3 Vererbung $\leftrightarrow$ Templates

Situation:

Es wird ein Datentyp A benötigt, der sehr ähnlich ist einem bereits definierten Datentyp B ($\rightarrow$ Klassen).

A soll

- einen Teil der Daten/Operationen wiederverwenden
- andere hinzufügen
- einige verändern (auf andere Wege implementieren)

Beispiel 3.1

Eine existierende Klasse: Polygon (B)
Eine neue Klasse: Rechteck (A)
Es handelt sich hierbei um eine Spezialisierung, da ein Rechteck ein spezielles Polygon ist.
Rechteck könnte alle Polygon-Operationen übernehmen (z.B. draw(), translate(dx,dy), rotate(x,y,ρ), area(), ...)

Einige Operationen können effizienter implementiert werden: Z.B. Flächenberechnung: area()
Andere Operationen sind nur für Rechtecke definiert: Z.B. Seitenverhältnis: ratio()

Allgemein: Klasse A (im Beispiel Rechteck) wird von Klasse B (im Beispiel Polygon) abgeleitet. B nennt man dabei die Basisklasse und A die abgeleitete Klasse.

Java

```
class A extends B{
   // Erweiterung: A-Teil
}
```

C++

```
class A: public B{
   // B-Teil:
   // Alle Daten und Methoden werden übernommen

   // A-Teil:
   // Zusätzliche Daten/Methoden,
   // modifizierte Operationen
}
```

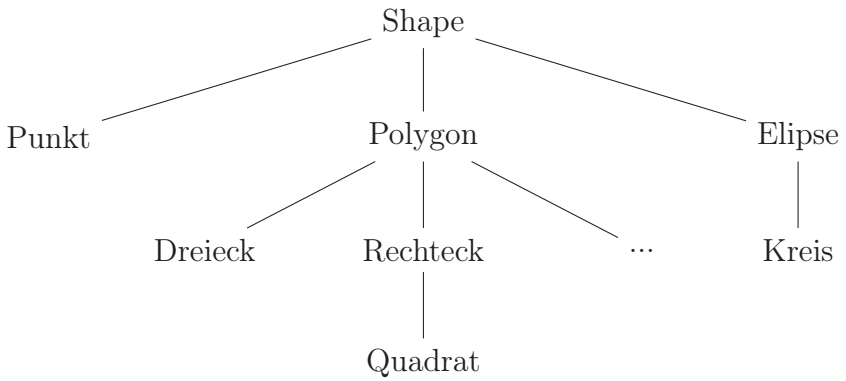
```
class Polygon{
   private // Daten

   double area();
   double umfang();
   void translate(double dx, double dy);
   void draw(...);
   ...
}
```

```
class Rechteck: public Polygon{
   // Konstruktoren -> Übung
   private:
      double b, h; // A-Teil
      // Überschreiben:
         double area(){return b*h;}
         double umfang(){return 2+(b+h);}
      // Hinzufügen:
      double ratio(){return b/h;}
}
```

Jedes Rechteck ist ein spezielles Polygon.

3.1 Allgemeiner: Ableitungsbaum



3.1.1 Typverträglichkeit

In C++ kann einer Variable von Typ B* (Pointer) oder B& (Referenz) ein Objekt (Pointer/Referenz) vom Typ A zugewiesen werden.
In Java sind alle Variablen vom Typ B.

C++

```
Polygon* p = new Rechteck(...);
void func(Polygon& poly){...}
Rechteck rect(...);
func(rect);
```

Java

```
Polygon p = new Rechteck(...);
```

Genauer: Alle im Ableitungsbaum verfügbaren Typen können zugewiesen werden.

Variable/Parameter hat zwei Typen:

- statischer Typ (zur Compile-Zeit bekannt (z.B. Polygon))
  → bestimmt, welche Methoden zur Verfügung stehen
- dynamischer Typ (zur Laufzeit bekannt (z.B. Rechteck))
  → bestimmt die Implementierung der Methoden

3.2 Polymorphe Datenstrukturen

Ein Beispiel wäre eine Feld von Polygonen.

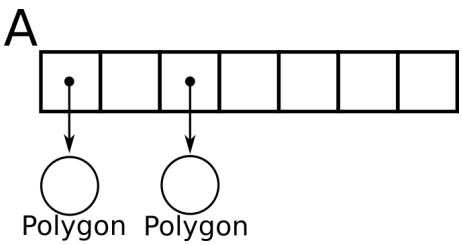
C++

```
Polygon** A = new Polygon*[100];
```

Java

```
Polygon[] A = new Polygon[100];
```

Polgon** A entspricht einem Feld von Pointern.




```
A[0] = new Polygon(...);
A[1] = new Rechteck(...);
A[2] = new Dreieck(...);
A[3] = new Quadrat(...);
```

→ polymorphes Array

Beispiel 3.2 Berechnung der Gesamtfläche

```
double total_area(Polygon** A){ // Java: (Polygon[] A)
   double total = 0;
   for (int i = 0; i < 100; i++){
      total += A[i]->area();
      // Oder: total += (*A[i]).area();
   }
   return total;
}
```

Der dynamische Typ bestimmt, welche Area-Implementierung verwendet wird.
→ Dynamische Bindung

Dynamische Bindung

```
polygon * p = new rechteck(...)
```

oder

```
double func(polygon& poly){
   return poly.area();
}
rechteck rect(...)
func(rect)
```

In C++ ist die statische Bindung der Standard. Für die dynamische Bindung wird eine virtuelle Methode benötigt.

```
virtual double area(){...}
```

`virtual` ist dabei das Schlüsselwort für die dynamische Bindung.

Beispiel 3.3 Grafik-Editor (erweiterbar)

Der Grafik-Editor speichert eine Menge von allgemeinen Flächen (→ shape)
Abstrakte Klasse „shape“

C++

```
class chape{
   virtual void draw(...) = 0
   virtual void area() = 0
   virtual void rotate(double phi, ...) = 0
}
```

Java

```
abstract class shape{
   abstract void draw(...);
   ...
}
```

Abstrakte Klassen werden verwendet, um Interfaces zu definieren.
Der Grafik-Editor speichert eine Liste von `shape*`.
`shape * p = new shape();` ergibt einen Fehler, da `shape` abstrakt ist.

Wenn `polygon` von `shape` abgeleitet ist:
C++

```
shape * p = new polygon(...)
```

```
class polygon: public shape{
   double area(){...}
}
```

Alle rein virtuellen (abstrakten) Methoden müssen definiert werden.

Java:

```
class polygon implements shape{...}
```

```
Interface shape{
   void draw();
}
```

Im Editor
`void draw_objects(){`
Iteriere über alle shapes s und rufe `s.draw()` auf.

Anwendung auf Algorithmen
Lineare Ordnung durch ein Interface (Java comparable)

C++

```
class comparable{
   virtual int compare (comparable& x) = 0;
```

Die Funktion vergleicht das Objekt, welches die Funktion aufruft, mit x und liefert

$$\begin{cases} +1, & \text{falls dieses } > x \\ 0, & \text{falls dieses } = x \\ -1, & \text{falls dieses } < x \end{cases}$$

Anwendung in generischen Sortieralgorithmen
Quicksort(comparable * A[])
zum Vergleich if(x.compare(y) ≤ 0) ...
Sortiere ein Feld von point:

```
class point : public comparable{
   point ...
   int compare(comparable & p){}
      // p ist tatsächlich ein Point!
   double px = ((point&) p).x;
   double py = ((point&) p).y;
   (x,y) < (px,py)
   // lexikographische Ordnung
   return ... ;
};
```

Aufruf von Quicksort

```
point* A [] = new point*[100];
(point**)
for( i=o; i<100; i++){
   A[i] = new point(i, i*i);
}
Quicksort(A, 100);
```

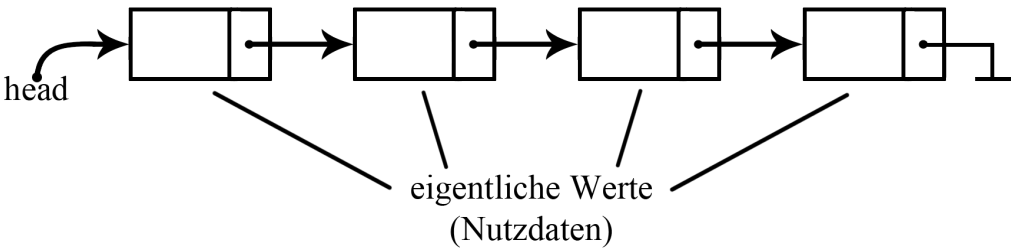
Datenstrukturen: → binäre Suchbäume, Schlüssel vom Typ „comparable“

3.3 Weitere Anwendungen von Vererbung

3.3.1 Generische Datenstrukturen

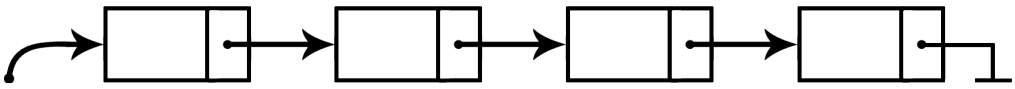
Eine generische Datenstruktur ist zum Beispiel eine Liste von beliebigen Objekten. Diese Liste soll leicht wiederverwendbar sein z.B. eine einfach verkettete Liste. Dadurch muss sie nicht immer neu implementiert werden.

Einfach verkettete Listen



Beobachtung:
Implementierung der Operationen (push, pop, ...) ist unabhängig vom Typ der Werte (int, string, point, ...).

Abstrahieren: Liste ohne Werte



1.) Basisklasse für allgemeine Listenelemente

```
class slist_elem{
   slist_elem* next;
   slist_elem(slist_elem* p){
      next = p;
   }
}
```

2.) Basisklasse für allgemeine Liste

```
class slist{
   slist_elem* first;
   slist(){
      first = NULL;
   }

   // hier eigentlich Destruktor

   void push(slist_elem * p){
      p->next = first
      first = p;
   }

   slist_elem* pop(){
      if(first == NULL) return first;
      slist_elem* p = first;
      first = first.next;
      return p;
   }
}
```

`slist` funktionier auch für alle von slist_elem abgeleiteten Klassen.

Beispiel 3.4 Liste von points

```
class point: public slist_elem{
   // wie vorher ...
}
point p
```

```
class point_list: public slist{
   // neues Interface, das nur Points erlaubt
   // -> Filter
   void push(point* p) {
      slist :: push(p);
   }
   point* pop(){
```