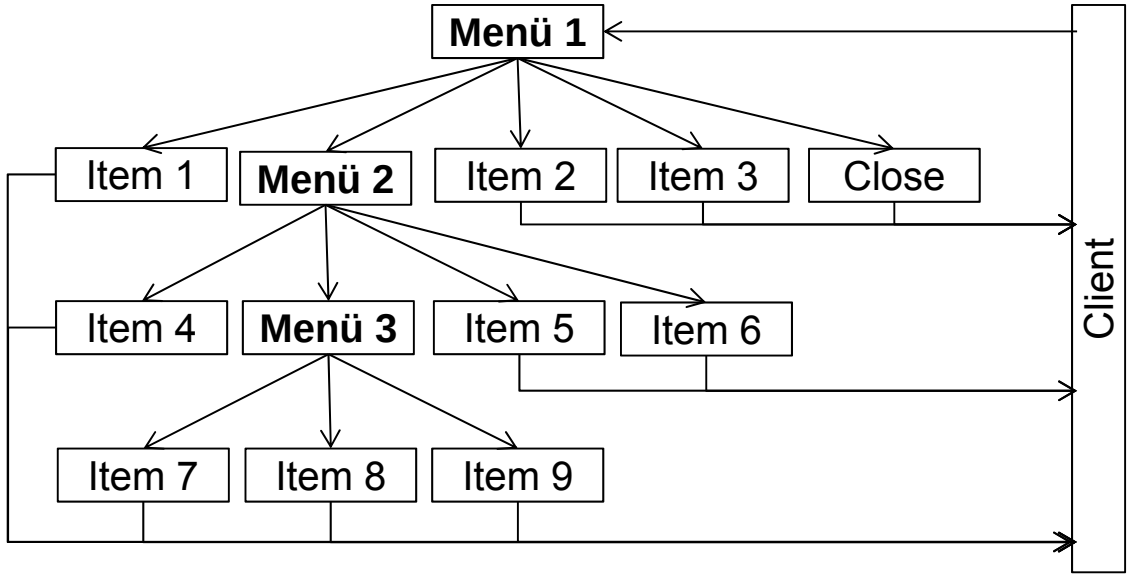
Nichts Neues!

Zusammenfassung: Regeln für virtuelle Methoden (spätes Binden)

1. Spätes Binden funktioniert nur in Verbindung mit Objekt-Referenzen und -Zeigern.
2. Alle Methoden, die in abgeleiteten Klassen virtuell sein sollen, müssen in der Basisklassen-Deklaration als `'virtual'` gekennzeichnet werden.
3. In der Implementation darf das Schlüsselwort `'virtual'` nicht verwendet werden.
4. Redefinierte virtuelle Methoden sind in den abgeleiteten Klassen automatisch virtuell (auch ohne das Schlüsselwort `'virtual'`)
5. Basisklassen enthalten oft virtuelle Methodendeklarationen, die erst in abgeleiteten Klassen benötigt werden, bzw. erst in diesen eine sinnvolle Verwendung finden. Um für solche Methoden keine Implementations-Attrappe vorsehen zu müssen, kann der Methodenrumpf `'{...}'` auch durch `'=0'` ersetzt werden. Dadurch wird diese Methode eine sogenannte **'rein virtuelle'** Methode.
6. Klassen mit mindestens einer rein virtuellen Methode nennt man "**Abstrakte Klasse**". Nicht abstrakte Klassen nennt man dagegen "**Konkrete Klasse**". Von abstrakten Klassen kann man keine Objekte erzeugen. Sie haben vor allem die Aufgabe, ihre Schnittstelle zu vererben.
7. Abgeleitete Klassen müssen rein virtuelle Methoden implementieren, sonst sind sie ebenfalls abstrakt. Dieser Mechanismus erzwingt, dass der Nutzer eine durch die rein virtuellen Methoden definierte Minimalfunktionalität realisiert.
8. **Konstruktoren** können nicht als virtual deklariert werden. **Virtuelle Destruktoren** sollten immer dann verwendet werden, wenn von der betreffenden Klasse aktuell oder evtl. zukünftig abgeleitet wird.
9. `friend`-Beziehungen werden nicht vererbt.
10. Vermeiden Sie `protected`-Attribute.

Fallstudie „Menü-Framework“

- Anforderungen an das Menüsystem
  - Besteht aus Menüs (`Menue`) und deren Menüeinträgen (`MenuItem`)
  - Mit jedem Menüeintrag ist eine Operation (`onChoice`) verbunden, die ausgeführt wird, wenn der Eintrag vom Benutzer ausgewählt wird
  - Hierarchische Menüstrukturen mit Untermenüs sollen möglich sein



Gewünschter Kontrollfluss des Menüsystems

Menükopf

Menüeinträge

Benutzeraufforderung

```
*** Menü 1 ***
1…Item 1
2…Menü 2
3…Item 2
4…Item 3
5…Close
Your choice:
```

Benutzerdialog (Beispiel)

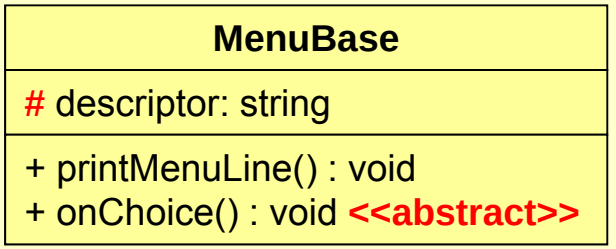
Workflow:

1. Menükopf anzeigen
2. Menüeinträge anzeigen
3. Benutzeraufforderung anzeigen
4. Benutzerauswahl entgegen nehmen
5. Benutzerauftrag ausführen
6. Weiter bei 1.

Abstrakte Basisklasse MenuBase


```
class MenuBase
{
protected:
  std::string descriptor;
public:

  MenuBase( const std::string& descriptor ) {
    this->descriptor = descriptor;
  }

  virtual ~MenuBase() { }

  virtual void printMenuLine() const {
    // Gibt den ShortCut und die Aufgabenbezeichnung auf dem Bildschirm aus
    std::cout << std::left << std::setw(5) << descriptor << std::endl;
  }

  virtual void onChoice() =0;
    // Definiert die zu erledigende Aufgabe und führt sie aus

}; // END class MenuBase
```

Generalisierung: Alle Menüeinträge
- haben eine Aufgabenbeschreibung
- müssen diese ausgeben können
- haben, falls ausgewählt individuelle Aufgaben zu erfüllen



Anwendungsbezogene Menüeinträge (Beispiele)


```
class MenuItem_Dies : public MenuBase
{
public:
  MenuItem_Dies( const std::string& descriptor ) : MenuBase(descriptor) { }

  virtual void onChoice() {
    // Definiert die zu erledigende Aufgabe und führt sie aus
    // ToDo: Direkte oder indirekte (Delegation) Aufgabenerledigung
    std::cout << "Jetzt tue ich dies!" << std::endl;
  }
};

class MenuItem_Das : public MenuBase
{
public:
  MenuItem_Das( const std::string& descriptor ) : MenuBase(descriptor) { }

  virtual void onChoice() {
    // Definiert die zu erledigende Aufgabe und führt sie aus
    // ToDo: Direkte oder indirekte (Delegation) Aufgabenerledigung
    std::cout << "Jetzt tue ich das!" << std::endl;
  }
};
```

Jetzt fehlt noch ein `Menue`-Objekt, das mehrere Menüeinträge verwalten kann.

Der Menüverwalter class Menu


```
class Menu : public MenuBase
{
protected:
  std::vector< MenuBase*> pitems;
public:
  Menu( const std::string& descriptor ) : MenuBase(descriptor) { }

  void Menu::add( MenuBase* pmi )
  {
    if(pmi)
      pitems.push_back(pmi); // Kompositionsbeziehung
    else
      ; // Fehlerbehandlung
  }

  ~Menu()
  {
    for ( size_t i=0; i<pitems.size(); ++i  )
      delete pitems[i];  // Kompositionsbeziehung
  }

private:
  void Menu::printMenuHeader() const
  {
    std::cout << std::string(5,'*') << ' '<< descriptor << std::string(5,'*') << std::endl;
  }
```

MenuBase
descriptor: string
+ printMenuLine() : void + onChoice() : void <<abstract>>

1. Menükopf anzeigen

Der Menüverwalter class Menu


```
private:
  void Menu::printItems() const {
    for ( size_t i=0; i<pitems.size(); ++i ) {
      std::cout << i << '\t';
      pitems[i]->printMenuLine();
    }
  }

  size_t Menu::queryUser( const std::string& prompt ) {
    std::cout << prompt;
    size_t input;
    std::cin >> input;
    return input;
  }

  virtual void Menu::onChoice()
  {
    size_t input;
    while(true) {
      this->printMenuHeader();
      this->printItems();
      input = this->queryUser("Bitte wählen: ");
      if ( input>=0 && input<pitems.size() ) {
        pitems[input]->onChoice();
        break;
      } else
        std::cout << "Unzulaessige Eingabe!" << std::endl;
    }
  }
```

2. Menüeinträge anzeigen

Gibt alle zulässigen Auswahlmöglichkeiten zusammen mit ihrem (automat. gewählten) numerischen shortcut i auf der Konsole aus

3. Benutzeraufforderung anzeigen

4. Benutzerauswahl entgegen nehmen

1. Menükopf anzeigen
2. Menüeinträge anzeigen
3. Benutzeraufforderung anzeigen
4. Benutzerauswahl entgegen nehmen
5. Benutzerauftrag ausführen
6. Weiter bei 1.

Das Stammmenü: Exit und Cancel


```
class MenuItem_Cancel : public MenuBase
{
public:
  MenuItem_Cancel() : MenuBase("Zurück") { }
  virtual void onChoice() { ; }
};

class MenuItem_Close : public MenuBase
{
public:
  MenuItem_Close() : MenuBase("Schließen") { }
  virtual void onChoice() { throw ExceptionClose( "Anwendung wurde vom  Benutzer beendet!" ); }
};
```

```
class ExceptionClose : public std::exception {
public:
 ExceptionClose( const char* msg ) : std::exception(msg) {}
};
```

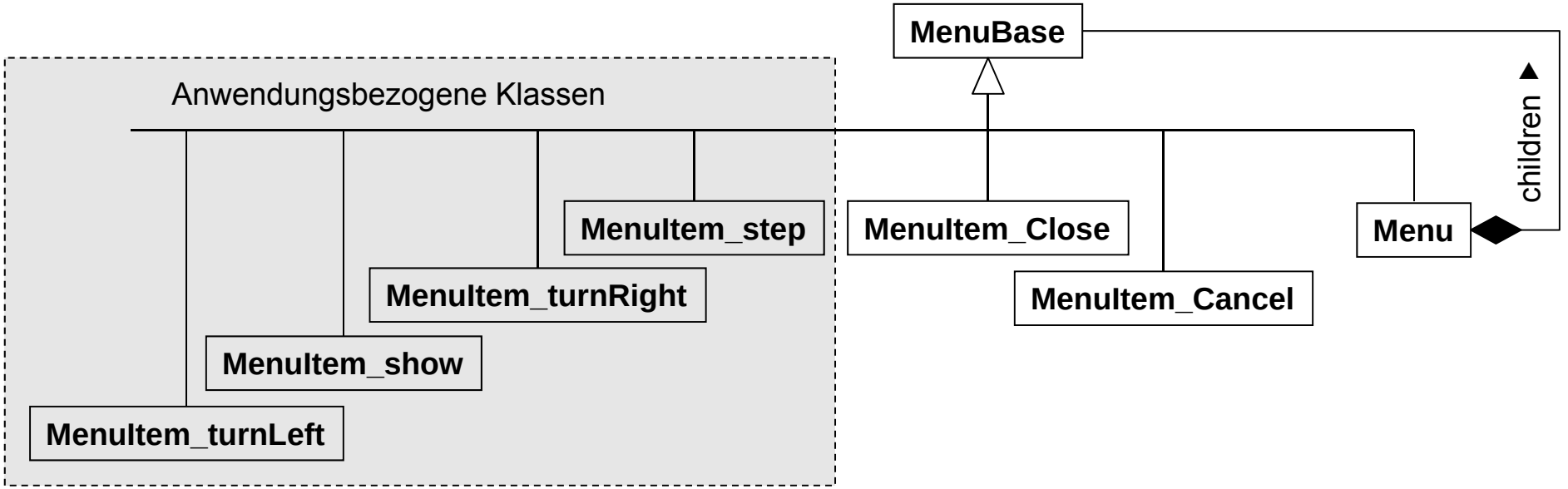
```
#include <exception>
```

Beispiel: Anwendungssystem Teil 1


```
enum Direction {N,E,S,W};

class FC
{
  Direction dir;
  int x,y;
public:
  FC(void) { x=y=0; dir=S; }

  ~FC(void) { };

  void show()
  {
    std::cout << '(' << x << ',' << y << ',';
    switch(dir) {
    case N: std::cout << 'N'; break;
    case E: std::cout << 'E'; break;
    case S: std::cout << 'S'; break;
    case W: std::cout << 'W'; break;
    }
    std::cout << ')' << std::endl;
  }
```

```
  void step() {
    switch(dir) {
    case N: --y; break;
    case E: ++x; break;
    case S: ++y; break;
    case W: --x; break;
    }
  }

  void turnRight() {
    switch(dir) {
    case N: dir=E; break;
    case E: dir=S; break;
    case S: dir=W; break;
    case W: dir=N; break;
    }
  }

  void turnLeft() {
    switch(dir) {
    case N: dir=E; break;
    case E: dir=S; break;
    case S: dir=W; break;
    case W: dir=N; break;
    }
  }
};
```

Anwendungsbezogene Menüerweiterung




```
class MenuItem_step : public MenuBase
{
  FC* pfc;
public:
  MenuItem_step( const std::string& descriptor, FC* pfc0 ) : MenuBase(descriptor), pfc(pfc0)  { }
  virtual void onChoice() { pfc->step(); }
};
```

```
class MenuItem_turnRight { /* Analog zu MenuItem_step */ };

class MenuItem_turnLeft { /* Analog zu MenuItem_step */ };

class MenuItem_show { /* Analog zu MenuItem_step */ };
```

Beispiel: Anwendungssystem Teil 2


```
int main()
{
  FC* pfc = new FC();
  // Menü konfigurieren
  MenuBase* pMenuTest = new Menu( "Unit Tests" );
  pMenuTest->add( new MenuItem_Cancel() );
  MenuBase* pMainMenu = new Menu("FutureCar");
  pMainMenu->add( pMenuTest );
  pMainMenu->add( new MenuItem_step( "Drive 1 step", pfc ) );
  pMainMenu->add( new MenuItem_turnRight( "Turn Right", pfc ) );
  pMainMenu->add( new MenuItem_turnLeft( "Turn Left", pfc ) );
  pMainMenu->add( new MenuItem_show( "Show", pfc ) );
  pMainMenu->add( new MenuItem_clearScreen( "Clear Screen" ) );
  pMainMenu->add( new MenuItem_Close() );

  try
  {
    do
    {
      pMainMenu->onChoice();
      cout << endl;
    } while(true);
  }
  catch ( const ExceptionClose& e )
  {
    cout << e.what() << endl;
  }
  return 0;
}
```

MenuBase
descriptor: string
+ printMenuLine() : void + onChoice() : void <<abstract>> + add(pmi : MenuBase*) : void

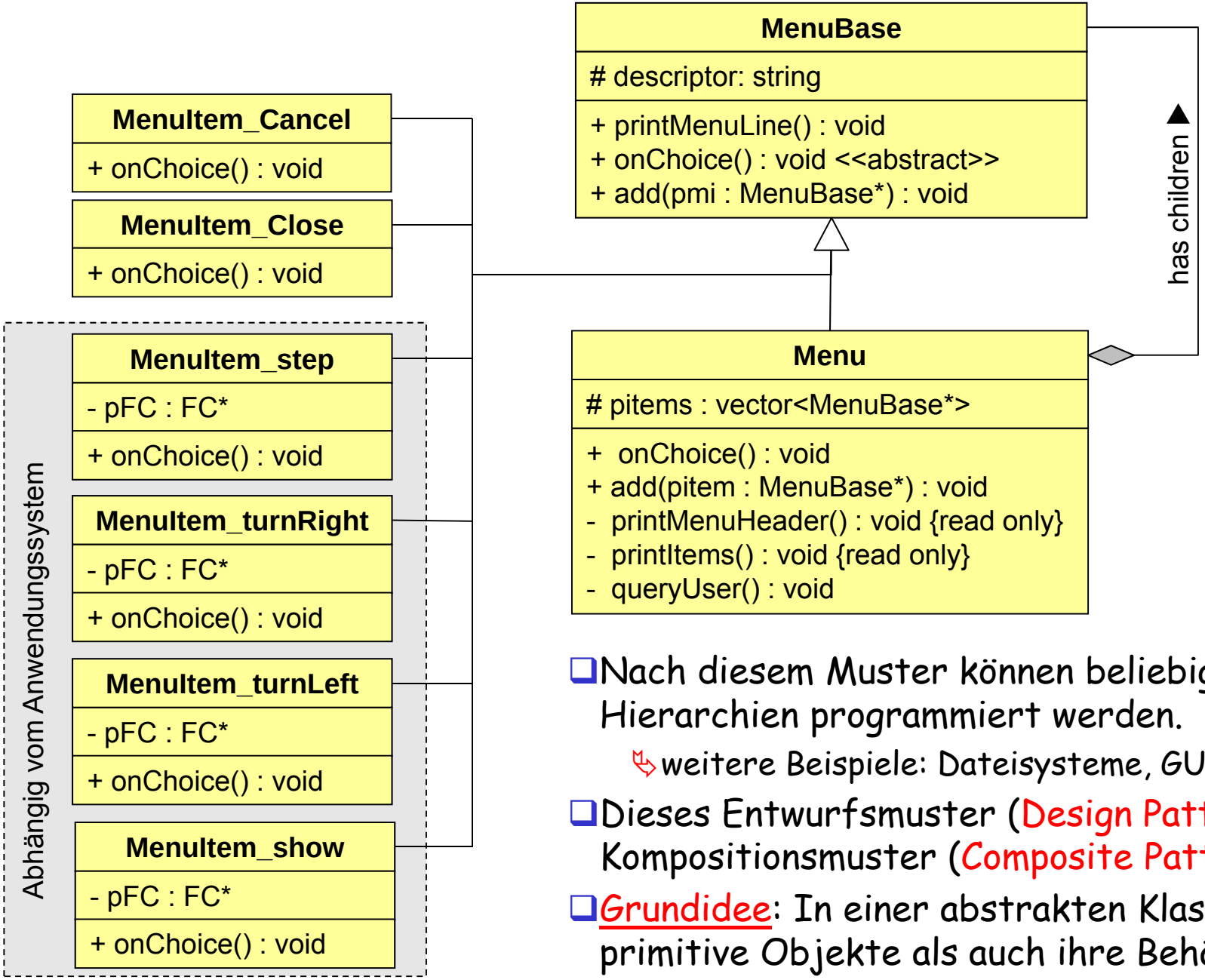
Compiler: error C2039: 'add': Ist kein Element von 'MenuBase'

Der Zeiger `pMenuTest` ist vom Typ `MenuBase*`.
Ihm steht nur die `MenuBase`-SS zur Verfügung.
`MenuBase` muss Methode `add` deklarieren.
Dummy-Implementierung ist zweckmäßig, weil sonst alle abgeleiteten Klassen eine Dummy-Implementierung bräuchten (bis auf `class Menu`)

```
virtual void MenuBase::add( MenuBase* pmi ) {
  throw ExceptionUnsupportedOperation("ERROR in MenuBase::add!");
}
```

Menüsystem als Beispiel einer Teile-Ganzes-Hierarchie

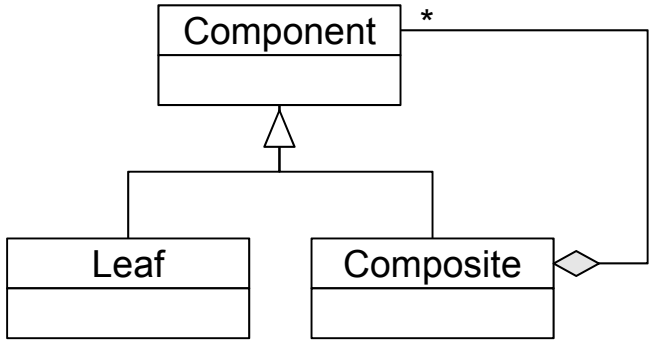
h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
FB Informatik
Prof. Dr. R.Nitsch



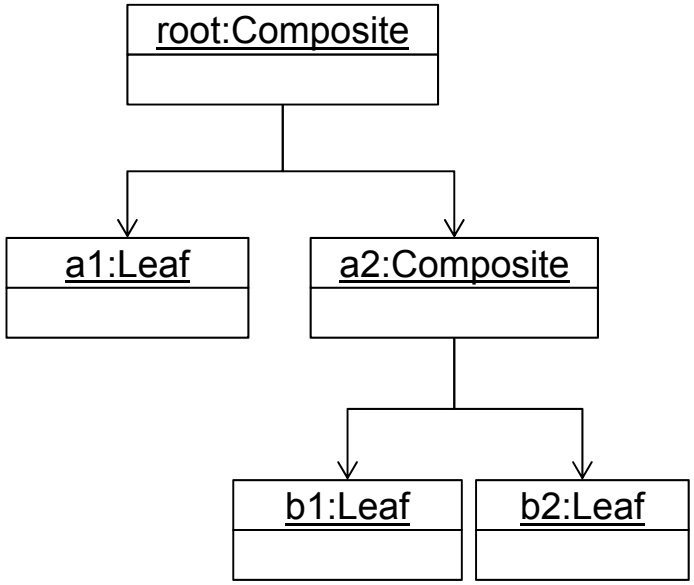
- Nach diesem Muster können beliebige Teile-Ganzes-Hierarchien programmiert werden.
  - weitere Beispiele: Dateisysteme, GUIs, …
- Dieses Entwurfsmuster (Design Pattern) wird Kompositionsmuster (Composite Pattern) genannt.
- Grundidee: In einer abstrakten Klasse werden sowohl primitive Objekte als auch ihre Behälter zusammengefasst.

Design-Pattern "COMPOSITE" - Beteiligte

- Composite = Zusammensetzung, Verbund
- Component
  - implementiert das Standardverhalten aller abgeleiteten Klassen (`printMenuLine`)
  - deklariert die Schnittstelle der Composite-Objekte (`printMenuTitle`, `onChoice`)
  - deklariert die Schnittstelle zum Verwalten der Kinder (`add`)
- Leaf
  - Repräsentiert Blatt-Objekte im COMPOSITE Pattern. Ein Blatt hat keine Kinder.
  - definiert das Verhalten der Nicht-Composite-Objekte (`onChoice`)
- Composite
  - Definiert das Verhalten der Komponenten mit Kindern (`onChoice`, `printItems`, `queryUser`)
  - speichert Kind-Komponenten (`vector<MenuBase*>`)
  - implementiert Kind-bezogene Operationen der Component-Schnittstelle (`add`)
- Client
  - Manipuliert Objekte der Zusammensetzung mit Hilfe der `Component`-Schnittstelle



Klassendiagramm



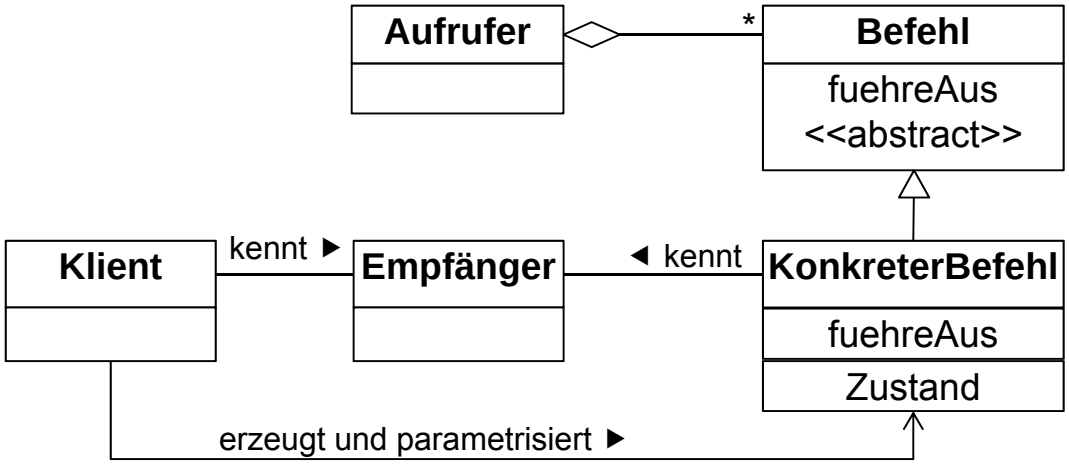
Objektdiagramm

Nachteile der bisherigen Implementierung eines Menüsystems

- Für jede neue Anwendung muss die Menü-Klassenhierarchie modifiziert werden, d.h. Anwendung und Menüsystem sind voneinander abhängig.
- Deshalb ist das bisherige Menüsystem auch kein Framework!
- Konsequenz:
  - Das COMPOSITE Pattern alleine reicht nicht aus, um ein Menüframework zu gestalten.
- Lösung:
  - Die zusätzliche Verwendung des COMMAND Pattern löst die verbliebene Abhängigkeit auf und ermöglicht das gewünschte Framework.

Design Pattern "COMMAND"

Grundidee: Ein Command-Objekt kapselt einen Befehl



Akteure

- *Befehl:* Basisklasse aller Kommandos.
- *Konkreter Befehl:* implementiert die Befehlsschnittstelle (`fuehreAus`). Kennt den/die Befehlsempfänger und hat weitere Informationen zur Befehlsausführung (Zustand).
- *Klient:* erzeugt einen *konkreten Befehl* und versieht ihn mit einem Verweis auf den *Empfänger* und allen anderen nötigen Informationen. Er gibt dem *Aufrufer* eine Referenz auf den *konkreten Befehl*.
- *Aufrufer:* besitzt einen oder mehrere Verweise auf *Befehle* und fordert diese bei Bedarf auf, ihre Aktion auszuführen.
- *Empfänger:* keine besonderen Anforderungen. Muss nichts über die anderen Akteure wissen. Somit kann jede Klasse als *Empfänger* dienen. Der *konkrete Befehl* kann Methoden des *Empfängerobjektes* aufrufen, um seine Aktion auszuführen.

Bewertung und Anwendungen

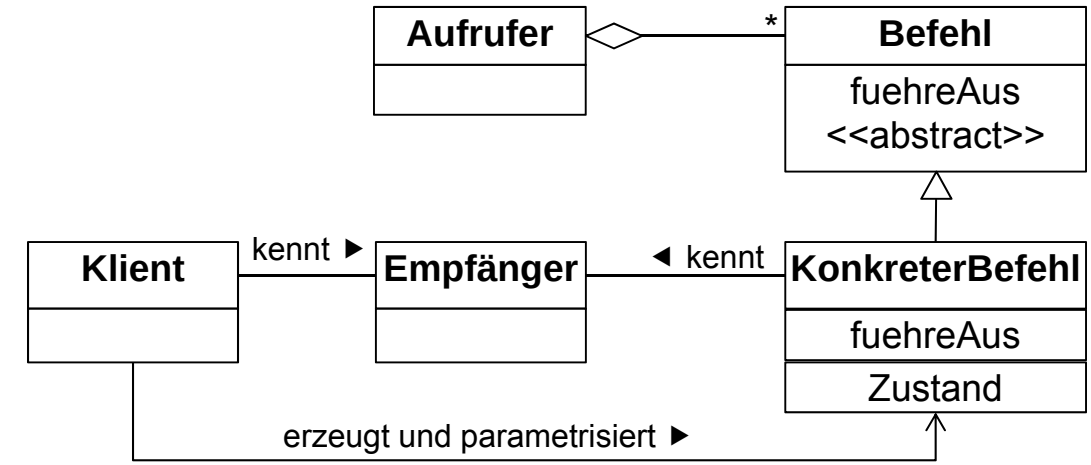
Vorteile

- Befehle können parametrisiert werden
  - Verweis auf Befehlsempfänger und auszuführende Operation
- Befehle können in Warteschlangen gespeichert werden
  - Befehlserstellung und Ausführung zu verschiedenen Zeiten oder in unterschiedlichen Kontexten
  - Befehle können protokolliert werden (Logdatei)
  - Befehle können rückgängig gemacht werden (Undo, Redo)
- Befehlsgeber und -Ausführende sind entkoppelt
  - Vorausetzung für Bibliotheken und Frameworks

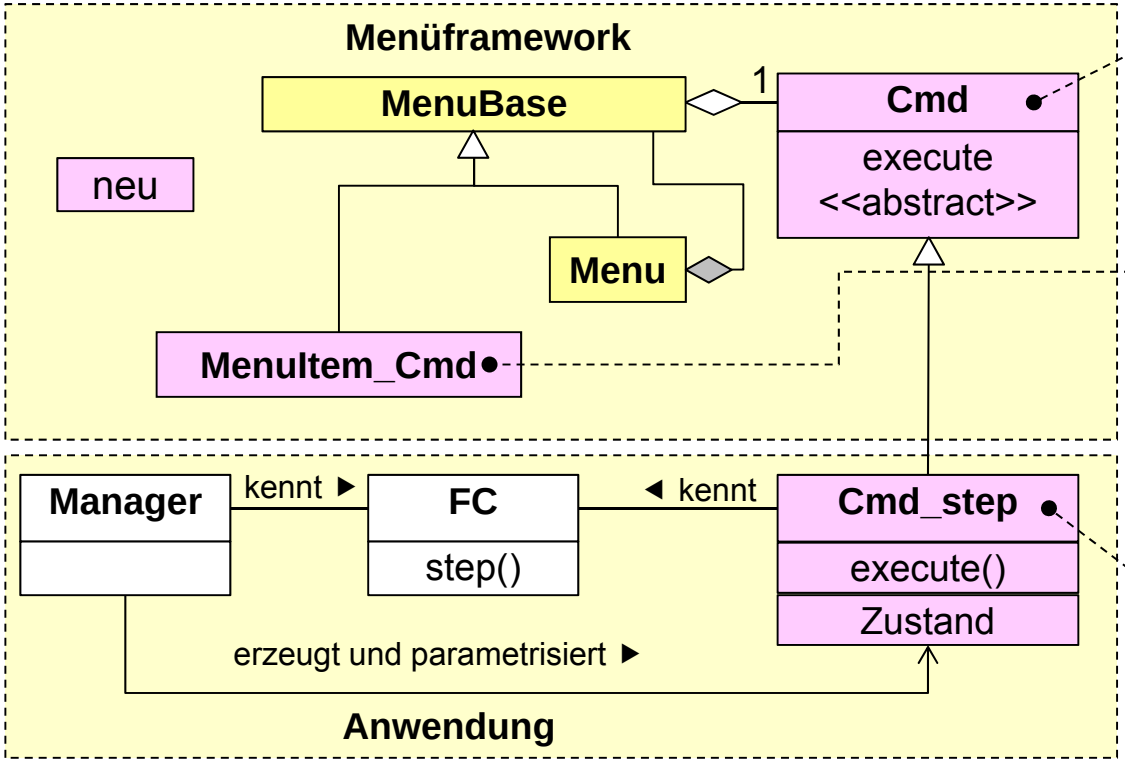
Nachteile

- Eine Klasse pro Kommando
  - Evtl. unübersichtliche Implementierung

Weg zum Menüframework



Command Pattern




```
class Cmd { public: virtual void execute() =0; };
```

```
class MenuItem_Cmd {
  Cmd* pcmd;
public:
  MenuItem_Cmd(const std::string& descriptor,Cmd* pcmd0)
     : MenuBase(descriptor), pcmd(pcmd0) { }
  virtual void onChoice() { pcmd->execute(); }
};
```

```
class Cmd_step {
  FC* pfc;
public:
  Cmd_step( FC* pfc0 ): pfc(pfc0) { }
  virtual void execute() { pfc->step(); }
};
```

Anwendung des Menuframeworks


```
#include "Menu.h"
#include "Cmd.h"

int main() {
  FC* pfc = new FC();
  // Menüs konfigurieren
  MenuBase* pMenuTest = new Menu( "Unit Tests" );
  pMenuTest->add( new MenuItem_Cancel() );
  MenuBase* pMainMenu = new Menu("FutureCar");
  pMainMenu->add( pMenuTest );
 pMainMenu->add( new MenuItem_Cmd( "Drive 1 step", new Cmd_step( pfc ) ) );
 pMainMenu->add( new MenuItem_Cmd( "show", new Cmd_show( pfc ) ) );
 pMainMenu->add( …
  // … Rest wie bisher
}
```