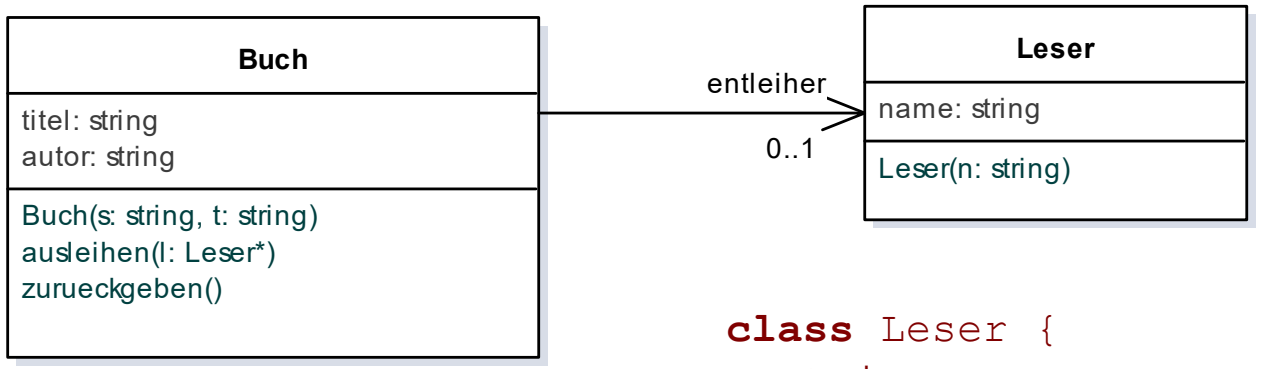
Assoziationen

- Implementierung von Assoziationen mit Zeigern
  - Realisierung jeder Richtung einer Assoziation mittels Zeigern zwischen Objekten
    - Die Multiplizität und die Navigierbarkeit müssen berücksichtigt werden
    - Jedes Objekt hat über einen Zeiger Zugriff auf die mit ihm verknüpften Objekte
  - Konsistentes Auf- und Abbauen der Verbindungen durch die Operationen muss sichergestellt werden
  - Multiplizität von 0..1 oder 1 (→ *Beispiel*)
    - Ein Objekt enthält ***einen*** Zeiger auf das verknüpfte Objekt (ggfs. NULL)
  - Multiplizität größer 1: Menge von Zeigern
    - Ein Objekt enthält ***eine Menge*** von Zeigern auf die verknüpften Objekte
    - Realisierung der Menge durch
      - *Container*-Klassen (s.u.)
        - » Bei Eigenschaftswerten {ordered} oder {sequence}: Verwendung von Container-Klassen, die die Ordnung ihrer Elemente ermöglicht
        - » z.B. Container-Klassen der C++-Standard-Bibliothek: `list, set, vector`
      - Felder (Arrays), insbesondere bei fester Obergrenze (→ *Türme v. Hanoi*)

Assoziationen

- Implementierung von Assoziationen mit Multiplizität <= 1
  - *Beispiel:*




```
class Buch {
  string titel;
  string autor;
  Leser * entleiher;
public:
  Buch(string t,string a);
  ausleihen(Leser * pL);
  zurückgeben();
};

Buch::Buch(string t, string a)
{  titel = t; autor = a;
   entleiher = NULL;
}
```

```
class Leser {
  string name;
public:
  Leser(string n)
  { name = n; }
};


// Link-Operation
Buch::ausleihen(Leser * pL)
{  entleiher = pL; }

// Unlink-Operation
Buch::zurueckgeben()
{ entleiher = NULL; }
```

Container-Klassen

- **Generische Container-Klassen**:
  - Klassen, deren Objekte eine **Mengen von Elementen eines bestimmten Typs** repräsentieren
  - Der Typ der Elemente ist Parameter der Container-Klasse
  - Container-Klassen der **C++-Standardbibliothek**:
    - `set<T>:`
      - Mengen von Elementen des Typs `T` ohne Duplikate
      - Reihenfolge der Elemente unabhängig vom Einfügen
    - `list<T>:`
      - Mengen von Elementen des Typs `T`, Duplikate möglich
      - Einfügen/Löschen an beliebigen Positionen möglich
      - Durchlaufen der Liste vorwärts und rückwärts möglich
      - Entspricht einer Doppelt verketteten Liste
    - `vector<T>`
      - Mengen von Elementen des Typs `T`, Duplikate möglich
      - Einfügen/Löschen an beliebigen Positionen möglich
      - Indizierter Zugriff möglich (`v[i] = ...`)
      - Entspricht einem dynamisch wachsenden Array

Container-Klassen

- *Beispiel:* die Klasse `list<T>`
  - `list<T>::list()`

    Konstruktor; initialisiert eine leere Liste.
  - `list<T>::~list()`

    Destruktor; gibt den Speicherplatz für die Listenelemente frei.
    - Vorsicht: wenn die Listenelemente Pointer auf Objekte sind, müssen diese Objekte explizit mit `delete` freigegeben werden
  - `void list<T>::push_back (const T& val)`
  - `void list<T>::push_front (const T& val)`

    Fügt ein neues Element `val` am Ende bzw. am Anfang der Liste ein
  - `void list<T>::pop_back ()`
  - `void list<T>::pop_front ()`

    Entfernt das Element am Ende bzw. am Anfang der Liste

Container-Klassen

- **Iteratoren**
  - Auf die Elemente von Container-Klassen kann mit Hilfe von ***Iteratoren*** (Werte des Typs `iterator`) zugegriffen werden
    - `iterator`-Werte repräsentieren Positionen im Container
    - Sind de facto Zeiger auf die Listenelemente
  - Eine Container-Klasse definiert ihren eigenen Iterator-Typ, z.B.:
    - `list<T>::iterator`
  - Methoden zum berechnen von Positionen, z.B.:
    - `iterator list<T>::begin()`

      Liefert die Position des ersten Elements der Liste
    - `iterator list<T>::end()`

      Liefert die Position *nach* dem letzten Element der Liste (zeigt *hinter* die Liste)
  - Inkrementieren eines `iterator`-Werts (`i++`) bewirkt, dass dieser anschließend auf das nächste Element zeigt
  - Dereferenzieren eines `iterator`-Werts (`*i`) ermöglicht Zugriff auf das Element

Container-Klassen

- **Iteratoren**
  - Typische Schleife zum iterieren durch einen Container
    - z.B. zur Ausgabe aller Elemente auf `cout`:

```
list<T> L;
list<T>::iterator i;
for (i = L.begin(); i != L.end(); i++) cout << *i;
```

Voraussetzung: "**<<**" ist für den Elementtyp `T` definiert.

  - **`iterator`**-bezogene Listenoperationen, z.B.:
    - **`iterator list<T>::insert(iterator pos, const T& val);`**

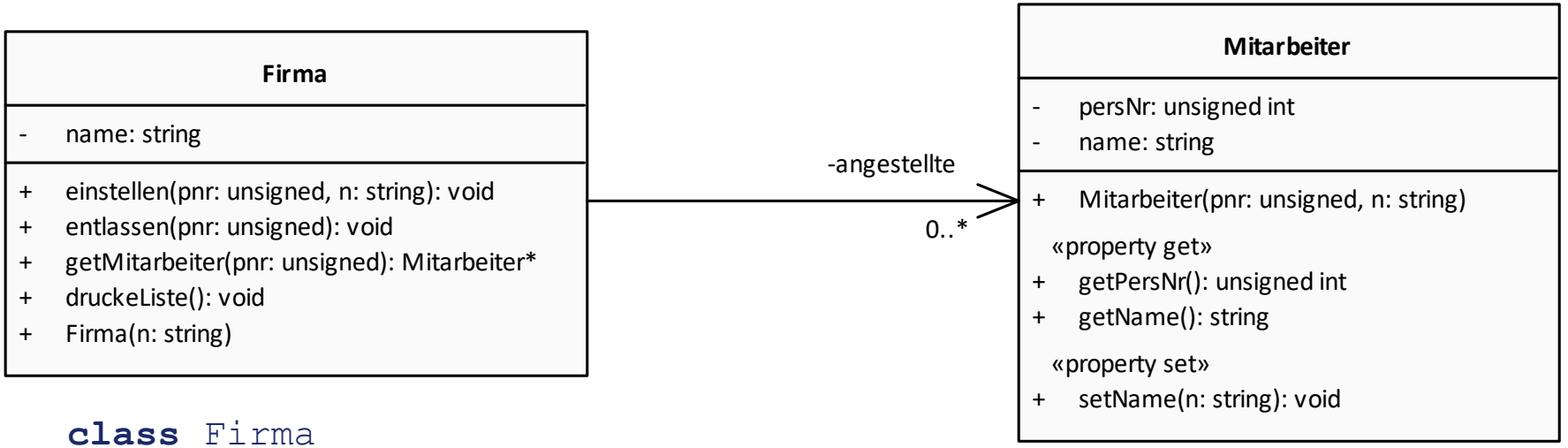
      Fügt ein neues Element **`val`** *vor* der Position **`pos`** ein;
      Resultat ist die Position des eingefügten neuen Elements
    - **`iterator list<T>::erase(iterator pos);`**

      Entfernt das Element an der Position **`pos`**;
      Resultat ist die Position *nach* dem entfernten Element.

- Weitere Details zu diesen und anderen Container-Klassen:
  `www.cplusplus.com/reference/stl/`

Assoziationen

- Beispiel:

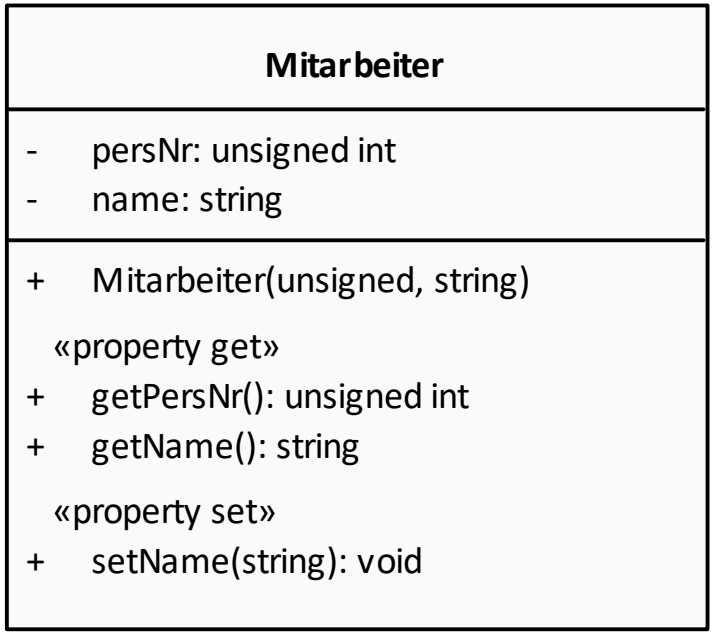



```
class Firma
{public:
  void einstellen(unsigned pnr, string n);
  void entlassen(unsigned pnr);
  Mitarbeiter* getMitarbeiter(unsigned pnr);
  void druckeListe();
  Firma(string n) // Konstruktor
private:
  string name;
  list<Mitarbeiter *> angestellte; // Implementierung der
                                   // Assoziation
};
```

Assoziationen

- Beispiel (Forts.): Implementierung der Klasse Mitarbeiter
  - Alle Methoden sind „inline“ definiert

```
class Mitarbeiter
{ public:
    void Mitarbeiter(unsigned pnr, string n)
    { persNr = pnr; name = n); }
    unsigned int getPersNr()
    { return persNr; }
    string getName();
    { return name; }
    void setName(string n)
    { name = n; }
  private:
    unsigned int persNr;
    string name;
};
```



Assoziationen

- Beispiel (Forts.): Implementierung der Klasse Firma

```
Firma::Firma(string n)
{ name = n; }

void Firma::einstellen(unsigned pnr, string n)
{  Mitarbeiter * ma = new Mitarbeiter(pnr,n);
   angestellte.push_back(ma);
}

Mitarbeiter * Firma::getMitarbeiter(unsigned pnr)
{  bool gefunden=false;
   list<Mitarbeiter *>::iterator i = angestellte.begin();
   while (!gefunden && (i!= angestellte.end()))
   { gefunden = (*i)->getPersNr() == pnr;
     if (!gefunden) i++;
   }
   if (gefunden) return (*i); else return 0;
}
```

Assoziationen

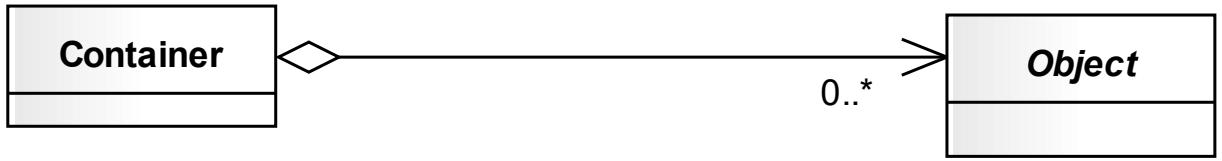
- Beispiel (Forts.): Implementierung der Klasse Firma

```
void Firma::entlassen(unsigned pnr)
{  bool gefunden=false;
   list<Mitarbeiter *>::iterator i = angestellte.begin();
   while (!gefunden && (i != angestellte.end()))
   { gefunden = (*i)->getPersNr() == pnr;
     if (!gefunden) i++;
   }
  if (gefunden) { delete *i; angestellte.erase(i);}
}

void Firma::druckeListe()
{ list<Mitarbeiter *>::iterator i;
  for(i = angestellte.begin();  i != angestellte.end(); i++)
   { cout << (*i)->getPersNr() << ": "
          << (*i)->getName() << endl;
   }
}
```

Container-Klassen

- Alternative Realisierung von Container-Klassen
  - Die Mengen enthalten **Zeiger** (Pointer, Referenzen) auf Objekte einer **abstrakten Basisklasse**
    - Java: Object
    - C++/MFC: CObject



  - Alle Objekte, die Spezialisierungen der Abstrakten Basisklasse sind, können in den Containern untergebracht werden

Container-Klassen

- *Beispiel:* MFC-Container-Klassen
  - MFC: Microsoft Foundation Classes
    - Umfangreiche Klassenbibliothek zur Realisierung von Windows-Anwendungen mit grafischer Benutzungsschnittstelle
    - Definiert die Abstrakte Klasse **`CObject`**
    - Definiert einige Container-Klassen, z.B.
    - **`CObList`**
      - Menge (Liste) von `CObject`-Pointern
      - Auf Listenelemente wird über eine „Position“ zugegriffen
        - » Datentyp **`POSITION`**
        - » Der numerische Wert einer Position ist nicht bekannt
        - » Speziell: Position am Anfang („head“) und Ende („tail“) der Liste
    - **`CObArray`**
      - Feld (Array) von `CObject`-Pointern
      - Auf die Listenelemente wird über Indizes [0..n-1] zugegriffen

Container-Klassen

- *Beispiel:* Methoden der MFC-Klasse **`CObList`** (u.a.)
  - **`CObList::CObList( );`**
    - Konstruiert eine "leere" Liste.
  - **`BOOL CObList::IsEmpty( );`**
    - Liefert *TRUE*, falls die Liste leer ist, sonst *FALSE*.
  - **`POSITION CObList::GetHeadPosition() ;`**
    `POSITION CObList::GetTailPosition() ;`
    - Liefert die Position des ersten (*GetHeadPosition*) bzw. des letzten (*GetTailPosition*) Elements der Liste, oder *NULL*, falls die Liste leer ist.
  - **`CObject * CObList::GetNext(POSITION& pos) ;`**
    - Liefert einen Zeiger auf das Element mit Position *pos*. Danach ist *pos* die Position des folgenden Elements (Referenzparameter!).
  - **`POSITION CObList::AddHead(CObject * E) ;`**
    `POSITION CObList::AddTail(CObject * E) ;`
    - Fügt den Zeiger *E* am Anfang der Liste (*AddHead*) bzw. am Ende der Liste (*AddTail*) ein. Ergebnis ist die Position des eingefügten Elements.
  - **`CObject * RemoveHead( );`**
    `CObject * RemoveTail( ) ;`
    - Liefert einen Zeiger auf das erste (*RemoveHead*) bzw. letzte (*RemoveTail*) Element der Liste und entfernt diesen Zeiger aus der Liste.

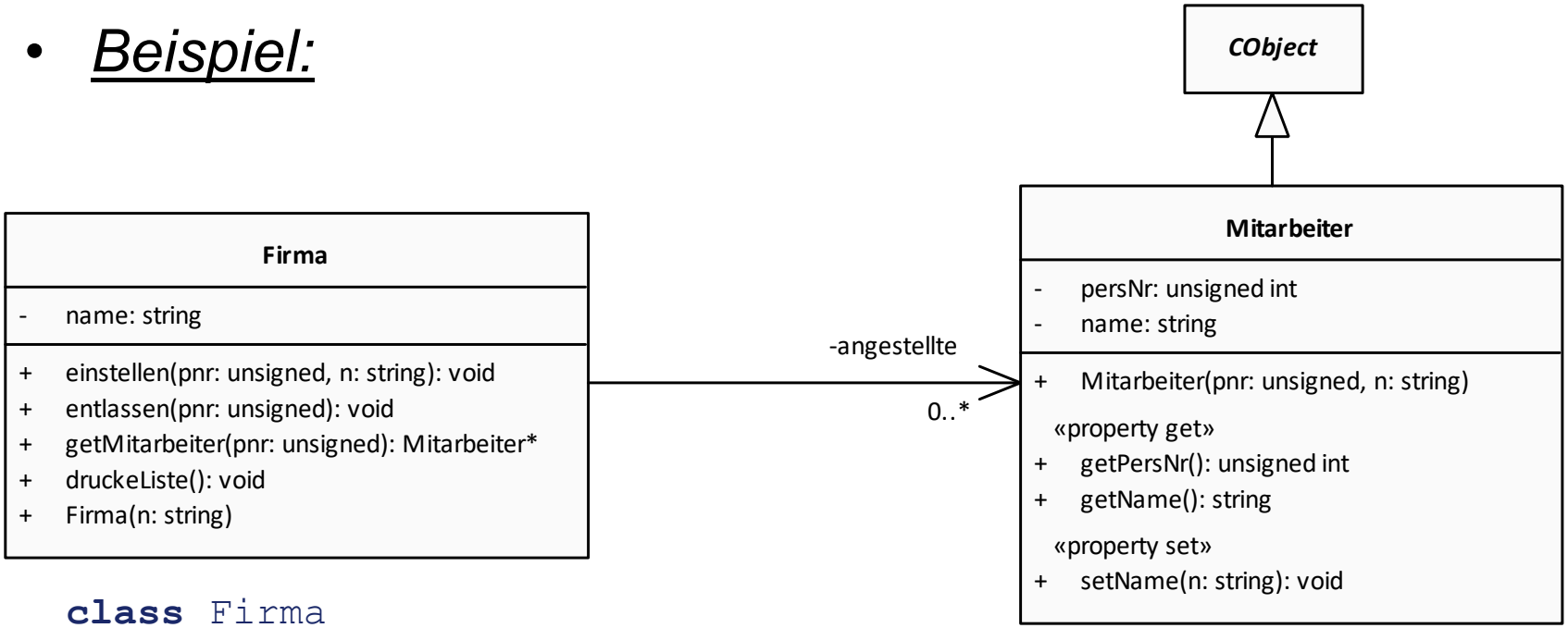
Container-Klassen

- *Beispiel:* Methoden der MFC-Klasse **`CObList`** (u.a.)

  - **`POSITION CObList::Find (CObject * E);`**
    - Liefert die Position des Elements E in der Liste, falls *E* Element der Liste ist, sonst *NULL*.
  - **`void CObList::RemoveAt(POSITION pos) ;`**
    - Entfernt das Element (CObject-Pointer) mit Position *pos* aus der Liste. (*Beachte:* Das Objekt, worauf der Zeiger zeigt, wird nicht gelöscht!)
  - **`void CObList::SetAt(POSITION pos, CObject * E) ;`**
    - Ersetzt das Element an der Position *pos* durch *E*;
  - **`POSITION InsertAfter(POSITION pos, CObject * E);`**
    `POSITION InsertBefore(POSITION pos, CObject * E) ;`
    - Fügt das Element *E* nach (*InsertAfter*) bzw. vor (*InsertBefore*) der Position *pos* in die Liste ein.

Assoziationen

- *Beispiel:*

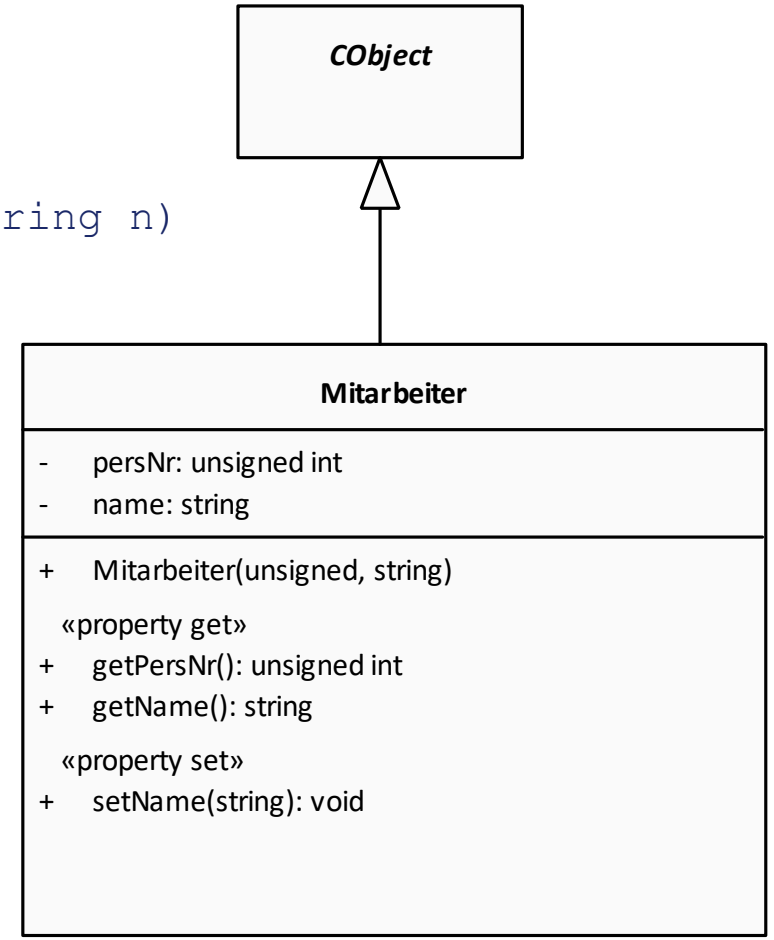



```
class Firma
{public:
  void einstellen(unsigned pnr, string n);
  void entlassen(unsigned pnr);
  Mitarbeiter* getMitarbeiter(unsigned pnr);
  void druckeListe();
  Firma(string n) // Konstruktor
private:
  string name;
  CObList angestellte; // Implementierung der Assoziation
};
```

Assoziationen

- Beispiel (Forts.): Implementierung der Klasse Mitarbeiter
  - Alle Methoden sind „inline“ definiert

```
class Mitarbeiter : public CObject
{ public:
    void Mitarbeiter(unsigned pnr, string n)
    { persNr = pnr; name = n); }
    unsigned int getPersNr()
    { return persNr; }
    string getName()
    { return name; }
    void setName(string n)
    { name = n; }
  private:
    unsigned int persNr;
    string name;
};
```



Assoziationen

- Beispiel (Forts.): Implementierung der Klasse Firma

```
Firma::Firma(string n)
{ name = n; }

void Firma::einstellen(unsigned pnr, string n)
{  Mitarbeiter * ma = new Mitarbeiter(pnr,n);
   angestellte.AddTail(ma);
}

Mitarbeiter * Firma::getMitarbeiter(unsigned pnr)
{  Mitarbeiter * ma; bool gefunden=false;
   POSITION pos = angestellte.GetHeadPostion();
   while (!gefunden && (pos != NULL))
   { ma = (Mitarbeiter *)angestellte.GetNext(pos);
     gefunden = ma->getPersNr() == pnr;
   }
   if (gefunden) return ma; else return 0;
}
```

Assoziationen

- Beispiel (Forts.): Implementierung der Klasse Firma

```
void Firma::entlassen(unsigned pnr)
{ POSITION pos, merkePos; bool gefunden=false; Mitarbeiter * ma;
  pos = angestellte.GetHeadPostion();
  while (!gefunden && (pos != 0))
   { merkePos = pos; // aktuelle Position merken
     ma = (Mitarbeiter *)angestellte.GetNext(pos);
     gefunden = ma->getPersNr() == pnr;
   }
  if (gefunden) { angestellte.RemoveAt(merkePos); delete ma; }
}

void Firma::druckeListe()
{ Mitarbeiter * ma; POSITION pos = angestellte.GetHeadPosition();
  while (pos != 0)
   { ma = (Mitarbeiter *)angestellte.getNext(pos);
     cout << ma->getPersNr() << ": "
          << ma->getName(s) << endl;
   }
}
```

Assoziationen

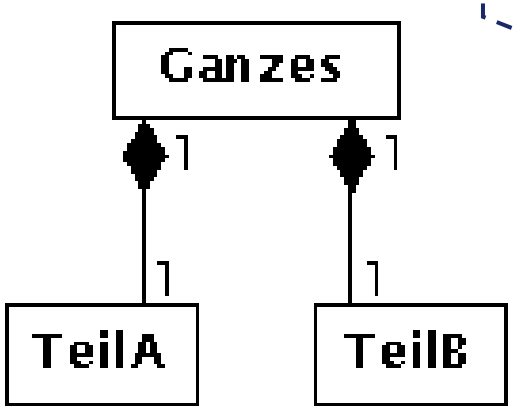
- Aggregation
  - Zu realisieren wie "normale" Assoziation
  - Navigation vom Ganzen zu den Teilen muss existieren
- Komposition
  - Navigation vom Ganzen zu den Teilen muss existieren
  - Operationen, die das Ganze betreffen, wirken sich auch auf seine Teile aus
  - Der Zugriff auf und das Erzeugen der Teile erfolgt immer über das Aggregatobjekt

Assoziationen

- Realisierungsmöglichkeiten für Kompositionen in C++
  - *by value* (physisches Enthaltensein)
    - Erzeugung und Löschung der Teile automatisch mit dem Ganzen
    - Kopieren des Aggregatobjekts bezieht sich immer automatisch auf seine Teile
      - Anmerkung: dies ist in Java nicht möglich, vgl. S. 57
  - *by reference* (Zeiger)
    - Durchführung des Erzeugen und Löschen des Teils durch den Konstruktor bzw. Destruktor
    - Beim Kopieren muß die entsprechende Operation auch das Kopieren der Teile realisieren

Assoziationen

- Realisierung Komposition in C++
  - Beispiel



Komposition "by value"

Komposition "by reference"

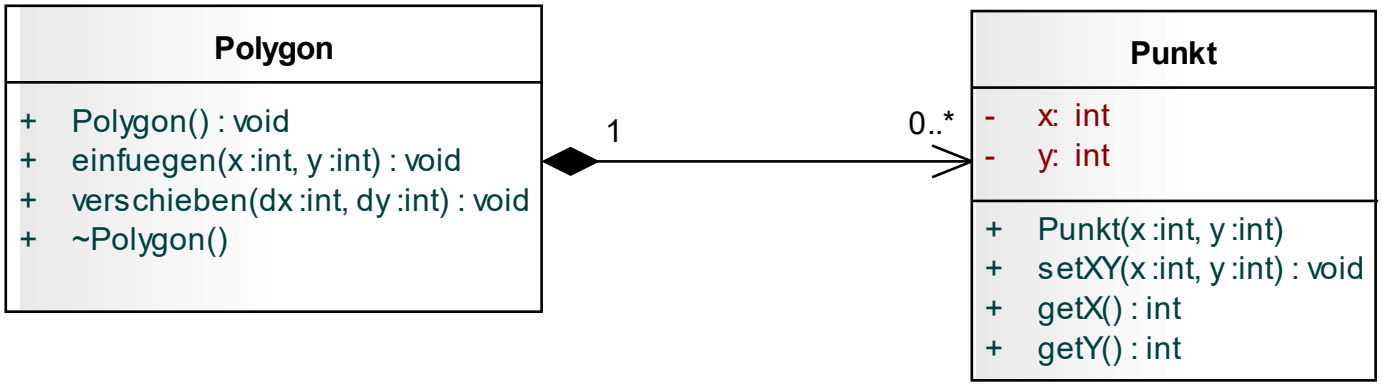
```
class Ganzes
{  TeilA   einTeilA;
   TeilB* einTeilB;

public:
   Ganzes()
   { einTeilB = new TeilB;}
   ~ Ganzes()
   {delete einTeilB;}
};
```

Quelle: [Balzert 2005], Abb. 6.5-7

Assoziationen

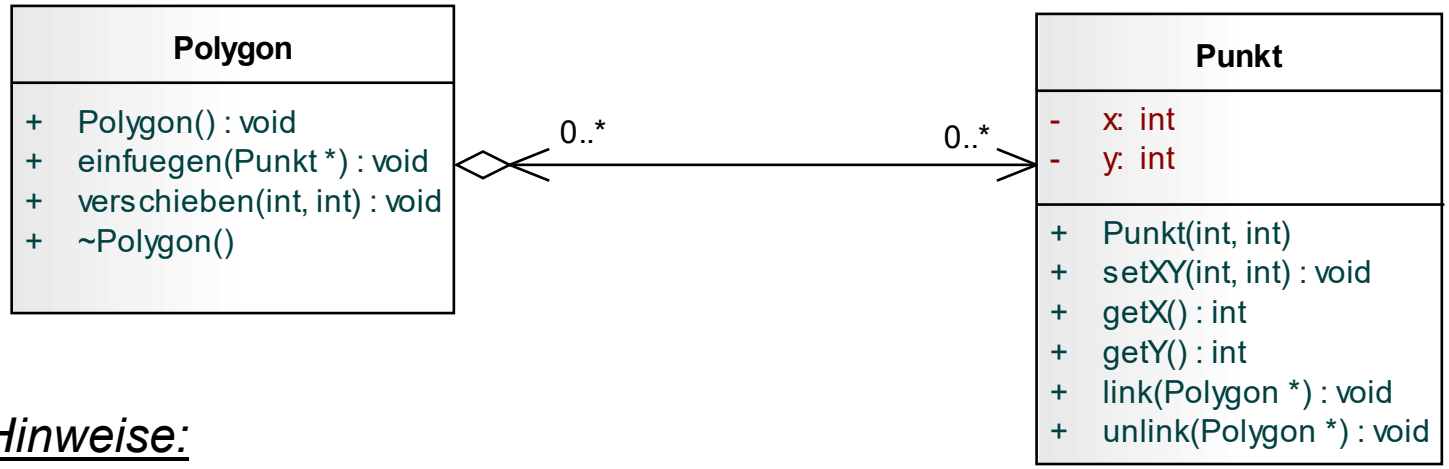
- **Aufgabe 1** (Vgl. Übungsblatt 1!)
  - Implementieren Sie folgendes Modell in C++



  - *Hinweise:*
    - Ein Polygon besteht mathematisch gesehen aus mindestens drei Punkten. Hier muss allerdings zunächst ein "leeres" Polygon erstellt werden. Anschließend können mit der Operation *einfuegen* mindestens drei Punkte eingefügt werden.
    - Die Operation *void verschieben(dx:int,dy:int)* addiert die Werte (dx,dy) auf die Koordinaten aller Punkte eines Polygons.
  - Schreiben Sie zum Testen ein Hauptprogramm, welches
    - zwei Polygone mit jeweils drei Punkten und einer gemeinsamen Seite erzeugt und
    - anschließend eines der beiden Polygone verschiebt.
    - Verfolgen Sie, welche Objekte wann im Speicher erzeugt werden und wie sie miteinander zusammen hängen. Fertigen Sie eine entsprechende Skizze an.

Assoziationen

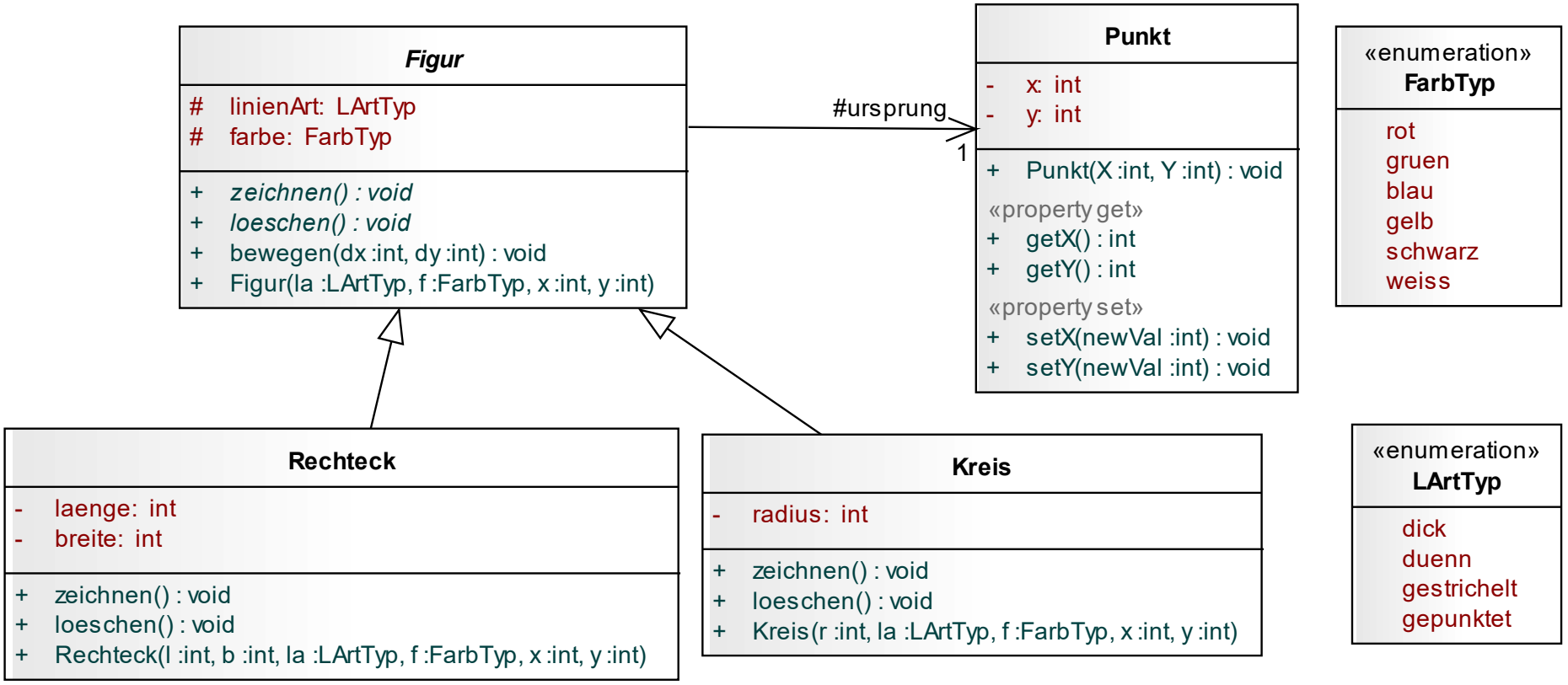
- **Aufgabe 2** (Vgl. Übungsblatt 1!)
  - Implementieren Sie folgendes Modell in C++



  - *Hinweise:*
    - Die Operation *Punkt::link(Polygon * pPol)* stellt eine Verknüpfung her von einem Punkt zu dem übergebenen Polygon.
    - Die Operation *Punkt::unlink(Polygon *pPol)* entfernt die Verknüpfung zwischen einem Punkt und dem übergebenen Polygon
  - Schreiben Sie zum Testen ein Hauptprogramm, welches
    - zwei Polygone mit jeweils drei Punkten und einer gemeinsamen Seite erzeugt, und
    - anschließend eines der Polygone verschiebt.
    - Verfolgen Sie, welche Objekte wann im Speicher erzeugt werden und wie sie miteinander zusammen hängen. Fertigen Sie eine entsprechende Skizze an.

Vererbung

- **Beispiel**: Eine Klassenhierarchie für geometrische Figuren



Vererbung in C++

- **Syntax**

```
class Vorfahr
{
   Deklaration der Attribute und Methoden
   der Klasse Vorfahr
} ;

class Nachkomme : modus Vorfahr
{
  Deklaration der speziellen Attribute und
  Methoden der Klasse Nachkomme

  evtl. Neudeklaration von Methoden der
  Klasse Vorfahr (Überschreibungen)
};
```

Vererbungsmodus:
private, public oder protected

  - Attribute und Methoden einer Klasse gehören zu einem **Schutzbereich**
    - ***private***, ***public*** oder ***protected*** (Default: *private*)
    - nur ***public*** und ***protected***-Elemente werden vererbt!
  - Die Vererbung wird durch einen ***Vererbungsmodus*** eingeschränkt
    - ***private***, ***public*** oder ***protected*** (Default: *private*)

Vererbung in C++

- **Beispiel**: Eine Klassenhierarchie für geometrische Figuren

```
enum LArtTyp { dick, duenn, gstrichelt, gepunktet } ;
enum FarbTyp { rot, gruen, blau, gelb, schwarz, weiss } ;

class Punkt
{   int x, y ;
  public:
    Punkt(int,int) ; // Konstruktor
    int getX() { return x; }
    void setX(int newVal) { x = newVal; }
    int getY() { return y; }
    void setY(int newVal) { y = newVal; }
} ;
```

Vererbung in C++

- Beispiel: Eine Klassenhierarchie für geometrische Figuren

```
class Figur
{
  protected: // Attribute sollen in Nachkommen sichtbar sein!
    LArtTyp linienArt ;
    FarbTyp farbe ;
    Punkt * ursprung ; // Assoziation zum Ursprungspunkt
  public:
    void zeichnen() ;
    void loeschen() ;
    // Figur in x- und y-Richtung verschieben
    void bewegen(int,int) ;
    // Konstruktor mit Parametern für Linienart, Farbe,
    // und die Koordinaten des Ursprungs
    Figur(LArtTyp,FarbTyp,int,int) ;
    // Destruktor
    ~Figur() ;
} ;
```

Vererbung in C++

- Beispiel: Eine Klassenhierarchie für geometrische Figuren

```
class Rechteck : public Figur  // Vererbungsmodus public
{
    int laenge, breite ; // private Attribute
  public:
    void zeichnen() ;
    void loeschen() ;
    // Konstruktorparameter: Länge, Breite und Attribute der Basisklasse
    Rechteck(int,int,LArtTyp,FarbTyp,int,int) ;
} ;

class Kreis : public Figur  // Vererbungsmodus public
{ private:
    int radius;  // privates Attribut
  public:
    void zeichnen() ; // Überschreiben der geerbten Methode
    void loeschen() ; // Überschreiben der geerbeten Methode
    // Konstruktorparameter: Radius und Attribute der Basisklasse
    Kreis(int,LArtTyp,FarbTyp,int,int) ;
} ;
```

Vererbung in C++

- Schutzbereiche
  - Allgemeine Zugriffsregeln
    - Zugriff auf Attribute und Methoden eines Objekts:
      - nur auf **öffentliche** Attribute und Methoden der Klasse erlaubt
      - Beispiel:

```
Kreis K(12,dick,rot,0,0);
K.zeichnen(); // o.k., zeichnen ist public!
K.radius = 5; // Nicht o.k.: radius ist private!
```

    - Zugriff auf Attribute und Methoden in einer Methodendefinition
      - Zugriff auf **alle** Attribute und Methoden der Klasse erlaubt
      - Beispiel:

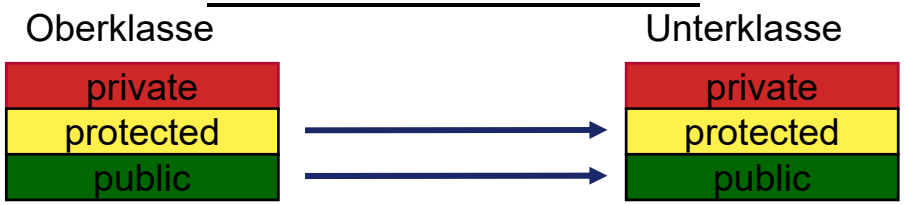
```
void Kreis::bewegen(int dx, int dy)
{
  loeschen(); // Kreis löschen
  ursprung -> setX(ursprung -> getX() + dx);
  ursprung -> setY(ursprung->getY() + dy);
  zeichnen(); // Kreis zeichnen
}
```

Vererbung in C++

- Schutzbereiche und Vererbung
  - Fragen:
    - Welche Merkmale einer Klassen werden von Vorfahren an Nachkommen vererbt?
    - Welche Sichtbarkeit haben vererbte Merkmale im Nachkommen?
  - Antworten:
    - Nur öffentliche (*public*) und geschützte (*porotected*) Merkmale werden vererbt
    - Der **Vererbungsmodus** legt fest, welche Sichtbarkeit vererbte Merkmale im Nachkommen haben

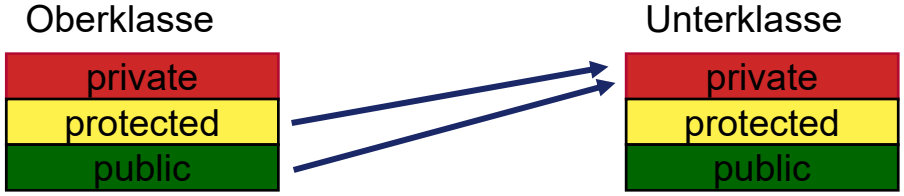
Vererbung in C++

- Schutzbereiche und Vererbungsmodi
  - Vererbungsmodus ***public:*** Vererbte Merkmale haben im Nachkommen *dieselbe Sichtbarkeit* wie im Vorfahren.



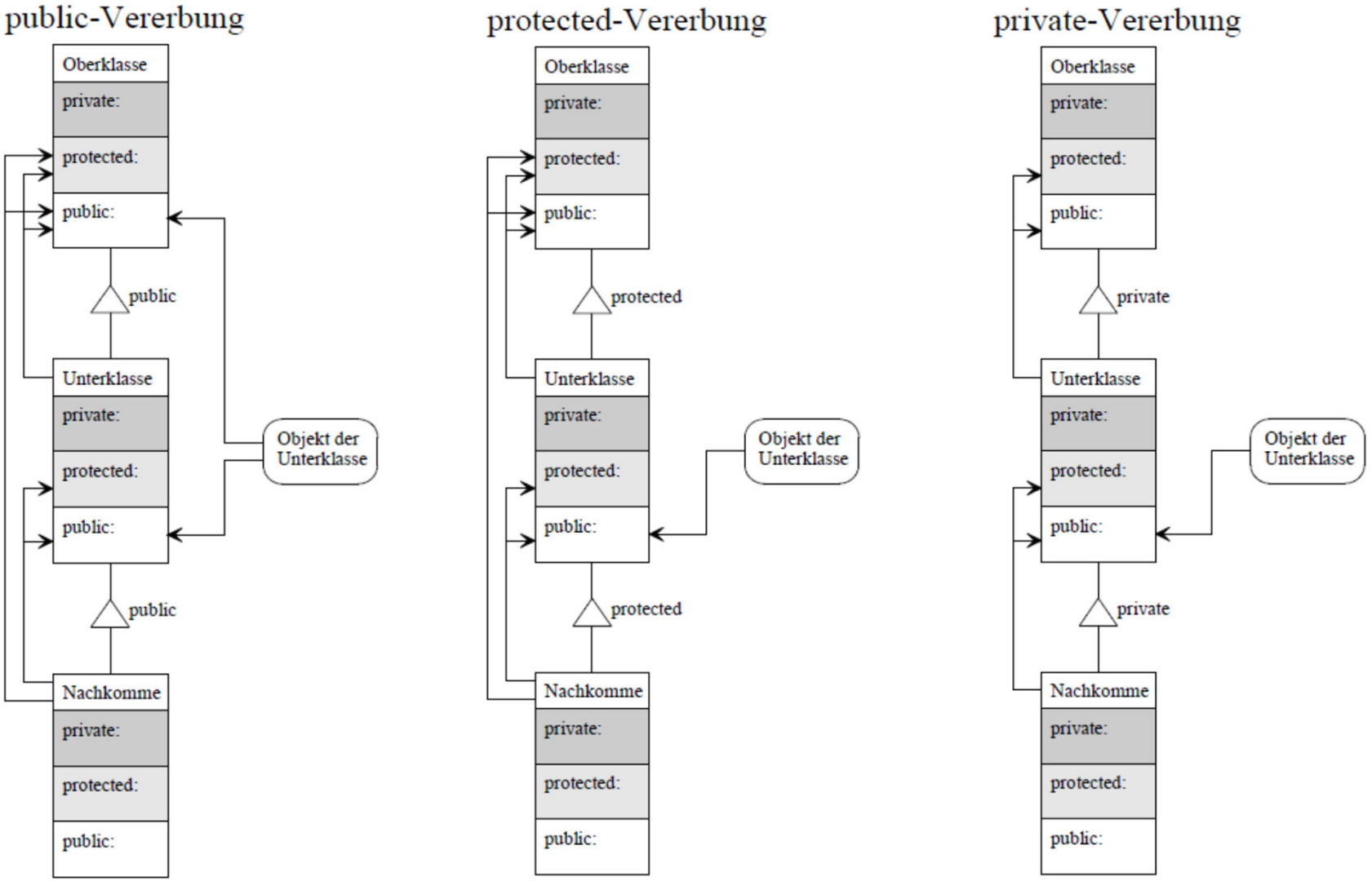
  - Vererbungsmodus ***protected:*** Vererbte Merkmale haben im Nachkommen die Sichtbarkeit *protected*.

  - Vererbungsmodus ***private:*** Vererbte Merkmale haben im Nachkommen die Sichtbarkeit *private*.



Vererbung in C++

- Schutzbereiche und Vererbungsmodi: Zugriffsrechte



Vererbung in C++

- ***Konstruktoren in Vererbungshierarchien***
  - Bei der ***Erzeugung*** eines Objekts einer abgeleiteten Klasse werden die Konstruktoren ***aller*** Vorfahren bis zur Basisklasse aufgerufen (Reihenfolge: **„von oben nach unten“**).
  - In der Konstruktordefinition der abgeleiteten Klasse kann die Weitergabe von Parametern an den Konstruktor der Vorfahr-Klasse spezifiziert werden.
  - *Syntax:*

```
Nachkomme::Nachkomme( p1,...,pk , pk+1,...,pn): Basisklasse(pk+1,...,pn)
{  // Die Parameter pk+1,...,pn sind für den Konstruktor der Basisklasse
   // Hier können die Parameter p1,…,pk verarbeitet werden.
   …
}
```

Vererbung in C++

- *Konstruktoren in Vererbungshierarchien*
  - *Beispiel:* Klassenhierarchie für geometrische Figuren

```
Punkt::Punkt(int X, int Y)
{
  x = X ; y = Y ;
}

Figur :: Figur(LArtTyp la , FarbTyp f, int x, int y)
{
  linienArt = la ; farbe = f ;
  ursprung = new Punkt(x,y) ; // Ursprung konstruieren
}
```

Vererbung in C++

- *Konstruktoren in Vererbungshierarchien*
  - *Beispiel:* Klassenhierarchie für geometrische Figuren

```
Rechteck::Rechteck(int l,int b, LArtTyp la, FarbTyp f, int x, int y):
                                                Figur(la,f,x,y)
// die Parameter LA,F,x,y werden an den Figur-Konstruktor übergeben
{
   laenge = l ; breite = b ;
}

Kreis::Kreis(int r, LArtTyp la, FarbTyp f, int x, int y):
                                                Figur(la,f,x,y)
// die Parameter LA,F,x,y werden an den Figur-Konstruktor übergeben
{
   radius = r ;
}
```

Vererbung in C++

- ***Destruktoren in Vererbungshierarchien***
  - Bei der ***Zerstörung*** eines Objekts einer abgeleiteten Klasse werden die Destruktoren ***aller*** Vorfahren bis zur Basisklasse aufgerufen (in umgekehrter Reihenfolge: **„von unten nach oben“**)
  - *Beispiel:* Klassenhierarchie für geometrische Figuren

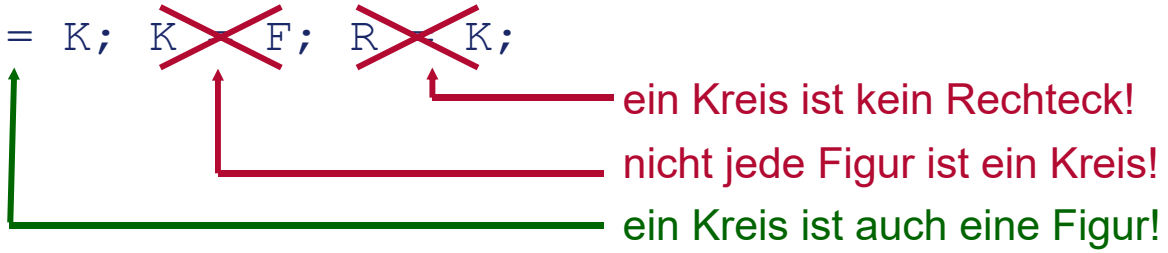
```
Figur::~Figur()
{
  delete ursprung ; // Ursprung freigeben
}
```

- Weitere Destruktoren (Kreis, Rechteck) sind nicht erforderlich!
  - Es existiert ein Default-Destruktor

Vererbung in C++

- Zuweisungen und Vererbungshierarchien
  - Welche Zuweisungen sind zulässig? Welche nicht?

```
Figur F(...); Kreis K(...); Rechteck R(...);
F = K; K = F; R = K;
```



  - Regeln:
    - Ein Nachkomme-**Objekt** kann an eine Vorfahr-Variable zugewiesen werden
      - umgekehrt nicht!
    - Ein **Zeiger auf ein Nachkomme**-Objekt kann an eine Zeigervariable mit dem Bezugstyp Vorfahre zugewiesen werden
      - umgekehrt nicht!

Anmerkung: Die Erzeugung eines Objekts der Klasse Figur (`Figur F(…)`) ist grundsätzlich nicht sinnvoll, da Figur gemäß Modell eine abstrakte Klasse ist. In dem bisher gezeigten C++-Code wäre dieses aber möglich. – Allerdings wäre es problematisch, eine Methode *Figur::zeichnen()* zu implementieren!

Vererbung in C++

- Beispiel:
  - Ein Figur-Zeiger kann auch auf Kreise oder Rechtecke zeigen

```
Figur * pF; char antwort;

cout << "Welche Figur? (K/R):";
cin >> antwort;

if ((antwort == 'K') || (antwort == 'k'))
   pF = new Kreis(12,dick,rot,0,0);
else if ((antwort == 'R') || (antwort == 'r'))
   pF = new Rechteck(7,5,duenn,blau,1,1);
else pF = NULL;

if (pF != NULL) pF -> zeichnen();
// Wenn pF != NULL zeigt pF auf ein Rechteck oder einen Kreis;
// Beides kann man zeichnen!
```

Vererbung in C++

- Abstrakte Operationen
  - Abstrakte Operationen haben in einer abstrakten Basisklasse keine Implementierung
  - Sie **müssen** in konkreten Unterklassen implementiert werden
    - *Beispiel:* Die Operationen *Figur::zeichnen()* und *Figur::loeschen()*
  - Deklaration von abstrakten Operationen in C++ in der Klassendefinition: "Null setzen"

```
class Figur {
  ...
public:
  void zeichnen() = 0;
  void loeschen() = 0;
... };
```

- Abstrakte Klassen
  - Von abstrakten Klassen können keine Instanzen gebildet werden
  - In C++: sobald eine Klasse eine abstrakte Operation hat, ist sie abstrakt

Vererbung in C++

- Polymorphismus
  - unterschiedliche Implementierungen für dieselbe Operation in verschiedenen Unterklassen
    - Griechisch: *polymorph* = *vielgestaltig*
  - Beispiel:
    - Unterschiedliche Implementierung der abstrakten Operation *Figur::zeichnen()* in den Unterklassen *Kreis* und *Rechteck* *

```
Kreis::zeichnen()
{ gotoXY(ursprung->getX(),ursprung->getY());
  drawCircle(linienArt,farbe,radius);
}
Rechteck::zeichnen()
{ gotoXY(ursprung->getX(),ursprung->getY());
  drawRectangle(linienArt,farbe,laenge,breite);
}
```

* *gotoXY()*, *drawCircle()* und *drawRectangle()* seien entsprechende Funktionen einer Grafik-Bibliothek

Vererbung in C++

- Polymorphismus
  - Problem: Binden von Funktions-Definitionen an Funktions-Aufrufe
  - Beispiel 1:

```
Figur * pF; char antwort;

cout << "Welche Figur? (K/R):"; cin >> antwort;

if ((antwort == 'K') || (antwort == 'k'))
  pF = new Kreis(12,dick,rot,0,0);
else if ((antwort == 'R') || (antwort == 'r'))
  pF = new Rechteck(7,5,duenn,blau,1,1);
else pF = NULL;

if (pf != NULL) pF -> zeichnen();
// Welche Definition von "zeichnen" soll hier angewandt werden?
// Kreis oder Rechteck zeichnen?
```

Vererbung in C++

- Polymorphismus
  - Problem: Binden von Funktions-Definitionen an Funktionsaufrufe
  - Beispiel 2: Implementierung der Methode *Figur::bewegen()*

```
Figur::bewegen(int dx,int dy)
{
  loeschen(); // Kreis oder Rechteck löschen?
  ursprung->setX(ursprung->getX()+dx);
  ursprung->setY(ursprung->getY()+dy);
  zeichen();  // Kreis oder Rechteck zeichen?
}
```

  - Lösung: **Dynamisches Binden**
    - Erst zur Laufzeit wird entschieden, welche Implementierung einer polymorphen Methode an einen Aufruf gebunden wird
      - Erst zur Laufzeit ist bekannt, mit welchen Objekten eine Methode aufgerufen wird
      - Zur Übersetzungszeit ist keine Entscheidung möglich (auch in Beispiel 1)

Vererbung in C++

- Dynamisches Binden – virtuelle Methoden
  - Polymorphe Methoden, die dynamisch gebunden werden sollen, müssen in der Klassendefinition als **virtuelle Methoden** deklariert werden:

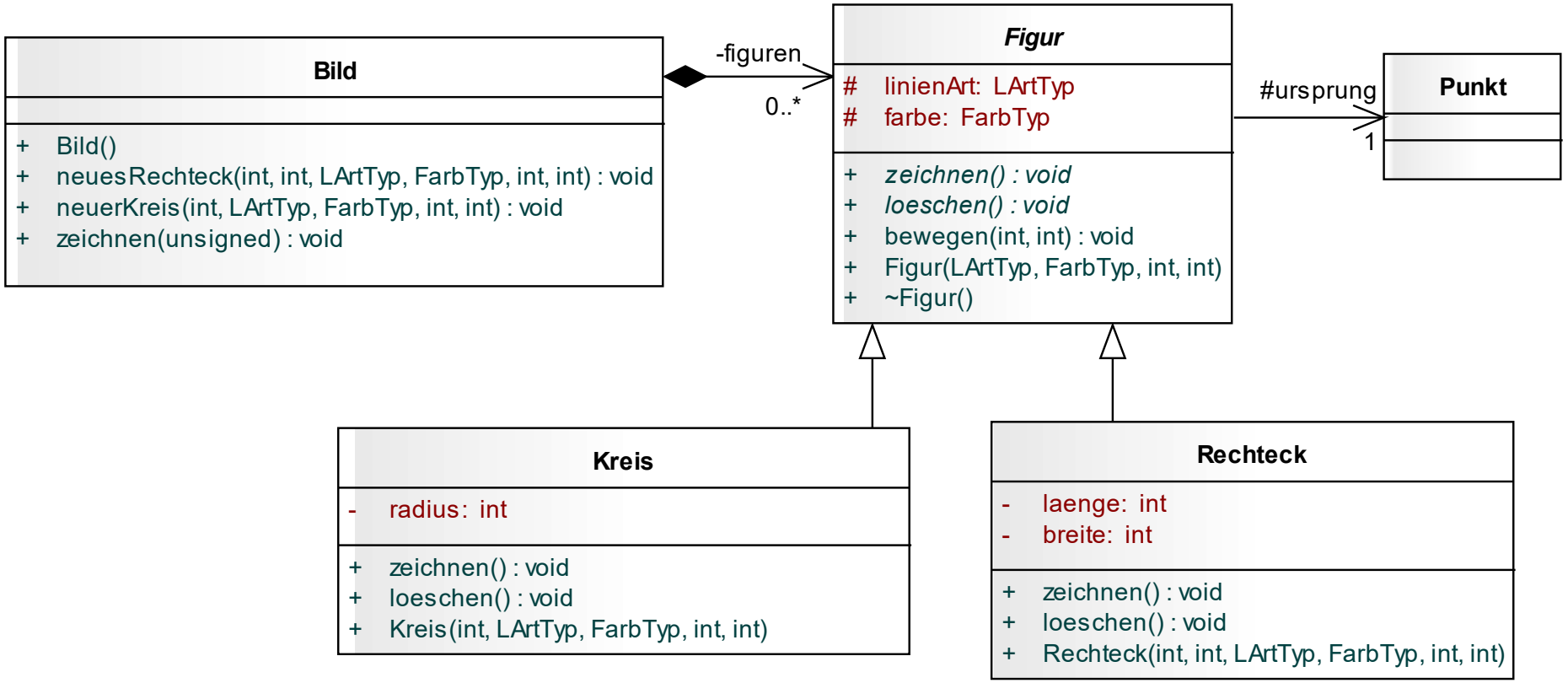
```
class Figur {
  ...
public:
  virtual void zeichnen() = 0;
  virtual void loeschen() = 0;
... };
```

Anmerkung: Virtuelle Methoden müssen nicht unbedingt gleichzeitig abstrakt sein.

  - Typische Anwendungen
    - Konkrete Methoden in abstrakten klassen, die abstrakte Operationen aufrufen (→ *Figur::bewegen()*)
    - Generische Container-Klassen (z.B. Klasse *Bild*, s.u.):
      - Eine Menge von Objekten, die zu einer gemeinsamen Basisklasse gehören
      - Bestimmte Operationen können mit allen Objekten durchgeführt werden, werden jedoch für Objekte unterschiedlicher Unterklassen unterschiedlich implementiert.

Vererbung in C++

- Beispiel: Ein Bild aus vielen Figuren



Vererbung in C++

- Beispiel: Ein Bild aus vielen Figuren
  - Implementierung der Klasse Bild

```
class Bild {

  // Ein Bild besteht aus maximal 100 Figuren
  // Implementierung der Komposition als Zieger-Feld
  Figur * Figuren[100];

public:

  Bild(); // Konstruktor
  // Neues Rechteck einfügen:
  void neuesRechteck(int l, int b,
                     LArtTyp la, FarbTyp f, int x, int y);
  // Neuen Kreis einfügen:
  void neuerKreis(int r, LArtTyp la, FarbTyp f, int x, int y);
  // das ganze Bild zeichen:
  void zeichnen();
};
```

Vererbung in C++

- Beispiel: Ein Bild aus vielen Figuren
  - Rechtecke und Kreise einfügen

```
void Bild::neuesRechteck(int l, int b,
                         LArtTyp la, FarbTyp f, int x, int y){
  // geeigneten Index i bestimmen ...
  // Rechteck konstruieren und eintragen:
  Figuren[i] = new Rechteck(l,b,la,f,x,y) ;
                  // Zuweisung Vorfahre = Nachkomme erlaubt!
}

void Bild::neuerKreis(int r,
                      LArtTyp la, FarbTyp f, int x, int y){
  // geeigneten Index i bestimmen ...
  // Rechteck konstruieren und eintragen:
  Figuren[i] = new Kreis(r,la,f,x,y) ;
                  // Zuweisung Vorfahre = Nachkomme erlaubt!
}
```

Vererbung in C++

- Beispiel: Ein Bild aus vielen Figuren
  - Das ganze Bild zeichnen

```
void Bild::zeichnen()
{ // ein Bild zeichnen, heißt alle Figuren zeichnen
  int i ;
  for (i=0;i<100;i++)
    if (Figuren[i] != NULL)
       Figuren[i] -> zeichnen() ;
       // Rechteck oder Kreis zeichnen ?
       // → dynamisches Binden erforderlich
}
```

Vererbung in Java

- Vererbung: Schlüsselwort **`extends`**

  class *Unterklasse* ***extends*** *Oberklasse*

- Abstrakte Klassen/Operationen:
  - Schlüsselwort **`abstract`** kennzeichnet abstrakte Klassen und abstrakte Operationen

```
public abstract class Basisklasse {

   public abstract void operation(…);
   …
}
```

  - Eine Klasse mit mindestens einer abstrakten Operation muss als **`abstract`** deklariert werden
- Dynamisches Binden
  - Alle Methodenaufrufe werden dynamisch gebunden
  - Keine Deklaration von "virtuellen Methoden" erforderlich

Vererbung in Java

- Beispiel: Abstrakte Basisklasse *Figur*

```
public abstract class Figur {
   protected LArtTyp linienArt;
   protected FarbTyp farbe;
   protected Punkt ursprung;

   public Figur(LArtTyp la, FarbTyp f, int x, int y){ ... }
   public abstract void zeichnen();
   public abstract void loeschen();
   public void bewegen(int dx, int dy){ ... };
}
```

- Beispiel: von *Figur* abgeleitete Klasse *Kreis*

```
public class Kreis extends Figur {
   private int radius;
   public Kreis(int r, LArtTyp la, FarbTyp f, int x, int y){...}
   public void zeichnen(){...}
   public void loeschen(){...}
}
```

Vererbung in Java

- Konstruktoren
  - Im Konstruktor einer Unterklasse kann als erstes der Basisklassen-Konstruktor explizit aufgerufen werden durch

    super(parameter-für-den-Basisklassen-Konstruktor);
  - Falls *super* nicht aufgerufen wird, wird implizit ein *super*-Aufruf ohne Parameter (default-Konstruktor) ergänzt

```
public abstract class Figur { ...
  public Figur(LArtTyp la, FarbTyp f, int x, int y){
    linienArt = la; farbe = f;
    ursprung = new Punkt(x,y);
  } ... }

public class Kreis extends Figur { ...
  public Kreis(int r, LArtTyp la, FarbTyp f, int x, int y) {
    super(la,f,x,y);
    radius = r;
  } ... }
```

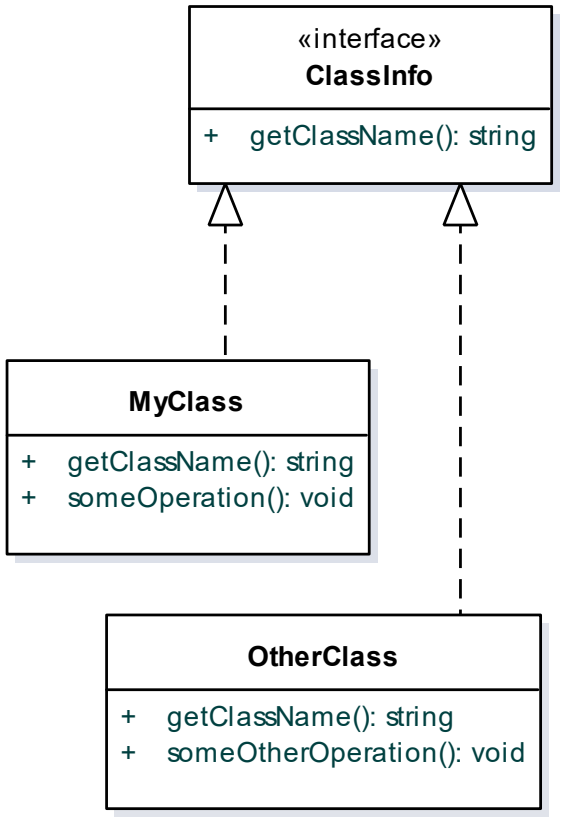
Schnittstellen

- **Schnittstelle** in C++
  - Konzept existiert nicht, kann jedoch durch Klassen mit ausschließlich virtuellen, abstrakten Operationen nachgebildet werden (***pure virtual member function***)

```
class Interface
{ //nur abstrakte Operationen
  public:
        virtual void operation1() = 0;
        virtual void operation2() = 0;
};
```

Schnittstellen

- Schnittstelle in C++ - Beispiel




```
class ClassInfo {
public:
  virtual string getClassName() = 0;
};

class MyClass : public ClassInfo {
public:
    void someOperation();
    virtual string getClassName()
                {return "MyClass" ;}
};

class OtherClass : public ClassInfo {
public:
    void someOtherOperation();
    virtual string getClassName()
                {return "OtherClass" ; }
};
```

Schnittstellen

- Schnittstelle in Java
  - Kann Konstanten und abstrakte Operationen enthalten
  - Wird mit dem Schlüsselwort **`interface`** deklariert
  - Wird von Klassen mit dem Schlüsselwort **`implements`** realisiert

```
interface ClassInfo
{public String getClassName();}

class MyClass implements ClassInfo
{public String getClassName()
    { return "MyClass"; }
...
}
```

Dateiströme in C++

- `<fstream>`
  - enthält Klassen zur formatierten Text-Ein-/Ausgab auf Dateien (**Stromklassen**)
    - `ofstream` *output file stream* zur Ausgabe
    - `ifstream` *input file stream* zur Eingabe
    - `fstream` *file stream* zur Ein- und Ausgabe
- Dateiströme
  - Dateien können repräsentiert werden durch Variablen der Stromklassen
    - `ofstream os;` Stromobjekt zur Ausgabe in eine Datei (Schreiben)
    - `ifstream is;` Stromobjekt zur Eingabe von einer Datei (Lesen)
    - `fstream fs;` Stromobjekt zur Ein- und Ausgabe von/in Dateien

Dateiströme in C++

- **Öffnen** von Dateien über den Konstruktor
  - Bei der Definition von Stromobjekten kann per **Paramterübergabe an den Konstruktor** dem Stromobjekt ein **Dateiname** und ein **Öffnungsmodus** zugeordnet werden.
  - Dadurch werden die Dateien **geöffnet** und mit dem Stromobjekt verknüpft.
  - Für den Öffnungsmodus (und weitere Parameter) gibt es default-Werte:

```
ofstream::ofstream(const char *name,
                   int modus = ios::out, ...);
ifstream::ifstream(const char *name,
                   int modus = ios::in, ...);
fstream::fstream(const char *name,
                 int modus, ...);
```

Dateiströme in C++

- Öffnen von Dateien: **Öffnungsmodi**
  - Öffnungsmodi sind als int-Konstanten der Klasse **`ios`** definiert

`ios::in`	Öffnen zum Lesen (Lese-Position am Anfang)
`ios::out`	Öffnen zum Schreiben (Schreib-Position am Anfang)
`ios::ate`	Schreib-/Leseposition am Dateiende
`ios::app`	Schreiben nur am Dateiende
`ios::trunc`	Dateiinhalt beim Öffnen löschen

  - Öffnungsmodi sind numerisch 2er-Potenzen und können mit dem ODER-Operator (|) kombiniert werden
  - *Beispiele:*

```
ifstream ein("Beispiel1.txt"); // Datei zum Lesen öffnen(default)
ofstream aus("Beispiel2.txt",ios::out | ios::trunc);
     // Datei zum Schreiben öffnen, Inhalt löschen
fstream einaus("Beispiel3.txt", ios::in | ios::out | ios::ate);
          // Datei zum Lesen und Schreiben öffnen,
          // Schreib-/Leseposition am Ende
```

  - Öffnen erfogreich? – Abfrage des Stromobjekts:

```
if (einaus) { // Datei bearbeiten } else { // Fehlerbehandlung }
```

Dateiströme in C++

- **Lesen** und **Schreiben**
  - E-/A-Operatoren >> bzw. <<
    - Ausgabe von Werten in eine Datei

```
aus << Wert1 << Wert2 ...   ;
```

    - Eingabe von Werten von einer Datei und Speichern in Variablen

```
ein >> Variable1 >> Variable2 >> ... ;
```

    - Zu Beachten:
      - Bei der Ein- bzw. Ausgabe mit >> bzw. << erfolgt eine Wandlung von Textzeichen in Zahlen bzw. umgekehrt.
      - Gelesene Zeichen werden aus dem Strom entfernt.
      - White-Spaces (Leerzeichen, Zeilenwechsel, Tabulator) werden vor der Eingabe einer Zahl überlesen.

Dateiströme in C++

- Lesen und Schreiben
  - Methoden der Stromklassen – Beispiele
    - `peek()`: nächstes Zeichen lesen ohne die Schreib-/Lese-Position zu verändern

```
int c; c = ein.peek();
```

    - `getline(char *, int)`: eine ganze Zeile lesen

```
char buffer[128]; ein.getline(buffer,128);
// Eine Zeile aus dem Strom nach buffer übertragen (maximal
// 128 Zeichen, höchstens bis zum nächsten Zeilenwechsel)
```

    - `ignore(int,char)`: Zeichen überlesen (S-/L-Position weiter schieben)

```
ein.ignore(10,'\n');
  // 10 Zeichen im Strom überlesen
  // höchstens bis zum nächsten Zeilenwechsel
```

Dateiströme in C++

- **Schließen** von Dateien
  - Mit dem Konstruktor geöffnete Dateien werden durch den **Destruktor** automatisch wieder **geschlossen**
  - Alternative
    - Deklaration des Datei-Stromobjekts mit Standard-Konstruktor
      - Der Strom ist noch nicht mit einer Datei verknüpft

```
ofstream os;
```

    - explizites Öffnen mit *open()* (Parameter wie Konstruktor)

```
os.open("Datei.txt");
```

    - Explizites Schließen mit *close()*

```
os.close();
```

- **Beispiel**
  - Speichern und Laden für die Personalkartei-Anwendung

Zustandsautomaten in C/C++

- **Explizite Zustandsvariablen**
  - Der aktuelle Zustand des Automaten wird in einer **Variablen** gespeichert

    ```
    Z = Anfangszustand
    ```

  - In einer **Schleife** mit einer **Fallunterscheidung** über alle Zustände wird
    - die Verarbeitung im aktuellen Zustand und
    - die Bestimmung des Folgezustands

    wiederholt, solange bis der Endzustand erreicht ist
  - Typ der Zustandsvariablen ("Qualitative Zustände", vgl. S. 5-90))
    - entweder **int** (Zustände durchnummerieren)
    - oder **Aufzählungstyp** definieren und Zustände benennen
    - Terminierende Automaten müssen einen Endzustand enthalten
  - Quantitative Zustände
    - Weitere globale Variablen können verwendet werden

Zustandsautomaten in C/C++

- Explizite Zustandsvariablen
  - Struktur der Schleife

```
enum Zustand {z1,z2,…,zn,ende};
enum Zustand Z = z1; /* z1 ist Startzustand */

do {
   switch(Z) {
   ...
   case(Zi): Verarbeite(Z);
             Z = Folgezustand(Z);
             break;
   ...
   }
} while(Z != ende);
```

  - Bei nicht-terminierenden Automaten Endlosschleife:

```
do { switch(Z) { ... } } while 1;
```

Zustandsautomaten in C/C++

- Explizite Zustandsvariablen
  - Verarbeitung in einem Zustand (`Verarbeite(Z)`)
    - Eintrittsaktivitäten ausführen
    - do-Aktivitäten ausführen
  - Berechnung des Folgezustands (`Z=Folgezustand(Z)`)
    - Ereignisse abfragen und Guard-Bedingungen prüfen
    - entsprechenden Folgezustand zuweisen
    - entsprechende Exit-Aktivitäten oder Aktivitäten am Übergang ausführen
    - Besonderheiten:
      - Unter Umständen muss die Berechnung des Folgezustands in einer "Warteschleife" erfolgen, um zu verhindern, dass Eintritts- und do-Aktivitäten wiederholt ausgeführt werden:

        Wiederhole { Z=Folgezustand(Z) } bis ein Übergang feuert
      - Interne Transitionen (vgl. S. 6-69): werden im Rahmen der Bearbeitung in einem Zustand abgearbeitet (s.u. Beispiel 2)
      - Pseudo-Zustände, wie z.B. Entscheidungen oder Kreuzungen (s.u. Beispiel 3)
        - » Duch "echte Zustände" ersetzen oder
        - » Ausprogrammieren (*if (… & …) … else*)

Zustandsautomaten in C/C++

- Explizite Zustandsvariablen
  - Voraussetzung
    - Alles kann effektiv berechnet werden, d.h. es stehen Funktionen oder Ausdrücke zur Verfügung zur
      - Ausführen der Aktivitäten (do-, entry-, exit-, Übergang)
      - Prüfen von Ereignissen und Bedingungen
        - » Falls erforderlich, globale Variablen ("quantitative Zustände") verwenden
    - Typisch:
      - Ein Zustandsdiagramm beschreibt den Lebenszyklus eines Objekts einer Klasse, und die benötigten Funktionen sind Operationen der Klasse
      - Ein Zustandsdiagramm beschreibt den Ablauf einer Operation einer Klasse, und die benötigten Funktionen sind Operationen der Klasse
- Aufgaben:
  - Siehe Übungsblatt 2, Aufgabe 2-4 (2)
  - Türme von Hanoi: Implementieren Sie die Methode **Spiel**::spielen()

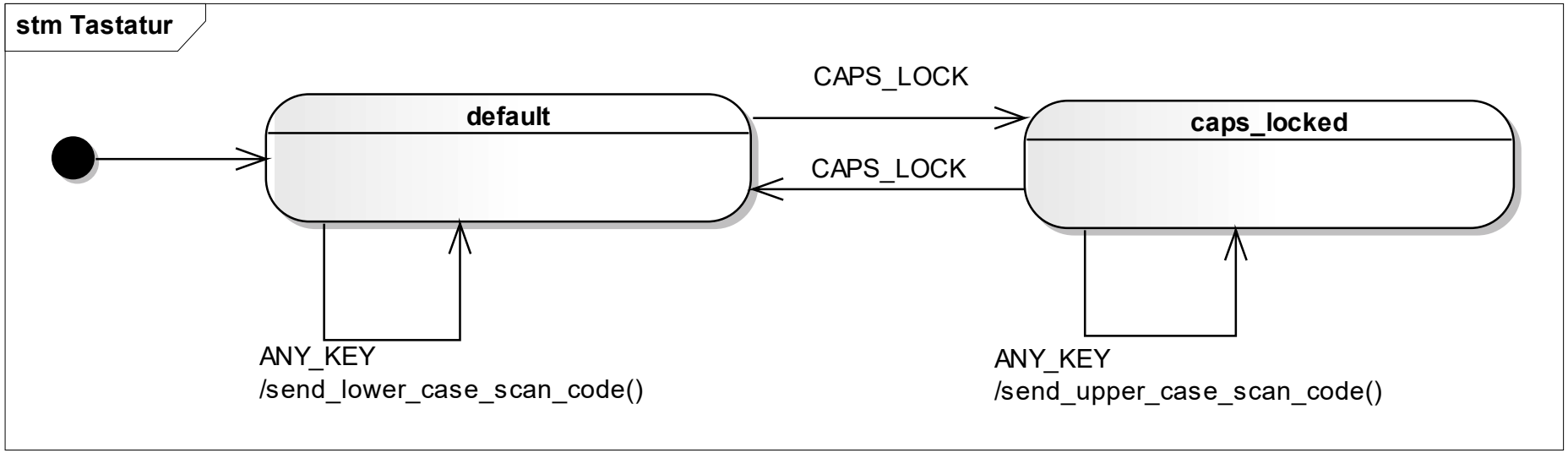
- **Beispiel 1** (vgl. S. 5-86)
  - Funktionen und zur Abfrage von Ereignissen:

`int get_key();` *Wartet, bis an der Tastatur eine Taste gedrückt wurde und liefert dann die "Nummer" der gedrückten Taste*

```
void send_lower_case_scan_code(int nr);
void send_upper_case_scan_code(int nr);
```

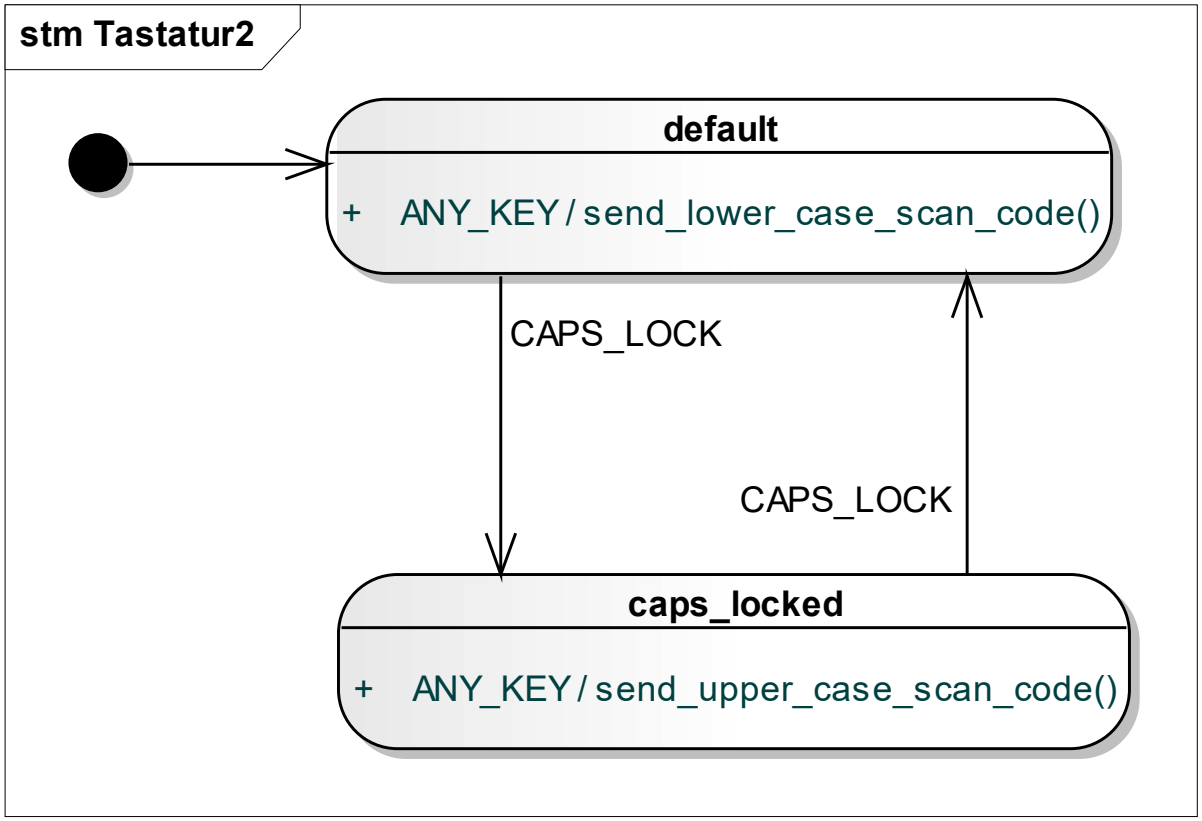
Schickt den Code für den mit das Taste Nr. nr verbundene Zeichen an das Betriebssystem (Upper-Case bzw. Lower-Case-Code)

`CAPS_LOCK` *Konstante, deren Wert die Nummer der CAPS_LOCK-Taste ist*



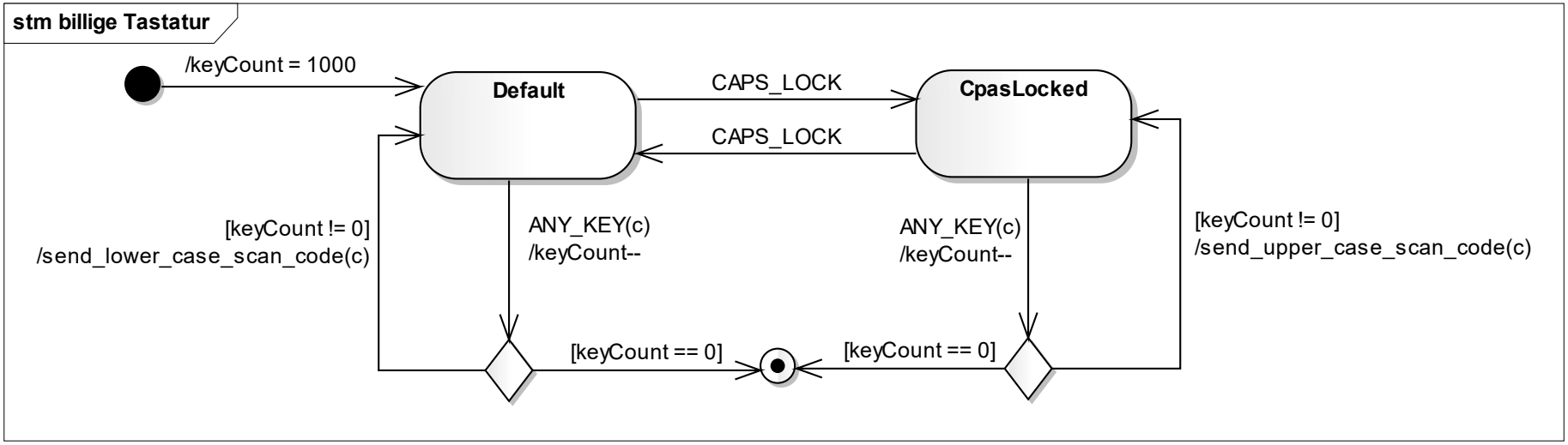
Zustandsautomaten in C/C++

- **Beispiel 2** (vgl. S. 5-87)
  - Mit internen Transitionen



Zustandsautomaten in C/C++

- **Beispiel 3** (vgl. S. 5-88)
  - Mit Pseudozustand "Entscheidung"
    - Wird Implementiert durch eine `if`-Anweisung
  - Mit quantitativem Zustand *keyCount*
    - Wird implementiert durch eine globale Variable `keyCount`
  - Mit Endzustand



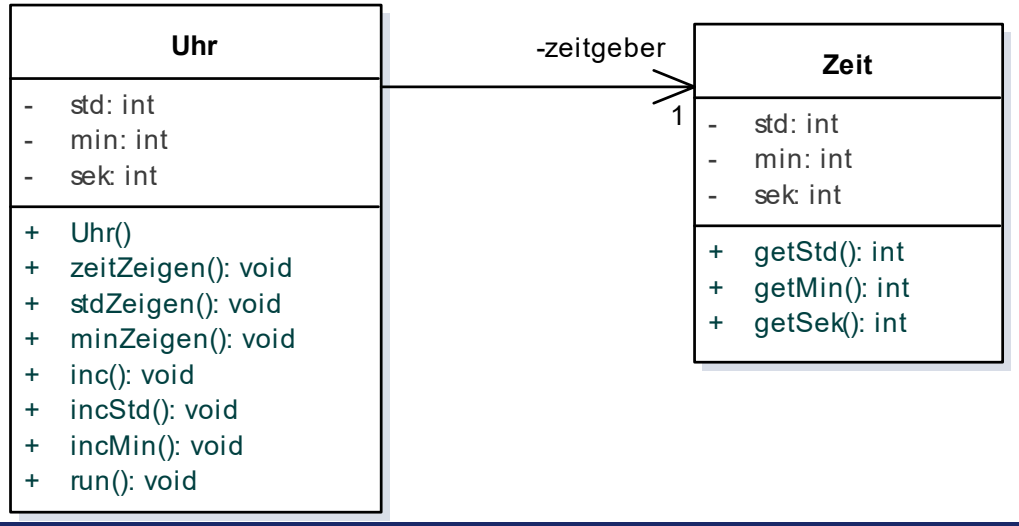
Zustandsautomaten in C/C++

- **Aufgabe**: Zustandsautomat für eine einfache Digitaluhr

  (vgl. Übungsblatt 2, Aufgabe 2 (a))

  – Implementieren Sie die folgende Klasse *Uhr* in C

  - Die Operationen *zeitZeigen*, *minZeigen*, *stdZeigen* geben jeweils die aktuell eingestellte Uhrzeit *(std::min::sek)*, die Stunden *(std::)* und die Minuten *(::min)* auf dem Bildschirm aus.
  - Die Operationen *incStd, incMin, inc* erhöhen jeweils die eingestellte Stunde, Minute bzw. die Uhrzeit (d.h. die Sekunden) um 1.
  - Die Uhr wird gesteuert durch drücken von „Knöpfen“: A, B und Z.
  - Um den Verlauf der „echten“ Zeit zu verfolgen, hat die Uhr einen Zeitgeber; hierzu kann ein Objekt der Klasse Zeit verwendet werden (Quellcode: → moodle)



Zustandsautomaten in C

- Aufgabe (Forts.)
  - Zustandsautomat für die Operation *run*
    - Die Operation *run()* beschreibt das Verhalten eines Objekts der Klasse Uhr und ist durch einen Zustandsautomaten beschrieben
    - Besonderheiten gegenüber Übung 3, Aufgabe 2 (a))
      - Neben Stunden und Minuten zeigt die Uhr auch einen Sekundenwert an
      - Zu Beginn wird die eingestellte Uhrzeit auf den Wert *00::00 ::00* gesetzt (Operation *init()*)
      - Bei Eintritt in den Zustand Anzeige wird in der globalen Variable (quantitativer Zustand) *tick* die aktuelle Sekunde vom *zeitgeber* übernommen
      - Immer wenn sich dieser Wert verändert, wird die eingestellte Uhrzeit inkrementiert (Operation *inc*)
    - Hinweise
      - Simulieren das Drücken der Knöpfe durch entsprechende Tastatureingaben
      - Bei der Implementierung der *inc*-Operationen sind die Grenzen 24 Std. bzw. 60 Min. bzw. 60 Sek. zu beachten