
Algorithmen und Datenstrukturen

Eine systematische Einführung in die

Programmierung

11 Suchalgorithmen

Josef Pichler

Version 10.1, 2024

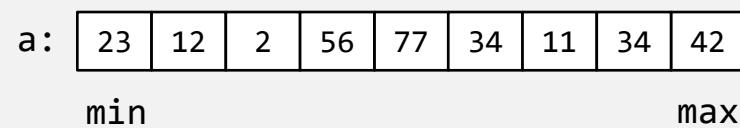
Einleitung

- Sinn und **Zweck der Speicherung** von Datenbeständen ist es, später auf sie als Ganzes oder auf Teile davon **zugreifen** zu können
- Dabei ist es oft erforderlich, nach jenen Datenobjekten zu suchen, die ein bestimmtes **Kriterium** erfüllen
- Die Suche nach Datenobjekten, zum Zweck der Informationsbeschaffung, ist ein **zentraler Bestandteil** nahezu jedes Softwaresystems. Es besteht daher ein besonderes Interesse an möglichst **effizienten Suchverfahren**
- Im Allgemeinen haben Datenobjekte mehrere **Merkmale** (Attribute), von denen im Kontext einer **bestimmten Suchaufgabe** einem die (Schlüssel-) Rolle zukommt, d. h. nach dessen Ausprägung (Wert) soll gesucht werden. Man nennt dieses Merkmal deshalb in der Regel auch **Suchschlüssel** (**key**)

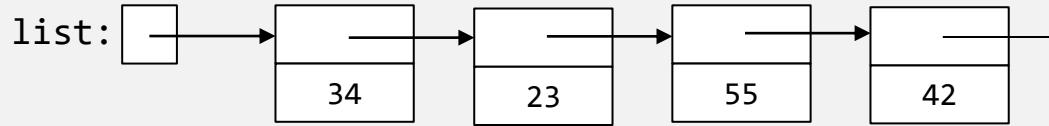
11.1 Sequentielle Suche

- Die sequentielle Suche (*sequential search*) ist das **einfachste** aber auch das **ineffizienteste** Suchverfahren
 - Es hat jedoch den Vorteil, dass es bei **jeder Art von Behältern**, unabhängig von der verwendeten Datenstruktur und der Anordnung der Elemente (also insbesondere auch bei **unsortierten Behältern**) anwendbar ist

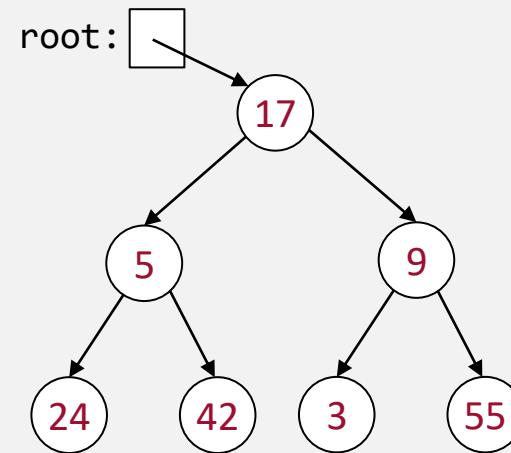
Feld:



Verkettete Liste:



Binärbaum:



(1) Sequentielle Suche in Feldern

Gegeben seien folgende Datentypen

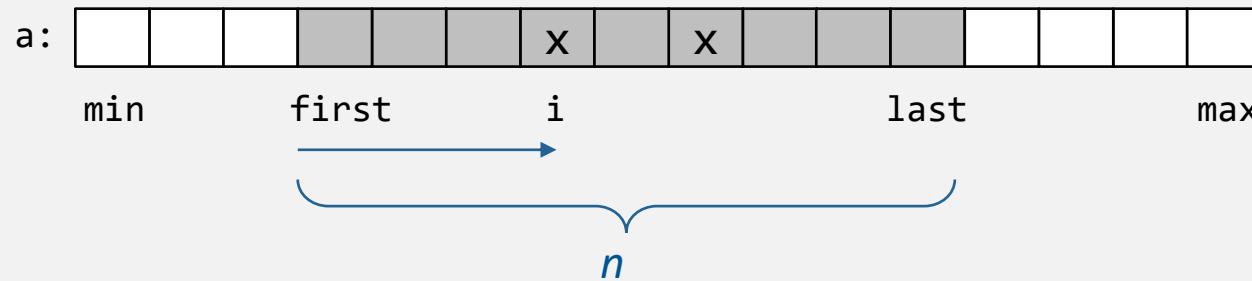
```
type
  KeyType = ... -- type which allows operation =
                -- and operation < for "sorted" containers
ObjectType = compound
  key: KeyType
  data1: DataType1    -- any type
  data2: DataType2    -- any type
  ...
  dataN: DataTypeN    -- any type
end -- ObjectType

const
  min = ...
  max = ... -- min ≤ max

type
  ElementType = ObjectType
  ArrayType = array [min:max] of ElementType
```

Sequentielle Suche in Feldern

Aufgabe: Suche nach dem Index des ersten Elements mit dem Schlüsselwert x in einem Teilbereich eines Felds



Gesucht ist ein Algorithmus `FirstIndexOf`, der in dem durch die beiden Grenzindizes `first` und `last` (mit $\min \leq \text{first} \leq \text{last} \leq \max$) festgelegten Teilbereich des Felds a „von vorne weg“ sequentiell nach dem Index des ersten Elements sucht, das in seiner Komponente `key` den gesuchten Schlüsselwert x enthält.

Der Algorithmus soll als Ergebnis den Wert $\min - 1$ liefern, falls kein solches Element gefunden werden konnte.

Sequentielle Suche in Feldern

Algorithmus:

```
FirstIndexOf(↓a: ArrayType ↓first: int ↓last: int ↓x: KeyType): int
    var
        i: int
    begin
        i := first
        while (i ≤ last) and (a[i].key ≠ x) do
            i := i + 1
        end -- while
        if i ≤ last then
            return i
        else
            return min - 1
        end -- if
    end FirstIndexOf
```

(2) Sequentielle Suche in verketteten Listen

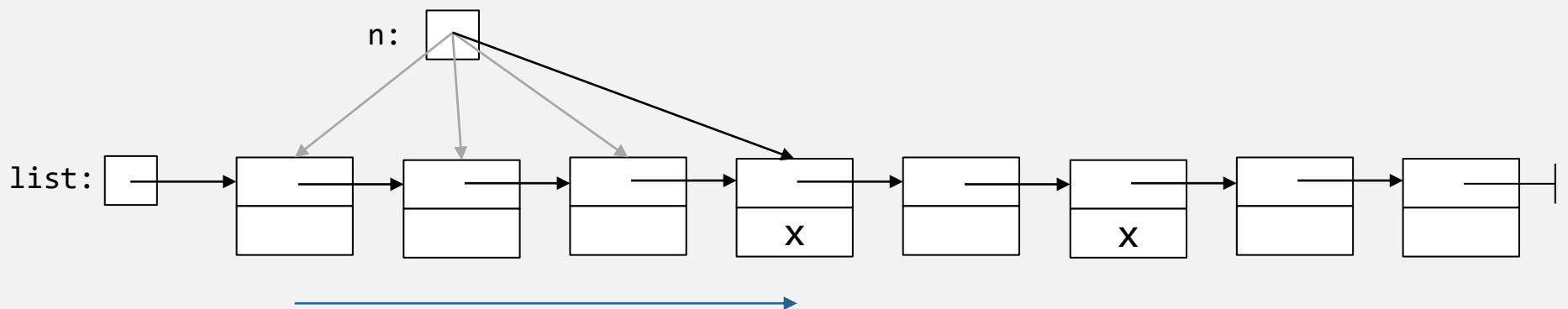
Gegeben seien folgende Datentypen

```
type
  ListNodePtr = →ListNode
  ListNode = compound
    next, prev: ListNodePtr
    key: KeyType
    dataI: DataTypeI
  end -- ListNode
  ListPtr = ListNodePtr
```

Gesucht ist ein Algorithmus `FirstNodeWith` der in einer einfach- oder doppelt-verketteten Liste `list` sequentiell nach dem ersten Knoten sucht, der in der Komponente `key` den gesuchten Wert `x` enthält.

Der Algorithmus soll als Ergebnis den Wert `null` liefern, falls kein solcher Knoten gefunden werden konnte.

Sequentielle Suche in verketteten Listen



Algorithmus:

```
FirstNodeWith(↓list: ListPtr ↓x: KeyType): ListNodePtr
  var
    n: ListNodePtr
begin
  n := list
  while (n ≠ null) and (n→key ≠ x) do
    n := n→next
  end -- while
  return n
end FirstNodeWith
```

(3) Sequentielle Suche in Binärbäumen

Gegeben seien folgende Datentypen

```
type
    TreeNodePtr = →TreeNode
    TreeNode = compound
        left, right: TreeNodePtr
        key: KeyType
        dataI: DataTypeI
    end -- TreeNode
    TreePtr = TreeNodePtr
```

Gesucht ist ein (rekursiver) Algorithmus `FirstInOrderNodeWith` der in einem Binärbaum `tree` in der durch den In-Order-Durchlauf gegebenen Reihenfolge sequentiell nach dem ersten Knoten sucht, der in der Komponente `key` den gesuchten Wert `x` enthält.

Der Algorithmus soll als Ergebnis den Wert `null` liefern, falls kein solcher Knoten gefunden werden konnte.

Sequentielle Suche in Binärbäumen

Algorithmus

```
FirstInOrderNodeWith(↓tree: TreePtr ↓x: KeyType): TreeNodePtr
    var
        n: TreeNodePtr
    begin
        if tree ≠ null then
            n := FirstInOrderNodeWith(↓tree→left ↓x)
            if n ≠ null then
                return n
            end -- if
            if tree→key = x then
                return tree
            end -- if
            n := FirstInOrderNodeWith(↓tree→right ↓x)
            if n ≠ null then
                return n
            end -- if
        end -- if
        return null
    end FirstInOrderNodeWith
```

1. look in left subtree

2. look in root node

3. look in right subtree

Laufzeitkomplexität sequentieller Suchalgorithmen

Im günstigsten Fall: Die minimale Anzahl von Suchschritten ist $S_{min}(n) = 1$.

- Dieser Fall tritt ein, wenn der zu durchsuchende Bereich des Behälters nur ein Element umfasst oder der gesuchte Schlüsselwert gleich im ersten besuchten Element gefunden wird.

Im ungünstigsten Fall: Die max. Anzahl von Suchschritten ist $S_{max}(n) = n$.

- Dieser Fall tritt ein, wenn der gesuchte Schlüsselwert in keinem der Elemente vorkommt oder erst im letzten besuchten Element gefunden wird.

Im typischen oder durchschnittlichen Fall:

- Wenn wir annehmen, dass alle n zu durchsuchenden Elemente des Behälters unterschiedliche Schlüsselwerte enthalten und nach jedem Schlüsselwert genau einmal gesucht wird, ergibt sich folgende

$$\text{Gesamtanzahl von Suchschritten } S_{total}(n) = 1 + 2 + \dots + n = \frac{n(n+1)}{2}.$$

Laufzeitkomplexität sequentieller Suchalgorithmen

Im typischen oder durchschnittlichen Fall

- Wenn wir weiter annehmen, dass nach jedem Schlüsselwert mit der gleichen Wahrscheinlichkeit gesucht wird, so beträgt die durchschnittliche Anzahl von Suchschritten

$$S_{avg}(n) = \frac{S_{total}}{n} = \frac{1}{n} \cdot \frac{n(n+1)}{2} = \frac{n+1}{2}$$

- Für große n gilt also

$$S_{avg}(n) \approx \frac{n}{2}$$

- Die asymptotische Laufzeitkomplexität der sequentiellen Suchalgorithmen ist demnach $O(n)$, sie ist also linear.
- Das bedeutet z. B. dass die Suche in einem Behälter mit doppelt so vielen Elementen, doppelt so viele Suchschritte erfordert.

11.2 Binäre Suche

Eine deutliche Verbesserung des Laufzeitverhaltens gegenüber jenem der sequentiellen Suche, erreichen wir durch ein Suchverfahren, das nach dem **Prinzip Teile und Herrsche** (*divide and conquer*) arbeitet:

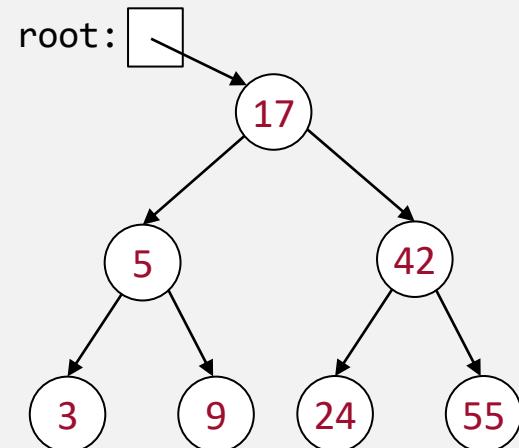
das binäre Suchen (*binary search*)

Voraussetzung für die Anwendung dieses Suchverfahrens ist allerdings, dass die Elemente im Behälter **sortiert angeordnet** sind, und ein direkter Zugriff auf das „mittlere“ Element möglich ist.

Sortiertes Feld:

a:	2	11	12	23	34	34	42	56	77
	min					max			

Binärer Suchbaum:



Binäres Suchen

Lösungsidee

- Es wird geprüft, ob der gesuchte Schlüsselwert mit dem des mittleren Elementes übereinstimmt
- Wenn das nicht der Fall ist, kann wegen der Sortierung der Behälterelemente durch den Vergleich des gesuchten Schlüsselwerts mit dem Wert der Schlüsselkomponente des gerade betrachteten Elements feststellt werden, ob in der „linken Hälfte“ oder in der „rechten Hälfte“ des Behälters analog (ev. rekursiv) weitergesucht werden muss
- Dadurch vermindert sich nach jedem nicht erfolgreichen Suchschritt die Anzahl der noch zu durchsuchenden Elemente um die Hälfte, was zu einer signifikanten Reduktion der Suchschritte im Vergleich zur sequentiellen Suche führt

(1) Binäre Suche in einem Feld

Die Transformation der Lösungsidee des binären Suchens in einem als Feld organisierten (sortierten) Datenbestandes führt zu folgendem Algorithmus:

```
IndexOf(↓a: ArrayType ↓first: int ↓last: int ↓x: KeyType): int
    var
        mid: int
    begin
        while first ≤ last do
            mid := (first + last) div 2
            if x = a[mid].key then
                return mid
            else
                if x < a[mid].key then
                    last := mid - 1
                else -- x > a[mid].key
                    first := mid + 1
                end -- if
            end -- if
        end -- while
        return min - 1
    end IndexOf
```

(2) Binäre Suche in einem binären Suchbaum

Gegeben seien folgende Datentypen

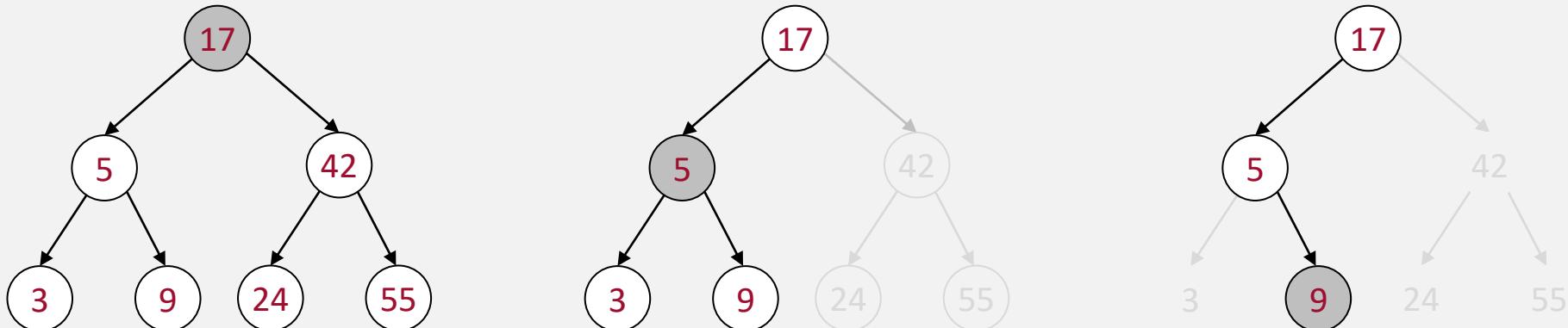
```
type
    TreeNodePtr = →TreeNode
    TreeNode = compound
        left, right: TreeNodePtr
        key: KeyType
        dataI: DataTypeI
    end -- TreeNode
    TreePtr = TreeNodePtr
```

Auf Basis dieser Deklarationen kann, wenn ein nach den Werten der Schlüsselkomponente (key) sortierter binärer Suchbaum tree vorliegt, ein einfacher rekursiver Algorithmus NodeWith formuliert werden, mit dem ein Knoten, dessen Komponente key den gesuchten Wert x enthält, gefunden werden kann

Binäre Suche in einem binären Suchbaum

Grundprinzip der Lösungsidee für den Algorithmus NodeWith ist, dass im jeweils betrachteten Baum `tree`, sofern dieser nicht leer ist, zunächst der Wurzelknoten untersucht wird, und wenn der gesuchte Wert darin nicht enthalten ist, die Suche durch einen rekursiven Aufruf entweder im linken (`tree→left`) oder im rechten Teilbaum (`tree→right`) fortgesetzt wird, je nachdem, ob der gesuchte Wert x kleiner oder größer als der Wert der Komponente `key` im Wurzelknoten ist

Beispiel: Suche nach Knoten mit dem Schlüsselwert $x = 9$



Binäre Suche in einem binären Suchbaum

Der Algorithmus `NodeWith` der diese Lösungsidee umsetzt lautet:

```
NodeWith(↓tree: TreePtr ↓x: KeyType): TreeNodePtr
begin
    if tree = null then
        return null
    elsif tree→key = x then
        return tree
    elsif x < tree→key then
        return NodeWith(↓tree→left ↓x)
    else -- x > tree→key
        return NodeWith(↓tree→right ↓x)
    end -- if
end NodeWith
```

Laufzeitkomplexität binärer Suchalgorithmen

Im günstigsten Fall: Die minimale Anzahl von Suchschritten ist $S_{min}(n) = 1$.

- Dieser Fall tritt ein, wenn der zu durchsuchende Bereich des Behälters nur ein Element umfasst oder der gesuchte Wert auf Anhieb im mittleren Element des Felds bzw. im Wurzelknoten des Suchbaums gefunden wird.

Im ungünstigsten Fall: Die maximale Anzahl von Suchschritten ist

$$S_{max}(n) = \lfloor \lg(n) \rfloor + 1.$$

- Dieser Fall tritt ein, wenn der gesuchte Wert nicht im zu durchsuchenden Behälter enthalten ist oder erst im letzten besuchten Element/Knoten gefunden wird.

Im typischen oder durchschnittlichen Fall:

- Der Einfachheit halber betrachten wir den Spezialfall eines sortierten Felds oder eines balancierten binären Suchbaums mit $n = 2^k - 1$ (für $k > 1$) Elementen/Knoten und nehmen an, dass die n Elemente/Knoten des Behälters in ihren Schlüsselkomponenten unterschiedliche Werte enthalten.

Laufzeitkomplexität binärer Suchalgorithmen

- Die Gesamtzahl der Suchschritte S_{total} , die notwendig sind, um jedes Element bzw. jeden Knoten der beiden Behälter zu lokalisieren, beträgt:

$$\begin{aligned} S_{total}(2^4 - 1) &= 1 \cdot 1 + 2 \cdot 2 + 3 \cdot 4 + 4 \cdot 8 && \Rightarrow \text{Grobanalyse,} \\ &= 1 \cdot 2^0 + 2 \cdot 2^1 + 3 \cdot 2^2 + 4 \cdot 2^3 = 49 && \text{Kapitel 8!} \end{aligned}$$

- Verallgemeinern wir das Berechnungsergebnis (mit $n = 2^k - 1$ und n unterschiedlichen Schlüsselwerten) so erhalten wir:

$$S_{total}(2^k - 1) = \sum_{i=1}^k i \cdot 2^{i-1} = \dots = (k-1) \cdot 2^k + 1$$

- Unter der Annahme, dass nach jedem Element des Behälters mit der gleichen Wahrscheinlichkeit gesucht wird, beträgt die durchschnittliche Anzahl von Suchschritten:

$$S_{avg}(2^k - 1) = \frac{S_{total}(2^k - 1)}{2^k - 1} = \frac{(k-1) \cdot 2^k + 1}{2^k - 1}$$

- Für große k gilt $S_{avg}(n) \approx (k-1)$ (+1, -1 vernachlässigen)

Laufzeitkomplexität binärer Suchalgorithmen

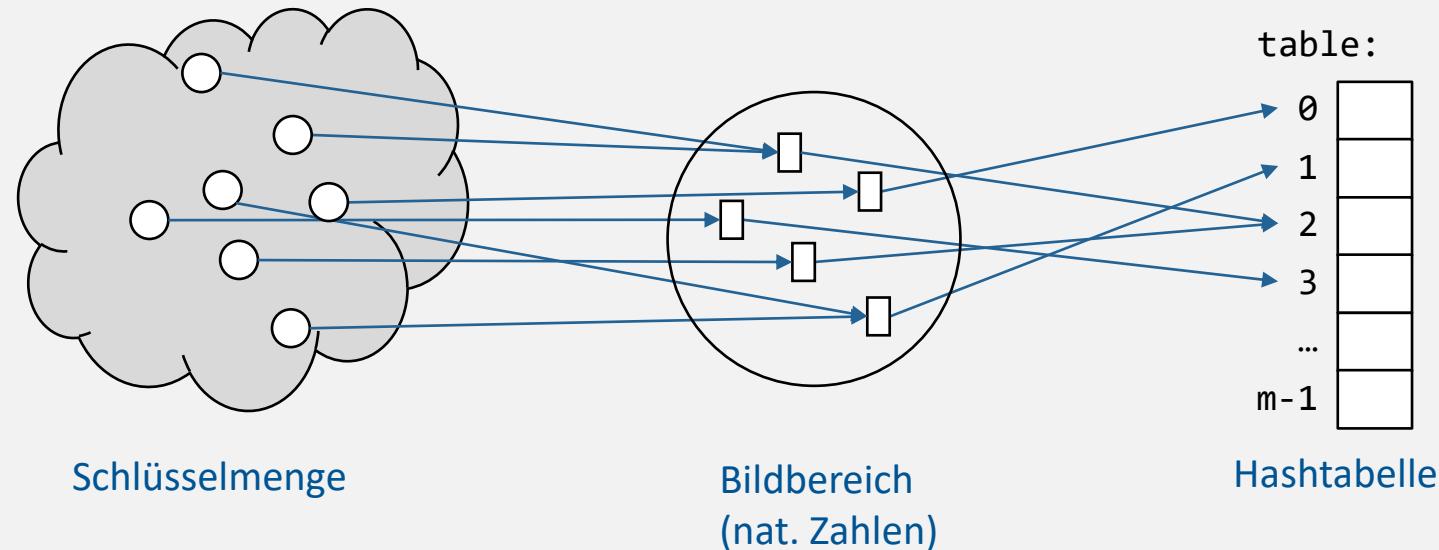
- Dieses Ergebnis bedeutet, dass sich das Laufzeitverhalten binärer Suchverfahren im durchschnittlichen Fall nur unwesentlich vom ungünstigsten Fall unterscheidet: nur etwa ein Suchschritt weniger wird benötigt.
- Die asymptotische Laufzeitkomplexität binärer Suchalgorithmen ist demnach $O(\lg n) = O(\log n)$, sie ist also logarithmisch.

11.3 Hashing-basierte Speicherung und Suche

- Bei den bisher erörterten Suchverfahren, der sequentiellen und der binären Suche, hängt der **Aufwand** für die Lokalisierung eines gesuchten Elements von der **Anzahl n der Elemente** im Behälter ab
 - Die asymptotische Laufzeitkomplexität von sequentiellen Suchverfahren ist $O(n)$.
 - Die asymptotische Laufzeitkomplexität binärer Suchverfahren ist günstiger, sie ist nur $O(\log n)$
- Obwohl es sich bei binären Suchverfahren bereits um effiziente Verfahren handelt, ist es wünschenswert und in bestimmten Fällen erforderlich, **noch effizientere Suchverfahren** zur Verfügung zu haben
- Wünschenswert wäre ein Suchverfahren, das **unabhängig von der Anzahl** der Elemente im Behälter mit einer asymptotischen Laufzeitkomplexität von annähernd $O(1)$ arbeitet, dessen Laufzeitverhalten also konstant ist

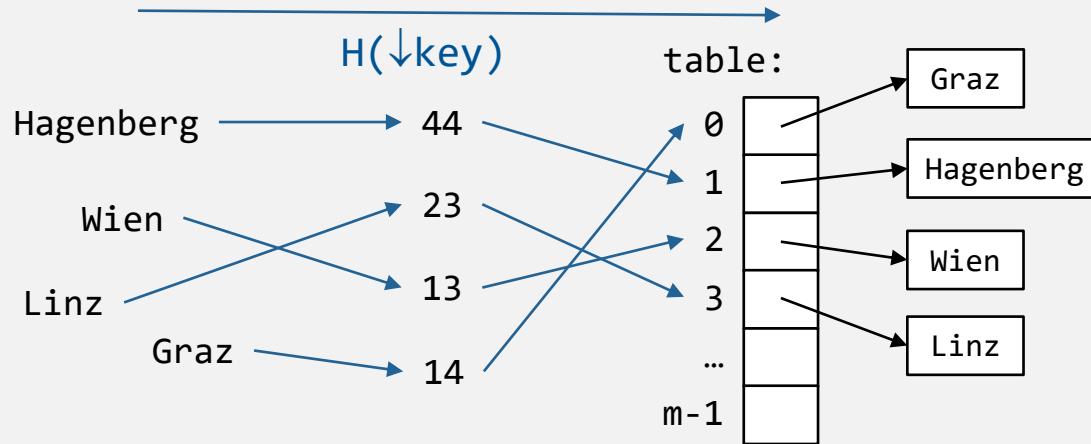
Grundidee des Hashing

- Das sogenannte **Hashing** (auch Streuspeicherung, *scatter storage* genannt), ist ein Verfahren, mit dem die Forderung einer asymptotischen Laufzeitkomplexität von $O(1)$ – unter bestimmten, aber realistischen Umständen – weitgehend erfüllt werden kann
- Hashing basiert auf der Idee, dass man bereits beim Speichern von Datenobjekten in einem Behälter dafür Sorge trägt, dass die einzelnen Datenobjekte in Abhängigkeit von einem Suchschlüssel schnell, möglichst auf Anhieb, gefunden werden können



Grundidee des Hashing

- Verwendung eines Felds fixer Größe m
- Transformation des Schlüsselinhalts in eine natürliche Zahl
- Abbildung der Zahl auf das Index-Intervall $[0..m-1]$
- Speichern der Datenobjekte, sodass Zugriff über berechn. Feldindex möglich ist



- Hashing bedeutet also die Abbildung von Schlüsselwerten auf einen Wert (Index) aus dem Intervall $[0..m-1]$
- Die Funktion dazu heißt Hashfunktion $H(\downarrow \text{key})$: int, ihr Ergebnis nennen wir Hashcode

Bildung einer Hashtabelle

- Gegeben seien Datenobjekte in Form von Knoten mit einer Schlüsselkomponente vom Datentyp string

```
type  
  NodePtr = →Node  
  Node = compound  
    key: string  
    data: ...  
  end -- Node
```

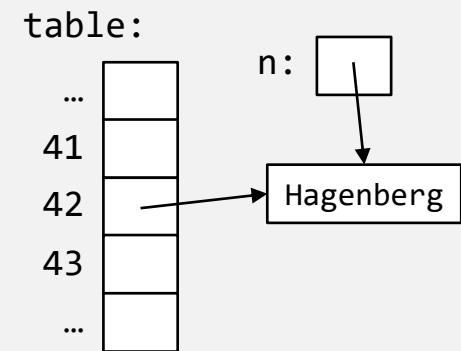
- Gebildet wird eine Hashtabelle, d. h. ein Feld fixer Größe

```
const m = ...  
type HashTable = array [0:m-1] of NodePtr  
var table: HashTable
```

- Die Elemente repräsentieren Zeiger auf Knoten

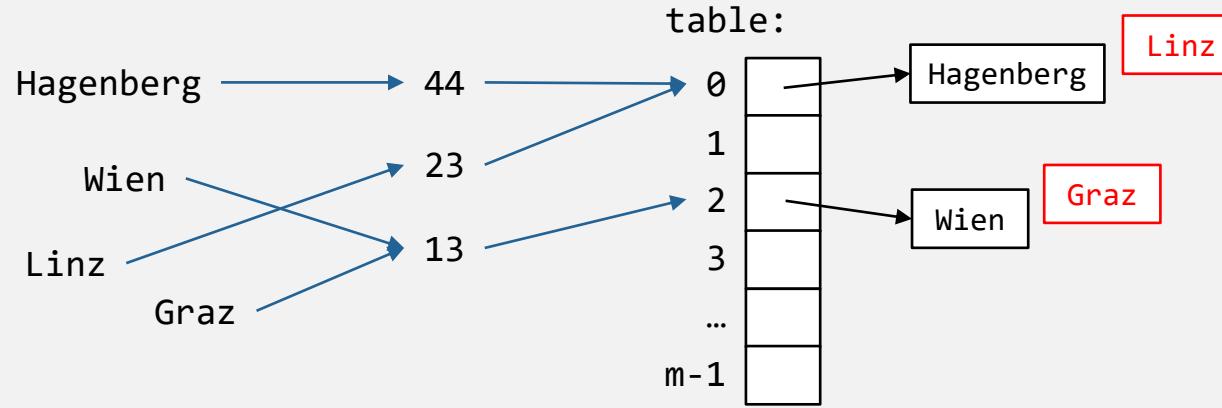
```
var  
  n: NodePtr
```

```
n := New(↓Node)  
n→key := "Hagenberg"  
table[42] := n
```



Das Problem der Kollisionen

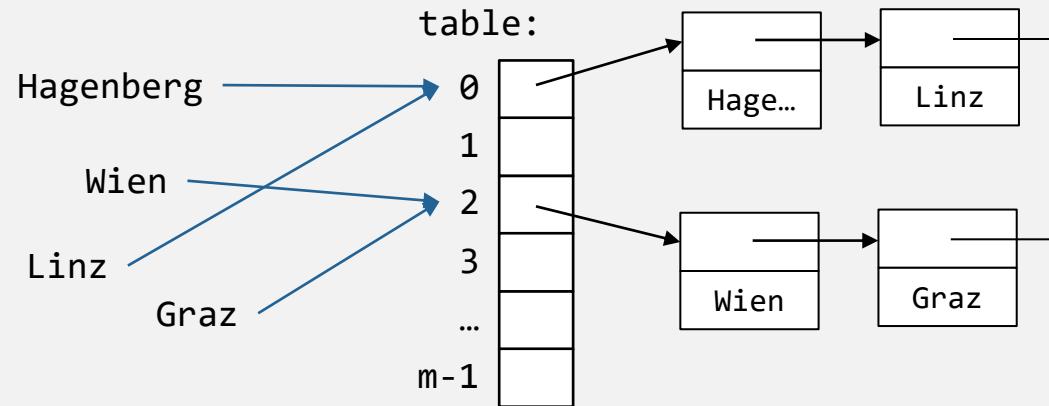
- Wenn für Schlüsselwerte $k_1 \neq k_2$ die Hashfunktion H identische Ergebnisse liefert, $H(k_1) = H(k_2)$, nennen wir dies eine Kollision
- Verschiedene Schlüsselwerte können auf die gleiche Zahl und/oder verschiedene Zahlen können auf den gleichen Index abgebildet werden



- Lösung des Kollisionsproblems
 - durch Verkettung von Kollisionsobjekten
 - durch offene Adressierung

11.3.1 Kollisionsbehandlung: Verkettung von Kollisionsobj.

Datenobjekte mit gleichem Hashcode werden in einer einfach-verketteten Liste gespeichert



Zusätzliche Komponente `next` erforderlich

```
type
    NodePtr = →Node
    Node = compound
        next: NodePtr
        key: string
        data: ...
    end -- Node
```

Speicherung von Datenobjekten

Deklarationen

```
const  
  m = ...  
type  
  HashTable = array [0:m-1] of NodePtr  
var  
  table: HashTable
```

Gedächtnisvariable

Initialisierung/Hilfsalgorithmen

```
InitHashTable()  
  var  
    i: int  
begin  
  for i := 0 to m - 1 do  
    table[i] := null  
  end -- for  
end InitHashTable
```

```
NewNode(↓key: string): NodePtr  
  var  
    n: NodePtr  
begin  
  n := New(↓Node)  
  n→key := key  
  n→next := null  
  return n  
end NewNode
```

Speicherung von Datenobjekten

Algorithmus

```
Insert(↓key: string ↓data:...)
  var
    n: NodePtr; i: int
begin
  i := H(↓key)
  if table[i] = null then
    table[i] := NewNode(↓key)
    table[i]→data := data
  else
    n := table[i]
    while (n→key ≠ key) and (n→next ≠ null) do
      n := n→next
    end -- while
    if n→key ≠ key then
      n→next := NewNode(↓key)
      n→next→data := data
    end -- if
  end -- if
end Insert
```

noch kein Eintrag mit Hashcode i

suche ob Knoten für Schlüssel key bereits vorhanden

noch kein Eintrag eines Datenobjektes mit Schlüssel key

Suchen des Datenobjekts mit Schlüsselwert key

Algorithmus

```
Find(↓key: string): NodePtr
    var
        n: NodePtr
    begin
        n := table[H(↓key)]
        while (n ≠ null) and (n→key ≠ key) do
            n := n→next
        end -- while
        return n
    end Find
```

suche Knoten mit
Schlüsselwert key

Optimierungsmöglichkeit

- Datenobjekte sortiert nach Schlüsselwerten in Listen einfügen
- Suche kann abgebrochen werden, wenn aktueller Schlüsselwert größer als gesuchter Schlüsselwert ist

Entfernen des Datenobjekts mit Schlüsselwert key

Algorithmus

```
Delete(↓key: string)
  var
    i: int; n, prev: NodePtr
begin
  i := H(↓key)
  n := table[i]; prev := null
  while (n ≠ null) and (n→key ≠ key) do
    prev := n
    n := n→next
  end -- while
  if n ≠ null then
    if prev = null then
      table[i] := n→next
    else
      prev→next := n→next
    end -- if
    Dispose(↓n)
  end -- if
end Delete
```

prev zeigt auf Vorgänger des Knoten mit Schlüssel

erster Knoten wird entfernt

innerer oder letzter Knoten wird entfernt

Zur Wahl der Größe der Hashtabelle

- Unter Annahme einer Gleichverteilung der Hashcodes ist die durchschnittliche Listenlänge $L = n/m$
 - mit n = Anzahl der Datenobjekte und m = Tabellengröße
- Kennt man n , kann man m passend wählen



11.3.2 Kollisionsbehandlung: Offene Adressierung

- Datensätze auch bei Kollisionen in Tabelle speichern (für $n < m$)
- Verschiedene Strategien um "Ersatzposition" zu finden

Lineare Kollisionsstrategie

- Wenn Position belegt ist, wird erste freie Position dahinter gesucht

```
KeyPos(↓key: string): int
    var
        i: int
    begin
        i := H(↓key)
        while (table[i] ≠ null) and (table[i]→key ≠ key) do
            i := (i + 1) mod m
        end -- while
        return i
    end KeyPos
```

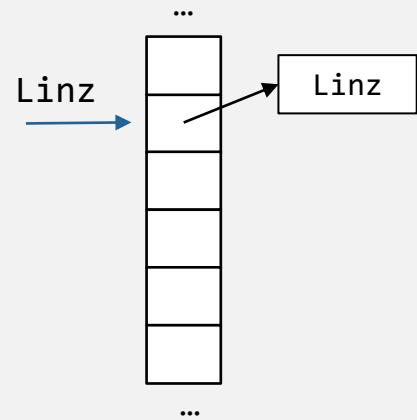
Ohne Prüfung, ob überhaupt noch
eine Position frei ist!

Einfügen und Suchen von Datenobjekten

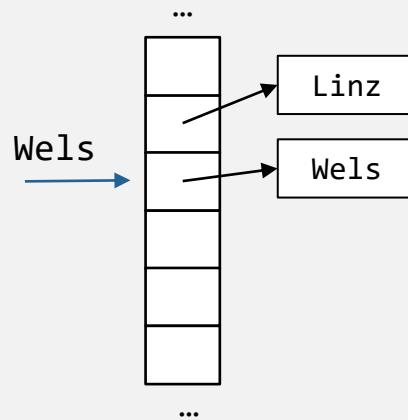
Algorithmus

```
Insert(↓key: string)
begin
    i := KeyPos(↓key)
    table[i] := NewNode(↓key)
end Insert
```

Beispiel

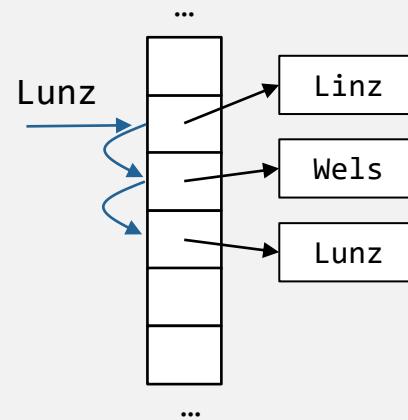


1. Insert Linz



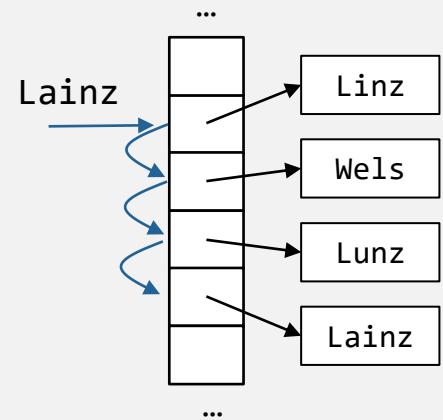
2. Insert Wels

$$H("Wels") \neq H("Linz")$$



3. Insert Lunz

$$H("Lunz") = H("Linz")$$



4. Insert Lainz

$$H("Lainz") = H("Linz")$$

Einfügen und Suchen von Datenobjekten

Algorithmus

```
Find(↓key: string): NodePtr
begin
    i := KeyPos(↓key)
    return table[i]
end Find
```

- Bei dieser „linearen“ Platzierung und Suche ist nach Morris die mittlere Anzahl e von erforderlichen Versuchen das $n + 1$. Datenobjekt zu speichern oder zu finden

$$e = \frac{1 - \frac{a}{2}}{1 - a} \text{ mit } a = \frac{n}{m}$$

- Daraus folgt, dass die mittlere Anzahl von Suchschritten nicht von der Anzahl der Datenobjekte, sondern nur vom **Füllfaktor a** der Hashtabelle abhängt
- Der Füllfaktor lässt sich steuern

Einfügen und Suchen von Datenobjekten

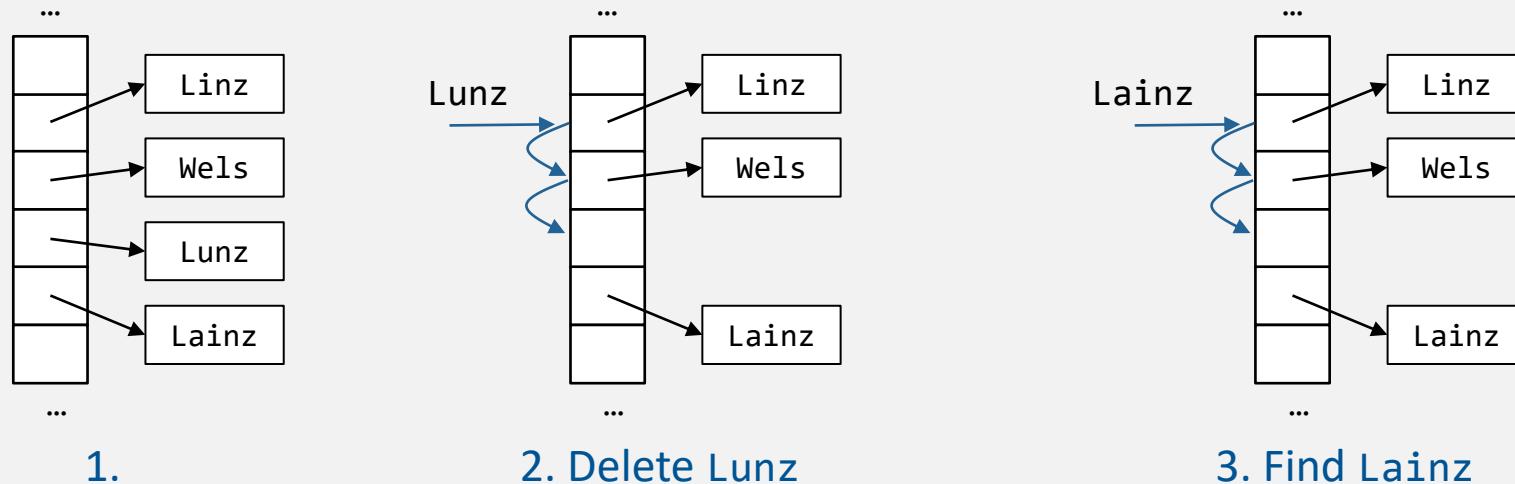
- Man kann auch so vorgehen, dass bei einer Kollision nicht die nächste freie Zelle der Hashtabelle gewählt wird, sondern die Adresse mit einem Zufallszahlengenerator bestimmt wird
- Ist auch diese Zelle belegt, wird der Vorgang solange wiederholt, bis eine leere Zelle gefunden ist
- Der Zufallszahlengenerator muss so konstruiert sein, dass er alle Zellen der Tabelle einmal berührt
- Zum Suchen muss derselbe Generator verwendet werden, wie zum Speichern (gleicher Startwert)

a	lineares Speichern	zufälliges Speichern
0.10	1.06	1.05
0.50	1.5	1.39
0.75	2.5	1.83
0.90	5.5	2.56

Entfernen von Datenobjekten

- Nicht trivial
- Eintrag darf nicht auf null gesetzt werden, falls dahinter Datensätze mit gleichem Hashcode gespeichert sind

Beispiel



Lösungsmöglichkeiten

- spezieller Datensatz für gelöschte Elemente
- spezielles Flag für gelöschte Elemente

Spezielles Datenobjekt anstelle entfernter Datenobjekte

Deklaration

```
const  
    nodeDeleted: NodePtr = NewNode("")
```

Schlüsselwert darf sonst
nicht vorkommen

Algorithmus

```
Delete(↓key: string)  
begin  
    i := KeyPos(↓key)  
    if (table[i] ≠ null) and (table[i] ≠ nodeDeleted) then  
        Dispose(↓table[i])  
        table[i] := nodeDeleted  
    end -- if  
end Delete
```

Erfordert Änderungen in KeyPos sowie in Find

Änderungen in KeyPos sowie in Find

```
KeyPos(↓key: string): int
begin
    i := H(↓key); del := -1
    while (table[i] ≠ null) and (table[i]→key ≠ key) do
        if (del < 0) and (table[i] = nodeDeleted) then
            del := i
        end -- if
        i := (i + 1) mod m
    end -- while
    if (table[i] = null) and (del ≥ 0) then
        return del
    else
        return i
    end -- if
end KeyPos

Find(↓key: string): NodePtr
var n: NodePtr
begin
    n := table[KeyPos(↓key)]
    if n = nodeDeleted then return null
    else return n end
end Find
```

11.3.3 Wahl der Hashfunktion

Eine Hashfunktion ist dann gut, wenn der Hashcode **schnell** berechnet werden kann, möglichst **gleichverteilte** Hashcodes geliefert werden und daher möglichst wenig Kollisionen auftreten

Einfache Hashfunktionen für Zeichenketten als Schlüssel

Beispiel 1: Ordnungszahl des 1. Zeichens

```
h := Int(↓key[1])
```

- Besonders kollisionsanfällig

bauer	berger	böhm	raab
-------	--------	------	------

linz	lunz	langau	zwettl
------	------	--------	--------

- Schnell zu berechnen jedoch schlechte Verteilung

Einfache Hashfunktionen für Zeichenketten als Schlüssel

Beispiel 2: Ordnungszahlen des ersten und letzten Zeichens

```
h := Int(↓key[1]) + Int(↓key[Length(↓key)])
```

- Ebenfalls kollisionsanfällig

bauer	berger	böhm	raab
linz	lunz	langau	zwettl

Beispiel 3: Ordnungszahlen aller Zeichen

```
h := 0
for i := 1 to Length(↓key) do
    h := h + Int(↓key[i])
end -- for
```

- Kollisionen bei Permutationen, da Position der Zeichen nicht berücksichtigt wird

lampe = 108 + 97 + 109 + 112 + 101 = 527
palme = 112 + 97 + 108 + 109 + 101 = 527

Einfache Hashfunktionen für Zeichenketten als Schlüssel

Beispiel 4: Hashfunktion nach Rabin u. Karp: Ordnungszahlen aller Zeichen + Position

```
h := 0
for i := 1 to Length( $\downarrow$ key) do
    h := (h * d + Int( $\downarrow$ key[i])) mod m
end -- for
```

- Mit d = Anzahl der möglichen Werte von $key[i]$ (ASCII: $d = 256$) und m = gewünschter Wertebereich (Tabellengröße)
- Die Größen d und m müssen teilerfremd sein
- Liefert gleich verteilte Werte aus $0 \dots m - 1$ (unter der Bedingung, dass $Gcd(\downarrow d \downarrow m) = 1$)

Abbildung auf das Indexintervall der Hashtabelle [0..m-1]

- Hashfunktion nach Rabin u. Karp liefert gleich passende Werte zwischen 0 und $m - 1$, die anderen Funktionsergebnisse müssen erst noch in diesen Bereich gebracht werden.

```
h := h mod m
```

- Schlecht, da benachbarte Elemente benachbart bleiben

```
2041 mod 1000 = 41  
2042 mod 1000 = 42
```

- Kann also zu Häufungen führen, liefert nur gute Ergebnisse wenn m eine Primzahl ist
- Besser Multiplikation mit Konstante a wie bei linearer Kongruenzmethode

```
h := (a * h) mod m
```

```
2041 * 621 mod 16384 = 5893  
2042 * 621 mod 16384 = 6514
```

Algorithmen und Datenstrukturen

Eine systematische Einführung in die

Programmierung

12 Suche in Zeichenketten

Josef Pichler

Version 10.1, 2024

12.1 Aufgabe

Gegeben sind zwei Zeichenketten s (Text/String der Länge n) und p (Muster/Pattern der Länge m , wobei $m < n$)

s: array [1:n] of char
p: array [1:m] of char

Gesucht ist die Position pos des ersten Auftretens von p in s, sodass

- $s[pos:pos + m - 1] = p[1:m]$ wenn $p \in s$
 - $pos = 0$ wenn $p \notin s$

Schnittstelle

Search($\downarrow s$: array[1:n] of char $\downarrow p$: array[1:m] of char $\uparrow pos$: int)

Beispiel

pos = 10

1 m

s: Land der Berge, Land am Strom

p: Berg

1 m

Geschichte

- Offensichtliche Lösung benötigt im ungünstigsten Fall $m \cdot n$ Vergleiche
- A. Cook hat bewiesen (1970), dass ein Algorithmus existieren muss, der mit $m + n$ Vergleichen auskommt (**Problemkomplexität** $O(m + n)$)
- D.E. Knuth und V.R. Pratt gelang es, einen solchen Algorithmus zu finden
- J.H. Morris hatte zur selben Zeit die gleiche Lösungsidee
- Knuth, Morris und Pratt veröffentlichten 1977 ihren Algorithmus
- R.S. Boyer und J.S. Moore veröffentlichten 1977 anderen Algorithmus, der ebenfalls mit $m + n$ Vergleichen auskommt
- R.M. Karp und M.O. Rabin schlugen 1980 einen auf der Idee des Hashings basierenden Algorithmus vor

12.2 Ein einfacher Algorithmus BruteForceMatching

Lösungsidee

- Muster p solange über jede Position in Text s legen,
 - bis eine Übereinstimmung der Länge m gefunden ist,
 - bei einer Nichtübereinstimmung ($s[i] \neq p[j]$ mit $j \leq m$, *mismatch*) wird p gegenüber s um eine Position nach rechts verschoben
 - Sedgewick bezeichnet diese Lösung als *brute force* („rohe Gewalt“)

Beispiele

	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5
s:	L	a	n	d		d	e	r		B	e	r	g	e	,
p:	B	e	r	g		B	e	r	g		B	e	r	g	
	B	e	r	g		B	e	r	g		B	e	r	g	
	B	e	r	g		B	e	r	g		B	e	r	g	
	B	e	r	g		B	e	r	g		B	e	r	g	
	B	e	r	g		B	e	r	g		B	e	r	g	
	B	e	r	g		B	e	r	g		B	e	r	g	
	B	e	r	g		B	e	r	g		B	e	r	g	
	B	e	r	g		B	e	r	g		B	e	r	g	
Übereinstimmung											B	e	r	g	

	1	2	3	4	5	6	7	8	9	0
s:	A	B	A	C	A	D	A	B	R	A
p:	<u>A</u>	<u>B</u>	<u>R</u>	A						
	A	B	R	A						
	<u>A</u>	<u>B</u>	R	A						
	A	B	R	A						
	<u>A</u>	<u>B</u>	R	A						
	A	B	R	A						
	<u>A</u>	<u>B</u>	R	A						
	A	B	R	A						
	<u>A</u>	<u>B</u>	R	A						

Übereinstimmung

12.2.1 BruteForceMatching – Variante 1

Muster wird von links nach rechts über Text gelegt

i	
s:	a c a a b c

p:	a a b
j	

Beginn mit $i = 1$ und $j = 1$

i	
s:	a c a a b c

p:	a a b
j	

Bei Übereinstimmung
wird j um 1 erhöht

i	
s:	a c a a b c

p:	a a b
j	

Bei Nichtübereinstimmung
wird Muster um eine Stelle
nach rechts verschoben
($i := i + 1, j := 1$)

i	
s:	a c a a b c

i	
s:	a c a a b c

p:	a a a
j	

Muster in Text an Position i
enthalten, wenn $j > m$

Muster in Text nicht enthalten,
wenn $i > (n - m + 1)$

BruteForceMatching – Variante 1

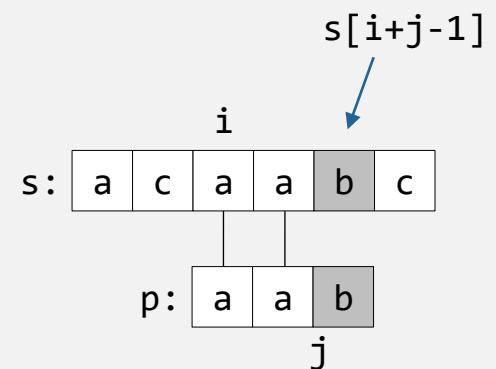
Algorithmus

```
BruteForceMatching(↓s: array[1:n] of char
                    ↓p: array[1:m] of char
                    ↑pos: int)
    var i, j: int
begin           solange nicht gefunden
    i := 1
    pos := 0
    while (pos = 0) and (i ≤ n - m + 1) do
        j := 1
        while (j ≤ m) and (s[i + j - 1] = p[j]) do
            j := j + 1
        end -- while
        if j > m then
            pos := i
        else
            i := i + 1
        end -- if
    end -- while
end BruteForceMatching
```

solange Ende nicht erreicht

hier gilt: $s[i:i + j - 1] = p[1:j]$

Muster p an Position i gefunden



12.2.2 BruteForceMatching – Variante 2

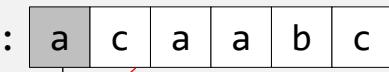
In Variante 1 sind bei jedem Vergleich Indexberechnungen erforderlich
Überlegungen zu Variante 2 (... $s[i] = p[j]$...):

i	
s:	a c a a b c

j	
p:	a a b

Beginn mit $i = 1$ und $j = 1$

i	
s:	a c a a b c

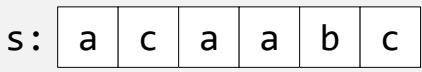


A red arrow points from the label 'i' to the first character 'a' in the string 's'. Another red arrow points from the label 'j' to the first character 'a' in the string 'p'.

| j | |
| p: | a | a | b |

Bei Übereinstimmung werden i und j um 1 erhöht

i	
s:	a c a a b c



A red arrow points from the label 'i' to the second character 'c' in the string 's'. Another red arrow points from the label 'j' to the second character 'a' in the string 'p'.

| j | |
| p: | a | a | b |

Bei Nichtübereinstimmung wird Muster um eine Stelle nach rechts verschoben
($i := i - j + 2$, $j := 1$)

i	
s:	a c a a b c

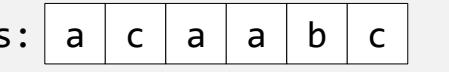


A red arrow points from the label 'i' to the third character 'a' in the string 's'. Another red arrow points from the label 'j' to the second character 'a' in the string 'p'.

| j | |
| p: | a | a | b |

Muster in Text an Position $i - m$ enthalten, wenn $j > m$

i	
s:	a c a a b c



A red arrow points from the label 'i' to the fourth character 'a' in the string 's'. Another red arrow points from the label 'j' to the third character 'a' in the string 'p'.

| j | |
| p: | a | a | a |

Muster in Text nicht enthalten, wenn $i > n$

BruteForceMatching – Variante 2

Algorithmus

```
BruteForceMatching2(↓s: array[1:n] of char
                     ↓p: array[1:m] of char
                     ↑pos: int)
    var i, j: int
begin
    i := 1
    j := 1
repeat
    if s[i] = p[j] then
        i := i + 1; j := j + 1
    else
        i := i - j + 2; j := 1
    end -- if
until (i > n) or (j > m)
if j > m then
    pos := i - m
else
    pos := 0
end -- if
end BruteForceMatching2
```

hier gilt: $s[i-j+1:i] = p[1:j]$

Muster um 1 Stelle nach
rechts verschieben
(bezogen auf Position
mit $j = 1$)

Muster p an Position $i - m$ gefunden

Laufzeitkomplexität

Günstigster Fall bei erfolgreicher Suche: $O(m)$

- Muster wird bereits an erster Position in Text gefunden

Günstigster Fall bei erfolgloser Suche: $O(n)$

- Erstes Zeichen von Muster kommt in Text nicht vor

Ungünstigster Fall: $O(m \cdot n)$

- Muster der Länge m muss mit jeder Teilkette aus dem Text verglichen werden ($m \cdot (n - m + 1)$) Vergleiche)

```
s := "aaaaaaaaaaaaaaaaaaaaab"
```

```
p := "aaaab"
```

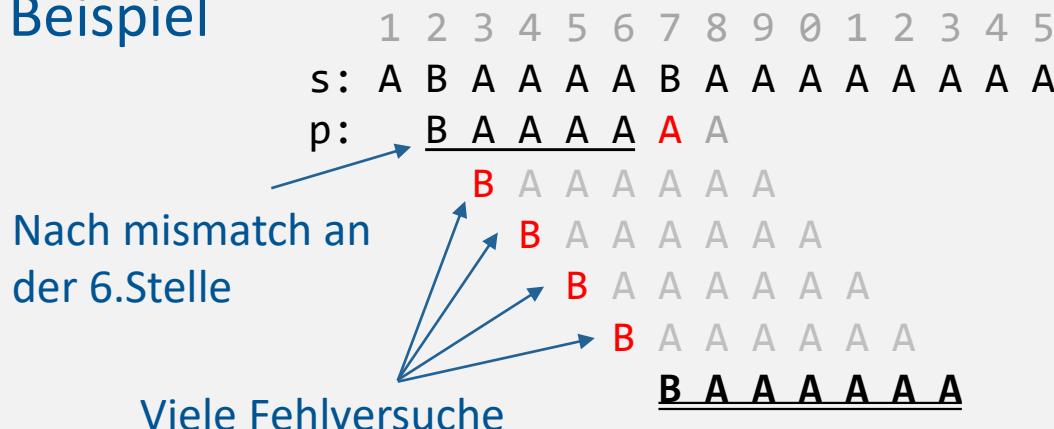
Bei großem Alphabet wird Muster nur selten vollständig mit Teilkette verglichen, Wahrscheinlichkeit $\left(\frac{1}{d}\right)^m$ mit d verschiedenen Zeichen

12.3 Knuth-Morris-Pratt Mustersuchalgorithmus

„this is one of the coolest algorithm that we'll cover in this course“ R. Sedgewick

- Beruht auf der Feststellung, dass wertvolle Zeit verloren geht, wenn bei jeder Verschiebung des Musters (relativ zum Text) stets wieder mit dem ersten Zeichen des Musters begonnen wird
- Wenn an Position j im Muster eine Nichtübereinstimmung festgestellt wird, kennt man die Zeichen $p[1], \dots, p[j]$
- Aus diesem Wissen lässt sich die Information gewinnen, ob das Muster um mehr als eine Position nach rechts verschoben werden kann

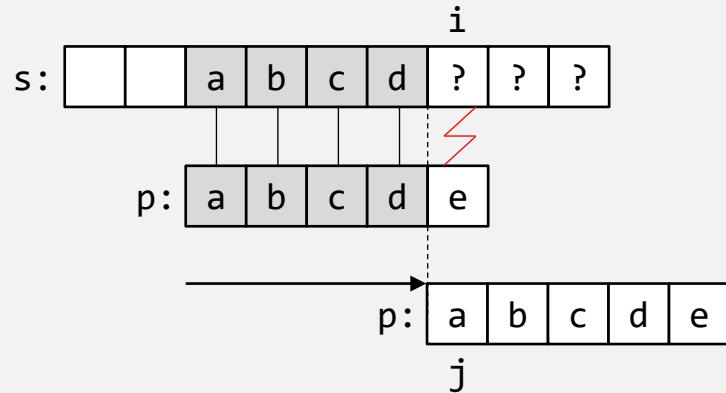
Beispiel



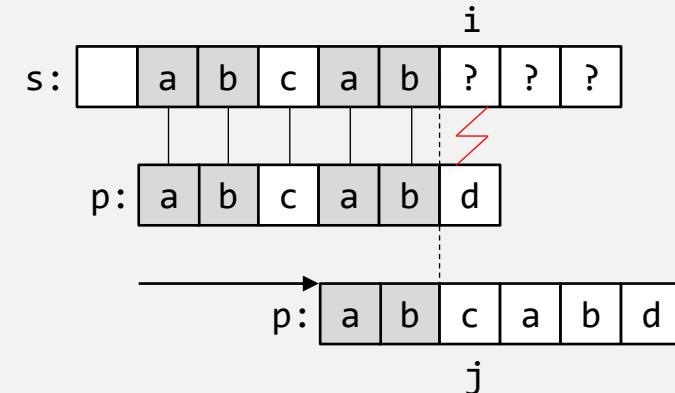
Idee: Wenn erstes Zeichen (hier B) im Rest des Musters nicht mehr vorkommt, können Zeichen AAAA keine gültigen Startpositionen sein

Knuth-Morris-Pratt Mustersuchalgorithmus

Lösungsidee für das Verschieben des Musters um mehr als eine Stelle



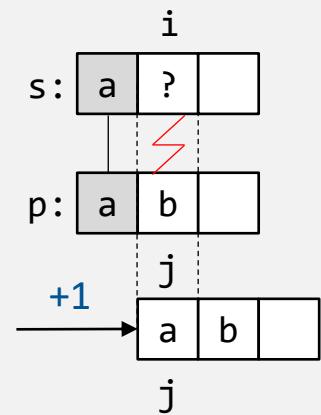
Fall 1: Erstes Zeichen in Muster kommt vor mismatch-Stelle nicht vor.



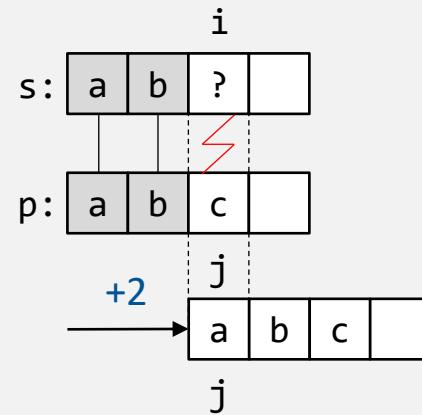
Fall 2: Teilkette vor mismatch-Stelle ist auch am Beginn des Muster zu finden.

- Ein ganzes Muster überspringen funktioniert nicht, wenn das Muster am Punkt des „mismatch“ auf sich selbst passt (Fall 2)
- Vor der Suche kann aufgrund des Musters p alleine bereits ermittelt werden, wie weit zurück die Suche wieder aufgesetzt werden muss
- Idee: Im Falle eines mismatch gibt Feld next Auskunft, wo im Muster die Suche fortgesetzt werden kann (dh nächster Wert für Position j)

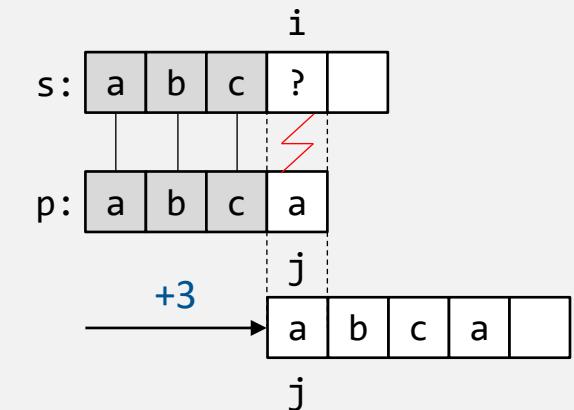
Ermittlung der Elemente des next-Felds



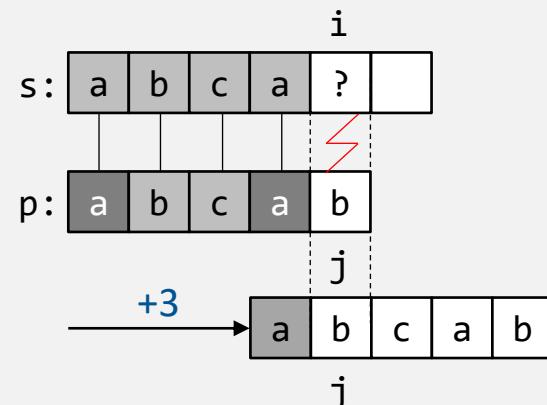
next[2] := 1



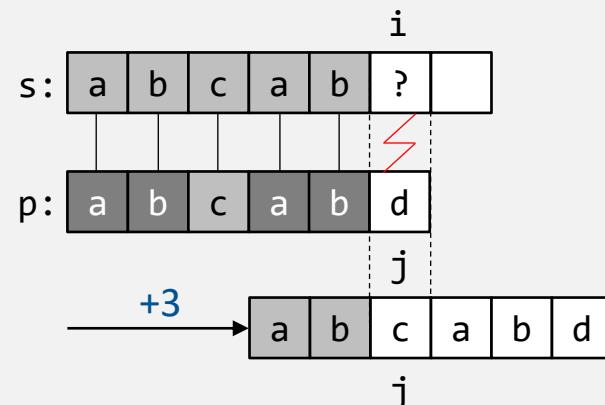
next[3] := 1



next[4] := 1



next[5] := 2



next[6] := 3

p:

a	b	c	a	b	d
---	---	---	---	---	---

next:

0	1	1	1	2	3
---	---	---	---	---	---

j:

1	2	3	4	5	6
---	---	---	---	---	---

shift

1	1	2	3	3	3
---	---	---	---	---	---

(j-next[j])

Ermittlung der Elemente des next-Felds

Im Falle eines mismatch gibt also next Auskunft, wo im Muster die Suche fortgesetzt werden kann

Bei mismatch an der Position $j = 1$ wird Muster um eine Position verschoben
(dh $i = i + 1$ und $j = 1$, dazu wird $\text{next}[1] = 0$ gesetzt)

Beispiele

p: [a b c a b d]	p: [b e r g]	p: [T T T T T F]	p: [a a a b b b]
next: [0 1 1 1 2 3] 1 m	next: [0 1 1 1] 1 m	next: [0 1 2 3 4 5] 1 m	next: [0] 1 m

Unter der Annahme, dass wir next kennen, erhalten wir somit folgenden Algorithmus (siehe nächste Seite)

Algorithmus

```
KMPMatching( $\downarrow s$ : array[1:n] of char  $\downarrow p$ : array[1:m] of char  $\uparrow pos$ : int)
var
    next: array[1:m] of int
    i, j: int
begin
    InitNext( $\downarrow p$   $\uparrow next$ )
    i := 1; j := 1
repeat
    if (j = 0) or (s[i] = p[j]) then
        i := i + 1
        j := j + 1
    else
        j := next[j]
    end -- if
until (i > n) or (j > m)
if j > m then
    pos := i - m
else
    pos := 0
end -- if
end KMPMatching
```

Variable i wird nicht verändert!

Muster p im Text s beginnend an Position pos

Muster p nicht im Text s enthalten

Aufbau der Tabelle next

Erfolgt nach dem gleichen Prinzip wie KMPMatching (weil es sich ja ebenfalls um eine Mustersuche handelt)!

```
InitNext(↓p: array[1:m] of char ↑next: array[1:m] of int)
    var i, j: int
begin
    i := 1;
    j := 0
    next[1] := 0
repeat
    if (j = 0) or (p[i] = p[j]) then
        i := i + 1
        j := j + 1
        next[i] := j
    else
        j := next[j]
    end -- if
until i ≥ m
end InitNext
```

Laufzeitkomplexität

- Die Autoren Knuth, Morris und Pratt zeigen, dass ihr Verfahren höchstens $m+n$ Vergleiche benötigt
- Der KMPSearch hat somit die asymptotische Laufzeitkomplexität $O(m+n)$, ist also linear
- Die asymptotische Laufzeitkomplexität des Algorithmus InitNext für die Ermittlung der Elementwerte des Felds next ist $O(m)$

12.4 Rabin-Karp Mustersuchalgorithmus

Zur Effizienzsteigerung (um unnötige Vergleiche zu vermeiden) werden Hashfunktionen herangezogen:

- Berechnung des Hashcodes h_p für das Muster p und für jede Teilkette h_s der Länge m aus dem Text s
- Wenn $h_p \neq h_s$ ist, dann kann an der entsprechenden Stelle in s das Muster p nicht beginnen, daher wird Hashcode für nächste Teilkette in s berechnet (d. h. Muster wird nach rechts verschoben)
- Wenn $h_p = h_s$ werden Muster und Teilkette zeichenweise verglichen (z. B. mittels BruteForce)
- Um Zeit zu sparen werden die Hashcodes inkrementell berechnet

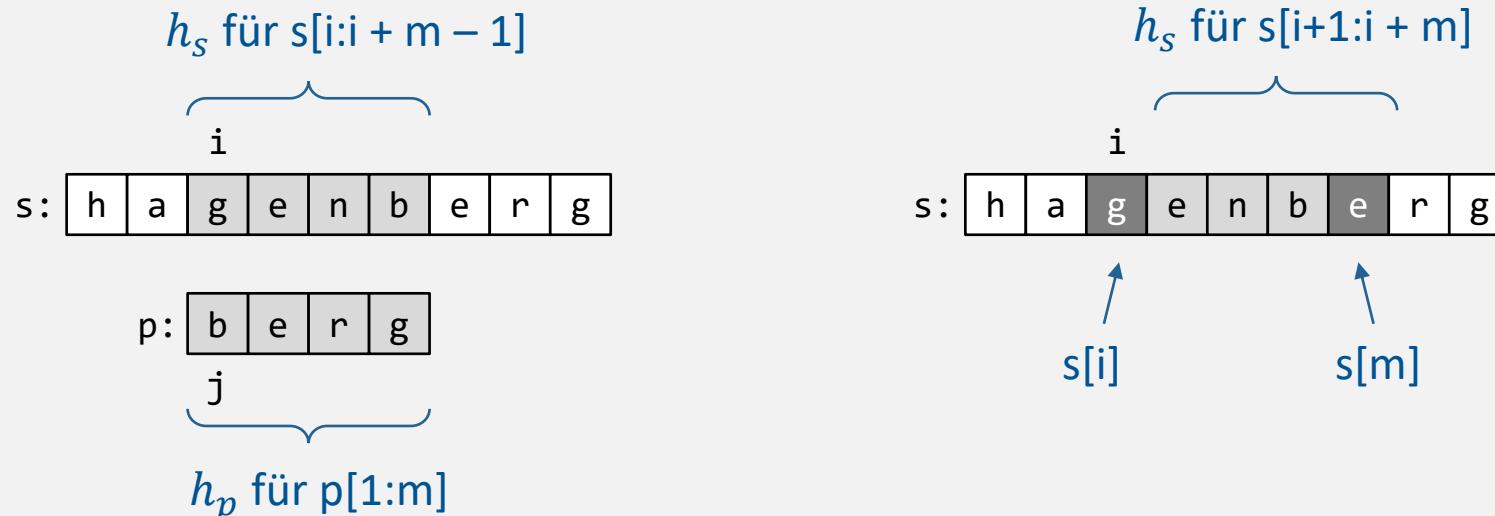
Vorteil:

- Durch das frühe Erkennen einer Nichtübereinstimmung, sind weniger Vergleiche erforderlich

12.4.1 Inkrementelle Berechnung der Hashcodes

- Berechnung des Hashcode für Teilkette $s[i], \dots s[i + m - 1]$ soll möglich sein, ohne alle Zeichen $s[i], \dots s[i + m - 1]$ "besuchen" zu müssen (sonst könnte man auch vergleichen)
- Man berechnet einmal den Hashcode h_s für $s[1..m]$
- Aus h_s für $s[i:i + m - 1]$ wird h_s für $s[i + 1:i + m]$ berechnet
 - Anteil von $s[i]$ aus Hashcode entfernen und
 - Anteil von $s[i + m]$ hinzufügen

Beispiel



Inkrementelle Berechnung mit einfacher Hashfunktion

Wenn wir die Summe aller Ordinalwerte der einzelnen Zeichen heranziehen, ergibt sich der Hashcode für die Zeichenkette beginnend an Position i

$$h(i) = s[i] + s[i + 1] + \dots + s[i + m - 1]$$

und der Hashcode für die Zeichenkette beginnend an Position $i + 1$

$$h(i + 1) = s[i + 1] + \dots + s[i + m - 1] + s[i + m]$$

Daraus folgt:

$$h(i + 1) = h(i) - s[i] + s[i + m]$$

Beispiel mit $s = "Hagenberg"$, $i = 3$, $m = 4$:

$$h(3) = 103 + 101 + 110 + 98 = 412$$

$$h(3 + 1) = 101 + 110 + 98 + 101 = 410$$

$$h(3 + 1) = h(3) - 103 + 101 = 410$$

Hashfunktion von Rabin-Karp

Muster $p[1], \dots p[m]$ und Teilketten $s[i], \dots s[i + m - 1]$ werden als m -stellige Zahlen zur jeweiligen Basis d (Größe des Alphabets) aufgefasst

```
h := 0
for i := 1 to m do
    h := (h * d + Int(↓key[i])) mod q
end -- for
```

Beispiel

- mit Alphabet = {A,B,C,D,E,F,G,H,I,J}, $d = 10$
- und Codierung $a \rightarrow 1, b \rightarrow 2, c \rightarrow 3 \dots$
- $ACCD = 1 \cdot 10^3 + 3 \cdot 10^2 + 3 \cdot 10^1 + 4 \cdot 10^0 = 1334$
- Wenn wir eine Zeichenkette mit gleicher Länge, gleichen Zeichen aber anderer Anordnung haben, erhalten wir ein anderes Ergebnis:
- $ACDC = 1 \cdot 10^3 + 3 \cdot 10^2 + 4 \cdot 10^1 + 3 \cdot 10^0 = 1343$

Inkrementelle Berechnung mit Hashfunk. von Rabin-Karp

Hashfunktion von Rabin-Karp

```
h := 0
for i := 1 to m do
    h := (h * d + Int(↓key[i])) mod q
end -- for
```

Berechnung ($s[i]$ bedeutet hier $\text{Int}(\downarrow\text{key}[i])$):

$$h(i) = s[i] \cdot d^{m-1} + s[i+1] \cdot d^{m-2} + \dots + s[i+m-1] \cdot d^0$$

$$h(i+1) = s[i+1] \cdot d^{m-1} + \dots + s[i+m-1] \cdot d^1 + s[i+m] \cdot d^0$$

Inkrementelle Berechnung von $h(i+1)$:

$$h = h - s[i] \cdot d^{m-1} \quad \xleftarrow{\text{entfernt „Anteil“ des 1. Elements}}$$

$$h = h \cdot d \quad \xleftarrow{\text{erhöht Exponent}}$$

$$h = h + s[i+m] \quad \xleftarrow{\text{addiert neues Element}}$$

$$h = h \bmod q$$

Probleme bei inkrementeller Berechnung

Problem 1: $h = h - s[i] \cdot d^{m-1}$

- d^{m-1} kann den gültigen Wertebereich übersteigen
- Statt mit d^{m-1} wird mit $d^{m-1} \bmod q$ multipliziert

```
dm := 1
for i := 2 to m do
    dm := (dm * d) mod q
end
```

Problem 2: $h = h - s[i] \cdot d^{m-1}$

- Vorangehendes h kann sehr klein sein, sodass Subtraktion zu negativen Wert führt
- Vor Subtraktion wird Wert addiert, der ein Vielfaches von q und sicher größer als $s[i] \cdot d^{m-1}$ ist; da $d^{m-1} < q$ ist kann man $d \cdot q$ nehmen

Das führt zu folgender Berechnung:

```
h := (h + d * q - dm * Int(↓s[i])) mod q
h := (h * d + Int(↓(s[i + m]))) mod q
```

12.4.2 Der Rabin-Karp-Mustersuchalgorithmus

```
RKMatching( $\downarrow s$ : array[1:n] of char  $\downarrow p$ : array[1:m] of char  $\uparrow pos$ : int)
const
    q = 8355967
    d = 256
begin
    dm := 1
    for i := 2 to m do
        dm := (d * dm) mod q
    end -- for
    hs := 0
    hp := 0
    for i := 1 to m do
        hs := (hs * d + Int( $\downarrow s[i]$ )) mod q
        hp := (hp * d + Int( $\downarrow p[i]$ )) mod q
    end -- for
    ...

```

berechnet $dm = d^{m-1} \text{mod } q$

Hashcode für die ersten m Zeichen der Zeichenkette

Hashcode für Muster

12.4.2 Der Rabin-Karp-Mustersuchalgorithmus

```
...
i := 1
while i ≤ n - m + 1 do
    if hs = hp then
        j := 1
        while (j ≤ m) and (s[i + j - 1] = p[j]) do
            j := j + 1
        end -- while
        if j > m then
            pos := i
            return
        end -- if
    end -- if
    if i < n - m + 1 then
        hs := (hs + d * q - dm * Int(↓s[i])) mod q
        hs := (hs * d + Int(↓s[i + m])) mod q
    end -- if
    i := i + 1
end -- while
pos := 0
end RKMatching
```

Vergleich der Zeichen in Muster und Teilkette, wenn Hashcodes übereinstimmen

inkrementelle Berechnung des nächsten Hashwerts

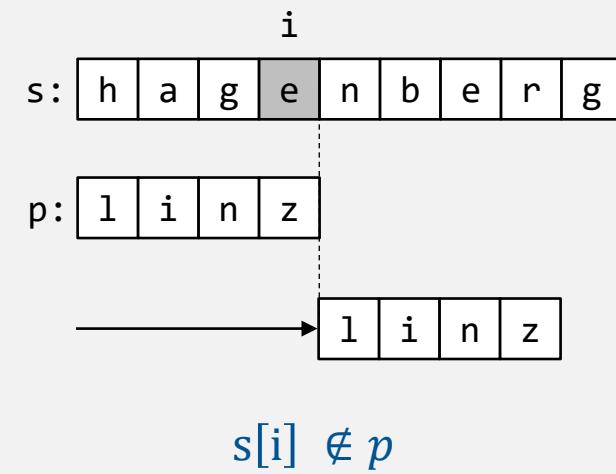
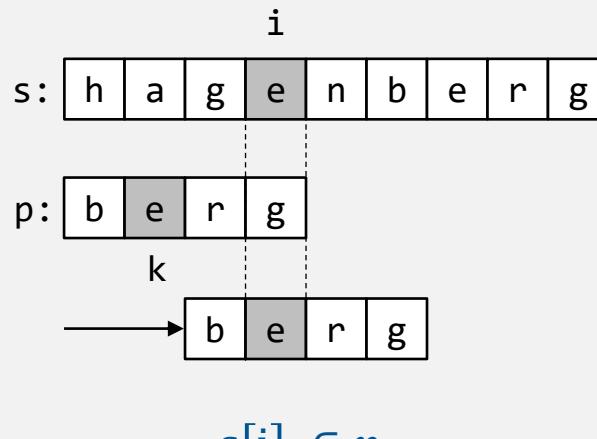
Laufzeitkomplexität: $O(n + m)$

12.5 Boyer-Moore Mustersuchalgorithmus

Muster wird von links nach rechts über Zeichenkette gelegt, der Zeichenvergleich erfolgt jedoch von rechts nach links (d. h. von hinten)

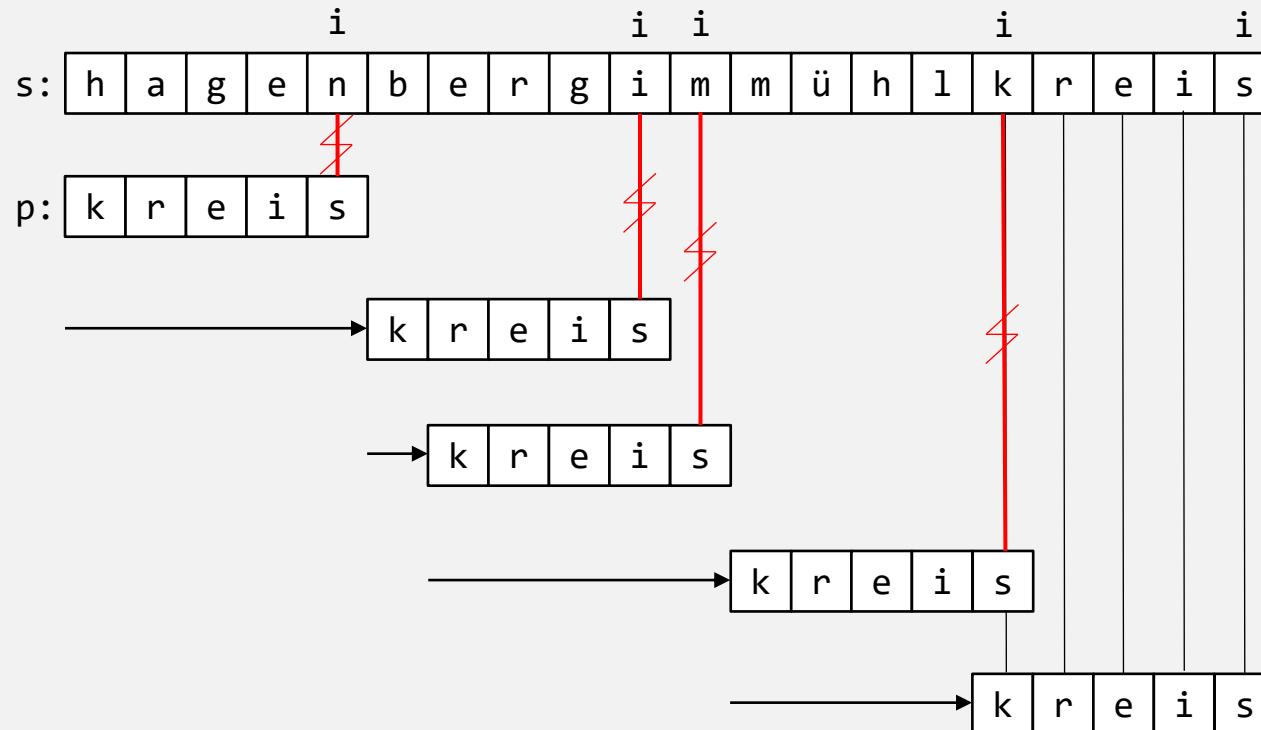
Bei mismatch an Position i Muster verschieben, uzw:

- Wenn mismatch-Zeichen $s[i]$ in p an Position k vorkommt, wird Muster so über Text gelegt, dass $s[i]$ und $p[k]$ übereinander liegen
- Wenn mismatch-Zeichen $s[i]$ in Muster p nicht vorkommt, scheiden die Positionen $\leq i$ als Startpositionen aus



Boyer-Moore Musteralgorithmus

Beispiel



Insgesamt nur 9 Vergleiche

- 4 Vergleiche mit mismatch
 - 5 Vergleiche mit match

12.5.1 BruteForceMatching Variante 3

Muster p der Länge m wird von links nach rechts über Text s der Länge n gelegt und bei Nichtübereinstimmung nach rechts verschoben

i	
s:	b a c a b c

j	
p:	a b c

Beginn mit $i = m$ und $j = m$

i	
s:	b a c a b c

j	
p:	a b c

Bei Übereinstimmung werden i und j dekrementiert

i	
s:	b a c a b c

j	
p:	a b c

Bei Nichtübereinstimmung wird Muster um eine Stelle nach rechts verschoben
($i := i + m - j + 1$, $j := m$)

i	
s:	b a c a b c

j	
p:	a b c

Muster in Text an Position $i + 1$ enthalten, wenn $j < 1$

i	
s:	b a c a b c

j	
p:	a a a

Muster in Text nicht enthalten, wenn $i > n$

BruteForceMatching Variante 3

Algorithmus

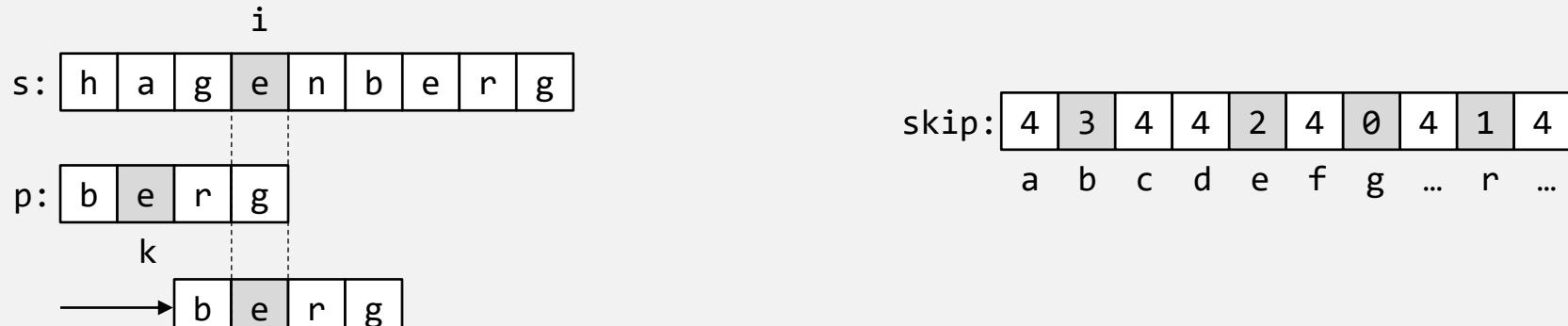
```
BruteForceMatching3(↓s: array[1:n] of char ↓p: array[1:m] of char ↑pos: int)
  var
    i, j: int
  begin
    i := m           ← Beginne rechts
    j := m
    repeat
      if s[i] = p[j] then
        i := i - 1 ; j := j - 1
      else
        i := i + m - j + 1; j := m ← Muster wird um 1 Stelle
                                nach rechts verschoben
      end -- if
    until (i > n) or (j < 1)
    if j < 1 then
      pos := i + 1
    else
      pos := 0           ← Muster gefunden
    end -- if
  end BruteForceMatching3
```

12.5.2 Der eigentliche Boyer-Moore-Mustersuchalgorithmus

- Neue Position des Musters nach mismatch hängt von $s[i]$ und Musterinhalt ab
- Einrichtung einer Tabelle skip (über Alphabet), die die Verschiebungsdistanzen in Abhängigkeit vom mismatch-Zeichen $s[i]$ enthält

```
var  
    skip: array[char] of int
```

- Ist mismatch-Zeichen c nicht in p enthalten, ist $skip[c] := m$
- Ist mismatch-Zeichen c mehrfach in p enthalten, muss die am weitesten rechts liegende Position genommen werden
- Bei mismatch an Position $p[m]$ wird p um eine Position verschoben



Aufbau der Tabelle skip

Algorithmus

```
InitSkip(↑skip: array[char] of int)
  var
    c: char
    j: int
  begin
    for c := Char(↓0) to Char(↓255) do
      skip[c] := m
    end -- for
    for j := 1 to m do
      skip[p[j]] := m - j
    end -- for
  end InitSkip
```

Beispiel: Einträge in die Tabelle skip für $p = "berg"$ und $m = 4$

j	p[j]	m - j
1	b	3
2	e	2
3	r	1
4	g	0

Algorithmus

```
BMMatching( $\downarrow s$ : array[1:n] of char  $\downarrow p$ : array[1:m] of char  $\uparrow pos$ : int)
var
    i, j: Integer
    skip: array[char] of int
begin
    InitSkip( $\uparrow skip$ )
    i := m; j := m
    repeat
        if s[i] = p[j] then
            i := i - 1; j := j - 1
        else
            i := i + skip[s[i]] ← ersetzt  $i := i + m - j + 1$ 
            j := m
        end -- if
    until (i > n) or (j < 1)
    if j < 1 then
        pos := i + 1
    else
        pos := 0
    end -- if
end BMMatching
```

Problem

- Wenn mismatch-Zeichen $s[i]$ auch rechts von der mismatch-Position auftritt, wird im bisherigen Algorithmus das Muster nach links verschoben

s:

b	c	b	d	c	d	b	c	d
---	---	---	---	---	---	---	---	---



p:

a	b	c	d
---	---	---	---

a	b	c	d
---	---	---	---

 ←

falsch

s:

b	c	b	d	c	d	b	c	d
---	---	---	---	---	---	---	---	---



p:

a	b	c	d
---	---	---	---

a	b	c	d
---	---	---	---

 →

richtig

- Der Wert von $\text{skip}[s[i]]$ wird nur verwendet, wenn rechts von $p[j]$ kein $s[i]$ steht
- Sonst wird Muster um eine Stelle nach rechts verschoben:

```
if ( $m - j + 1 > \text{skip}[s[i]]$ ) then
     $i := i + m - j + 1$ 
else
     $i := i + \text{skip}[s[i]]$ 
end -- if
```

Laufzeitkomplexität

Günstigster Fall ($\frac{n}{m}$)

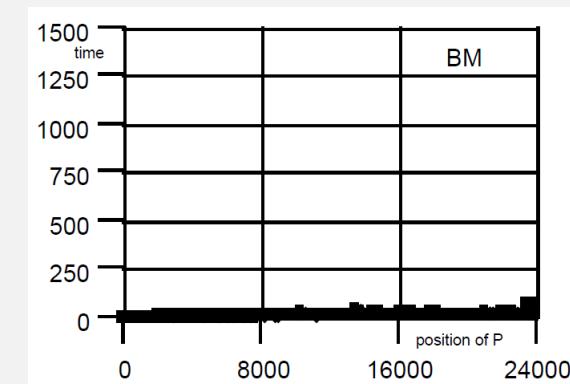
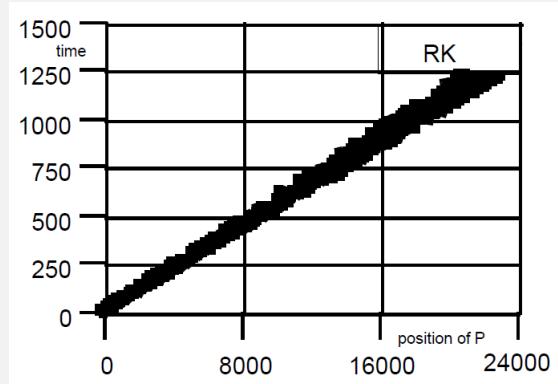
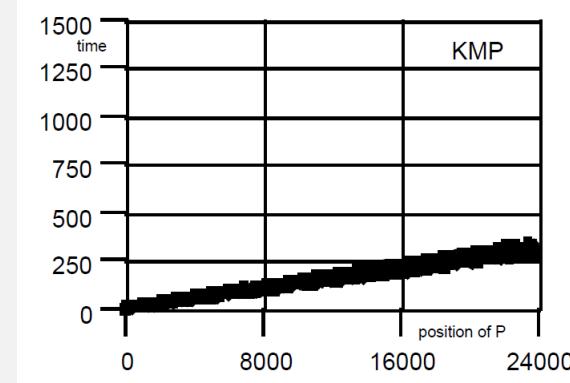
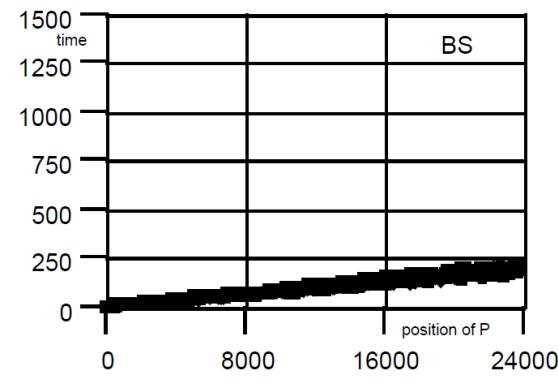
```
s := "aaaaaaaaaaaaaaaaaaaaaaabcdef"  
p := "bcdef"
```

Ungünstigster Fall ($n \cdot m$ Vergleiche)

```
s := "aaaaaaaaaaaaaaaaaaaaaaaabaaaa"  
p := "baaaa"
```

Für große Zeichensätze und nicht allzu lange Muster sind im Durchschnitt $\frac{n}{m}$ Vergleiche erforderlich

12.6 Vergleich der Laufzeiten



Pirklbauer: A Study of Pattern-Matching Algorithms, Structured Programming, Vol. 13, pp. 89-98, Springer-Verlag, Mai 1992.

Algorithmen und Datenstrukturen

Eine systematische Einführung in die

Programmierung

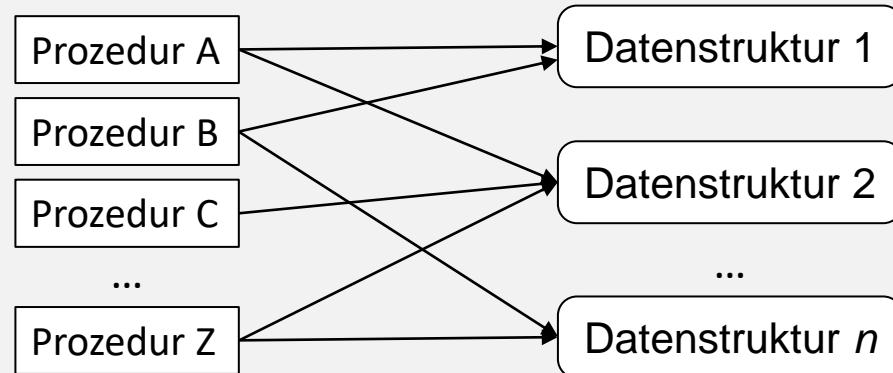
13 Datenkapseln und Module

Josef Pichler

Version 10.1, 2024

13.1 Unbeschränkter Zugriff auf Datenobjekte

Zerlegung von Algorithmen nach funktionalen Gesichtspunkten führt häufig zu unbeschränkten Datenobjektzugriffen



Konsequenz

- Alle Algorithmen (A bis Z) müssen Details der Datenobjekte (1 bis n) kennen
- Beim Ändern einer Datenstruktur ist es schwierig festzustellen, wo sie überall verwendet wird
- Unabsichtliche Verwendung (Zerstörung) der Daten möglich

Beispiel 1

Beispiel: Unbeschränkter Zugriff auf Stack (Realisierung mittels Feld)

Deklarationen

```
const  
    size = ...  
var  
    stack: array [1:size] of int  
    top: int
```

Initialisierung

```
top := 0
```

Operationen Push und Pop

```
top := top + 1  
stack[top] := 42
```

ohne `top:=top + 1` wird letztes Element überschrieben

```
x := stack[top]  
top := top - 1
```

Fehlerbehandlung (z.B. Stack leer oder voll) ist bei jedem Zugriff auf Datenobjekt `stack` erforderlich.

Beispiel 2

Beispiel: Unbeschränkter Zugriff auf Bankkonto-Datenobjekt

Deklarationen

```
const
  size = ...
type
  Timestamp = ...
  Change = compound
    time: Timestamp
    purpose: string
    amount: real
  end -- compound
  Account = compound
    balance: real
    initialCredit: real
    nChanges: int
    changes: array [1:size] of Change
  end -- compound
var
  a: Account
```

Anfangsstand bei Eröffnung,
Kontobewegungen, aktueller Saldo

Kontodaten müssen konsistent sein
 $balance = initialCredit + changes[1:nChanges].amount$

Anzahl der Kontobewegungen muss mit
tatsächlich vorhandenen Änderungen
übereinstimmen

Beispiel 2

Verwendung

```
-- init account  
a.initialCredit := 0.0  
a.balance := 0.0  
a.nChanges := 0
```

ohne Anweisung würde letzter
Eintrag überschrieben

```
-- new change  
a.nChanges := a.nChanges + 1  
a.changes[a.nChanges].time := ...  
a.changes[a.nChanges].purpose := "Von Oma"  
a.changes[a.nChanges].amount := 1000.0  
a.balance := a.balance + 1000.0
```

ohne Anweisung würde Kontostand nicht
stimmen

Probleme

- Gefahr von inkonsistenten Zuständen
- Undurchsichtiger Datenfluss
- Hoher Änderungsaufwand, z.B. Stack-Repräsentation Feld durch Liste ersetzen

Beispiel 3

Einfaches Zeichenprogramm (ca. 1984)

- ca. 5.800 Zeilen Pascal-Code
- ca. 3.800 Zeilen Assembler-Code

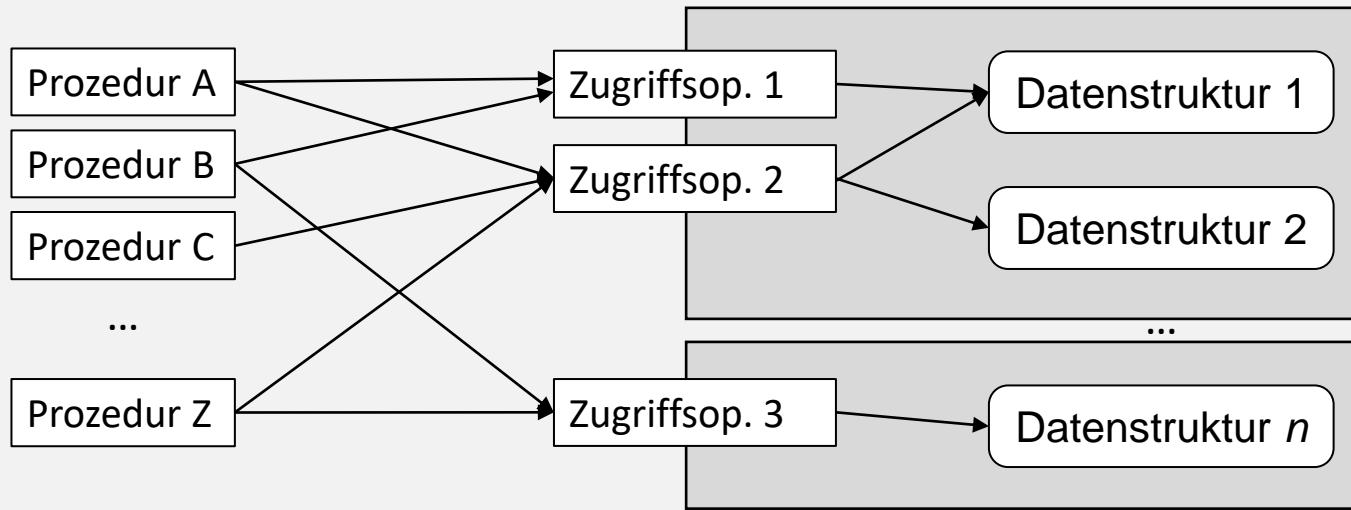
Pascal-Code

- eine Datei
- ca. 150 globale Variablen
- ca. 150 Prozeduren u. Funktionen



<https://computerhistory.org/blog/macpaint-and-quickdraw-source-code/>

13.2 Beschränkter Zugriff auf Datenobjekte



Konsequenz

- Nur noch wenige Algorithmen (Zugriffsoperationen) greifen auf die Datenobjekte zu
- Datenobjekte und Zugriffsoperationen bilden abgeschlossene Einheit
- Schutz vor Zerstörung des Inhalts der Datenobjekte/Datenstruktur (DS)
- Implementierung der gekapselten Datenstruktur kann geändert werden
- Indirekter Zugriff kostet Zeit

Geheimnisprinzip (*Principle of Information Hiding*)

nach David Parnas, 1972

Datenabstraktion

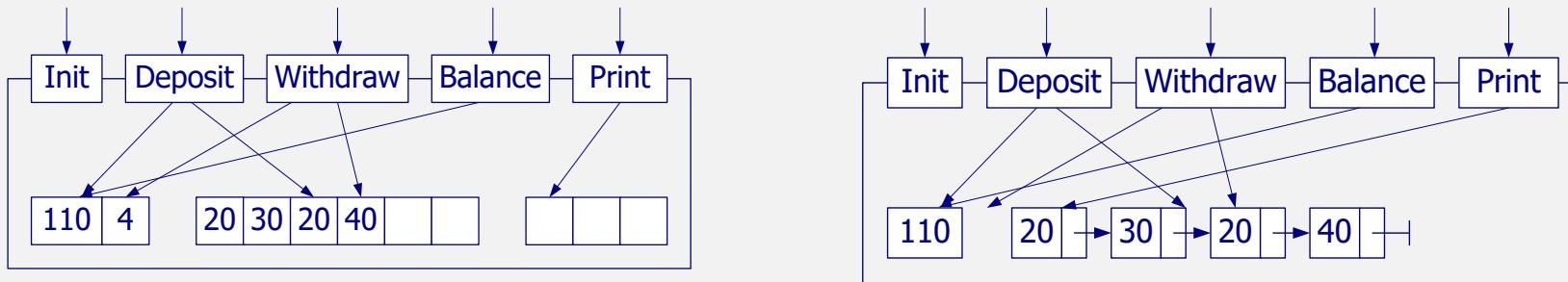
- Datenobjekte, d.h. ihr konkreter Aufbau, dürfen nur denjenigen Algorithmen bekannt sein, die sie zu ihrer Implementierung brauchen
- Unterscheidung zwischen Implementierungsaspekt und Anwendungsaspekt

Datensicherheit und Änderungsfreundlichkeit

- Die Datenobjekte können nicht direkt, sondern nur mittels zugelassener Operationen (Zugriffsoperationen) manipuliert werden
- Die Änderung der konkreten Implementierung von Datenobjekten erfordert keine Änderungen in den sie verwendenden Algorithmen (bei gleichbleibender Schnittstelle der Zugriffsoperationen)

Datenkapsel

- Die Datenobjekte selbst bleiben der "Außenwelt" verborgen (Daten sind gekapselt – *information hiding*)
- Zugriffsoperationen sind einzige Möglichkeit zur Manipulation des Inhalts und zum Zugriff auf den Inhalt der Datenobjekte



- Benutzer-Algorithmus kennt "Inhalt" der Datenkapsel nicht

```
Init()
Deposit("Spende von Oma" ↓1000.0)
Withdraw("iPhone" ↓199.0)
if Balance() > 499.0 then
    Withdraw("Mac Mini" ↓499.0)
end -- if
```

13.3 Module zur Realisierung von Datenkapseln

Datenkapselungsprinzip:

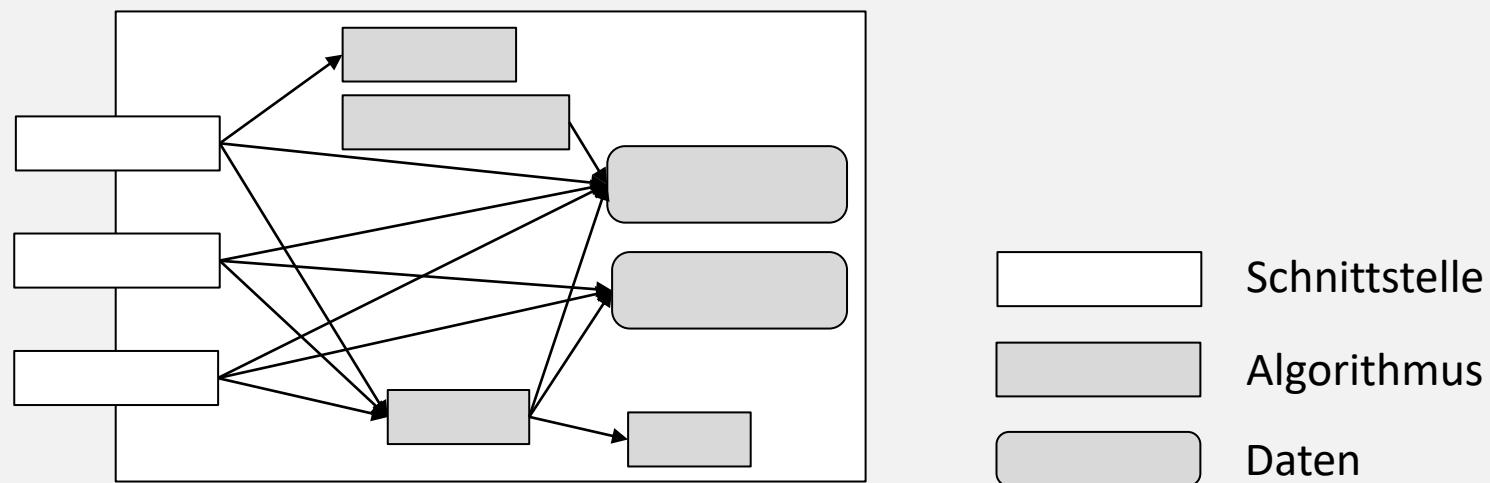
- Datenkapsel entspricht Sammlung von Algorithmen (inkl. Zugriffsoperationen) und ihnen gemeinsam zugänglichen Datenobjekten
- die Datenobjekte sind statische Variablen, auf die alle Algorithmen (inkl. Zugriffsoperationen) zugreifen können (lokale globale Variable)

Realisierungsform:

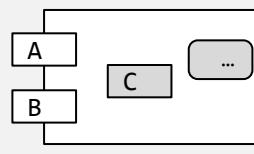
- Bildung einer Programmkomponente/Übersetzungseinheit
- Verschiedene Konzepte **Modul** (z.B. Modula-2), **Package** (z.B. Java), **Unit** (z.B. Free Pascal)
- Pascal-Dialekte (z.B. Free Pascal) bieten Möglichkeit, Programmsysteme auf mehrere Übersetzungseinheiten aufzuteilen
- Systemzerlegung wird Modularisierung genannt

Das Modulkonstrukt (Wdh. Kapitel 5, 1. Semester)

- Ein Modul ist eine Sammlung von Datenobjekten und Algorithmen.
- Er kommuniziert mit der Außenwelt nur über eine eindeutig definierte Schnittstelle.
- Die Benutzung eines Moduls setzt keine Kenntnisse seines inneren Aufbaus voraus.
- Die Implementierung eines Moduls setzt keine Kenntnisse über seine Benutzung voraus.
- Die Korrektheit ist ohne Kenntnis der Einbettung ins Gesamtsystem prüfbar.



Aufbau



Module bestehen aus zwei Teilen: *Schnittstelle* und *Implementierung*

```
interface of M
  const ...
  type ...
  A(...)
  B(...)
end M
```

Schnittstelle (*interface*)

- Konstanten
- Datentypen
- Schnittstellen der Zugriffsoperationen (Signatur)

```
implementation of M
  var ...
  A(...) begin ...C()... end A
  B(...) begin ...C()... end B
  C(...) begin ... end C

  init
  ...
end M
```

Implementierung (*implementation*)

- benötigte Datenobjekte/-strukturen
- Implementierung der Zugriffsoperationen
- weitere Algorithmen, die nicht direkt von außen benutzt werden können
- ggf. Initialisierung

Beispiel: Modul für Integer-Stack

```
interface of IntStack
  InitStack()
  Push( $\downarrow$ e: int)
  Pop( $\uparrow$ e: int)
  IsEmpty(): bool
end IntStack
```

```
implementation of IntStack
  const
    size = 100
  var
    stack: array[1:size] of int
    top: int

  InitStack()
  begin
    top := 0
  end InitStack

  IsEmpty(): bool
  begin
    return top = 0
  end IsEmpty
```

```
...
  Pop( $\uparrow$ e: int)
  begin
    e := stack[top]
    top := top - 1
  end Pop
```

```
Push( $\downarrow$ e: int)
begin
  top := top + 1
  stack[top] := e
end Push
```

```
init
  InitStack()
end IntStack
```

Beispiel: Modul für Integer-Stack

Verwendung Modul *IntStack*

```
program P
  import IntStack
  var
    i: int
begin
  Push(↓13)  Push(↓21)
  Push(↓34)  Push(↓55)
  while not IsEmpty() do
    Pop(↑i)
    Write(↓i ↓" ")
  end -- while
  top := 0
  Write(↓stack[4])
end P
```

interface of IntStack
InitStack()
Push(↓e: int)
Pop(↑e: int)
IsEmpty(): bool
end IntStack

implementation of IntStack
const size = 100
var stack: array[1:size] of int
var top: int
...

nicht möglich (Geheimnisprinzip!)

Verwendung von Modulen

Häufige Unterscheidung

- **funktionsorientierte** Module
 - ... fassen verwandte Funktionen zusammen
(z.B. Mathematik-Funktionen, grafische Ausgabe)
- **aufgabenorientierte** Module
 - ... lösen eine Teilaufgabe eines größeren Problems
(z.B. Mailverarbeitung)
- **datenorientierte** Module.
 - ... kapseln Daten von der Umgebung und stellen Zugriffsfunktionen zur Verfügung (z.B. Stack, Konto, Hashtabelle, Stichwortverzeichnis)

Beispiel: Hashtabelle (ohne Modul)

Mehrere Algorithmen mit Gedächtnis (table)

Deklarationen (ohne Modul):

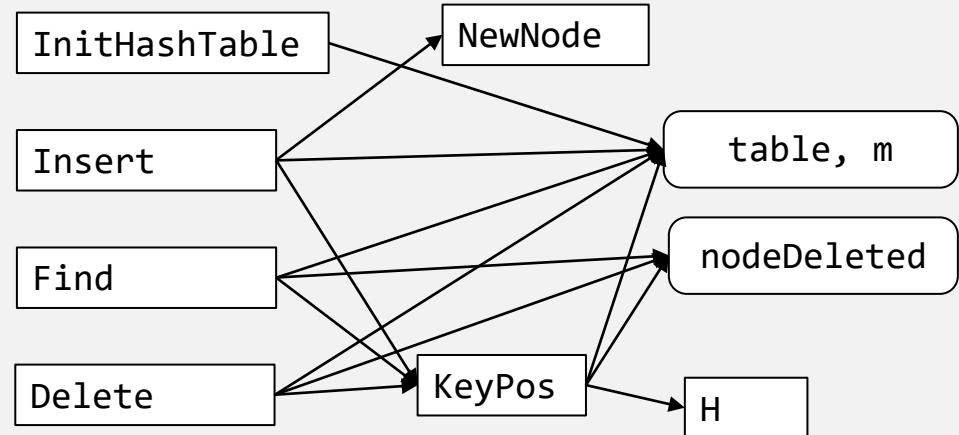
```
const
  m = ...
  nodeDeleted = NewNode("")

type
  HashTable = array...
  NodePtr = ...

var
  table: HashTable

InitHashTable()
NewNode(↓key: string): NodePtr
Insert(↓key: string ↓data...)
Find(↓key: string): NodePtr
Delete(↓key: string)
KeyPos(↓key: string): int
H(↓key: string): int
```

Verwendungsbeziehungen:



Beispiel: Hashtabelle (mit Modul)



Modul-Deklarationen

```
interface of MHashTable
```

```
end MHashTable
```

```
implementation of MHashTable
```

```
init
```

```
end MHashTable
```

Richtlinien zur Modulbildung

Sicherstellen der Modulgeschlossenheit

- möglichst starke Bindung zwischen Operationen und Daten,
- es soll keine Operationen und Daten geben, die in keinem Zusammenhang zueinander stehen (gilt für aufgaben- und datenorientierte Module)

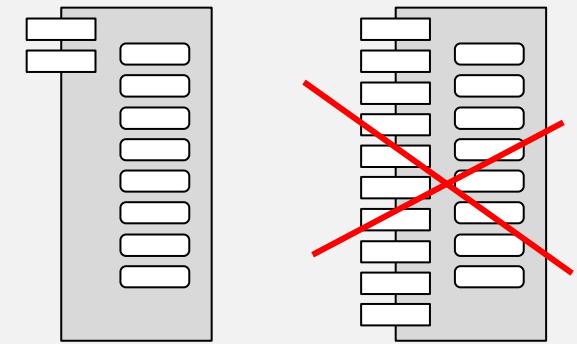
Sicherstellung einer geringen Modulkopplung (MK)

- Modulkopplung ... wie stark sind versch. Module untereinander verbunden
- geringe MK: Maß für die Unabhängigkeit der Module
- hohe MK: Hinweis darauf, das logisch zusammengehörende Operationen über mehrere Module verteilt sind
- globale (exportierte) Datenobjekte führen oft zu hoher MK

Richtlinien zur Modulbildung

Sicherstellung beherrschbarer Schnittstellen

- kleine Schnittstelle -> kleine Modulkopplung
- keine globale Datenobjekte



Sicherstellung der Testbarkeit

Sicherstellung von Interferenzfreiheit

- sie ist nicht gegeben, wenn ein Modul mehrere Aufgaben bearbeitet oder eine Aufgabe auf mehrere Module verteilt ist

13.4 Abstrakte Datenstruktur (ADS)

- Menge von Datenobjekten die bestimmte Abstraktion darstellen (z.B. Stack, Baum, Konto, Autorenverzeichnis, ...)
- Beschreibung, welche Datenobjekte verwaltet werden (z.B. Konto)
- Beschreibung der erlaubten Operationen
- Keine Beschreibung, wie Datenstruktur implementiert ist

Beispiel: Abstrakte Datenstruktur für Bankkonto

- Konto mit Anfangsguthaben, Liste von Kontobewegungen und Saldo
- Operationen
 - Initialisieren
 - Einzahlen und Abheben
 - Saldo abfragen
 - Kontoauszug drucken

Zugriffsoperationen

Implementierung der Operationen durch Prozeduren/Funktionen mit Parametern

```
interface of Account
  -- Initialisiere Konto
  InitAccount(↓value: real)
  -- Zugriffsoperation Einzahlung
  Deposit(↓purpose: string ↓amount: real)
  -- Zugriffsoperation Auszahlung
  Withdraw(↓purpose: string ↓amount: real)
  -- Zugriffsoperation Saldo
  Balance():real
  -- Zugriffsoperation Kontoübersicht
  PrintAccount()
end Account
```

Gesamtheit aller Zugriffsoperationen mit Parametern wird *Signatur* einer ADS genannt

Dokumentation der Zugriffsoperationen

Zugriffsoperationen müssen dokumentiert werden, um ihre Semantik und Benutzung zu erläutern

Schnittstelle	Withdraw(\downarrow purpose: string \downarrow amount: real)
Wirkung	Fügt neue Kontobewegung mit angegebenen Zweck, Betrag und der aktuellen Zeit hinzu und aktualisiert den Saldo.
Vorbedingungen	InitAccount() wurde aufgerufen purpose ist nicht leer amount > 0 Balance() ist größer als amount
Nachbedingung	Balance() liefert den um amount verringerten Betrag
Fehlerverhalten	Wenn eine Vorbedingung verletzt ist, hat diese Operation keine Wirkung.

- Keine Angabe über Aufbau der Datenstruktur
- Keine Angabe über Implementierung der Zugriffsoperationen

Dokumentation der Zugriffsoperationen

Beispiele

Schnittstelle

pop

```
public E pop()
```

Wirkung

Removes the object at the top of this stack and returns that object as the value of this function.

Returns:
The object at the top of this stack (the last item of the `Vector` object).

Throws:
`EmptyStackException` - if this stack is empty.

Vorbedingung und Fehlerbehandlung

get

```
E get(int index)
```

Returns the element at the specified position in this list.

Parameters:
`index` - index of the element to return

Returns:
the element at the specified position in this list

Throws:
`IndexOutOfBoundsException` - if the index is out of range (`index < 0 || index >= size()`)

13.5 Abstrakte Datentypen (ADT)

- Abstrakte Datentypen definieren Menge gleicher ADS
- ADT können wie echte Datentypen verwendet werden

Beispiel: Mehrere Bankkonten

```
var
    savingsAccount: Account
    account: Account
```

```
InitAccount(↓↑account ↓1000)
Withdraw(↓↑account ↓"iPhone" ↓449)
```

```
if Balance(↓↑account) > 200 then
    Withdraw(↓↑account ↓"Sparbuch" ↓200)
    InitAccount(↓↑savingsAccount ↓200)
else
    InitAccount(↓↑savingsAccount ↓0)
end -- if
```

jede Zugriffsoperation wird auf eine best. DS ausgeübt

Datenstrukturen (Aufbau
bleiben geheim!)

Schnittstelle

- Datenobjekte bleiben wieder vor der Außenwelt verborgen
- Manipulation der (abstrakten) Datenobjekte wieder nur über Zugriffsoperationen möglich
- Jede Zugriffsoperation hat einen zusätzlichen Parameter: das zu manipulierende Exemplar des ADT

Beispiel: Abstrakter Datentyp IntStack:

```
interface of IntStack
type
  Stack
  Name muss in Schnittstelle bekannt
  sein – Typ soll aber geheim bleiben
  Änderung im Vergleich zur Schnittstelle
  von abstrakten Datenstrukturen
  Aufräumarbeiten z.B. wenn als Liste realisiert
  InitStack(↑s: Stack)
  Push(↓↑s: Stack ↓e: int)
  Pop(↓↑s: Stack ↑e: int)
  IsEmpty(↓s: Stack): bool
  DisposeStack(↓↑s: Stack)
end IntStack
```

Implementierung

legt konkreten Typ der gekapselten Datenobjekte fest

```
implementation of IntStack
type
  Stack = compound
    stack: array[1:100] of int
    top: int
  end -- compound

InitStack(↑s: Stack)
begin
  s.top := 0
end InitStack

Push(↓↑s: Stack ↓e: int)
begin
  s.top := s.top + 1
  s.stack[s.top] := e
end Push
...
end IntStack
```

Konkreter Typ für Stack

Vorteile abstrakter Datentypen

Können verwendet werden, um flexibel beliebig viele Exemplare komplexer Datenstrukturen (z.B. Vektoren, Mengen, Verzeichnisse wie Hashtabellen) zu bilden und zu verwenden,

- deren konkreter Aufbau und deren Komplexität verborgen bleibt ([Abstraktion](#)),
- deren konkrete Implementierung ohne Wissen der Klienten geändert/ausgetauscht werden kann ([Änderungsfreundlichkeit](#)),
- deren missbräuchliche Verwendung verhindert wird ([Sicherheit](#)).

Zusammenfassung

- Eine **abstrakte Datenstruktur** (*abstract data structure*), kurz ADS, ist ein algorithmischer Baustein, der über eine spezielle Schnittstelle eine Menge von (abstrakten) Zugriffsoperationen zur Verfügung stellt, mit denen eine Menge gekapselter, d.h. abstrakter Datenobjekte (deren konkrete Realisierung dem Benutzer einer Zugriffsoperation verborgen bleibt) einzeln oder als Ganzes manipuliert werden kann.
- Ein **abstrakter Datentyp** (*abstract data type*), kurz ADT, definiert eine Menge von Datenobjekten, die alle dieselbe abstrakte Datenstruktur haben und auf die nur mittels (abstrakter) Zugriffsoperationen zugegriffen werden kann.
- Unter einem **Modul** (*module*) im Sinne der Informatik verstehen wir eine Sammlung von Algorithmen und Datenobjekten zur Bearbeitung einer in sich abgeschlossenen Aufgabe.

Ausblick auf objektorientierte Programmierung

Eines der wichtigsten Konzepte der objektorientierten Programmierung sind erweiterbare abstrakte Datentypen (z.B. in Form von Klassen)

Implementation:

```
const
  size = 100
type
  Stack = class
    data: array [1:size] of int
    top: int
    Init()
    Push(↓e: int)
    Pop(↑e: int)
    IsEmpty(): bool
  end -- Stack
```

```
Stack.Push(↓e: int)
begin
  top := top + 1
  data[top] := e
end Stack.Push
```

Verwendung:

```
var
  a, b: Stack
a.Push(↓42)
b.Push(↓12)
if a.IsEmpty() then
  ...
end -- if
```

Algorithmen und Datenstrukturen

Eine systematische Einführung in die

Programmierung

14 Systementwurf - Entwurfssparadigmen

Josef Pichler

Version 10.1, 2024

Motivation

Softwareentwickler·innen neigen dazu, voreilig mit der Implementierung zu beginnen

- unzureichendes Problemverständnis
- Details werden zu früh betrachtet

Folgen daraus sind

- systemspezifische (vom Rechner abhängige) Lösungen
- unüberschaubare Algorithmen
- Ideen nicht mehr erkennbar
- schlecht zu verstehen und zu warten

Entwurf ist wichtig

- je größer ein Problem, desto wichtiger ist der Entwurf
- je „sauberer“ der Entwurf, desto weniger Probleme treten in Implementierung auf

14.1 Grundsätzliche Entwurfsprinzipien

Wichtigstes Prinzip zur Meisterung der Komplexität, ist die **Abstraktion**

Wird erreicht durch

- Zerlegung von Systemen in Subsysteme und Komponenten
(Modellierung von System- und Komponentenschnittstellen)
- Zerlegung von Aufgaben in Teilaufgaben
- Sukzessive Konkretisierung von Subsystemen, Komponenten und erforderlichen Algorithmen und Datenstrukturen
- Vernachlässigung (Abstraktion) von Implementierungs- und Realisierungsdetails

Betrachtung jeweils nur jener Aspekte eines Systems, die für den nächsten Lösungsschritt von Bedeutung sind

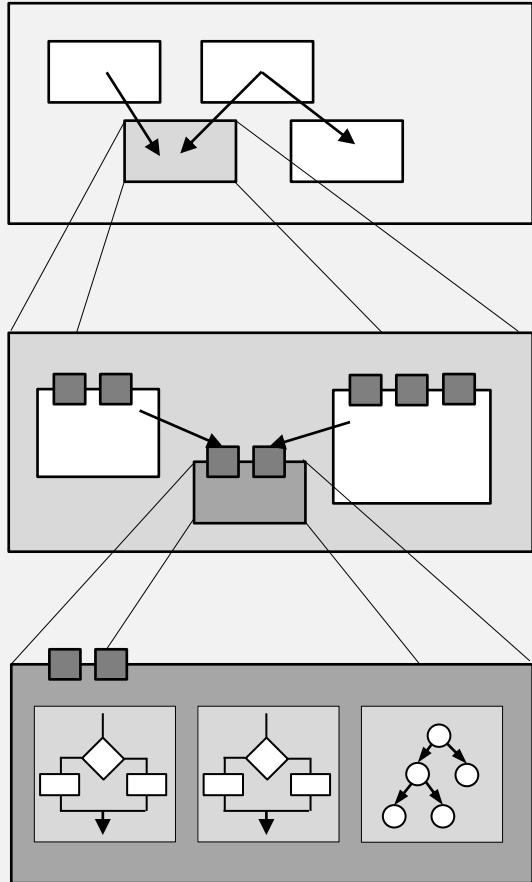


Grady Booch @Grady_Booch

The entire history of software engineering is about the rising levels of abstraction.

twitter.com, 29. Jan. 2023

Entwurfsebenen



Zerlegung in Subsysteme

- schwache Kopplung zwischen Subsystemen
- Kommunikation über Dateien, Datenbanken, Netzwerk

Modularisierung

- Zerlegung eines Subsystems in handhabbare Teile
- enge Kopplung
- Kommunikation über Aufrufsschnittstellen

Algorithmenentwurf

- Verfeinerung von Algorithmen und Datenstrukturen

Vorgehensprinzipien

Topdown-Entwurf

- vom Abstrakten zum Konkreten
- Zerlegung eines großen Systems (einer Aufgabe) in mehrere kleinere Subsysteme (Teilaufgaben)
- saubere Schnittstellen, gute Modularisierung
- kritische Integration am Ende

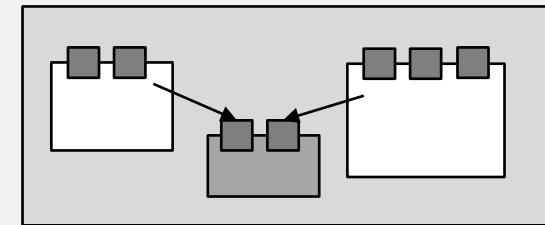
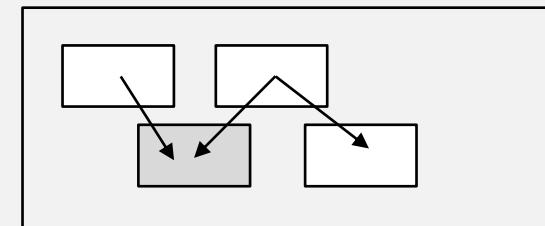
Bottomup-Entwurf

- vom Konkreten zum Abstrakten
- zusammenfassen mehrerer kleiner Bausteine zu einem größeren System (Subsystem)
- hoher Wiederverwendungsgrad
- inkrementelle Tests, schrittweise Integration
- beginnt mit „vermuteten Teilproblemen“

Architektur- vs. Komponenten-Design

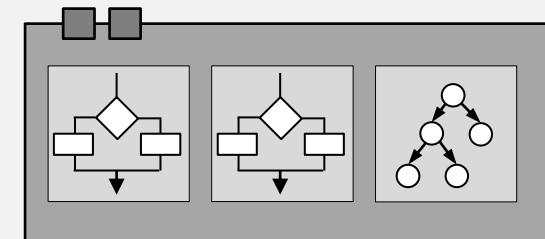
Architektur-Design (Grobentwurf)

- System-, Subsystementwurf
- Festlegung der System-, Subsystemstruktur
(Wechselwirkungen zwischen den Komponenten),
Komponentenschnittstellen



Komponenten-Design (Feinentwurf)

- Komponentenentwurf
- Festlegung der erforderlichen Algorithmen,
Datenstrukturen,



Zerlegungsarten – Programmierparadigmen

Aufgabenorientierte Zerlegung (Aufgaben- und modulorientierte Programmierung)

- geht von Aufgabe aus und zerlegt sie in Teilaufgaben
- (hierarchische) Zerlegung eines Systems in seine funktionalen Bestandteile
- im Mittelpunkt stehen die **funktionalen Aspekte**

Datenorientierte Zerlegung (Daten- und transformationsorientierte Programmierung)

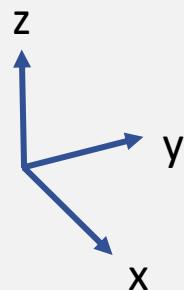
- geht von Ein/Ausgabe-Daten aus, deren Struktur und Komplexität die Systemstruktur beeinflussen
- Zerlegung eines Systems in Teile, gemäß den zu verarbeitenden Datenstrukturen
- im Mittelpunkt stehen die datenbezogenen Aspekte

Objektorientierte Zerlegung (Objekt- und komponentenorientierte Programmierung)

- geht davon aus, dass Datenobjekte und Operationen untrennbare Einheiten bilden
- Zerlegung eines Systems in Objekte/Komponenten (= Datenobjekt und darauf anwendbare Operationen – sogenannte aktive Datenobjekte)
- im Mittelpunkt steht die Auffassung, dass Datenobjekte und Operationen eine Einheit bilden

14.2 Aufgabenstellung für Entwurfsbeispiel

Beispiel. Zu Simulations- und Bedienungszwecken soll eine Applikation für einen 3D-Drucker entwickelt werden.



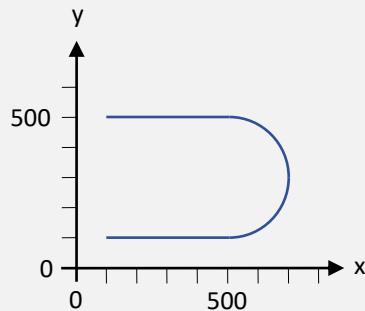
Bildquelle: <https://computerwelt.at/news/3d-drucker-zum-selberbauen/>

Aufgabenstellung Entwurfsbeispiel

- Der Bewegungsablauf des 3D-Druckers (mit 3 Achsen) wird durch eine spezielle (vorgegebene) Sprache beschrieben.
- Beschreibung der Bewegungsbahn: Die Bewegungsbahn ist eine Folge von linearen Teilschritten. Zwei Bahnformen können beschrieben werden:
 - Lineare Bewegung zum Punkt (x, y, z) in n Schritten: $n \ L \ (x, y, z)$
Beispiel: $5 \ L \ (100, 200, 300)$
 - Kreisbewegung zum Punkt (x, y, z) über einen Zwischenpunkt (x_1, y_1, z_1) in n Schritten: $n \ C \ (x_1, y_1, z_1) \ (x, y, z)$
Beispiel: $20 \ C \ (50, 50, 50) \ (100, 100, 100)$
- Der Endpunkt einer Bahn ist gleichzeitig der Anfangspunkt der nächsten Bahn.

Beispiel:

```
(100,100,100)
10 L (500,100,100)
20 C (700,300,100)(500,500,100)
10 L (100,500,100)
E
```



Aufgabenstellung Entwurfsbeispiel

Steuerung und schematische Visualisierung des 3D-Druckers:

- Der Bewegungsablauf soll am Bildschirm kontinuierlich und in Einzelschritten verfolgt werden können.
- Der 3D-Drucker soll von der Applikation aus angehalten werden können.

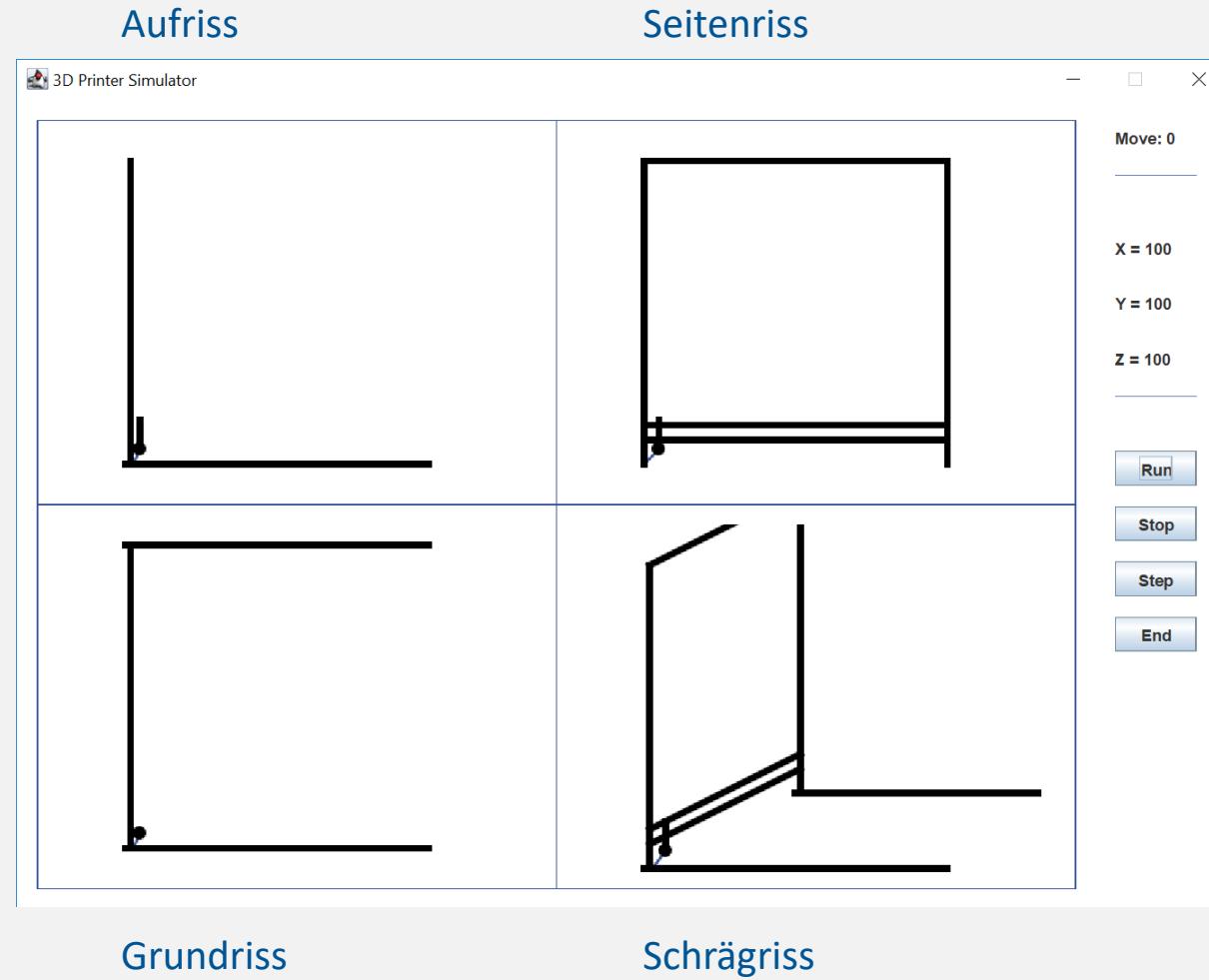
Nebenbedingungen:

- Der Druckkopf des 3D-Druckers soll auf einen würfelförmigen Raum von 20 cm Kantenlänge beschränkt sein.
- Der Druckkopf darf diesen Raum nicht verlassen.

Dem Prototyping-orientierten Entwicklungsansatz entsprechend, wird zunächst zur Exploration der Anforderungen an die zu entwickelnde Software ein Prototyp entwickelt

(zur Überprüfung, ob zwischen Auftraggeber und Auftragnehmer Konsens hinsichtlich der Anforderungen und Ausgestaltung des zu entwickelnden Softwaresystems besteht).

Illustration durch einen Prototyp



Algorithmen und Datenstrukturen

Eine systematische Einführung in die Programmierung

15 Aufgaben- und modulorientierter Systementwurf

Josef Pichler

Version 10.1, 2024

15.1. Aufgabenorientierte Zerlegung - Grundprinzip

Wiederholung des bereits in ADE1 vorgestellten Prinzips in einem größeren Problemkontext.

Entwurf der Systemarchitektur durch Anwendung des Prinzips der schrittweisen Verfeinerung (*stepwise refinement*)

- Zerlege eine Aufgabe in Teilaufgaben
- Betrachte jede Teilaufgabe für sich und möglichst unabhängig von anderen Teilaufgaben
- Zerlege sie wieder in Teilaufgaben, bis diese so einfach geworden sind, dass ihre Lösung sich auf einfache Weise durch einen Algorithmus beschreiben lässt

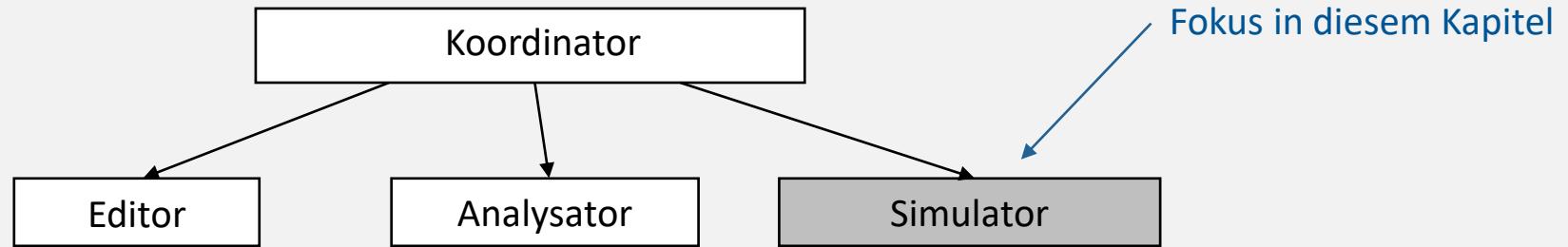
Das bedeutet

- zunächst nur Algorithmen-Schnittstellen festzulegen
- unwichtige Einzelheiten zurückzustellen
- zuerst wichtigste Aspekte der Aufgabe zu identifizieren
- die Aufgaben-Komplexität stetig zu vermindern

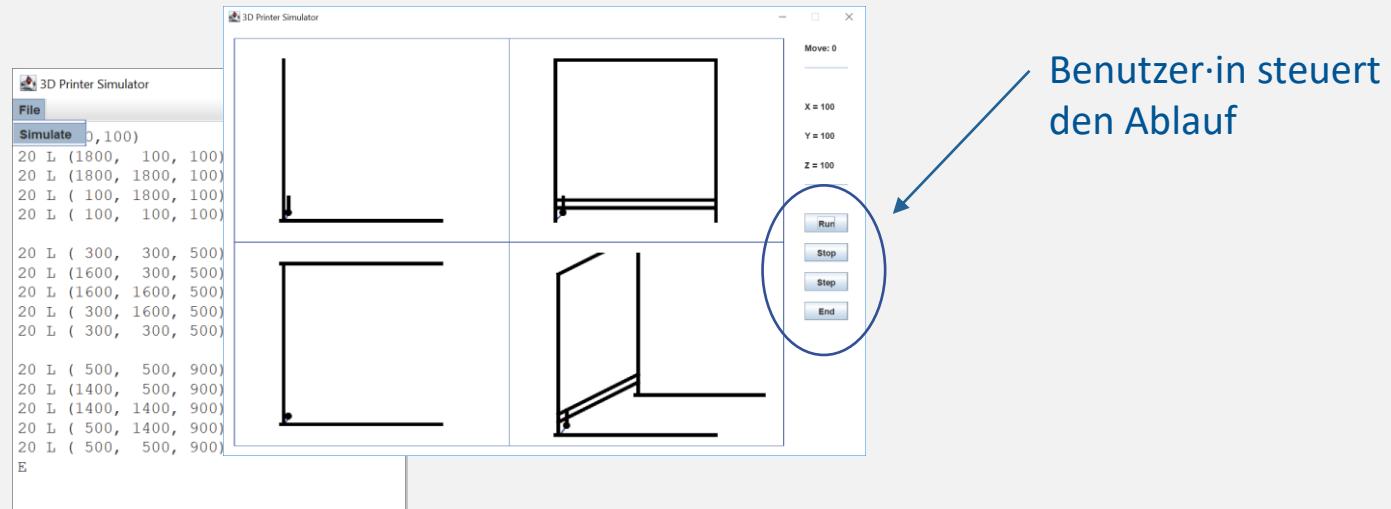
Hinweis: Schrittweise Verfeinerung ist ein iterativer Prozess!

15.2 Entwurf der Grobarchitektur

Gemäß Prototyp ist es sinnvoll, das System in vier Komponenten zu zerlegen:

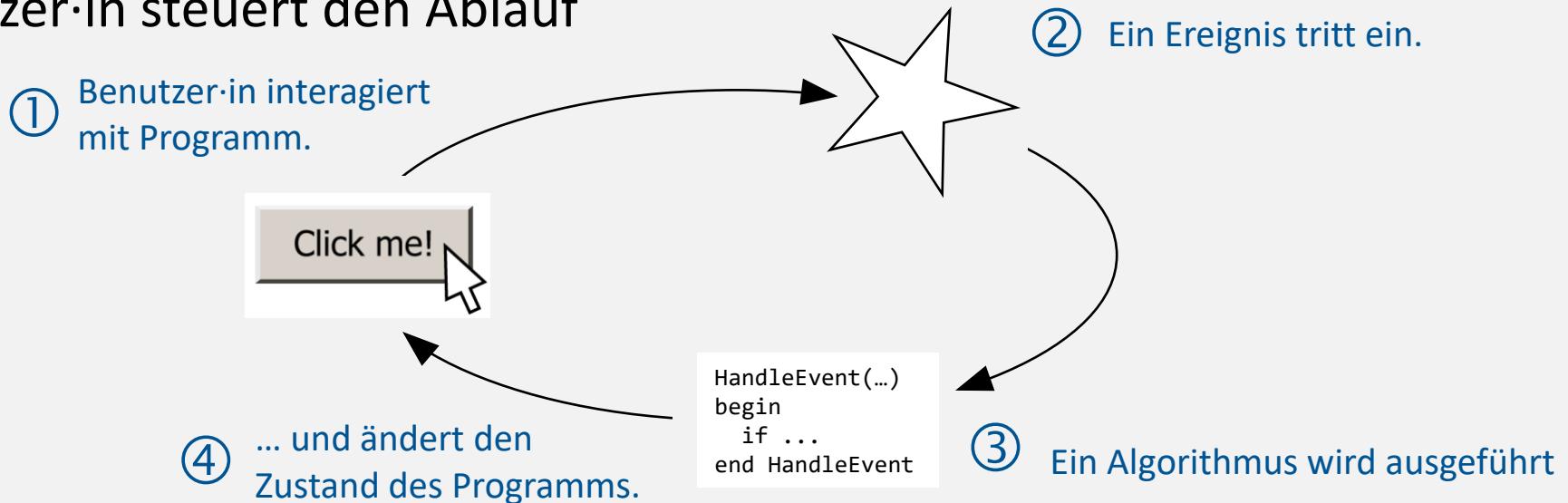


Simulator ist ein ereignisgesteuertes System: Benutzer·in steuert den Ablauf



Ereignisgesteuerte Programmierung

Benutzer·in steuert den Ablauf



Das Hauptprogramm besteht aus einer Ereignisschleife

```
import Windows  
  
exit := false  
while not exit do  
  e := GetNextEvent()  
  HandleEvent(↓e ↑exit)  
end -- while
```

Importiert aus Modul des Betriebssystems

```
interface of Windows type  
Event = compound  
kind: ...  
x, y: int  
end -- Event  
GetNextEvent(): Event  
end Windows
```

Identifikation der Ereignisarten

Schnittstelle der Komponente Simulator

Benutzergesteuerte Ereignisse:

- buttonEvent mit Code (runButton, stepButton etc.) für ged. Knopf



```
type  
CodeType = (runButton,  
            stepButton,  
            stopButton,  
            endButton)
```

Verwaltungsereignisse:

- initEvent
- backgroundEvent
- updateEvent

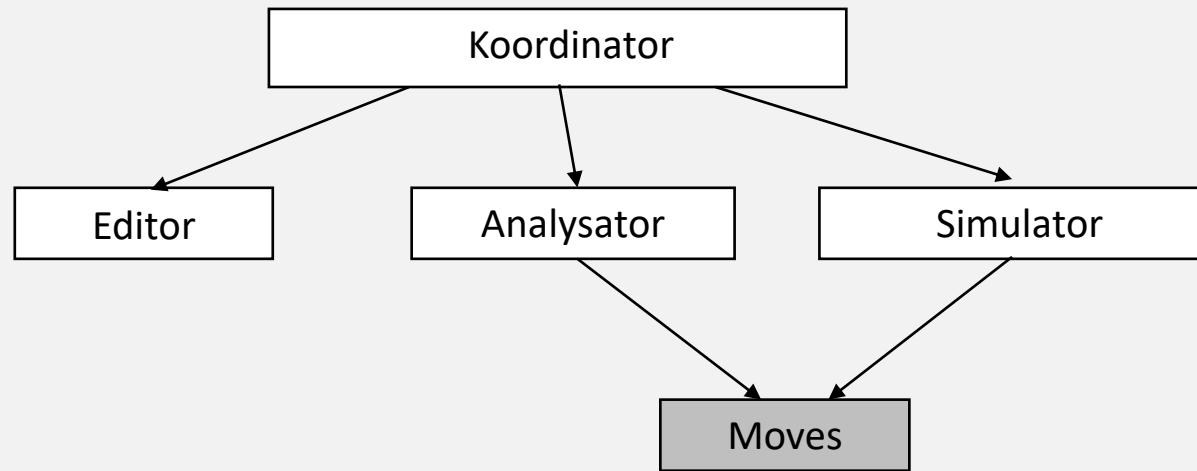
```
type  
EventType = (buttonEvent,  
              initEvent,  
              backgroundEvent,  
              updateEvent)
```

Event Handler – Schnittstelle:

```
HandleEvent(↓kind: EventType ↓code: CodeType ↑exit: bool)
```

Verfeinerung der Grobarchitektur

Einführung einer Datenkapsel **Moves** zur Kapselung der Bewegungsbeschreibung und als Kommunikationsplattform der Systemkomponenten Analysator und Simulator.



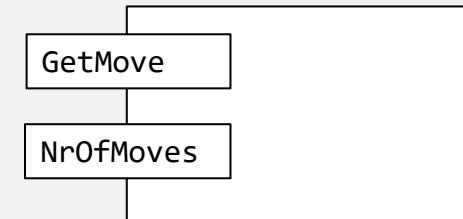
Verfeinerung der Grobarchitektur

Zugriffsoperationen der Datenkapsel Moves

GetMove(\downarrow nr: int \uparrow x: int \uparrow y: int \uparrow z: int)

Festlegung: GetMove(\downarrow 0...) liefert den Startpunkt

NrOfMoves(): int



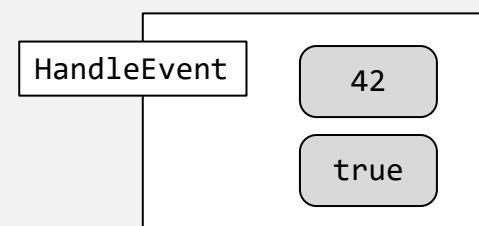
Datenobjekte für Simulation (Komponente Simulator)

var
stepNr: int

Zähler für Bewegungsschritte
(aktueller Bewegungsschritt)

var
running:boolean

Information, ob der Drucker
gerade in Bewegung ist
(oder stillsteht)



Verfeinerung der Grobarchitektur

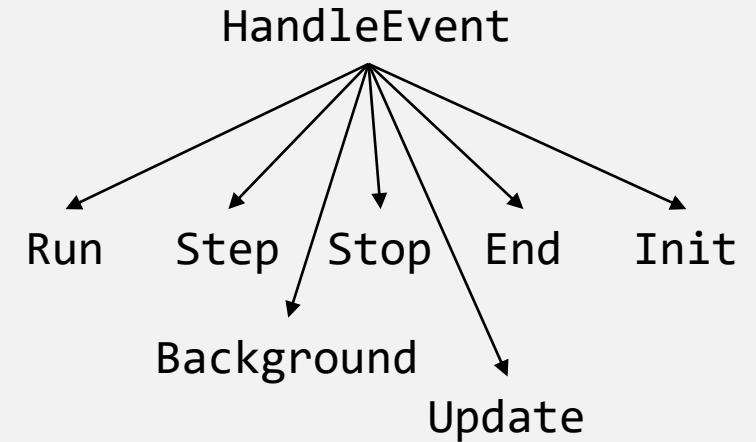
Algorithmus HandleEvent

```
HandleEvent(↓kind: EventType ↓code: CodeType ↑exit: bool)
begin
    exit := false
    case kind of
        buttonEvent:
            case code of
                runButton: Run()
                stepButton: Step()
                stopButton: Stop()
                endButton: End(); exit := true
                otherwise: "unknown code => handle error"
            end -- case
        initEvent: Init()
        backgroundEvent: Background()
        updateEvent: Update()
        otherwise: "unknown event => handle error"
    end -- case
end HandleEvent
```

Verfeinerung der Grobarchitektur

Ereignisverarbeitung

- der Algorithmus HandleEvent identifiziert und delegiert die Aufgabenbearbeitung (Run, Step, Stop, End, Init, Background, Update)



- Typisch für aufgabenorientierte schrittweise Verfeinerung
- Besonders charakteristisch bei ereignisorientierten Systemen

15.3 Aufgabenorientierter Zerlegungsprozess

Algorithmisierung der identifizierten Teilaufgaben Init, Run und Stop:

Initialisierung der
3D-Drucker-Applikation

```
Init()
  var
    x, y, z: int
begin
  stepNr := 0
  running := false
  OpenWindow()
  GetMove(↓0 ↑x ↑y ↑z)
  MovePrinterTo(↓x ↓y ↓z)
  Update()
end Init
```

Starten des 3D-Druckers

```
Run()
begin
  if stepNr < NrOfMoves() then
    running := true
  end -- if
end Run
```

Anhalten des 3D-Druckers

```
Stop()
begin
  running := false
end Stop
```

Aufgabenorientierter Zerlegungsprozess

Algorithmisierung der identifizierten Teilaufgaben Step und Background:

Bewegungsschritt des
3D-Druckers

```
Step()
  var
    x, y, z: int
begin
  if stepNr < NrOfMoves() then
    stepNr := stepNr + 1
    GetMove(↓stepNr ↑x ↑y ↑z)
    MovePrinterTo(↓x ↓y ↓z)
    Update()
  end -- if
end Step
```

```
Background()
begin
  if running then
    Step()
  end -- if
end Background
```

- `running` bleibt `true`, auch wenn die letzte Bewegung schon ausgeführt wurde
- Beim Entwurf von `Step` wurde dieser Fall (noch) nicht berücksichtigt
- Zurück zur Verfeinerung von `Step`

Aufgabenorientierter Zerlegungsprozess

Korrektur von Step:

Bewegungsschritt des
3D-Druckers

```
Step()
  var
    x, y, z: int
begin
  if stepNr < NrOfMoves() then
    stepNr := stepNr + 1
    GetMove(↓stepNr ↑x ↑y ↑z)
    MovePrinterTo(↓x ↓y ↓z)
    Update()
  else
    Stop()
  end -- if
end Step
```

Ergänzung: Nach der letzten Bewegung
wird running auf false gesetzt.

An sämtlichen Aufrufstellen von Step() muss kontrolliert werden,
ob diese Änderung Konsequenzen auf andere Prozeduren hat.

15.3 Aufgabenorientierter Zerlegungsprozess

Algorithmisierung der identifizierten Teilaufgaben End und Update:

Beenden der Simulation

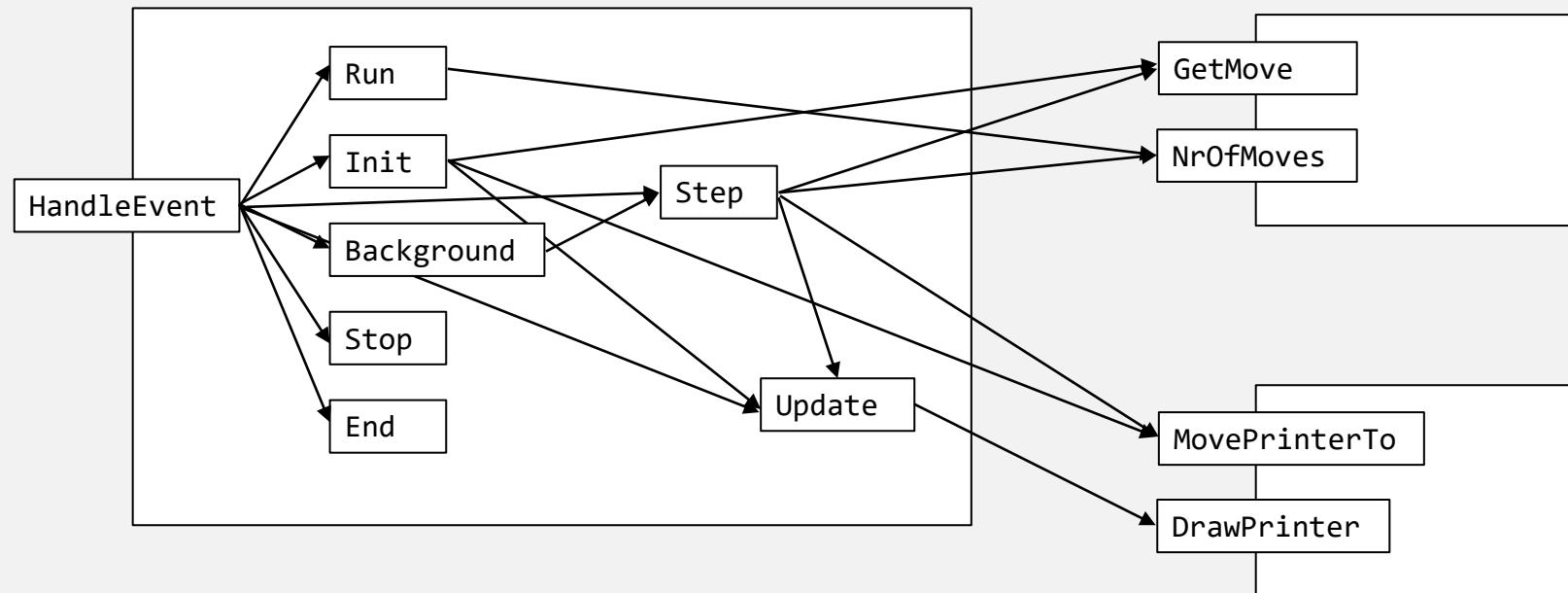
```
End()  
begin  
    CloseWindow()  
end End
```

Neuzeichnen des Simulations-Fensters

```
Update()  
begin  
    DrawPrinter()  
    ShowCoordinates()  
end Update
```

Aufgabenorientierter Zerlegungsprozess

Zur Bildung von Modulen expliziter Syntheseschritt erforderlich!



Aufgabenorientierter Zerlegungsprozess

Bisheriger Erfolg:

- Benutzerdialog vom eigentlichen Zeichnen abgekoppelt
- Teilprobleme mit kleiner Komplexität
- Verbleibender Rest ist einfacher als die Gesamtaufgabe
- Rest kann unabhängig von der bisherigen Lösung weiterentwickelt werden

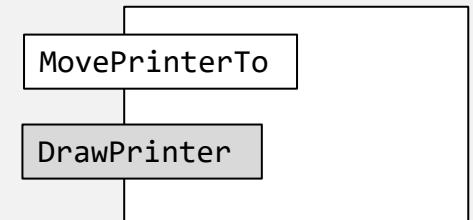
Bei Vornahme der Modulbildung (Syntheseschritt) zwei Arten von Architekturkomponenten:

- Module zur Strukturierung im Großen
- Prozeduren zur Strukturierung im Kleinen

Aufgabenorientierter Zerlegungsprozess

Teilproblem DrawPrinter

- Datenkapsel mit aktueller Position (x, y, z)
- Position wird durch MovePrinterTo eingestellt
- Zerlegungsproblem: Zwei Möglichkeiten der Primärzerlegung:
 - Vier Sichten getrennt behandeln; innerhalb jeder Sicht nach den einzelnen Teilen verfeinern
 - Teile getrennt behandeln; Zeichnen jedes Teils nach den vier Sichten verfeinern
- Kein Patentrezept möglich; durch Orthogonalität entstehende Reihenfolgeprobleme müssen intuitiv gelöst werden
- Erfahrung des Entwerfers ist für die Qualität des Ergebnisses von entscheidender Bedeutung
- Lösungsansatz: Primärzerlegung nach den vier Sichten (gezeigt am Beispiel des Aufrisses)



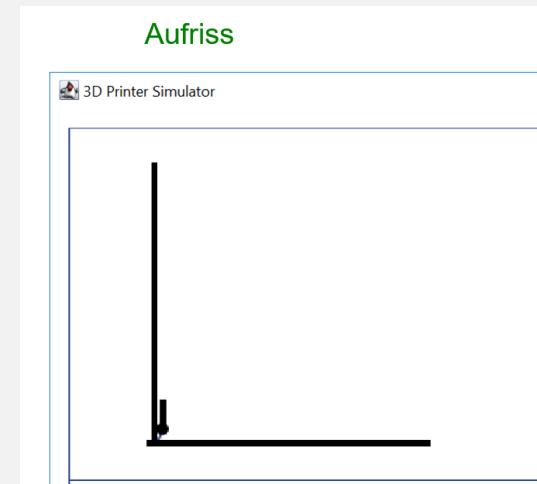
Zeichnen des Aufrisses

Zerlegung gemäß der Bestandteile

- Schienen
- Portal
- Druckkopf

Zeichnen des Aufrisses (...Front)

```
DrawPrinterFront()  
begin  
    DrawRailsFront()  
    DrawGateFront()  
    DrawPrintHeadFront()  
end DrawPrinterFront
```



Zeichnen des Aufrisses

Zeichnen der Schienen

```
DrawRailsFront()  
begin  
    MoveTo(↓0 ↓0)  
    LineTo(↓2100 ↓0)  
end DrawRailsFront
```

Die vordere Schiene verdeckt die hintere;
es muss nur eine gezeichnet werden

Zeichnen des Portals (Nur vorderer Pfosten muss gezeichnet werden)

```
DrawGateFront()  
begin  
    MoveTo(↓x ↓50)  
    LineTo(↓x ↓2100)  
end DrawGateFront
```

Nur vorderer Pfosten muss gezeichnet
werden

Zeichnen des Aufrisses

Zeichnen des Druckkopfs

```
DrawPrintHeadFront()  
begin  
    MoveTo(↓x + 50 ↓z + 50)  
    LineTo(↓x + 50 ↓z + 200)  
    DrawCircle(↓x + 50 ↓z + 50 ↓100)  
end DrawPrintHeadFront
```

Zeichnen des 3D-Druckers

```
DrawPrinter()
    DrawPrinterFront()
        DrawRailsFront()
        DrawGateFront()
        DrawPrintHeadFront()
    DrawPrinterTop()
        DrawRailsTop()
        DrawGateTop()
        DrawPrintHeadTop()
    DrawPrinterRight()
        DrawRailsRight()
        DrawGateRight()
        DrawPrintHeadRight()
    DrawPrinterReal()
        DrawRailsReal()
        DrawGateReal()
        DrawPrintHeadReal()
```

Insgesamt 17 Algorithmen mit
sehr geringer Komplexität

Aufgabenorientierter Zerlegungsprozess – Zusammenfassung

Ziele

- Systematisierung des Entwurfsprozesses
- Finden einer günstigen Zerlegung eines Programmsystems in Prozeduren
- Meisterung der Komplexität durch Abstraktion

Vorteile

- mit etwas Übung fast mechanisch anwendbar
- universell anwendbar (auch für Teilaufgaben, die nicht durch schrittweise Verfeinerung entstanden sind)
- ermöglicht arbeitsteiligen Entwurf

Nachteile

- unterstützt die Modularisierung nicht explizit (Syntheseschritt erforderlich, s. S. 14)
- unterstützt nicht die Zerlegung der Datenstrukturen
- konsequentes topdown-Vorgehen bei großen Aufgaben oft schwierig

Algorithmen und Datenstrukturen

Eine systematische Einführung in die Programmierung

16 Daten- und transformationsorientierter Systementwurf

Josef Pichler

Version 10.1, 2024

16.1 Zum Denkmodell des datenorientierten Systementwurfs

- In der Praxis findet man häufig Aufgabenstellungen, in denen es darum geht, dass **ein Strom strukturierter Datenobjekte** zu verarbeiten ist
- Softwaresysteme für solche Aufgabenstellungen können auf elegante und systematische Weise mit Hilfe von **attribuierten Grammatiken spezifiziert** werden
- Darauf aufbauend kann dann die **Systemarchitektur** und die **Systemimplementierung** abgeleitet werden
- Die solcherart entworfenen Lösungen beschreiben die Transformation eines gegebenen Stromes von Datenobjekten in eine andere Form
- Wir sprechen deshalb von **transformationsorientierter Software**
- Solche Systementwürfe können **automatisch** in Algorithmensysteme transformiert werden

Beispiele für Strom strukturierter Datenobjekte

Pascal-Programm

```
PROGRAM P;
  VAR a, b: INTEGER;
BEGIN
  Read(a); Read(b);
  IF a < b THEN ...
END.
```

Bewegungsdatei

```
(100,100,100)
10 L (500,100,100)
20 C (700,300,100)(500,500,100)
10 L (100,500,100)
E
```

Mail-Strom

marcel hirscher hat seine alpine
karriere offiziell beendet zzzz
ausbruch des corona... zzzz auftakt
der... zzzz zzzz

Arithmetische Ausdrücke

```
x * 3421 + 1
3 * (4 + 5)
(hs + d*q - dm*si) mod q
```

URL

<http://oeo.orf.at/stories>
<https://www.fh-oeo.at>
<ftp://lorem.ipsum.com:3343>

E-Mail-Adressen

alice@yahoo.org
bob@whitehouse.gov
gmail@chucknorris.com

Zum Denkmodell des datenorientierten Systementwurfs

- Das Denkmodell für den datenorientierten Entwurf in der Softwareentwicklung geht davon aus, dass aufbauend auf der **Struktur (Syntax)** eines Eingabedatenstroms seine **Bedeutung (Semantik)** und der angestrebte **Transformationsprozess** definiert und daraus die Systemarchitektur abgeleitet werden kann
- Zur Beschreibung der **syntaktischen Struktur** des Eingabedatenstromes können wir das aus dem Gebiet der formalen Sprachen bekannte Konzept der **Grammatiken** und für die Beschreibung des **Transformationsprozesses** das der **attribuierten Grammatiken** heranziehen
- Der **Syntaxanalyse** des Eingabedatenstroms kommt eine zentrale Bedeutung im Rahmen des Transformationsprozesses zu

Beispiele für Grammatiken

Pascal-Programm

```
PROGRAM P;
  VAR a, b: INTEGER;
BEGIN
  Read(a); Read(b);
  IF a < b THEN ...
END.
```

Program = ident [UsesClause] Block ".
Block = DeclPart BlockStat.
Stat = AssignStat | BlockStat | IfStat | ...
AssignStat = Lvalue ":=" Expr.
IfStat = "IF" Expr "THEN" Stat ["ELSE" Stat].
BlockStat = "BEGIN" Stat { ";" Stat } "END".
...

Mail-Strom

marcel hirscher hat seine alpine
karriere offiziell beendet zzzz
ausbruch des corona... zzzz auftakt
der... zzzz zzzz

S = { M } L.
M = W { W } "zzzz".
W = C { C } " " { " " } .
L = "zzzz".
C = "a" | "b" | ... | "z".

URL

<http://oeo.orf.at/stories>
<https://www.fh-oeo.at>
<ftp://lorem.ipsum.com:3343>

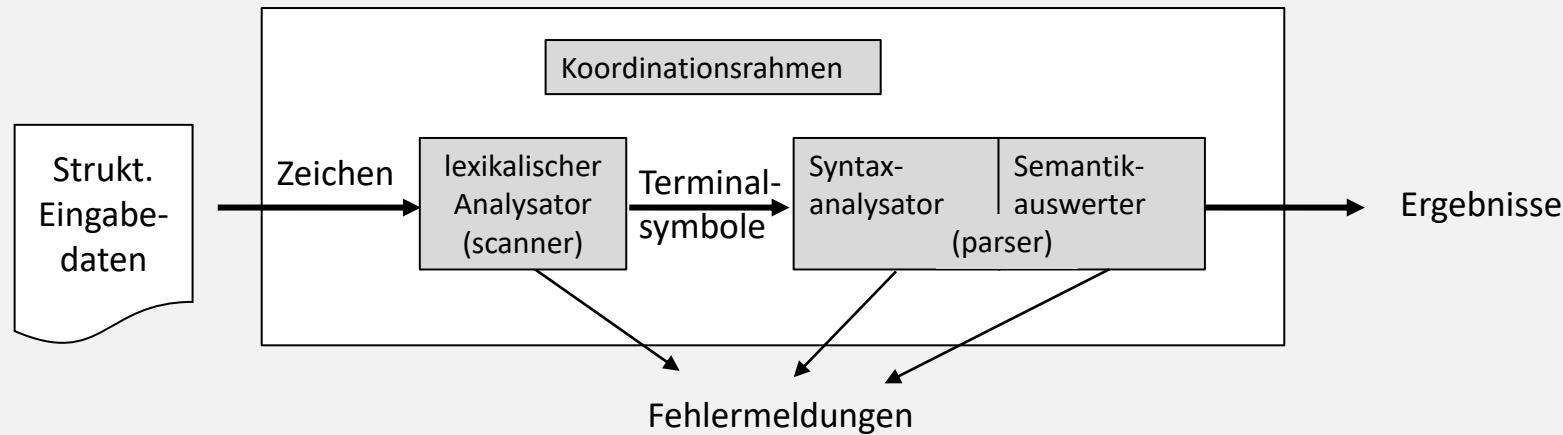
Url = Protocol "://" Host [":" Port] ["/" Path]
Protocol = "http" | "https" | "ftp".
Host = ident { "." ident }.
Port = number.
Path = ident.

Zum Denkmodell des datenorientierten Systementwurfs

- Die Konstruktion datenorientierter **Systemarchitekturen** erfordert die Entwicklung von **Analysatoren**, mit deren Hilfe die syntaktische Korrektheit der zu verarbeitenden Eingabedatenströme geprüft werden kann (den so genannten **lexikalischen Analysator** und den so genannten **Syntaxanalysator**)
- Um einen Eingabedatenstrom verarbeiten zu können, d. h. die gewünschten Ergebnisse aus ihm ermitteln bzw. diesen in eine andere Gestalt transformieren zu können, ist es notwendig, auch die semantischen Aspekte des Eingabedatenstroms zu berücksichtigen (deshalb benötigen wir den so genannten **Semantikauswerter**)

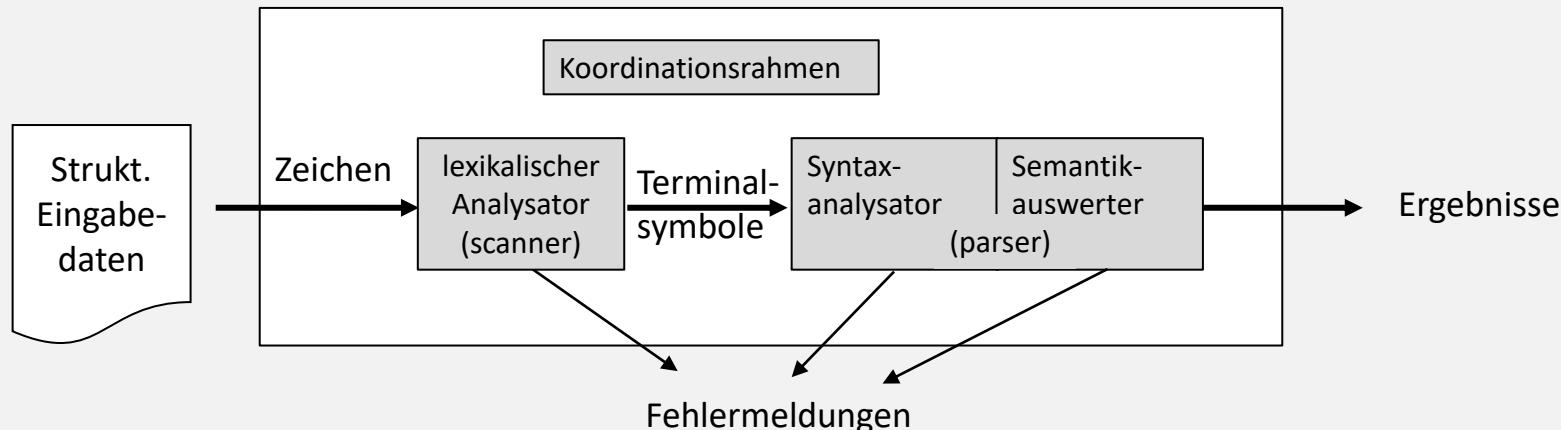
Zum Denkmodell des datenorientierten Systementwurfs

Grobarchitektur



- Grundstruktur der Systemarchitektur bleibt immer gleich (Referenzarchitektur)
- Architekturmuster Pipes & Filters

Zum Denkmodell des datenorientierten Systementwurfs



Beispiel

Eingabedaten	Zeichen	Terminalsymbole	Syntaxbaum	Ergebnis
"x * 34 + 1"	'x' '*' '3' '4' '+' '1'	ident mult number plus number	<pre>graph TD; plus["+"] --> mult["*"]; plus --> one["1"]; mult --> x["x"]; mult --> thirtyfour["34"];</pre> <p>Annahme: x = 2</p>	69

16.2 Grundlagen – Formale Sprachen und Grammatiken

- Grundelement ist ein Zeichen oder **Symbol (Terminalsymbol)**
- Das **Alphabet** ist eine endliche Menge von Zeichen oder Symbolen

$$V_1 = \{"0", "1"\}$$

$$V_2 = \{":", ";", "-", "(", ")"\}$$

$$V_3 = \{"if", "else", ":=", "<", \text{ident}, \dots\}$$

- Durch Hintereinanderschreiben entstehen Zeichen- bzw. **Symbolketten**
für V_1 z.B. `010` `0110` `01010` `0000111101010101`
- für V_2 z.B. `: -)` `: - (` `:-----` `))))))`
- für V_3 z.B. `if a < b then min := a else min := b`
- Einen **Eingabedatenstrom** fassen wir als Zeichen- bzw. **Symbolkette** auf
- die Menge V^+ über einem Alphabet V ist Menge aller nichtleeren Ketten, die sich aus Elementen von V bilden lassen
- die Menge V^* über einem Alphabet V ist Menge aller Ketten einschließlich der leeren Kette ϵ ($V^* = V^+ \cup \{\epsilon\}$)

Grundlagen – Formale Sprachen und Grammatiken

Definition Formale Sprache

- Eine formale Sprache L über einem Alphabet V ist eine Teilmenge von V^* :

$$L \subseteq V^*$$

- Ihre Elemente heißen Sätze
- Beispiel: Sprache mit vier Sätzen

$$V = \{":", ";", "-\}, "(", ")"\}$$

$$V^* = \{\varepsilon, ":", ";", "-\}, ")", "(", ":", ";", "-\}, :\}\}$$

$$L = \{":-)", ":-(", ";-)", ";-("}$$

- Teilmenge kann auf verschiedene Arten definiert werden
 - Aufzählung (für sehr einfache Sprachen mit wenigen Sätzen, siehe Beispiel)
 - Grammatik
 - Automaten

Definition Grammatik

Eine **Grammatik** G mit dem **Satzsymbol** S, geschrieben G(S), ist eine endliche, nicht leere Menge von **Ersetzungsregeln**, die zwei Arten von Symbolen enthalten:

- **Terminalsymbole**, die in Sätzen vorkommen,
- und **Nonterminalsymbole**, die der Strukturbeschreibung dienen.

Das Satzsymbol S ist ein ausgezeichnetes Nonterminalsymbol, das auf mindestens einer linken Seite der Ersetzungsregeln vorkommt.

Beispiel: Sprache mit vier Sätzen

	:-)	;-)	:-()	;-()
X	:-)	;-)	:-()	;-()
Y	:-)	;-)	:-()	;-()
Z	:-)	;-)	:-()	;-()

Grammatik G(S):

$$\begin{aligned}S &= X \ Y \ Z. \\X &= ":". \\X &= ";". \\Y &= "-". \\Z &= ")". \\Z &= "(".\end{aligned}$$

$$\begin{aligned}V_T &= \{":", ";", "-", "\)", "("\} \\V_N &= \{S, X, Y, Z\} \\&\text{Satzsymbol } S\end{aligned}$$

Definition Grammatik

- Eine Ersetzungsregel besteht aus einem Nonterminalsymbol A und einer Symbolkette α , einer Folge aus Terminal- und/oder Nonterminalsymbolen
- Eine Regel wird geschrieben als $A = \alpha.$ und gelesen als „A ist definiert als α “ oder „A kann ersetzt werden durch α “. Dabei heißt A „linke Seite“ und α „rechte Seite“ der Regel.
- Gibt es mehrere Möglichkeiten für die Ersetzung von A, kann anstelle von z. B. zwei Regeln $A = \alpha.$ und $A = \beta.$ auch $A = \alpha \mid \beta.$ geschrieben werden; das wird gelesen als „ α oder β “, dabei werden α und β als Alternativen von A bezeichnet

$$S = X \underbrace{YZ}_{\alpha}.$$

$$X = ":",$$

$$\begin{aligned} S &= X Y Z. \\ X &= ":". \\ X &= ";" \\ Y &= "-". \\ Z &= ")" \\ Z &= "(" \end{aligned}$$

$$\left. \begin{array}{l} X = ":" \mid ";" \\ Z = ")" \mid "(" \end{array} \right\} X = ":" \mid ";" \quad \rightarrow \quad Z = ")" \mid "("$$

$$\begin{aligned} S &= X Y Z. \\ X &= ":" \mid ";" \\ Y &= "-". \\ Z &= ")" \mid "(" \end{aligned}$$

$$\begin{aligned} S &= X "-" Z. \\ X &= ":" \mid ";" \\ Z &= ")" \mid "(" \end{aligned}$$

$$S = (":" \mid ";") "-" (")" \mid "(").$$

Beispiel Grammatik



Beispiel: Musiknoten (Version 1)

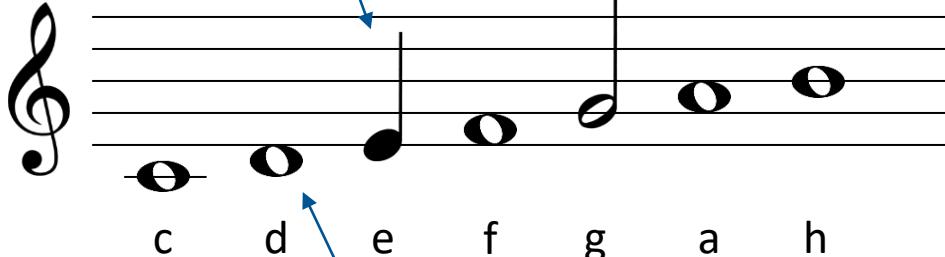
- Drei Noten
- Notennamen: c, d, e, f, g, a, h
- Notenwerte: 1, 2, 4, 8

g1, g2, a2

c1, d2, e4

a2, a4, a8

Viertel Note (4) Halbe Note (2)



Ganze Note (1)

- Grammatik G(Bar)

$V_T = \{",", "c", "d", "e", "f", "g", "a", "h", "1", "2", "4", "8\}$

Menge der Terminalsymbole

Menge der Nonterminalsymbole
und Satzsymbol

Ersetzungsregeln

Definition Grammatik

- **Nonterminalsymbole** sind all jene Symbole, die auf den linken Seiten der Ersetzungsregeln einer Grammatik stehen und als Abstraktionsmittel zur Gliederung der Sätze verwendet werden (das wichtigste Nonterminalsymbol ist das Satzsymbol)
- **Terminalsymbole** sind jene Symbole, die nur auf rechten Seiten von Grammatikregeln und somit als Elemente in den Sätzen der Sprache der Grammatik vorkommen. Die Menge der Terminalsymbole bildet das (Terminalsymbol-)Alphabet der Sprache
- Aus dem Satzsymbol S kann man durch **Ableitungen** (fortgesetzte Anwendung der Ersetzungsregeln) Symbolketten, so genannte Satzformen bilden. Eine Satzform, die nur mehr Terminalsymbole enthält, ist ein Satz
- Die **Sprache** $L(G)$ einer Grammatik $G(S)$ repräsentiert die Menge aller Sätze, die sich mit Hilfe ihrer Ersetzungsregeln aus dem Satzsymbol S erzeugen oder ableiten lassen

Beispiel Ableitung

Aus dem Satzsymbol S kann man durch Ableitungen Symbolketten, so genannte Satzformen bilden

Grammatik G(S):

$$\begin{aligned}S &= X Y Z. \\X &= ":" \mid ";" \\Y &= "-". \\Z &= ")" \mid "(".\end{aligned}$$

Ableitung

$$\begin{array}{lll}X = ":" & Y = "-" & Z = ")" \\ \downarrow & \downarrow & \downarrow \\ S \Rightarrow X Y Z \Rightarrow : Y Z \Rightarrow : - Z \Rightarrow & S \Rightarrow X Y Z \Rightarrow : Y Z \Rightarrow : - Z \Rightarrow & S \Rightarrow X Y Z \Rightarrow ; Y Z \Rightarrow ; - Z \Rightarrow \\ S \Rightarrow X Y Z \Rightarrow : Y Z \Rightarrow : - Z \Rightarrow & S \Rightarrow X Y Z \Rightarrow ; Y Z \Rightarrow ; - Z \Rightarrow & S \Rightarrow X Y Z \Rightarrow ; Y Z \Rightarrow ; - Z \Rightarrow\end{array}$$

: -)
: - (
; -)
; - (

Sätze

Die Sprache L(G) ist Menge aller Sätze

$$L(G) = \{":-)", ":-(", ";-)", ";-("\}$$

Beispiel Grammatik

Grammatik G(S)

$S = A \cdot ;$.
 $A = "a" \ B \mid B \ B \ "b".$
 $B = "b" \mid "a" \ "b".$

$V_T = \{"a", "b", ";"\}$
 $V_{NT} = \{S, A, B\}$

Ableitungen

$S \Rightarrow A \cdot ; \Rightarrow a \ B \cdot ; \Rightarrow a \ b \cdot ;$

$S \Rightarrow A \cdot ; \Rightarrow a \ B \cdot ; \Rightarrow a \ a \ b \cdot ;$

$S \Rightarrow A \cdot ; \Rightarrow B \ B \ b \cdot ; \Rightarrow b \ B \ b \cdot ; \Rightarrow b \ b \ b \cdot ;$

$S \Rightarrow A \cdot ; \Rightarrow B \ B \ b \cdot ; \Rightarrow b \ B \ b \cdot ; \Rightarrow b \ a \ b \ b \cdot ;$

$S \Rightarrow A \cdot ; \Rightarrow B \ B \ b \cdot ; \Rightarrow a \ b \ B \ b \cdot ; \Rightarrow a \ b \ b \ b \cdot ;$

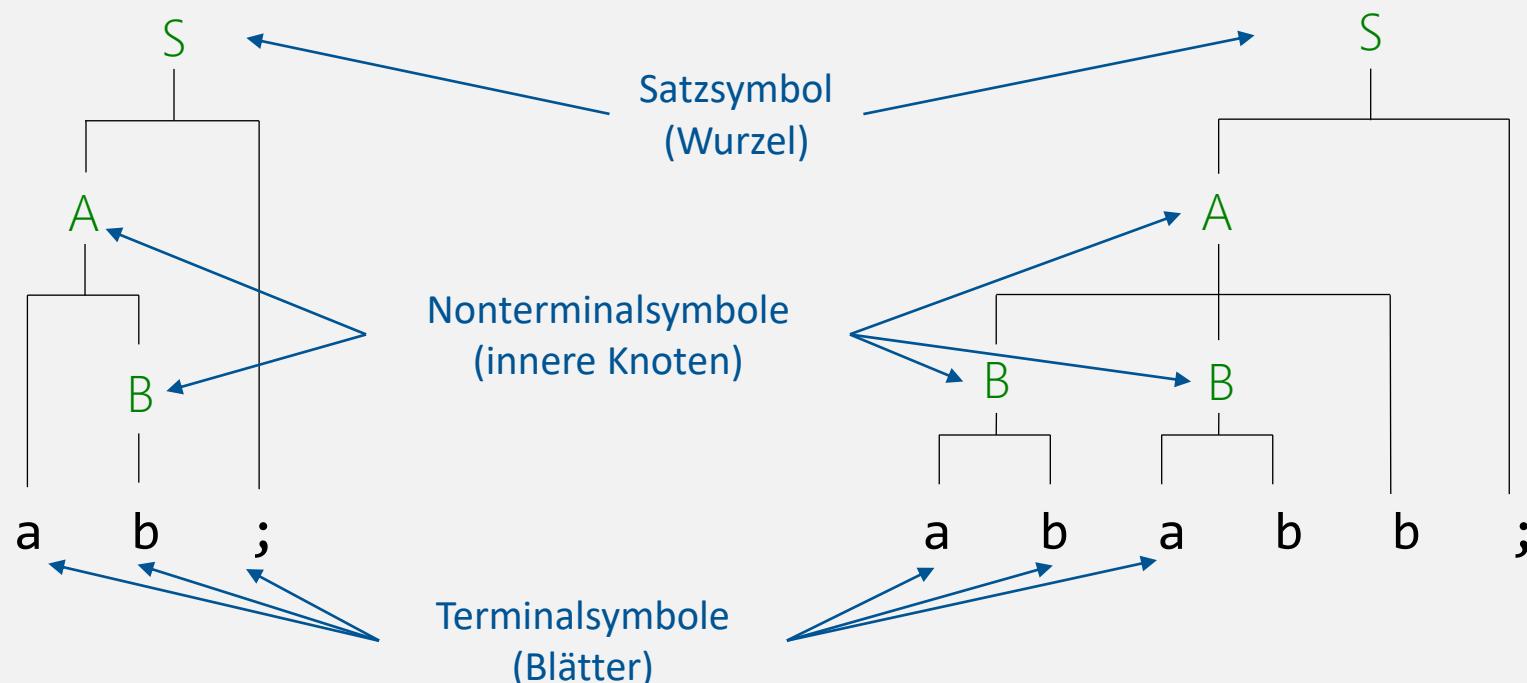
$S \Rightarrow A \cdot ; \Rightarrow B \ B \ b \cdot ; \Rightarrow a \ b \ B \ b \cdot ; \Rightarrow a \ b \ a \ b \ b \cdot ;$

Syntaxbaum (Parse-Baum)

Zeigt die Ableitungsstruktur für einen Satz einer Grammatik

Beispiele für Grammatik G(S)

```
S = A ";"  
A = "a" B | B B "b".  
B = "b" | "a" "b".
```



Rekursive Grammatiken

Grammatiken können auch rekursive Ersetzungsregeln haben

Man unterscheidet (mit $\alpha, \beta, \gamma \in V$)

- linksrekursive Regeln $A = \alpha \mid A\beta .$
- rechtsrekursive Regeln $A = \alpha \mid \beta A .$
- zentralrekursive Regeln $A = \alpha \mid \beta A \gamma .$

Beispiel: Grammatik $G(S)$ für Binärzahlen und leere Kette

$$\begin{aligned} S &= B. \\ B &= "0" \mid "1" \mid \varepsilon . \end{aligned}$$

$$\begin{aligned} V_T &= \{"0", "1"\} \\ V_{NT} &= \{S, B\} \end{aligned}$$

Ableitung (Beispiele)

$$S \Rightarrow 0 \ B \Rightarrow 0 \ \varepsilon = 0$$

$$S \Rightarrow 1 \ B \Rightarrow 1 \ \varepsilon = 1$$

$$S \Rightarrow 1 \ B \Rightarrow 1 \ 0 \ B \Rightarrow 1 \ 0 \ \varepsilon = 1 \ 0$$

$$S \Rightarrow 1 \ B \Rightarrow 1 \ 0 \ B \Rightarrow 1 \ 0 \ 1 \ B \Rightarrow 1 \ 0 \ 1 \ \varepsilon = 1 \ 0 \ 1$$

...

Beispiel: Grammatik einer einfachen Sprache



Menge aller Sätze, die mit "a" beginnen, mit "c" oder "d" enden und bei denen zwischen dem ersten und letzten Zeichen beliebig viele "b" vorkommen dürfen.

Grammatik:

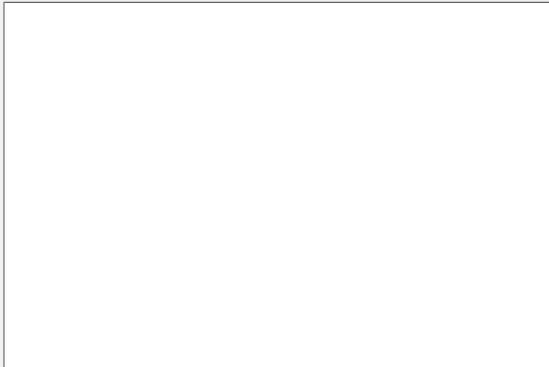
Beispiele:

Beispiel: Grammatik für Palindrom

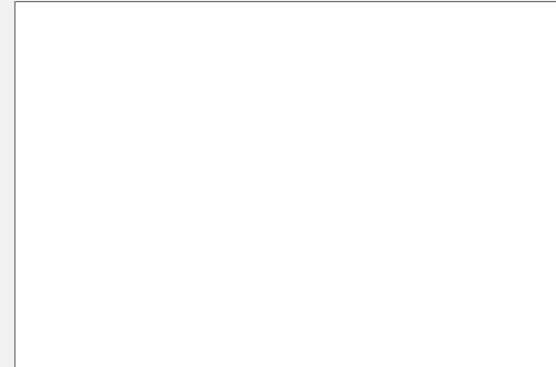
Ein Palindrom ist eine Zeichenkette, die vorwärts und rückwärts gelesen identisch ist, z.B. otto oder madamimadam

Der Einfachheit halber betrachten wir Palindrome, die sich mit dem Alphabet $V = \{"0", "1"\}$ beschreiben lassen, z.B. 0, 0110, 11011

Grammatik:



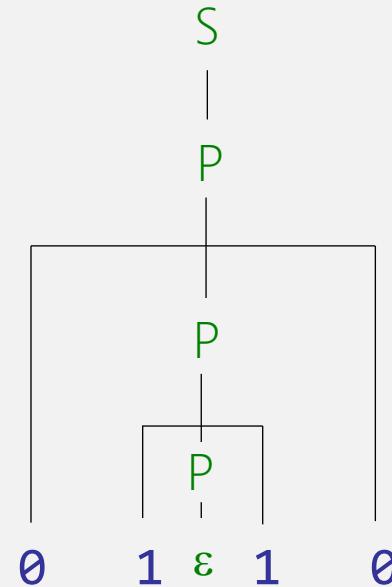
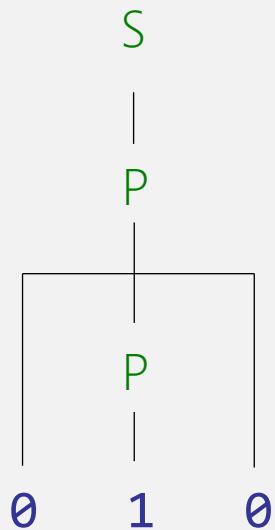
Beispiele:



Beispiel: Grammatik für Palindrom

Syntaxbäume

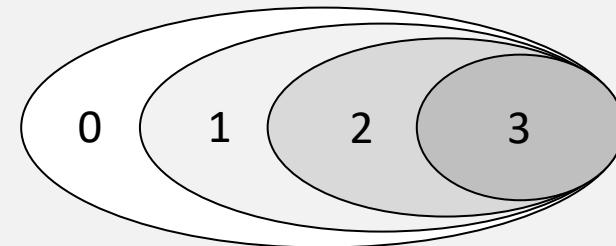
```
S = P.  
P = ε | "0" | "1" | "0" P "0" | "1" P "1".
```



Chomsky-Hierarchie

Der Linguistiker **Noam Chomsky** hat eine Sprachtypologie, die nach ihm benannte Chomsky-Hierarchie der Sprachen, definiert und die Grammatiken nach der Form ihrer Ableitungsregeln in vier **Typen** eingeteilt, die er von 0 bis 3 durchnummeriert hat:

- die unbeschränkten (Typ 0)
- die kontextsensitiven (Typ 1)
- die **kontextfreien** (Typ 2)
- die **regulären** (Typ 3)



Nur die beiden einfachsten Sprach- bzw. Grammatikklassen – jene vom Typ 3, die **regulären**, und jene vom Typ 2, die **kontextfreien** – sind für die datenorientierte Programmierung relevant.

16.2.1 Reguläre Grammatiken

Eine Grammatik heißt regulär (Typ 3), wenn sie sich durch Ersetzungsregeln der folgenden Art ausdrücken lässt:

$$A = b B.$$

$$A = a.$$

$$A = \epsilon.$$

$$V_T = \{a, b\}$$

$$V_{NT} = \{A, B\}$$

Beispiel: Grammatik der Bezeichner

- Ein Bezeichner besteht aus Buchstaben **letter** und Ziffern **digit**, von denen das erste Zeichen ein Buchstabe sein muss.
- Reguläre Grammatik $G(S)$

$$S = \text{letter } R.$$

$$R = \text{letter } R \mid \text{digit } R \mid \epsilon.$$

$$V_T = \{\text{letter}, \text{digit}\}$$

$$V_{NT} = \{S, R\}$$

- Reguläre Grammatiken können als **regulärer Ausdruck** definiert werden
 $\text{letter} (\text{ letter} \mid \text{digit})^*$

Reguläre Ausdrücke

- Äquivalent zu regulären Grammatiken
- Beschreibung einer Menge von Zeichenketten (formalen Sprache)

Definition

- Die leere Kette (ϵ) ist ein regulärer Ausdruck
- Ein einzelnes Zeichen oder *wildcard* . ist ein regulärer Ausdruck
- Wenn α und β reguläre Ausdrücke sind, dann sind auch folgende Ausdrücke regulär:

$\alpha \beta$	die Konkatenation von α und β
$\alpha \beta$	eine der beiden Alternativen von α und β (auch $\alpha + \beta$)
α^*	die null- oder mehrmalige Wiederholung von α (<i>closure</i>)
(α)	Zusammenfassung von Teilausdrücken (z.B. $(\alpha \beta)^*$)

Beispiel: $a b^* (c | d)$

- beschreibt die Menge {ac, ad, abc, abd, abbc, abbd, ...}

Reguläre Ausdrücke

Beispiele

Regulärer Ausdruck	Sätze der Sprache	Keine Sätze
aa baab	aa baab	jede andere Kette
.u.u.u.	cumulus	succubus
.*0....	1000234 98701234	111111111 403982772
ab*(c d)	ad abc abbbbbbc	abcd
a(a b)aab	aaaab abaab	jede andere Kette
(ab)*a	a abababa	aa abbba

Erweiterte reguläre Ausdrücke

Zusätzliche Operationen ermöglichen kürzere Ausdrücke

Operation	Beispiel	Sätze der Sprache	Keine Sätze
Zeichenmengen	[abc][xyz]	az bx	xa
	[A-Za-z][a-z]*	Hagenberg schnell	maxN
Negation	[^aeiou]*	Typ	Type
Meta-Zeichen	\[.*\]	[i]	[j [
Einmal/mehrmal	a(bc)+	abc abcbc	a
Null-/einmal	a(bc)?	a abc	abcbc
Genau n -mal	[0-9]{5}	31415	1234 123456

Reguläre Grammatiken

Anwendungen regulärer Grammatiken / regulärer Ausdrücke

- Pattern Matching
- Betriebssysteme (Unix, Windows) und Bibliotheken
- Lexikalische Analyse

Anwendung 1: Pattern Matching



Beispiel: E-Mail-Adresse

Eine E-Mail-Adresse ist folgendermaßen aufgebaut

1. Sequenz von Buchstaben
2. Zeichen "@"
3. Sequenz von Buchstaben, gefolgt von Zeichen "." (beliebige Anz. von Wdh.)
4. "at" oder "com" (vereinfacht)

Welche der folgenden Zeichenketten sind E-Mail-Adressen?

alice@yahoo.com	✓
hagenberg	✗
mail@chuck.norris.com	✓
bob@whitehouse	✗
john.doe@fhooe.at	✗

Regulärer Ausdruck (Alternativen werden oft mit | statt mit + geschrieben)

Anwendung 1: Pattern Matching



Beispiel: Datum in Form YYYY-MM-DD (z.B. 2020-04-01)

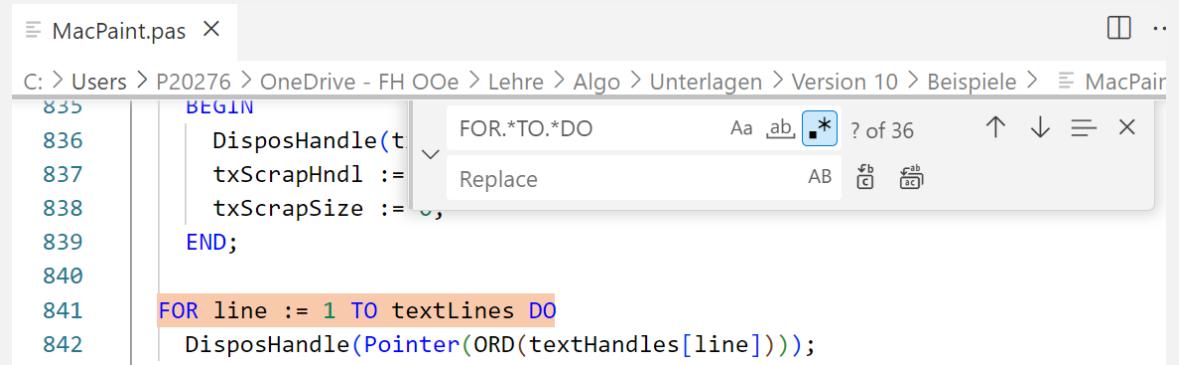
Beispiel: Telefonnummern (z.B. (800) 555-1212)

Anwendung 2: Betriebssysteme und Bibliotheken

- Standardlösungen für die Mustersuche in Zeichenketten (z.B. Dateien)
- Unix: grep (Generalized Regular Expression Pattern matcher)
 - entwickelt von Ken Thompson (Entwickler von Unix, Turing Award)
- Windows: findstr

```
Regular expression quick reference:  
  .          Wildcard: any character  
  *          Repeat: zero or more occurrences of previous character or class  
  ^          Line position: beginning of line  
  $          Line position: end of line  
  [class]    Character class: any one character in set  
  [^class]   Inverse class: any one character not in set  
  [x-y]     Range: any characters within the specified range  
  \x        Escape: literal use of metacharacter x  
  \<xyz>    Word position: beginning of word  
  xyz\>    Word position: end of word
```

- Suche in VSCode



Anwendung 2: Betriebssysteme und Bibliotheken

Beispiele zur Verwendung von findstr (Windows 10)

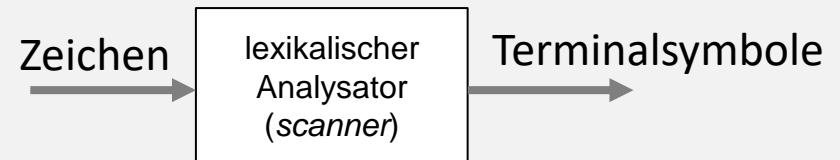
Pascal-Programm MacPaint.pas

Kommando	Treffer
findstr ".*" MacPaint.pas	IF result = 1 { save } THEN END; { ProcessEvent }
findstr "[0-9]" MacPaint.pas	brush[7] := \$0180;
findstr "[0-9]*" MacPaint.pas	TYPE Str63 = String[63];
findstr ".*" MacPaint.pas	THEN newFace := [] TextSize(fontSizes[theFontSize]);
findstr ".*+.*1;" MacPaint.pas	brush := brush + 1; vert := vert + 1;
findstr "FOR.*TO.*DO" MacPaint.pas	FOR i := 0 TO 9 DO

Anwendung 3: Lexikalische Analyse

Aufgaben der lexikalischen Analyse

- Liefert Terminalsymbole
- Überliest bedeutungslose Zeichen



Beispiel: Pascal-Programm

```
IF x <= min THEN BEGIN
  x := 42
END; (* IF *)
```

Terminalsymbole haben eine Struktur

```
then = "t" "h" "e" "n".
number = digit rest.
rest = ε | digit rest.
```

Erkennung ist nicht Teil der Syntaxanalyse

- einfache Syntaxanalyse ... "i" ("f" Expr ... | letter ...)
- Überlesen von Leerzeichen ... "if" { Blank } Expr { Blank } ...
- für Terminalsymbole reichen reguläre Grammatiken

Symbolfolge

Symbol	Wert
if	
ident	"x"
lessEq	
ident	"min"
then	
begin	
ident	"x"
assign	
number	42
end	
semicolon	

Anwendung 3: Lexikalische Analyse

Beispiel: Terminalklassen in Pascal

Terminalklasse	Reguläre Grammatik	Regulärer Ausdruck
Zeichen	letter = "a" "b" ... "z"	[a-zA-Z]
Ziffern	digit = "0" "1" ... "Z"	[0-9]
Ziffern (hex)	hex = "0" "1" ... "a" ... "F"	[0-9a-fA-F]
Zahlen (int)	uint = digit { digit } "\$" hex { hex }	[0-9]+ \\$\{[0-9a-fA-F]+\}
Zahlen (real)	ureal = digit {digit} ." digit {digit} ["E"] "e" ["+" "-"] digit {digit}]	[0-9]+.\{[0-9]+\{[eE]\[-+\]?\{[0-9]+\})?
Namen	ident = letter { letter digit }	[a-zA-Z][a-zA-Z0-9]*
Zeichenketten	string = """ regStr """	'[^']*'

16.2.2 Kontextfreie Grammatiken

2

3

Die Bezeichnung kontextfrei für eine Grammatik drückt aus, dass in allen Symbolketten, die aus dem Satzsymbol durch Anwendung der Ersetzungsregeln abgeleitet werden können, jedes darin vorkommende Nonterminalsymbol unabhängig von den Symbolen links oder rechts davon, also unabhängig von seiner Umgebung, dem Kontext, durch eine seiner Alternativen ersetzt werden kann.

Alle Ersetzungsregeln haben die folgende Form

$A = \dots$

$V_{NT} = \{A\}$

Beispiel

$S = A ";"$.
 $A = "a" B \mid B B "b"$.
 $B = "b" \mid "a" "b"$.

NT B kann unabhängig vom Kontext durch "b" oder "a" "b" ersetzt

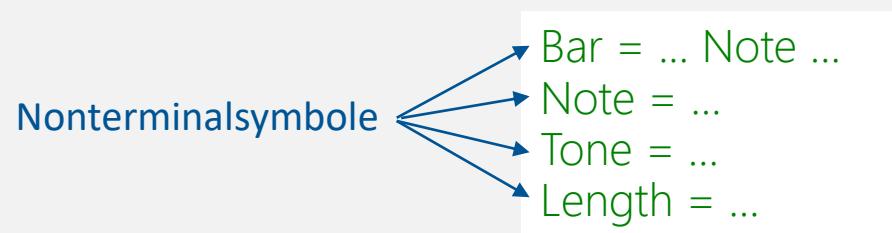
Kontextfreie Grammatiken

1. Terminalsymbole

- elementare Bestandteile der Sätze einer Sprache (lassen sich nicht weiter zerlegen)
- einfache Terminalsymbole (z.B. ";", ".", ":=", "<=", "BEGIN")
- Terminalklassen (z.B. ident, number)

2. Nonterminalsymbole

- sind Bezeichner von Abstraktionen und werden zur Gliederung von (Satz-)Strukturen verwendet



Kontextfreie Grammatiken

3. Ableitungsregeln

- zu jedem Nonterminalsymbol gibt es eine Ableitungsregel, die beschreibt, in welche Teile die durch das Nonterminalsymbol bezeichnete Abstraktion zerlegt werden kann.
- Beispiel:

Bar = Note "," Note "," Note.

Note = Tone Length.

Tone = "c" | "d" | "e" | "f" | "g" | "a" | "h" .

Length = "1" | "2" | "4" | "8".

4. Start- (Satz-)Symbol

- Linke Seite der "ersten" Ableitungsregel ("oberstes Nonterminalsymbol", "höchste Abstraktion")
- Beispiel: Bar

EBNF-Darstellung von Grammatiken

EBNF = Erweiterte Backus-Naur-Form (Wirthsche EBNF)

Symbol	Bedeutung
=	trennt linke und rechte Seite einer Ableitungsregel
	trennt Alternativen $a b \dots a$ oder b
.	schließt eine Ableitungsregel ab
(...)	klammert zusammengehörige Teile
[...]	Optionsklammern $[a] \dots a \varepsilon$
{ ... }	Wiederholungsklammern $\{a\} \dots a aa aaa \dots$

Beispiel 1: Grammatik der Binärzahlen und leere Kette (in EBNF)

$$\begin{aligned} S &= \{ B \}. \\ B &= "0" | "1". \end{aligned}$$

$$S = \{ "0" | "1" \}.$$

$$\begin{aligned} S &= B. \\ B &= "0" B | "1" B | \varepsilon. \end{aligned}$$

Beispiel 2: Grammatik der Bezeichner

$$S = \text{letter} \{ \text{letter} | \text{digit} \}.$$

$$\begin{aligned} S &= \text{letter} R. \\ R &= \text{letter} R | \text{digit} R | \varepsilon. \end{aligned}$$

Beispiel: Grammatik einer einfachen Sprache (in EBNF)



Menge aller Sätze, die mit "a" beginnen, mit "c" oder "d" enden und bei denen zwischen dem ersten und letzten Zeichen beliebig viele "b" vorkommen dürfen.

Grammatik:

Beispiele:

Rekursive Grammatik:

$$\begin{aligned}S &= "a" \ B \ E. \\B &= "b" \ B \mid \epsilon. \\E &= "c" \mid "d".\end{aligned}$$

Beispiel: Grammatik für eine Folge von Zeiteinheiten



Menge aller Sätze, die eine Folge von Zeiteinheiten enthalten. Eine Zeiteinheit hat die Form H:M oder M, ein Komma trennt Zeiteinheiten. H und M sind ein- oder zweistellige ganze Zahlen.

Grammatik:

Beispiele:

Beispiel: Grammatik für Musiknoten (Version 2)



Beispiel: Musiknoten

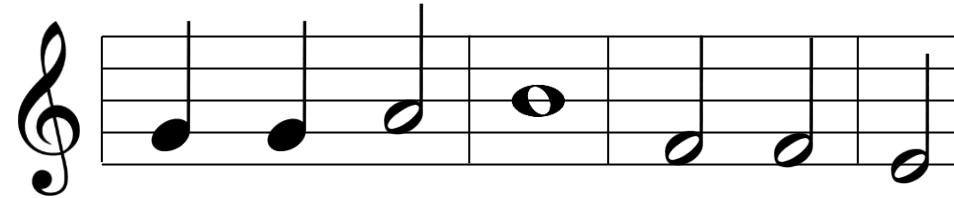
- Beliebige Anzahl von Takten
- Beliebige Anzahl von Noten pro Takt
- Notennamen: c, d, e, f, g, a, h
- Notenwerte: 1, 2, 4, 8

```
|g4,g4,a2|h1|f2,f2|e2,f2|
```

```
|g4,g4,g8,g8,e8,e8|
```

```
|a2|
```

- Grammatik G(Music)



Besondere Grammatiken: LL(1)-Grammatiken

Definition: Wenn bei der Analyse in jeder Situation das aktuelle Terminalsymbol (genannt Vorgriffssymbol) ausreicht, um in der aktuellen Ersetzungsregel die richtige Alternative auszuwählen, erfüllt die Grammatik die so genannte **LL(1)-Bedingung**.

Beispiele

Eingabestrom

1 0 1

▲
aktueller
Terminalsymbol

3:15

▲
aktueller
Terminalsymbol

Ersetzungsregel

$$P = \epsilon \mid "0" \mid "1" \mid "0" P "0" \mid "1" P "1".$$

▲
zwei Alternativen starten mit dem TS "1"
LL(1)-Bedingung ist nicht erfüllt

$$\text{Period} = \text{Number} \mid \text{Number} ":" \text{Number}.$$

▲
zwei Alternativen starten mit dem TS "digit"
LL(1)-Bedingung ist nicht erfüllt

Besondere Grammatiken: LL(1)-Grammatiken

LL(1)-Bedingung

- Das bedeutet, dass in jeder Ersetzungsregel der Grammatik alle Alternativen mit unterschiedlichen Terminalsymbolen beginnen.
- Auch bei Optionen und Wiederholungen muss mit einem Vorgriffssymbol die richtige Entscheidung getroffen werden können.

Beispiel: Beseitigung von LL(1)-Konflikten durch "Zusammenfassen" von Alternativen

Period = Number | Number ":" Number.

LL(1)-Bedingung ist nicht erfüllt

Period = [Number ":"] Number.

LL(1)-Bedingung ist nicht erfüllt

Period = Number [":" Number].

LL(1)-Bedingung ist erfüllt

16.2.3 Attributierte Grammatiken – Definitionen

- Eine kontextfreie Grammatik beschreibt lediglich die Syntax einer Sprache
- Ein Syntaxanalysator kann daher nur prüfen, ob ein Eingabetext korrekt ist
- Wenn man einen Eingabedatenstrom verarbeiten (das heißt, in eine andere Form transformieren und gegebenenfalls Ergebnisse aus ihm ermitteln) will, sind zusätzliche Aktionen erforderlich
- Neben den Erkennungsprozessen (zur Analyse der syntaktischen Korrektheit des Eingabedatenstroms) benötigen wir Verarbeitungs-/Transformationsprozesse
- Dazu erweitern wir kontextfreie Grammatiken zu **attribuierten Grammatiken**
- Eine attributierte Grammatik ist eine (kontextfreie) Grammatik, die mit **semantischen Aktionen** und **Attributen** so angereichert ist, dass damit der Verarbeitungs-/Transformationsprozess vollständig beschrieben ist

Semantische Aktionen

Semantische Aktionen beschreiben Operationen, die während der Erkennung des Eingabedatenstroms ausgeführt werden sollen

- Semantische Aktionen werden direkt an den **passenden Stellen** in die kontextfreie Grammatik eingefügt
- Eine semantische Aktion unmittelbar nach einem Grammatiksymbol bedeutet, dass sie **nach der Erkennung dieses Symbols** (im Zuge der Syntaxanalyse) **ausgeführt** werden soll
- Schreibweise: Pseudocode zwischen `sem` und `endsem`

Beispiel: Differenz zwischen der Anzahl von 0en und 1en einer Binärzahl

```
S = { "0" | "1" }.
```

```
S =      sem d := 0 endsem
        { "0"  sem d := d - 1 endsem
        | "1"  sem d := d + 1 endsem
        }      sem Write(↓d)  endsem
        .
```

Semantische Aktionen

Beispiel: Differenz zwischen der Anzahl von 0en und 1en einer Binärzahl

```
S =      sem d := 0 endsem
        { "0"  sem d := d - 1 endsem
        | "1"  sem d := d + 1 endsem
        sem Write(↓d) endsem
    }.
```

Eingabestrom	d	Ausgabe
11100	0	
11100	1	
11100	2	
11100	3	
11100	2	
11100	1	1

Mit semantischen Aktionen können also Ausführungen beliebiger Aktionen von der Struktur des Eingabedatenstroms abhängig gemacht werden

Attribute

Attribute beschreiben Eigenschaften von Grammatiksymbolen in Form von **Datenobjekten**, die den **Grammatiksymbolen** zugeordnet werden

Bei den Attributen unterscheiden wir nach ihrer Art zwischen

- **semantischen Attributen**, die mit Nonterminalsymbolen der Grammatik verknüpft sind und deren Werte mittels semantischer Aktionen im Zuge des Erkennungsprozesses ermittelt werden

Beispiele: Mail_{↑nrWords} Period_{↑value}

- **lexikalischen Attributen**, die mit Terminalklassen verknüpft sind und deren Werte vom lexikalischen Analysator ermittelt und zur Verfügung gestellt werden
- Beispiele: ident_{↑name} number_{↑value} digit_{↑value}

Attribute

Nonterminalsymbole können auch Eingangsattribute haben

- Ausgangsattribute ($S_{\uparrow x}$) ... entstehen bei der Erkennung eines Terminal- oder Nonterminalsymbols

$N_{\uparrow n} = \dots B_{\uparrow c} B_{\uparrow d}$ sem $n := 2*c + d$ endsem.

$B_{\uparrow v} = \dots | "1"$ sem $v := 1$ endsem.

$\dots = \text{digit}_{\uparrow \text{value}}$ sem Write($\downarrow \text{value}$) endsem.

... Terminalklasse $\text{digit}_{\uparrow \text{value}}$

- Eingangsattribute ($S_{\downarrow y}$) ... werden einem Nonterminalsymbol zu seiner Erkennung mitgegeben

$\dots = \dots$ sem $n := 42$ endsem $F_{\downarrow n} \dots$

$F_{\downarrow n} = \dots$ sem $\dots := n$ endsem.

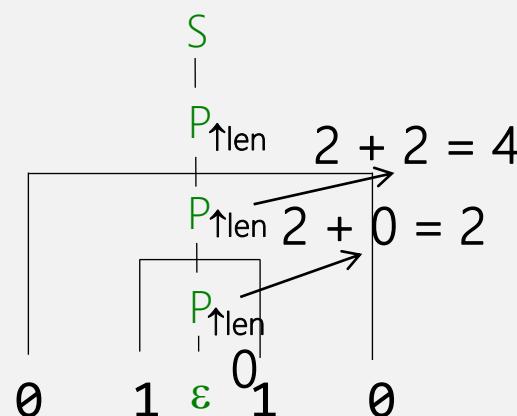
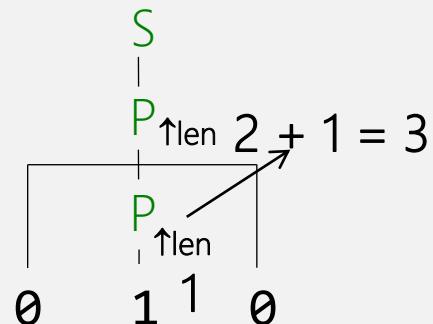
Beispiel: Länge eines Palindroms



Grammatik $G(S)$ $S = P. \quad P = \epsilon \mid "0" \mid "1" \mid "0" \text{ P } "0" \mid "1" \text{ P } "1".$

Attributierte Grammatik

"Dekorierter" Syntaxbaum



Beispiel: Berechnung des Werts einer Hex-Zahl



Grammatik (z.B. 1Bh)

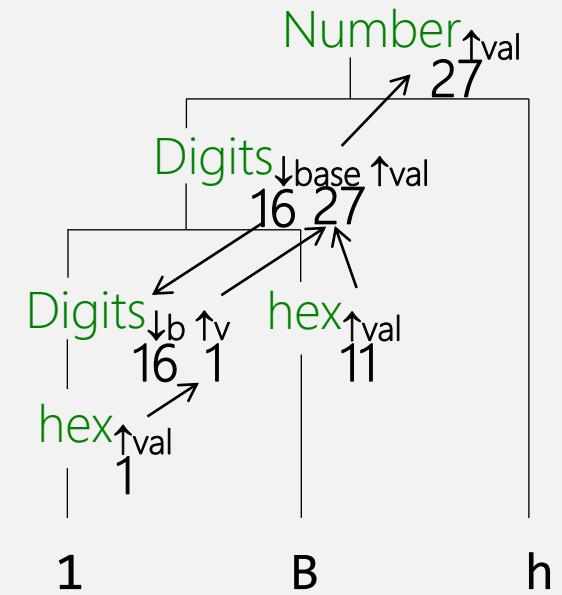
Number = Digits | Digits "h".
Digits = hex | Digits hex.

Terminalklasse hex ("0" | "1" | ... | "F")

Attribute

Number_{↑val} Digits_{↓base ↑val} hex_{↑val}

Attributierte Grammatik



Beispiel

Beispiel: Grammatik für Binärzahlen und leere Kette

$S = \{ B \}.$
 $B = "0" \mid "1" .$

Attributierte Grammatik

$S_{\uparrow \text{value}}$	=	sem value := 0 endsem
	{ $B_{\uparrow b}$	sem value := $2^* \text{value} + b$ endsem
	.	
$B_{\uparrow b}$	= "0" "1"	sem b := 0 endsem sem b := 1 endsem
	.	

Beispiel-Berechnung

Eingabestrom	b	value
1010	?	0
1010	1	1
1010	0	2
1010	1	5
1010	0	10

Beispiel: Summe einer Folge von Zeiteinheiten

Grammatik G(S):

S = Period {," Period }.

Period = Number [":" Number].

Number = digit [digit].

Terminalsymbole:

"," , ":" , digit ("0" | "1" | "2" | ... | "9")

Attribute: Period_{↑p} Number_{↑n} digit_{↑d}

Attributierte Grammatik:

S = Period_{↑p1}
 { "," Period_{↑p2}
 }

.

Period_{↑p} = Number_{↑n}
 [":" Number_{↑m}
].

Number_{↑n} = digit_{↑d}
 [digit_{↑d}
].

sem sum := p1 endsem
sem sum := sum + p2 endsem
sem WriteTime(↓sum) endsem

sem p := n endsem
sem p := 60 * n + m endsem

sem n := d endsem
sem n := 10 * n + d endsem

56
2:15
↑n / ↑m

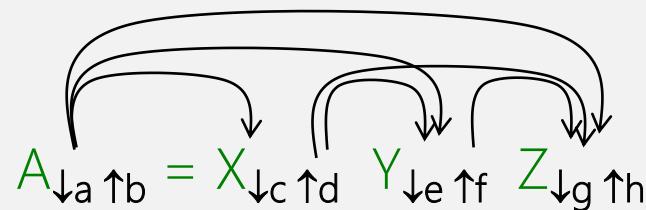
Attributierte Grammatiken – Definitionen

Eine attributierte Grammatik ist **L-attribuiert**, wenn für jede ihrer Regeln

$$A = X_1 \dots X_n$$

gilt: die Eingangsattribute von X_i , für alle $1 \leq i \leq n$ hängen nur von den Eingangsattributen von A und den Ausgangsattributen von $X_1 \dots X_{i-1}$ ab

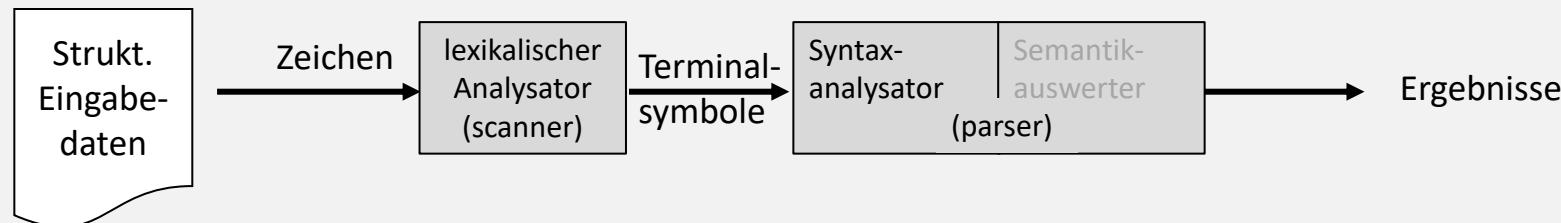
Beispiel:



L-attribuierte Grammatiken ermöglichen eine mit der Syntaxanalyse von links nach rechts schritthalrende Attributauswertung ohne expliziten, dekorierten Syntaxbaum

Algorithmische Interpretation von Grammatiken

- Ableitungsregeln werden als Erkennungsprozesse aufgefasst
- Durch die Grammatik sind alle (im Zuge der Verarbeitung eines Eingabedatenstromes) notwendigen Erkennungsprozesse festgelegt
- Die implementierten Erkennungsprozesse zusammengenommen nennt man Syntaxanalysator
- Der Syntaxanalysator setzt ein Programm (genannt lexikalischer Analysator) voraus, das Terminalsymbole erkennt und liefert



Algorithmische Interpretation von Grammatiken

Arbeitsweise

- jede Ableitungsregel wird von links nach rechts durchlaufen
- zur Alternativenauswahl wird das aktuelle Eingabesymbol verwendet
- nach Erkennung eines Terminalsymbols fordert der Syntaxanalysator vom lexikalischen Analysator das nächste Element (Terminalsymbol) des Eingabedatenstromes an
- bei jedem Nonterminalsymbol
 - unterbricht der Syntaxanalysator die Abarbeitung der laufenden Regel
 - arbeitet das angetroffene Nonterminalsymbol ab, d.h. führt den entsprechenden Erkennungsprozess aus
 - setzt dann an der Unterbrechungsstelle fort

16.3 Datenorientierter Systementwurf mit ATG

Vorgehensweise

Schritt 1: Syntaktische Struktur der Eingabe als kontextfreie Grammatik darstellen

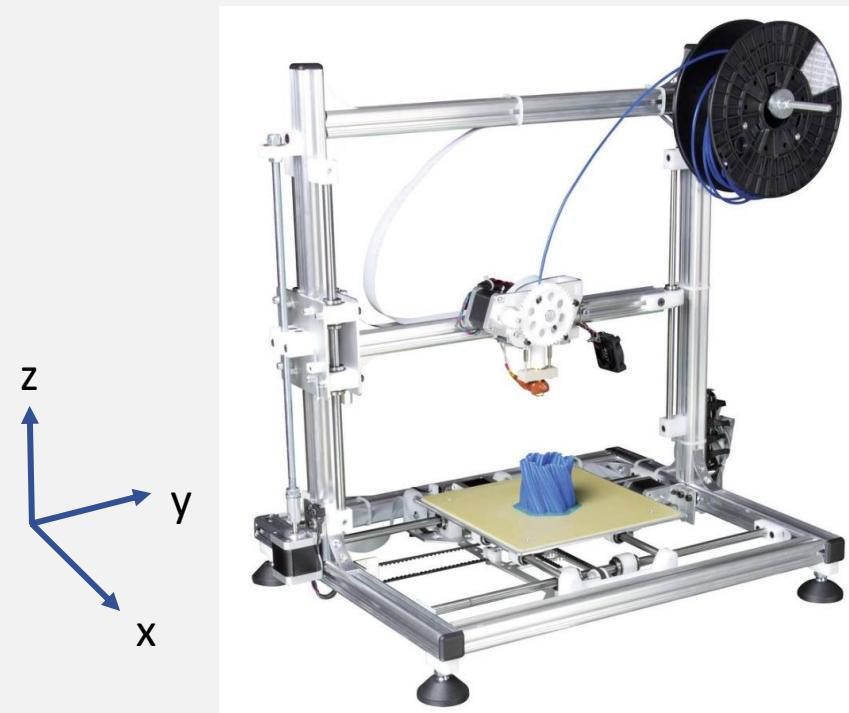
Schritt 2: Attribute für den Verarbeitungsprozess finden

Schritt 3: Semantische Aktionen entwerfen und in die Grammatik integrieren

Schritt 4: Code für Systemimplementierung ermitteln/generieren

Aufgabenstellung für Entwurfsbeispiel

Zu Simulations- und Bedienungszwecken soll eine Applikation für einen 3D-Drucker entwickelt werden.



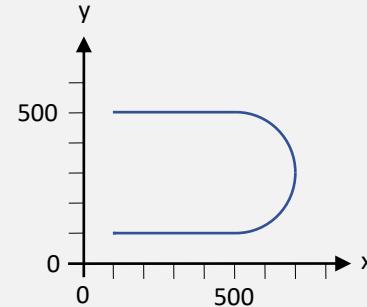
Aufgabenstellung für Entwurfsbeispiel

- Der Bewegungsablauf des 3D-Druckers (mit 3 Achsen) wird durch eine spezielle (vorgegebene) Sprache beschrieben
- Beschreibung der Bewegungsbahn: Die Bewegungsbahn ist eine Folge von linearen Teilschritten. Zwei Bahnformen können beschrieben werden:
 - Lineare Bewegung zum Punkt (x, y, z) in n Schritten: $n \ L \ (x, y, z)$
Beispiel: $5 \ L \ (100, 200, 300)$
 - Kreisbewegung zum Punkt (x, y, z) über einen Zwischenpunkt (x_1, y_1, z_1) in n Schritten: $n \ C \ (x_1, y_1, z_1) \ (x, y, z)$
Beispiel: $20 \ C \ (50, 50, 50) \ (100, 100, 100)$
- Koordinaten werden in $1/10$ mm angegeben (z.B. $123 = 12.3$ mm)
- Der Endpunkt einer Bahn ist gleichzeitig der Anfangspunkt der nächsten Bahn

Beispiele für Bewegungsbeschreibung

Beispiel 1

```
(100,100,100)  
10 L (500,100,100)  
20 C (700,300,100)(500,500,100)  
10 L (100,500,100)  
E
```



Beispiel 2

```
(200,100,100)  
5 L (-100,+200,-100)  
20 C (+700,+700,+1200) (+0,+1400,+0)  
E
```

der Punkt $x=800, y=1000, z=1200$
soll auf der Kreisbahn liegen

Start beim Punkt $x=200, y=100, z=100$

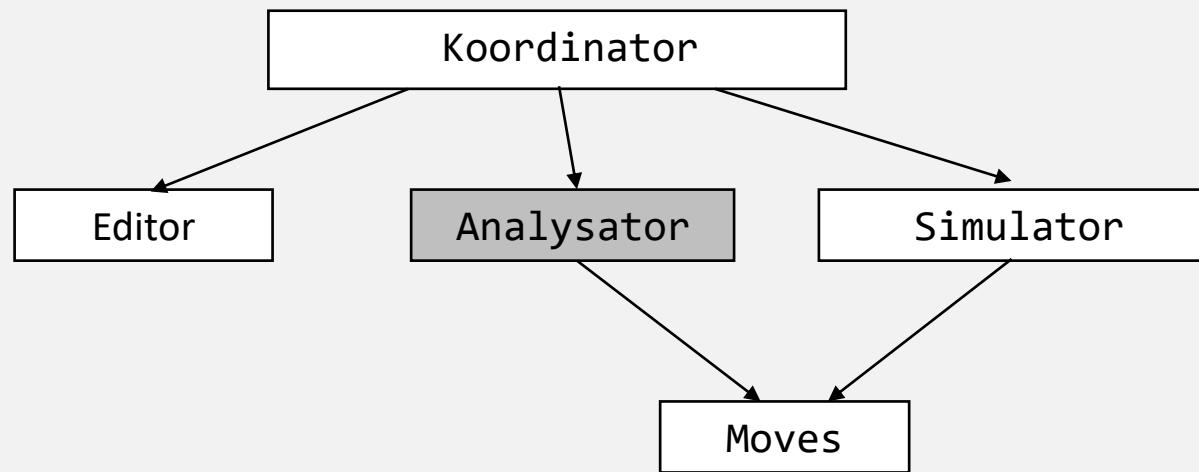
Lineare Bewegung in 5 Schritten zum
Punkt $x=100, y=300, z=0$

Kreisbewegung in 20 Schritten zum
Punkt $x=100, y=1700, z=0$

Anwendungsbeispiel für ATG

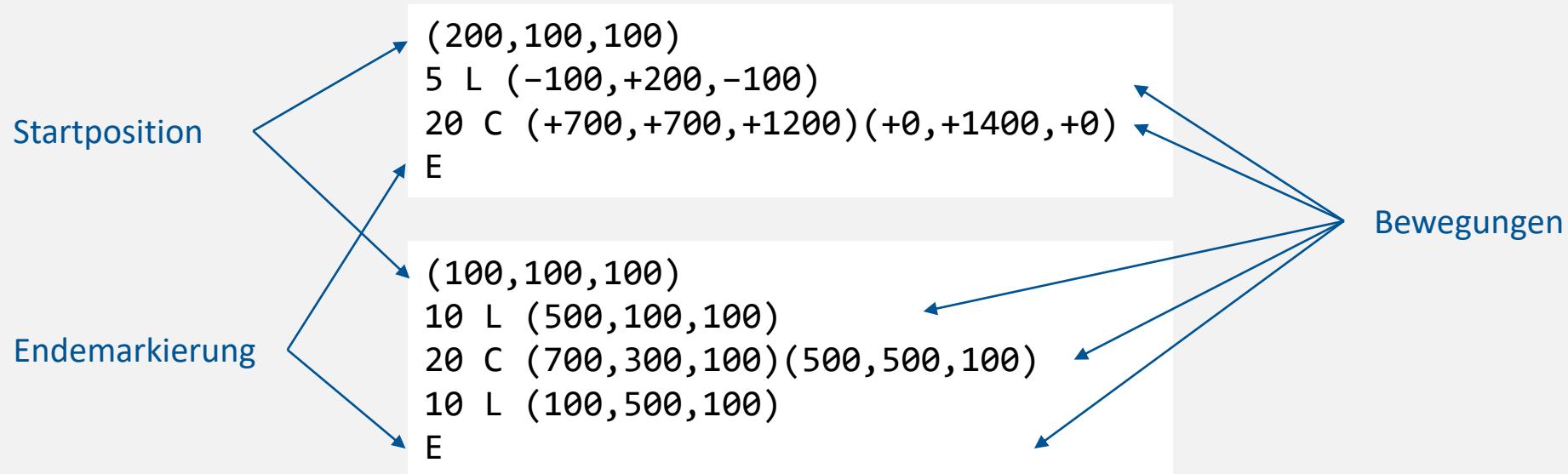
Entwurf der Analysator-Komponente der 3D-Drucker-Applikation

Die Analysator-Komponente dient dazu, eine gegebene Bewegungsbeschreibung auf syntaktische Korrektheit zu prüfen und die durch sie spezifizierten Bewegungs-Segmente in geeigneter Form in der Datenkapsel Moves abzuspeichern (damit die Simulator-Komponente auf eine syntaktisch korrekte Bewegungsbeschreibung zugreifen kann).



Entwurfs-Schritt 1: Kontextfreie Grammatik entwerfen

Gesamte Bewegungsbeschreibung (Startsymbol)



PrinterMoves = Position { Move } "E".

Entwurfs-Schritt 1: Kontextfreie Grammatik entwerfen

Positionsangabe

Beispiel: (200,100,100)

(200,100,100)

Position = "(" ... "," ... "," ... ")".

(200,100,100)

Coordinate

Position = "(" Coordinate "," Coordinate "," Coordinate ")".

Teilbewegung

10 L (-100,+200,-100)

Move = Number "L" Position.

20 C (700,300,100)(500,500,100)

Move = Number "C" Position Position.

Move = Number ("L" Position | "C" Position Position).

Koordinate 200 -100 +200

Coordinate = ["+" | "-"] Number.

Zahl 200 100

Number = digit { digit } .

Entwurfs-Schritt 1: Kontextfreie Grammatik entwerfen

Grammatik G(PrinterMoves):

PrinterMoves = Position { Move } "E".

Position = "(" Coordinate "," Coordinate "," Coordinate ")".

Move = Number ("L" Position | "C" Position Position).

Coordinate = ["+" | "-"] Number.

Number = digit { digit } .

Alphabet (Terminalsymbole):

- "L", "C", "E", "+", "-", "(", ")", ",", digit
- **Terminalklasse** digit = "0" | "1" | ... | "9"

Nonterminalsymbole (Startsymbol PrinterMoves):

- PrinterMoves, Position, Move, Coordinate, Number

Entwurfs-Schritt 2: Attribute festlegen

PrinterMoves

erkennt eine vollständige Bewegungsbeschreibung

Position $\downarrow x_0 \downarrow y_0 \downarrow z_0 \uparrow x \uparrow y \uparrow z$

erkennt eine Position mit den Koordinaten x, y und z. x_0, y_0 und z_0 bedeuten die Ursprungsposition, auf die sich relative Koordinatenangaben beziehen

Move $\downarrow x_0 \downarrow y_0 \downarrow z_0 \uparrow x \uparrow y \uparrow z$

erkennt eine Bewegung beginnend bei (x_0, y_0, z_0) und liefert den Endpunkt (x, y, z) der Bewegung

Coordinate $\downarrow v_0 \uparrow v$

erkennt eine Koordinate mit dem Wert v. v_0 ist ein Ursprungswert, auf den sich relative Angaben beziehen

Number $\uparrow v$

erkennt eine ganze Dezimalzahl mit dem Wert v

digit $\uparrow d$

erkennt eine Ziffer mit dem Wert d

Entwurfs-Schritt 3: Semantische Aktionen

Semantische Aktionen entwerfen und in Grammatik integrieren

```
PrinterMoves =  
    Position↓0 ↓0 ↓0 ↑x0 ↑y0 ↑z0 sem InitMoves(↓x0 ↓y0 ↓z0) endsem  
    { Move↓x0 ↓y0 ↓z0 ↑x ↑y ↑z sem x0 := x; y0 := y; z0 := z endsem  
    } "E".
```

```
interface of Moves  
    InitMoves(↓x0 ↓y0 ↓z0)  
end Moves
```

Entwurfs-Schritt 3: Semantische Aktionen

Semantische Aktionen entwerfen und in Grammatik integrieren

```
Position↓x0 ↓y0 ↓z0 ↑x ↑y ↑z =
  "(" sem v0 := x0 endsem Coordinate↓v0 ↑v sem x := v endsem
  "," sem v0 := y0 endsem Coordinate↓v0 ↑v sem y := v endsem
  ";" sem v0 := z0 endsem Coordinate↓v0 ↑v sem z := v endsem
  ")".
```

Einfacher

```
Position↓x0 ↓y0 ↓z0 ↑x ↑y ↑z =
  "(" Coordinate↓x0 ↑x
  "," Coordinate↓y0 ↑y
  ";" Coordinate↓z0 ↑z
  ")".
```

Entwurfs-Schritt 3: Semantische Aktionen

Semantische Aktionen entwerfen und in Grammatik integrieren

```
Move↓x0 ↓y0 ↓z0 ↑x ↑y ↑z =  
  Number↑steps  
  ("L" Position↓x0 ↓y0 ↓z0 ↑x ↑y ↑z sem AddLine(↓x ↓y ↓z ↓steps) endsem  
   | "C" Position↓x0 ↓y0 ↓z0 ↑x1 ↑y1 ↑z1  
     Position↓x0 ↓y0 ↓z0 ↑x ↑y ↑z sem AddArc(↓x ↓y ↓z ↓x1 ↓y1 ↓z1 ↓steps) endsem  
   ).
```

```
interface of Moves  
  InitMoves(↓x0 ↓y0 ↓z0)  
  AddLine(↓x ↓y ↓z ↓steps)  
  AddArc(↓x ↓y ↓z ↓x1 ↓y1 ↓z1 ↓steps)  
end Moves
```

Entwurfs-Schritt 3: Semantische Aktionen

Semantische Aktionen entwerfen und in Grammatik integrieren

```
Coordinate↓v0 ↑v =  
    sem sign := '' endsem  
    [ "+"      sem sign := '+' endsem  
    | "-"      sem sign := '-' endsem  
    ]  
Number↑v   sem if sign = '+' then  
            v := v0 + v  
        elseif sign = '-' then  
            v := v0 - v  
        end -- if  
    endsem  
.  
.
```

Entwurfs-Schritt 3: Semantische Aktionen

Semantische Aktionen entwerfen und in Grammatik integrieren

```
Numberv =  
    digitd    sem v := d endsem  
    { digitd    sem v := 10 * v + d endsem  
    }.
```

Entwurfs-Schritt 3: Semantische Aktionen entwerfen

Abstrakte semantische Aktionen spezifizieren

Datenkapsel Moves zur Verwaltung einer Liste von Teilbewegungen mit folgenden Zugriffs Routinen, die als semantische Aktionen im Systementwurf vorkommen:

InitMoves($\downarrow x_0 \downarrow y_0 \downarrow z_0$)

Initialisiert die Teilbewegungsliste mit dem Startpunkt (x_0, y_0, z_0).

AddLine($\downarrow x \downarrow y \downarrow z \downarrow \text{steps}$)

Erweitert die Teilbewegungsliste um eine lineare Bewegung zum Punkt (x, y, z) in steps Schritten.

AddArc($\downarrow x \downarrow y \downarrow z \downarrow x_1 \downarrow y_1 \downarrow z_1 \downarrow \text{steps}$)

Erweitert die Teilbewegungsliste um eine kreisförmige Bewegung über den Punkt (x_1, y_1, z_1) zum Punkt (x, y, z) in steps Schritten.

16.4 Transformation attributierter Grammatiken in Algo.

Grammatikelement	entspricht
Nonterminalsymbol	Interne Erkennungsprozedur
Terminalsymbol	Externe Erkennungsprozedur des lexikalischen Analysators
Ableitungsregel	Prozedurrumpf
Alternativen	if ... then ... else ... end
Option [...]	if ... then ... end
Wiederholung {...}	while ... do ... end
Attribute	Parameter d. Erkennungsprozeduren
■ Eingangsattribute	Eingangsparameter
■ Ausgangsattribute	Ausgangsparameter
Semantische Aktionen	Codestücke oder Algorithmen (gekapselt in einem “Semantik-Modul”)

Transformation attributierter Grammatiken in Algorithmen

■ Externe Erkennungsprozedur des lexikalischen Analysators (Scanner)

```
interface of Scanner  
type  
    Symbol = (digitSym, eSym, plusSym, minusSym, ... , eofSym)
```

```
    GetSymbol(↑sy: Symbol) -- provide code sy for next terminal symbol  
    GetDigitVal(↑val: int) -- provide value for current digit  
end Scanner
```

■ Interne Erkennungsprozedur

z.B. Move = ...

```
Move(): bool  
begin ...  
end Move
```

true ... Syntaxfehler erkannt
=> Abbruch

■ Erkennung von Terminal- und Nonterminalsymbolen

z.B. "+"

```
if symbol ≠ plusSym then return false end  
GetSymbol(↑symbol)
```

z.B. Position

```
if not Position(...) then  
    return false  
end -- if
```

Von der Grammatikregel zur Erkennungsprozedur (manuell)

```
PrinterMoves =  
Position↓0 ↓0 ↓0 ↑x0 ↑y0 ↑z0  
{ Move↓x0 ↓y0 ↓z0 ↑x ↑y ↑z  
}  
"E".
```

Move = Number ...

Number = digit { digit } .

```
PrinterMoves(): bool  
var  
    x0, y0, z0, x, y, z: int  
begin  
    if not Position(↓0 ↓0 ↓0 ↑x0 ↑y0 ↑z0) then  
        return false  
    end -- if  
  
    while symbol = digitSym do  
        if not Move(↓x0 ↓y0 ↓z0 ↑x ↑y ↑z) then  
            return false  
        end -- if  
  
    end -- while  
    if symbol ≠ eSym then return false end  
    GetSymbol(↑symbol)  
    return (symbol = eofSym)  
end PrinterMoves
```

Von der Grammatikregel zur Erkennungsprozedur (manuell)

```
PrinterMoves =  
Position $\downarrow_0 \downarrow_0 \downarrow_0 \uparrow_0 \uparrow_0 \uparrow_0$   
sem  
InitMoves( $\downarrow x_0 \downarrow y_0 \downarrow z_0$ )  
endsem  
{ Move $\downarrow x_0 \downarrow y_0 \downarrow z_0 \uparrow x \uparrow y \uparrow z$   
sem  
x0 := x; y0 := y; z0 := z  
endsem  
}  
"E".
```

```
PrinterMoves(): bool  
var  
x0, y0, z0, x, y, z: int  
begin  
if not Position( $\downarrow_0 \downarrow_0 \downarrow_0 \uparrow_0 \uparrow_0 \uparrow_0$ ) then  
return false  
end -- if  
InitMoves( $\downarrow x_0 \downarrow y_0 \downarrow z_0$ )  
while symbol = digitSym do  
if not Move( $\downarrow x_0 \downarrow y_0 \downarrow z_0 \uparrow x \uparrow y \uparrow z$ ) then  
return false  
end -- if  
x0 := x; y0 := y; z0 := z  
end -- while  
if symbol ≠ eSym then return false end  
GetSymbol( $\uparrow$ symbol)  
return (symbol = eofSym)  
end PrinterMoves
```

Von der Grammatikregel zur Erkennungsprozedur (manuell)

Coordinate_{v₀ ↑ v} =

 sem sign := '' endsem

["+" sem sign := '+' endsem
| "-" sem sign := '-' endsem
]

Number_{↑ v}

 sem

 if sign = '+' then

 v := v₀ + v

 elsif sign = '-' then

 v := v₀ - v

 end -- if

 endsem

.

Coordinate(_{↓ v₀} int _{↑ v} int): bool

 var

 sign: char

 begin

 sign := '

 if symbol = plusSym then

 sign := '+'

 GetSymbol(_{↑ symbol})

 elsif symbol = minusSym then

 sign := '-'

 GetSymbol(_{↑ symbol})

 end -- if

 if not Number(_{↑ v}) then return false end

 if sign = '+' then

 v := v₀ + v

 elsif sign = '-' then

 v := v₀ - v

 end -- if

 return true

 end Coordinate

Von der attributierten Grammatik zur Systemimplementierung

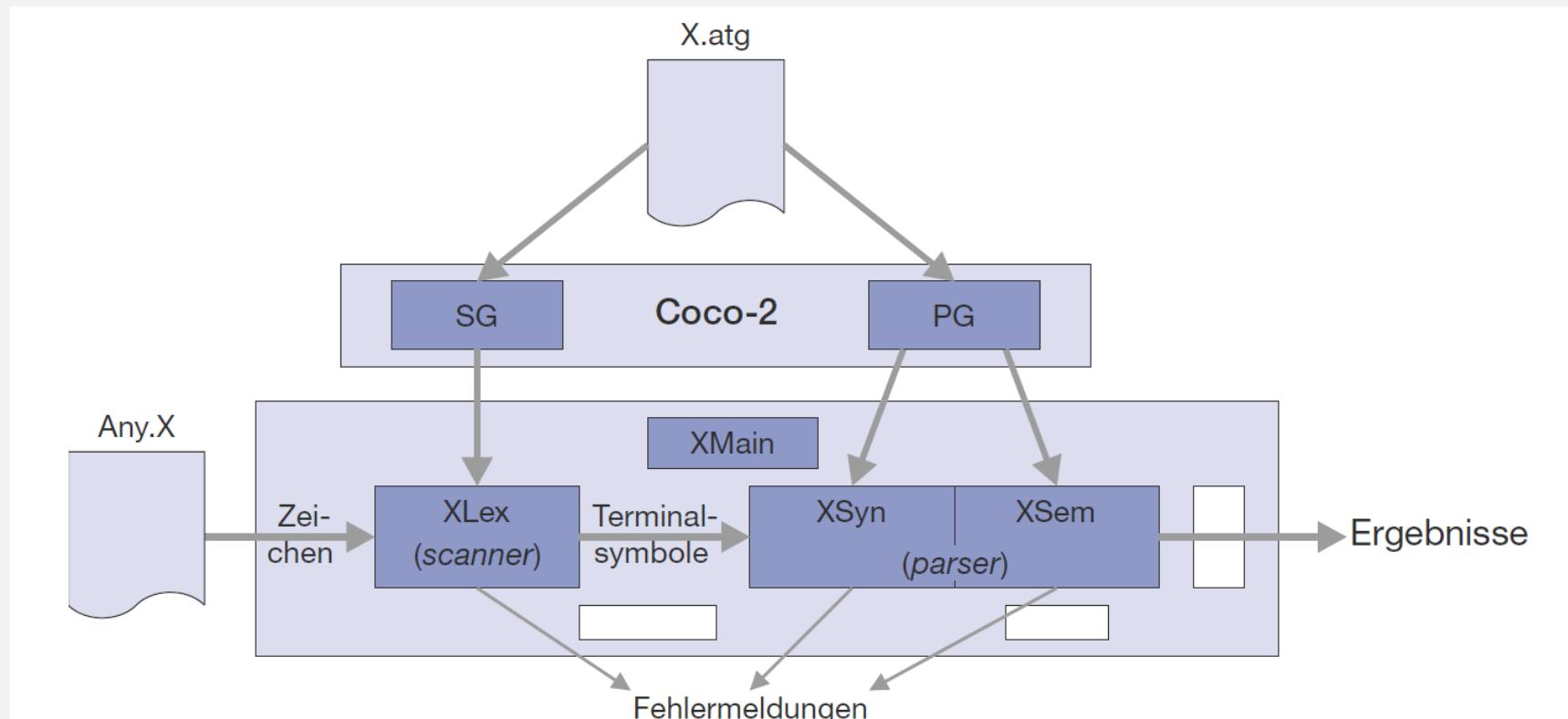
Schon in den 1970-er Jahren wurden Softwarewerkzeuge entwickelt, die den Compilerbauer bei der Herstellung wesentlicher Teile eines Compilers unterstützen, indem sie aus formalen Beschreibungen Quelltext für die dadurch spezifizierten Teile erzeugen.

Bekannte und verbreitete Compilerbau-Werkzeuge, auch Compiler-Generatoren oder Compiler-Compiler genannt, sind:

- Lex zur Generierung lexikalischer Analysatoren aus der Beschreibung lexikalischer Strukturen durch reguläre Ausdrücke.
- Yacc zur Generierung von Bottom-up-Syntaxanalysatoren aus attribuierten Grammatiken (kontextfreie LALR(1)-Grammatiken).
- Der Scanner-Generator Alex und der Parser-Generator Coco zur Herstellung von Top-down-Syntaxanalysatoren aus attribuierten Grammatiken auf Basis kontextfreier LL(1)-Grammatiken.
- Coco/R, der als Syntaxanalyseverfahren den rekursiven Abstieg nutzt und daher sehr effizient ist.
- ANTLR zur Herstellung von Syntaxanalyseverfahren und von Transformationsprozessen

Von der attributierten Grammatik zur Systemimplementierung

Beispiel: Coco-2 für unterschiedliche Programmiersprachen (z. B. Pascal, C/C++ und C#)



16.5 Vorgehensmodell zur datenorientierten Programmierung

1. Beschreiben der **lexikalischen Struktur** des Eingabedatenstroms (vorzugsweise mit **regulären Grammatiken** oder **regulären Ausdrücken**).
2. Generierung eines **lexikalischen Analysators** und Bereitstellung eines rudimentären Koordinationsrahmens, damit der lexikalische Analysator getestet werden kann.
3. Beschreiben der **syntaktischen Struktur** des Eingabedatenstroms mit einer **kontextfreien Grammatik**, die die **LL(1)-Bedingung** erfüllt.
4. Generierung eines **Syntaxanalysators** und Erweiterung des Koordinationsrahmens, damit der Syntaxanalysator getestet werden kann.

16.5 Vorgehensmodell zur datenorientierten Programmierung

5. Festlegen der **lexikalischen** und der **semantischen Attribute** und Erweiterung der kontextfreien Grammatik mit den für den beabsichtigten Transformationsprozess notwendigen **semantischen Aktionen** (unter Beachtung, dass die Bedingung für eine L-attributierte Grammatik erfüllt wird). Das Ergebnis ist eine **attributierte Grammatik**.
Hinweis: Unter Umständen müssen auch semantische Algorithmen entworfen werden, wenn man mit einfachen semantischen Aktionen nicht das Auslangen findet (es ist sinnvoll, diese in einem eigenen Modul zusammenzufassen).
6. Erweiterung des lexikalischen Analysators, so dass auch die **Werte der lexikalischen Attribute** ermittelt werden können und Generierung der vollständigen Parserkomponente (d.h. inklusive Semantikauswerter). Erweiterung des Koordinationsrahmens, so dass damit das gesamte Programmsystem getestet werden kann.

16.6 Zusammenfassung – Vor- und Nachteile

Attributierte Grammatiken sind (ein aus dem Compilerbau bekanntes Konzept) geeignet für:

- die Spezifikation von Softwaresystemen
- den Entwurf von Softwaresystemen
- die Dokumentation von Softwaresystemen

Attributierte Grammatiken bzw. datenorientierte Programmierung sind dann nutzbringend einsetzbar:

- wenn im Wesentlichen ein Eingabedatenstrom vorliegt
- wenn der Eingabedatenstrom genügend strukturiert ist
- wenn im Wesentlichen der Eingabedatenstrom in eine andere Form transformiert werden soll

Zusammenfassung – Vor- und Nachteile

Vorteile

- Deskriptive Entwurfstechnik
- Modularisierungstechnik
- zwingt zur Trennung von Syntax und Semantik, Syntax ist Routine, die/der Entwickler*in kann sich auf Semantik konzentrieren
- knappe und (mit etwas Übung) gut lesbare Darstellung (hervorragendes Dokumentationsmittel)
- Systementwurf lässt sich automatisch in Programme transformieren

Nachteile

- nicht anwendbar bei mehreren parallel zu verarbeitenden Eingabedatenströmen
- Anwendung setzt Grundkenntnisse des Übersetzerbaus voraus

Algorithmen und Datenstrukturen

Eine systematische Einführung in die

Programmierung

17 Objektorientierter Systementwurf

Josef Pichler

Version 10.1, 2024

17.1 Grundlegendes zur objektorientierten Programmierung

- Im Mittelpunkt steht die Auffassung, dass **Daten und Operationen Einheiten**, nämlich Objekte, bilden
- Ein Programmsystem kann als Sammlung von miteinander **kommunizierenden Objekten** angesehen werden
- Zur Laufzeit besteht ein solches Programmsystem aus einem **Netzwerk von Objekten**, die gemeinsam an der Lösung des Problems arbeiten

Wichtige Ziele:

1. Verminderung der semantischen Lücke
 - durchgängiger methodischer Ansatz verkleinert die semantische Lücke **zwischen Anwenderwelt und Implementierung**
 - Identifikation und Modellierung problemadäquater Objekte in allen Entwicklungsphasen

Wichtige Ziele (Fortsetzung)

2. Meisterung der Komplexität
 - klare, problemadäquate Modularisierung des Programmsystems
 - überschaubare Systemkomponenten mit geringer Kopplung
 - gute Schnittstellenabstraktion
3. Verbesserung der **Wiederverwendbarkeit**
 - Realisierung von Komponenten mit flexiblen Halbfertigfabrikaten
 - Konstruktion neuer Systeme durch Kombination bereits existierender Komponenten und Halbfertigfabrikaten
4. Verbesserung der **Änder- und Erweiterbarkeit**
 - Inkrementelle Erweiterung und Anpassung von Systemkomponenten ohne Änderung ihres Quellcodes
 - systemweit wirksame funktionale Anpassungen

Zum Objektbegriff

Ein Objekt repräsentiert eine **Laufzeitkomponente** eines objektorientierten Programmsystems und setzt sich aus zwei Arten von Bestandteilen zusammen: **Datenkomponenten** und **Methoden**

Datenkomponenten (*Attribute, members*)

- Sie repräsentieren die Eigenschaften eines Objekts und werden im Sinne der Datenkapselung vor der Außenwelt verborgen (*information hiding*)
- Sie sind vor dem direkten Zugriff von außen geschützt; sie können von außen nur mit entsprechenden Methoden manipuliert werden

Methoden (*methods*)

- Methoden repräsentieren Operationen, die ein Objekt auf „seinen“ Datenkomponenten ausführen kann
- Methoden werden in Form von Algorithmen realisiert
- Im Zuge der Ausführung einer Methode eines Objekts können auch andere Objekte erzeugt und benutzt (indem Messages an Objekte verschickt werden) werden
- Unter einer **Message** verstehen wir die abstrakte Form einer Methode, repräsentiert durch die Schnittstelle der Methode

Zum Objektbegriff

Ein Objekt repräsentiert eine **Laufzeitkomponente** eines objektorientierten Programmsystems und setzt sich aus zwei Arten von Bestandteilen zusammen: **Datenkomponenten** und **Methoden**

Beispiel	Datenkomponenten	Methoden
Account	balance, changes	Deposit, Withdraw, Balance, ...
Stack	data: array..., top	Push, Pop, IsEmpty
Random	seed	IntRand, RealRand, RangeRand, ...
Hashtable	data: array...	Insert, Find, Remove, Size, ...
Rectangle	x, y, width, height	MoveBy, Contains, Equals, ...
Rational	num, denom	Plus, Minus, Divides, Times, ...
Color	red, green, blue	Darker, Brighter, Intensity, ToGray, ...

Zum Klassenbegriff

Ein Objekt ist dadurch charakterisiert, dass

- es einen **Zustand hat** (Werte der Datenkomponenten)
- bestimmte **Aufträge erledigen kann** (Messages/Methoden) und
- es zu einer bestimmten **Klasse gehört**

Klasse

- Eine Klasse ist ein **benutzerdefinierter Datentyp**, der als Schablone zur Bildung von Objekten, die alle über die gleichen Eigenschaften (Datenkomponenten) und Methoden verfügen, verwendet wird
- Eine Klasse ist ein **abstrakter Datentyp (ADT)**
- Wie ein abstrakter Datentyp stellt sie **abstrakte Operationen** (Messages) und **Implementierung** (Methoden) zum Zugriff auf die Datenkomponenten zur Verfügung und ermöglicht es, **beliebig viele Exemplare** (Objekte) davon zu erzeugen

Zum Klassenbegriff

Klasse

- Im Unterschied zu den bisher eingeführten abstrakten Datentypen können **Klassen hierarchisch angeordnet** und „**erweitert**“, d. h. mit **zusätzlichen Eigenschaften und Methoden** ausgerüstet werden
- Klassen sind also **erweiterbare abstrakte Datentypen**
- Die Objekte einer erweiterten Klasse besitzen neben den durch die Erweiterung festgelegten Eigenschaften und Methoden auch die Eigenschaften und Methoden aller anderen, in der Klassenhierarchie vor ihnen liegenden Klassen (Basisklassen genannt)
- Eine Klasse kann also von einer anderen Klasse (der Basisklasse) Eigenschaften und Methoden „**erben**“
- Um dies umzusetzen, benötigen wir zusätzliche Konzepte

Zusätzliche benötigte Konzepte

- **Vererbung.** Klassen und damit die aus ihnen erzeugten Objekte können (z. B. durch Ableitung) die in anderen Klassen definierten Eigenschaften und Operationen erben und zu den erbten Eigenschaften und Operationen können weitere hinzugefügt werden.
- **Polymorphie.** Variablen vom Typ einer Klasse können nicht nur Objekte dieser, sondern auch Objekte aller direkt oder indirekt davon abgeleiteten Klassen referenzieren. Damit können Variablen Objekte unterschiedlicher (polymorpher) Gestalt repräsentieren.
- **Dynamische Bindung.** Die Bindung von (Methoden-)Aufrufen mit einem bestimmten Codestück kann auch erst zur Laufzeit durchgeführt werden. Für gleiche Operationen können also verschiedene Algorithmen vorgesehen werden.
- Darüber hinaus gehören zu OOP (aus Komfortgründen) auch noch weitere Konzepte.

Zur Historie

1970er Jahre: Strukturierte Programmierung

- Prozeduren zur Strukturierung von Softwaresystemen
- Vermeidung unbeschränkter Ablaufstrukturen in Algorithmen
- Einführung von Datentypen
- basierend auf den Sprachentwicklungen Algol, Pascal

Ziel: Qualitätssteigerung und Meisterung der Komplexität

1980er Jahre: Modulorientierte Programmierung

- Moduln zur Strukturierung von Softwaresystemen
- Abstrakte Datentypen, *information hiding*
- basierend auf den Sprachentwicklungen Ada, Modula-2

Ziel: Qualitätssteigerung und arbeitsteilige Softwareentwicklung

1990er Jahre: Objektorientierte Programmierung

- Klassen zur Strukturierung von Softwaresystemen
- erweiter- und veränderbare abstrakte Datentypen
- Möglichkeit der Faktorisierung von Gemeinsamkeiten durch Vererbungsmechanismus
- Steigerung der Flexibilität durch Polymorphie und dynamische Bindung
- basierend auf den Sprachentwicklungen Smalltalk, C++

Ziel: Produktivitäts- und Qualitätssteigerung

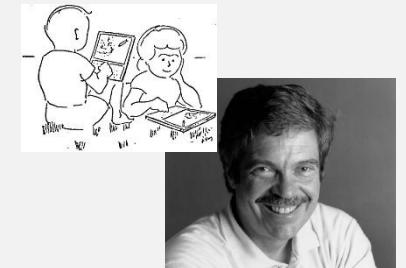
Zur Historie

Ursprung der objektorientierten Programmierung

- Simula (1967): Objekte und Klassen für Simulationstechnik
O. Dahl und K. Nygaard gelten als die Erfinder der OOP
Begriffe: *class, this, ...*
- Smalltalk (um 1975, A. Kay) ist auch heute noch eine bedeutende rein objektorientierte Programmiersprache
Begriffe: *sub class, super class, ...*
- Konzepte aus Smalltalk wurden in andere Sprachen übernommen



O. Dahl, K. Nygaard



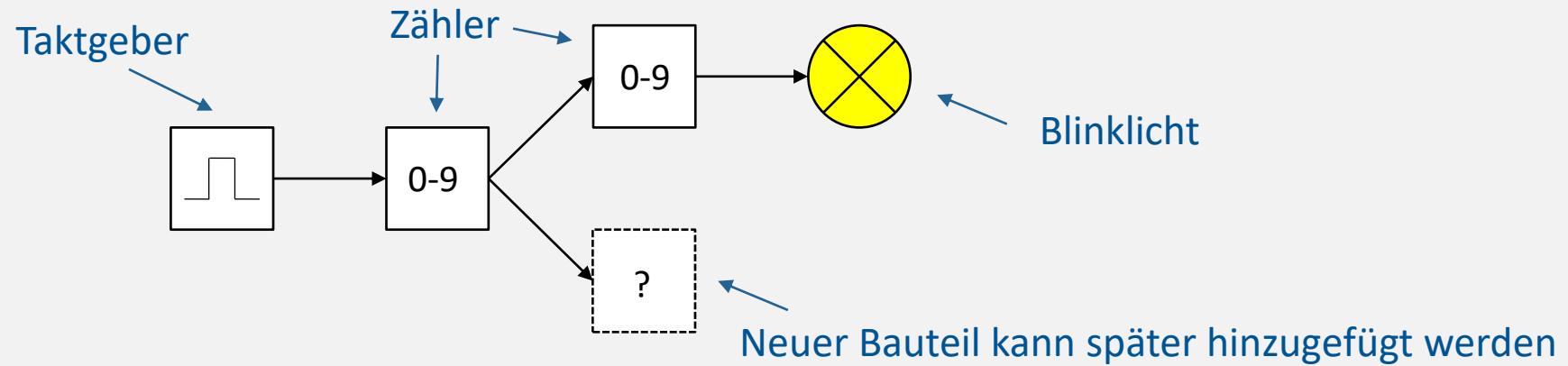
A. Kay

Man unterscheidet

- Rein objektorientierte Sprachen („alles ist ein Objekt!“)
- Hybridsprachen (Sprachen mit hinzugefügten Elementen der OOP)

Objekte

- Elemente aus der realen oder gedachten Welt (Problemwelt)
- Analogie zu technischen Bauteilen

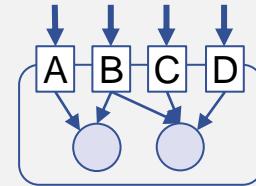


- Jedes Objekt weiß selbst, welche Operationen wie auszuführen sind
- Nur das Objekt selbst kennt seinen Zustand (Geheimnisprinzip)
- Alle Datenobjekte (im bisherigen Sinne) können als Objekte (im objektorientierten Sinne) aufgefasst werden

Von abstrakten Datenstrukturen zu Objekten

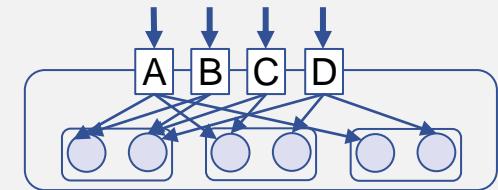
Abstrakte Datenstruktur

- Ein Exemplar mit Zugriffs routinen



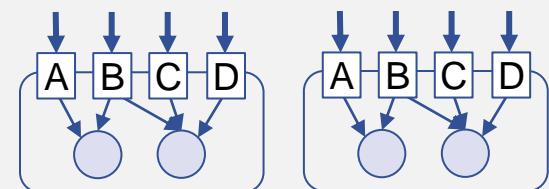
Abstrakter Datentyp

- Beliebig viele Exemplare mit Zugriffs routinen



Abstrakter Datentyp im oo. Sinne

- Beliebig viele Exemplare mit eingebauten Zugriffs routinen (Methoden)
- Objekt enthält wie ein Verbund Datenobjekt
- jedoch zusätzlich auch Methoden



Weiteres zum Klassenbegriff

Gleichartige Objekte werden zu einer **Klasse** zusammengefasst

Klassen sind Schablonen für gleichartige Objekte

- beschreibt Aufbau (Datenkomponenten) ihrer Objekte und
- die mit ihnen möglichen Operationen

Eine Klasse entspricht einem **abstrakten Datentyp** in dem Sinne, dass von ihr beliebig viele Objekte gebildet werden können

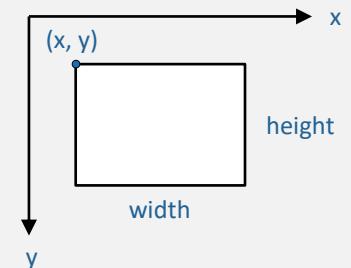
Beispiel: Klasse für Rechteck

```
type
  Rectangle = class
    x, y, width, height: int

    MoveBy(↓dx: int ↓dy: int)
    Contains(↓px: int ↓py: int): bool
    Draw()
  end -- Rectangle
```

Datenkomponenten
(vgl. Verbund-Datentyp)

mögliche Operationen
(Nachrichten, Messages)



Zum Methodenbegriff

- Methoden sind die Algorithmen, die **ein Objekt beim Empfang von Nachrichten ausführt**, um eine Aufgabe zu erledigen
- Methoden werden **im Kontext von Klassen** implementiert
- In Methoden wird auf die Datenkomponenten des Objekts zugegriffen

```
Rectangle.MoveBy(↓dx: int ↓dy: int)
begin
    x := x + dx
    y := y + dy
end Rectangle.MoveBy
```

x, y ... Zugriff auf Datenkomponenten

dx, dy ... Eingangsparameter der Methode

```
Rectangle.Contains(↓px: int ↓py: int): bool
begin
    return (x < px) and (px < (x + width)) and (y < py) and (py < (y + height))
end Rectangle.Contains
```

```
Rectangle.Draw()
begin
    DrawRect(↓x ↓y ↓width ↓height)
end Rectangle.Draw
```

DrawRect ... Standardprozedur

Variablen und Objekte

- Deklaration von Variablen zur Identifikation von Objekten

```
var  
    r1, r2, r3: →Rectangle
```

r1: r2: r3:

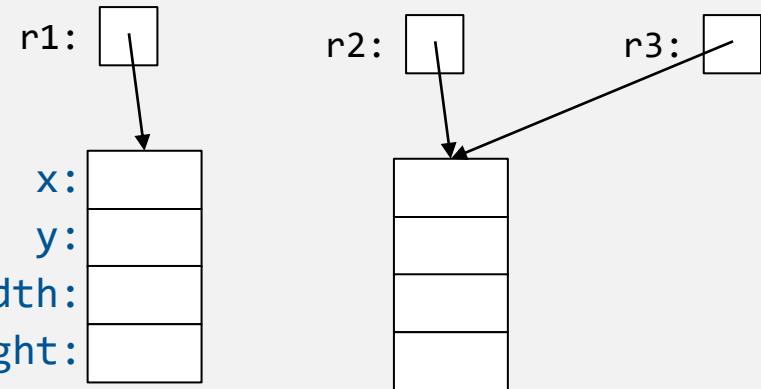
Variablen sind noch nicht initialisiert

- Erzeugung von Objekten

```
r1 := New(↓Rectangle)
```

```
r2 := New(↓Rectangle)
```

```
r3 := r2
```



- Freigeben von Objekten

```
Dispose(↓r1)
```

```
Dispose(↓r2)
```

- Variablen eines Klassentyps sind für uns immer Zeigervariablen
- Objekte sind dynamische Datenobjekte (liegen am Heap)
- Objekte enthalten nur Datenkomponenten; Methoden werden nur einmal pro Klasse gespeichert

Zum Begriff Message/Nachricht

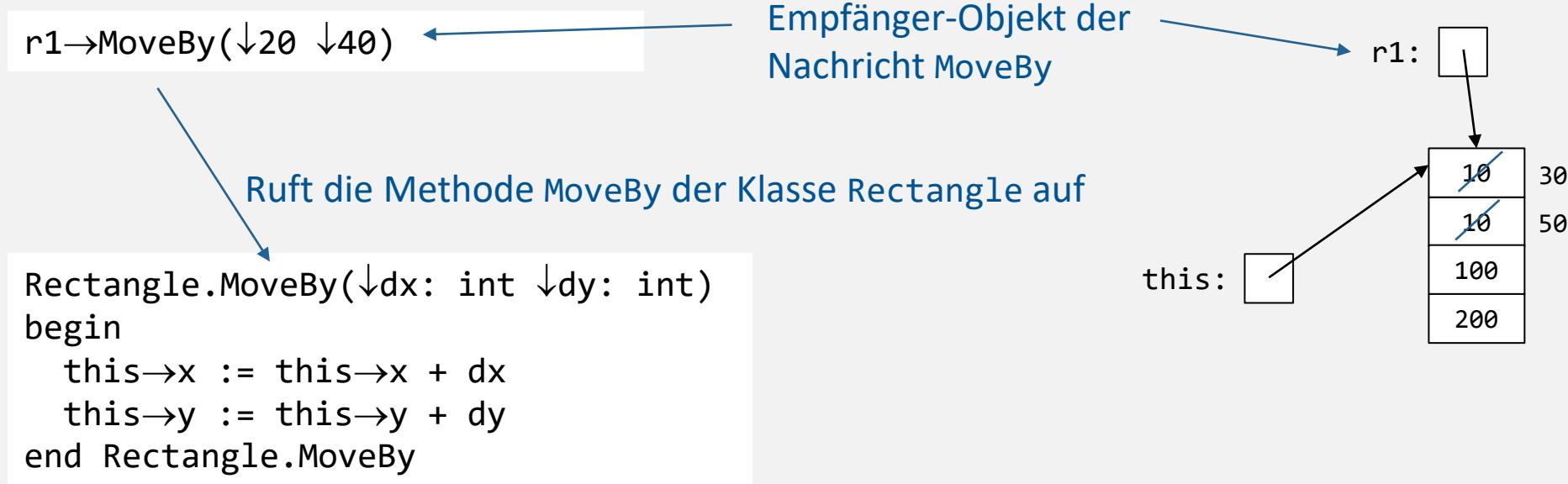
- Objekte kommunizieren über Nachrichten (*messages*) miteinander (Mechanismus, um Methoden von Objekten aufzurufen)

```
if r1→Contains(↓4 ↓3) then  
    r1→MoveBy(↓20 ↓30)  
    r1→Draw()  
end -- if
```

- Nachrichten können verschiedene Wirkung haben
 - Eigenschaft des Empfängerobjekts abfragen (z.B. Contains)
 - Zustand des Empfängerobjekts verändern (z.B. MoveBy)
 - Empfängerobjekt zu Aktion veranlassen (z.B. Draw)
 - ...
- Sender der Nachricht hat keine Kenntnis, wie die Methoden der Empfänger implementiert sind
- Empfängerobjekt entscheidet, welche Methode ausgeführt wird

Zum Begriff Message/Nachricht

Beispiel: Nachricht MoveBy an ein von Variable r1 referenziertes Objekt



- Innerhalb der Methode steht `this` für den Namen des Empfängerobjekts

```
x := x + dx
y := y + dy
```

Man kann `this` auch weglassen, wenn sich kein Namenskonflikt ergibt

Zum verwendeten Begriffssystem

Gegenüberstellung der verwendeten Begriffe

Bisher	Im Kontext der OOP
Abstrakter Datentyp (ADT)	Klasse
Exemplar eines ADT	Objekt
Zugriffsroutine	Message
Algorithmus/Prozedur	Methode
Algorithmen-/Prozeduraufruf	(üblicherweise) Senden einer Nachricht Es gibt auch noch Algorithmenaufruf

Zum Konstruktorbegriff

- Dient der Initialisierung von Objekten zum Zeitpunkt der Erzeugung
- Ist eine spezifische Methode, die beim Erzeugen eines Objekts (automatisch) aufgerufen wird
- Konvention: Name des Konstruktors ist identisch mit dem der Klasse

Beispiel (Deklaration)

```
type
    Rectangle = class
        x, y, width, height: int
        Rectangle(↓x: int ↓y: int ↓width: int ↓height: int)
        ...
    end -- Rectangle
```

Konstruktoren haben nur Eingangsparameter!

```
Rectangle.Rectangle(↓x: int ↓y: int ↓width: int ↓height: int)
begin
    this→x := x; this→y := y
    this→width := width; this→height := height
end Rectangle.Rectangle
```

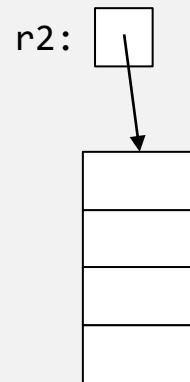
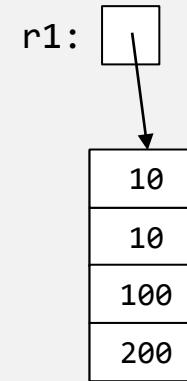
Zum Konstruktorbegriff

- Konstruktoren werden automatisch aufgerufen, wenn ein Objekt ihrer Klasse erzeugt wird

```
var  
    r1, r2: →Rectangle
```

```
r1 := New(↓Rectangle ↓10 ↓20 ↓100 ↓200)
```

1. Erzeugt ein Rectangle-Objekt
2. Ruft Konstruktor auf, der die Datenkomponenten initialisiert
3. Zuweisung der Startadresse des Objekts an Variable r1



- Erzeugung (wie bisher) ohne Konstruktor

```
r2 := New(↓Rectangle)
```

1. Erzeugt ein Rectangle-Objekt
2. Zuweisung der Startadresse des Objekts an Variable r2

Geheimnisprinzip (*information hiding*)

- Direkter Zugriff auf die Komponenten eines Objekts soll verhindert werden
- Zugriffsrechte werden z.B. durch `private` und `public` definiert

```
type
  Rectangle = class
  private
    x, y, width, height: int
  public
    Rectangle(↓x: int ↓y: int ↓width: int ↓height: int)
    ...
  end -- Rectangle
```

`private` ... nur innerhalb der Klasse sichtbar

`public` ... global sichtbar

- Verwendung

```
var
  r1: →Rectangle
```

```
r1 := New(↓Rectangle ↓10 ↓20 ↓100 ↓200)
r1→x := 42
Write(↓r1→x)
```

Wegen `private` nicht möglich!

Geheimnisprinzip: Get- und Set-Methoden

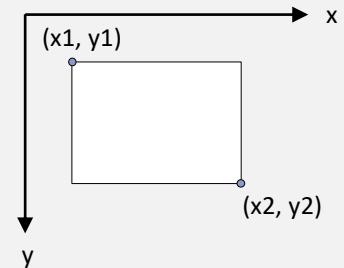
Zugriffsmethoden zu privaten Datenkomponenten

```
type
  Rectangle = class
  private
    x, y, width, height: int
  public
    Width(): int
    SetWidth(↓w: int)
    Height(): int
    SetHeight(↓h: int)
    X(): int
    Y(): int
    ...
  end -- Rectangle
```

Lese- und Schreibzugriff für
Eigenschaft Width

Nur Lesezugriff für Eigenschaft X

```
Rectangle.setWidth(↓w: int)
begin
  if w ≥ 0 then
    this→width := w
  else
    this→width := 0
end -- if
end Rectangle.setWidth
```



Konsequenzen

- Implementierung der Datenkomponenten kann ausgetauscht werden
- x1, y1, x2, y2: int
- Zusatzfunktionen (z.B. Vorbedingungen prüfen) beim Zugriff möglich

Ein erstes Anwendungsbeispiel: Realisierung eines Stacks

Beispiel: Gesucht ist eine Klasse IntStack für die Verwaltung von Integer-Objekten nach dem LIFO-Prinzip

Schritt 1: Welche Operationen werden benötigt?

```
s := New(↓IntStack)
s→Push(↓47)
s→Push(↓11)
while not s→IsEmpty() do
    e := s→Pop()
    Write(↓e)
end -- while
```

```
var
    s: →IntStack
    e: int
```

Schritt 2: Welche Datenkomponenten werden für die Implementierung benötigt?

```
const
    max = ...
var
    stack: array [1:max] of int
    top: int
```

Beispiel: Klasse IntStack

Schritt 3: Klassendeklaration

```
type
  IntStack = class
  private
    const max = 10
    stack: array [1:max] of int
    top: int
  public
    IntStack()
    Push( $\downarrow$ e: int)
    Pop(): int
    IsEmpty(): bool
  end -- IntStack
```

} Datenobjekte sind nicht von Außen zugänglich

} Konstruktor
verfügbare Operationen/Messages, die von Objekten dieses Typs „verstanden“ werden

Beispiel: Klasse IntStack

Schritt 4: Implementierung der Methoden (Klassenimplementierung)

```
IntStack.IntStack()
begin
    this→top := 0
end IntStack.IntStack
```

```
IntStack.Push(↓e: int)
begin
    if top < max then
        top := top + 1
        stack[top] := e
    else
        -- error handling
    end -- if
end IntStack.Push
```

```
IntStack.IsEmpty(): bool
begin
    return (top = 0)
end IntStack.IsEmpty
```

```
IntStack.Pop(): int
var e: int
begin
    if not IsEmpty() then
        e := stack[top]
        top := top - 1
        return e
    else
        -- error handling
        return -1
    end -- if
end IntStack.Pop
```

Beispiel: Klasse Color

- Eine Farbe ist ein durch das Auge/Gehirn vermittelter Sinneseindruck, der durch elektromagnetische Strahlung hervorgerufen wird
- Gesucht ist eine Klasse Color zur Repräsentation von Farben

```
type
  Color = class
  private
    red, green, blue: int
  public
    Color(↓r: int ↓g: int ↓b: int)
    Red(): int
    Green(): int
    Blue(): int
    -- make color darker
    Darker()
    -- make color brighter
    Brighter()
    -- is this color the same as c's?
    Equals(↓c: →Color): bool
  end -- Color
```

Datenkomponenten:

Intensität für Rot-, Grün-, Blauanteil (0-255)

R	255	0	0	255	0	51	51
G	0	255	0	255	0	51	102
B	0	0	255	255	0	153	0
Farbe							

Beispiel: Klasse Color

Konstruktor und Zugriffsmethoden der Klasse Color

```
Color.Color(↓r: int ↓g: int ↓b: int)
begin
    red := r; green := g; blue := b
end Color.Color
```

```
Color.Red(): int
begin
    return red
end Color.Red
```

```
Color.Green(): int
begin
    return green
end Color.Green
```

```
Color.Blue(): int
begin
    return blue
end Color.Blue
```

Beispiel: Klasse Color

Methode Darker

```
Color.Darker()
  const f = 0.95
begin
  red := Int(↓red * f)
  green := Int(↓green * f)
  blue := Int(↓blue * f)
end Color.Darker
```

Benutzung: 

```
var
  c: →Color; i: int
```

```
c := New(↓Color ↓0 ↓255 ↓0)
for i := 1 to 10 do
  x := 30 * i
  SetFillColor(↓c→Red() ↓c→Green() ↓c→Blue())
  FillRect(↓x ↓30 ↓20 ↓20)
  c→Darker()
end -- for
Dispose(↓c)
```

Beispiel: Klasse Color

Methode Equals

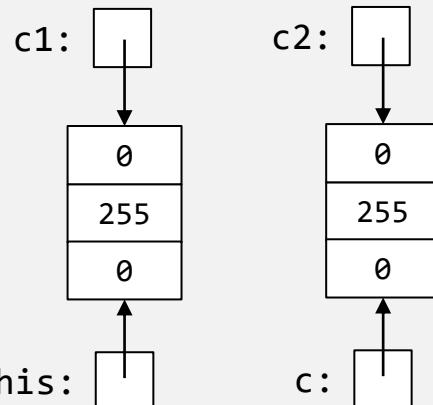
in Methoden der Klasse Color ist der Zugriff erlaubt!

```
Color.Equals(↓c: →Color): bool  
begin  
    return red = c→red and green = c→green and blue = c→blue  
end Color.Equals
```

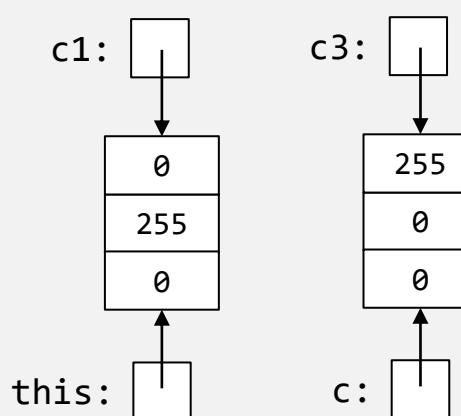
Benutzung:

```
var c1, c2, c3: →Color  
c1 := New(↓Color ↓0 ↓255 ↓0)  
c2 := New(↓Color ↓0 ↓255 ↓0)  
c3 := New(↓Color ↓255 ↓0 ↓0)
```

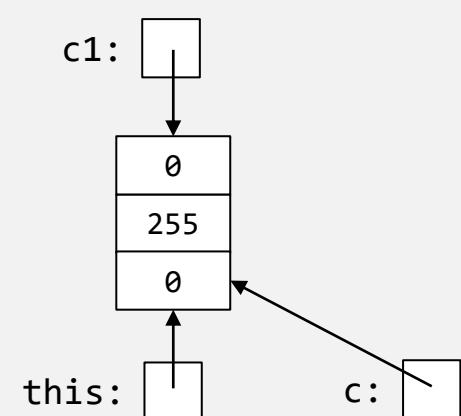
c1→Equals(↓c2)



c1→Equals(↓c3)



c1→Equals(↓c1)



Beispiel: Rechnen mit Farben: Helligkeit (Luminanz)

- Die Luminanz ist ein Maß für die Helligkeit von Bildpunkten
- NTSC-Standardformel für Helligkeit: $0.299r + 0.587g + 0.114b$

```
type
  Color = class
    ...
    Intensity(): real
    ...
  end -- Color
```

```
Color.Intensity(): real
begin
  return 0.299*red + 0.587*green + 0.114*blue
end Color.Intensity
```

Beispiele:

R	255	0	0	255	0	51	51
G	0	255	0	255	0	51	102
B	0	0	255	255	0	153	0
Farbe							
Lum.	76	150	29	255	0	62	75

Beispiel: Rechnen mit Farben: Graustufen

- Aufgabe: Konvertierung einer Farbe in entsprechende Graustufe
- Hinweis: wenn alle drei R/G/B-Komponenten den gleichen Wert haben, ergibt dies eine Graustufe zwischen 0 (schwarz) und 255 (weiß)
- Welchen Wert soll man für eine bestimmte Farbe wählen? Den Luminanz-Wert

```
type
  Color = class
  ...
  ToGray(): →Color
end -- Color
```

```
Color.ToGray(): →Color
var
  gray: →Color; i: int
begin
  i := Int(↓this→Intensity())
  gray := New(↓Color ↓i ↓i ↓i)
  return gray
end Color.ToGray
```

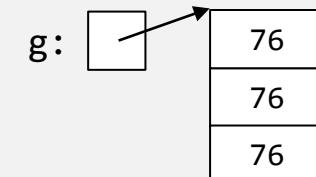
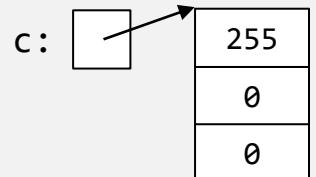
Beispiele:

Farbe							
Lum.	76	150	29	255	0	62	75
Graustufe							

Benutzung:

```
var c, g: →Color
c := New(↓Color ↓255 ↓0 ↓0)
```

```
g := c→ToGray()
```



17.2 Vererbung

Ein fundamentales und für OOP charakteristisches Konzept ist das **Vererbungskonzept**:

- Klassen (Typen) und damit die aus ihnen erzeugten Objekte können bereits **anderswo definierte Eigenschaften und Operationen erben** und zu diesen Eigenschaften und Operationen weitere hinzufügen
- Hypothese: Viele Klassen sind einander ähnlich und unterscheiden sich nur in Details
- Vererbung (*inheritance*) gestattet die Bildung neuer Klassen ausgehend von bestehenden Klassen

Beispiele

- Ein Objekt das Studentendaten repräsentiert hat alle Eigenschaften eines Objekts das Personendaten repräsentiert **und ein Personenkennzeichen**
- Ein farbiges Rechteck hat alle Eigenschaften eines gewöhnlichen Rechtecks **und zusätzlich eine Farbe**

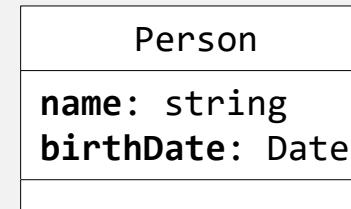
Klassen für Personen- und Studentenobjekte

Beispiel ohne Vererbung

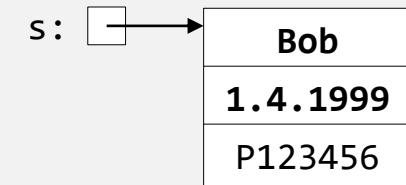
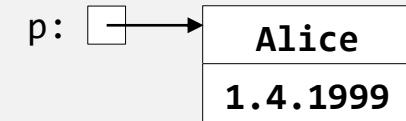
```
type
  Person = class
    name: string
    birthDate: Date
    ...
  end -- Person

  Student = class
    name: string
    birthDate: Date
    id: string
    ...
  end -- Student
```

Klassen:



Objekte:



- Operationen der Klasse Person zur Manipulation der Komponenten name und birthDate müssen auch in der Klasse Student implementiert werden (doppelter Code)
- eine Änderung der Klasse Person (z.B. neue Komponenten, Operationen) zieht meist auch Änderung der Klasse Student mit sich

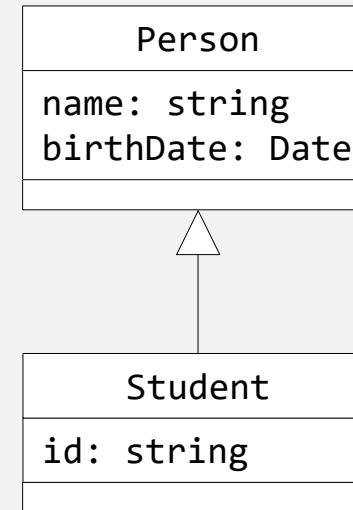
Klassen für Personen- und Studentenobjekte

Beispiel mit Vererbung

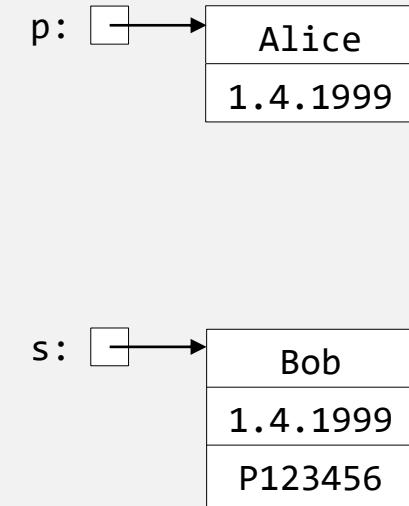
```
type
  Person = class
    name: string
    birthDate: Date
    ...
  end -- Person

  Student = class based on Person
    id: string
    ...
  end -- Student
```

Klassen:



Objekte:

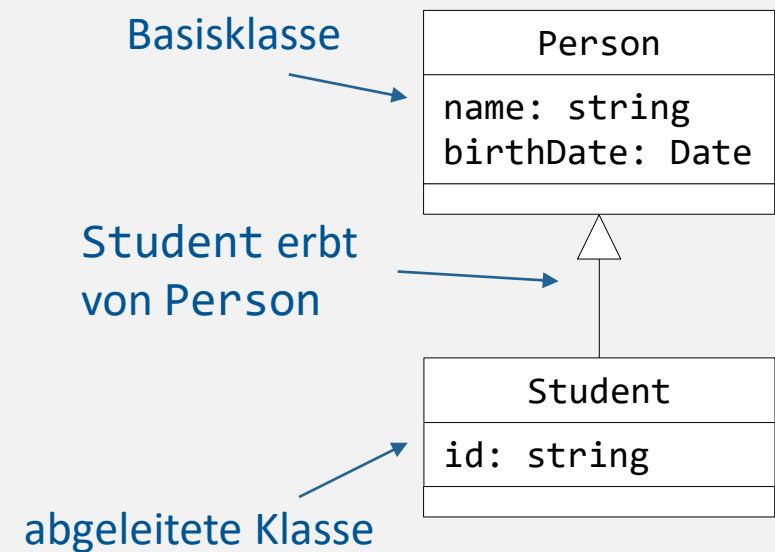


- die Klasse Student erweitert die Klasse Person und „erbt“ alle Eigenschaften und Operationen; in der Klasse Student wird neue Komponente hinzugefügt
- ein Objekt der Klasse Student besitzt alle Eigenschaften der Klasse Person
- ein Objekt der Klasse Student ist auch ein Objekt der Klasse Person

17.2.1 Bezeichnungen

Klassen

- Person ... Basisklasse
(base class, super class)
- Student ... abgeleitete Klasse
(derived class, sub class)



„ist-ein“ - Beziehung

- Durch Vererbung entsteht eine **ist-ein**-Beziehung zwischen Objekten:
ein Objekt vom Typ B ist auch ein Objekt vom Typ A, wenn die Klasse B
von der Klasse A (direkt od. indirekt) abgeleitet wurde
- anders ausgedrückt:
ein Studentenobjekt kann alles, was ein Personenobjekt kann

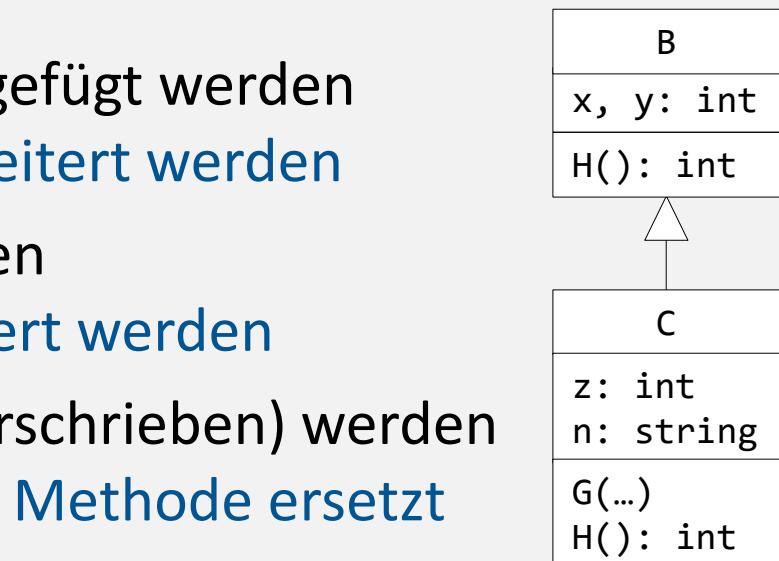
17.2.2 Prinzip der Vererbung

Wenn von einer Basisklasse B eine neue Klasse C abgeleitet wird, erbt C alle Eigenschaften (Komponenten) und Operationen (Messages/Methoden), die B schon hat

Die abgeleitete Klasse C kann gegenüber ihrer Basisklasse B erweitert und adaptiert werden

- neue Datenkomponenten (z, n) können hinzugefügt werden
 - ... damit kann das Eigenschaftsspektrum verbreitert werden
- neue Methoden (G) können hinzugefügt werden
 - ... damit kann das Funktionsspektrum verbreitert werden
- geerbte Methoden (H) können angepasst (überschrieben) werden
 - ... geerbte Funktionalität wird durch eine neue Methode ersetzt

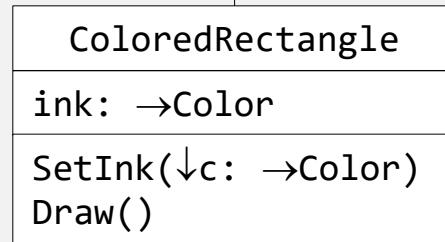
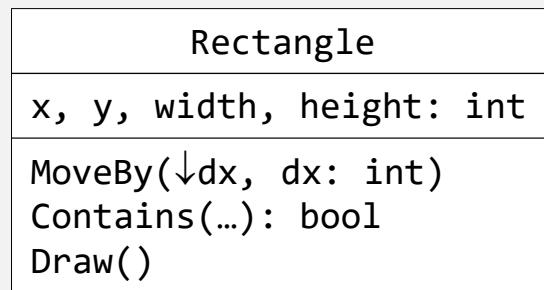
Nicht möglich ist das Entfernen oder Ändern von Datenkomponenten oder Methoden



Überschreiben von Methoden: für die Message H an ein C-Objekt gibt es zwei Methoden!

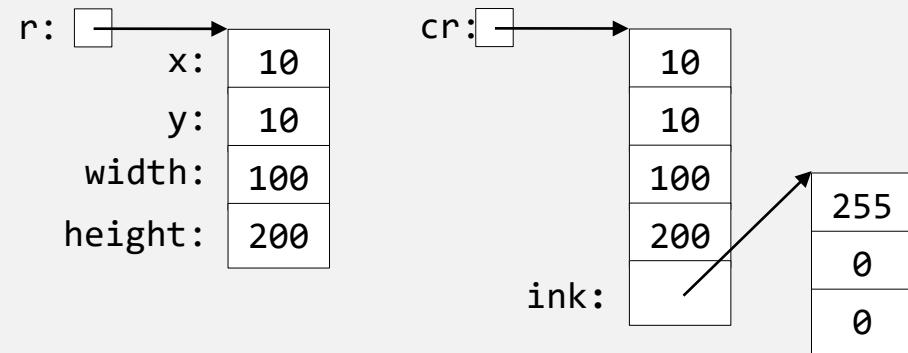
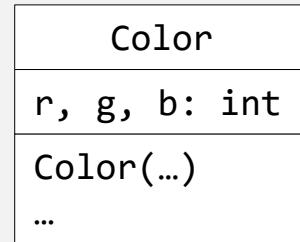
Illustration des Vererbungsprinzips

- Bisherige Rectangle-Objekte wurden in einer Farbe (schwarz) gezeichnet
- Gesucht: eine Klasse ColoredRectangle für farbige Rechtecke
- Es ist naheliegend, die Klasse Rectangle als Basisklasse für die neue Klasse zu wählen



ColoredRectangle ist gegenüber *Rectangle* erweitert und adaptiert worden:

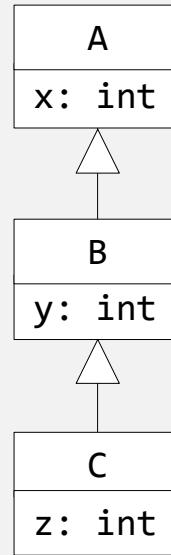
- eine neue Datenkomponente (`ink`) ist hinzugekommen
- eine neue Operation (`SetInk`) ist zur Verfügung gestellt worden
- eine geerbte Operation (`Draw`) ist modifiziert (überschrieben) worden



Neue Datenkomponenten hinzufügen

Deklarationen

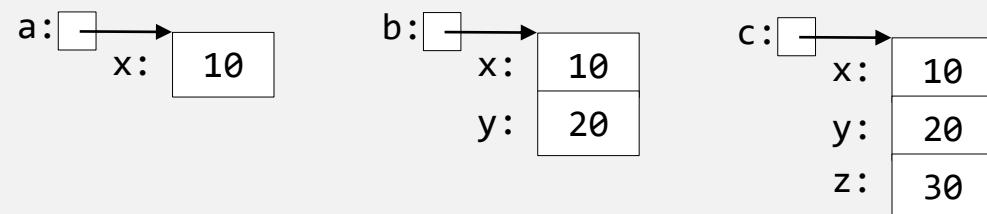
```
type
  A = class
    x: int
  end -- A
  B = class based on A
    y: int
  end -- B
  C = class based on B
    z: int
  end -- C
```



Verwendung

```
var
  a: →A; b: →B; c: →C
```

```
a := New(↓A)
b := New(↓B)
c := New(↓C)
```



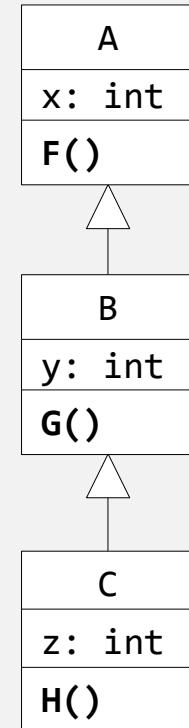
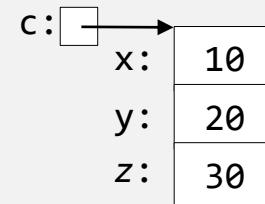
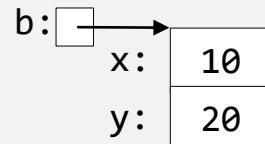
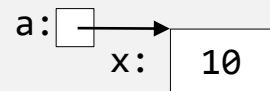
Neue Methoden hinzufügen

Verwendung

```
var  
  a: →A; b: →B; c: →C
```

```
a := New(↓A)  
b := New(↓B)  
c := New(↓C)
```

Objekte



Nachrichten

a→F() ✓

a→G() ✗

a→H() ✗

b→F() ✓

b→G() ✓

b→H() ✗

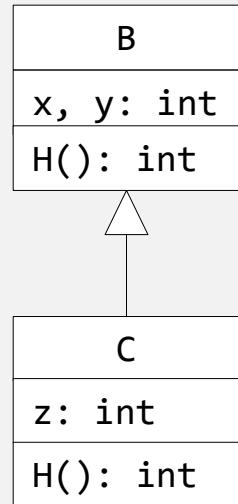
c→F() ✓

c→G() ✓

c→H() ✓

Geerbte Methoden modifizieren (überschreiben)

Klassen und Methoden



für die Message H an ein
C-Objekt gibt es zwei Methoden!

Methoden H berechnen Hash-
Werte für B- oder C-Objekte

```
B.H(): int
begin
    return 31*x + y
end B.H
```

```
C.H(): int
begin
    return 31*31*x + 31*y + z
end C.H
```

Objekte und Nachrichten

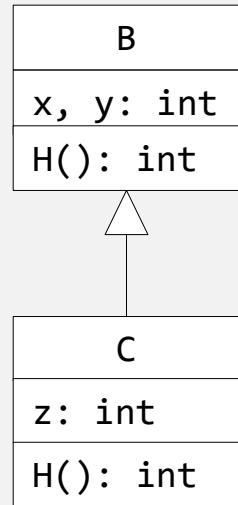
```
var
    b: →B; c: →C
```

```
b := New(↓B ↓10 ↓20)
c := New(↓C ↓10 ↓20 ↓30)
```

Message	Methoden	
b→H()	B.H	$31*10 + 20$
c→H()	C.H	$31*31*10 + 31*20 + 30$

Geerbte Methoden überschreiben mit super

Klassen und Methoden



für die Message H an ein
C-Objekt gibt es zwei Methoden!

Methoden H berechnen Hash-
Werte für B- oder C-Objekte

```
B.H(): int
begin
    return 31*x + y
end B.H
```

```
C.H(): int
begin
    return 31*super→H() + z
end C.H
```

Objekte und Nachrichten

```
var
    b: →B; c: →C
```

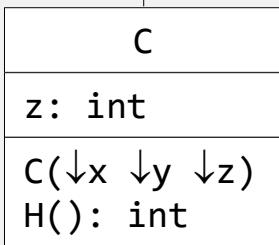
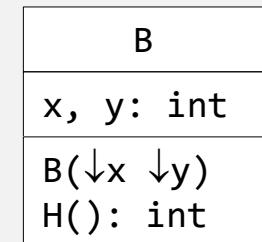
```
b := New(↓B ↓10 ↓20)
c := New(↓C ↓10 ↓20 ↓30)
```

Message	Methoden	
b→H()	B.H	$31*10 + 20$
c→H()	C.H, B.H	$31*(31*10 + 20) + 30$

Konstruktoren(-aufruf) bei Vererbung

Konstruktoren der Basisklasse werden explizit aufgerufen

Klassen:



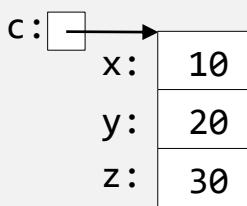
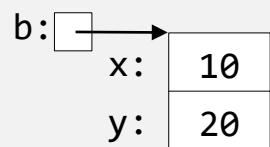
var
b: →B; c: →C

b := New(↓B ↓10 ↓20)
c := New(↓C ↓10 ↓20 ↓30)

Konstruktoren:

B.B(↓x: int ↓y: int)
begin
 this→x := x
 this→y := y
end B.B

C.C(↓x: int ↓y: int ↓z: int)
begin
 super→B(↓x ↓y)
 this→z := z
end C.C

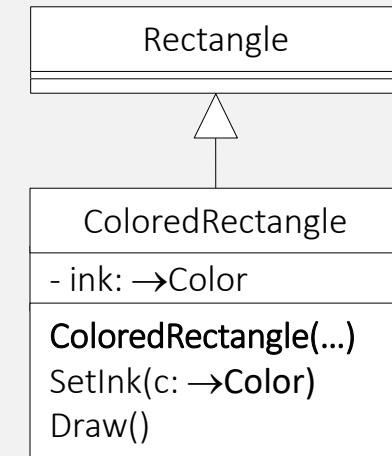


Beispiel: Klasse ColoredRectangle

Vervollständigung der Deklarationen

Konstruktor

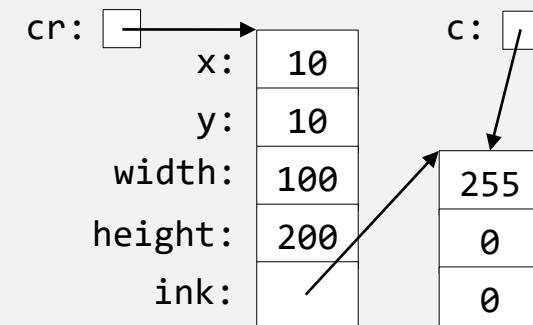
```
ColoredR....ColoredRectangle(↓x: ... ↓ink: →Color)
begin
    super→Rectangle(↓x ↓y ↓width ↓height)
    this→ink := ink
end ColoredRectangle.ColoredRectangle
```



Benutzung

```
var
    c: →Color
    cr: →ColoredRectangle

    c := New(↓Color ↓255 ↓0 ↓0)
    cr := New(↓ColoredR... ↓10 ↓10 ↓100 ↓200 ↓c)
```



Beispiel: Klasse ColoredRectangle

Methode Draw in Basisklasse

```
Rectangle.Draw()  
begin  
    DrawRect(↓x ↓y ↓width ↓height)  
end Rectangle.Draw
```



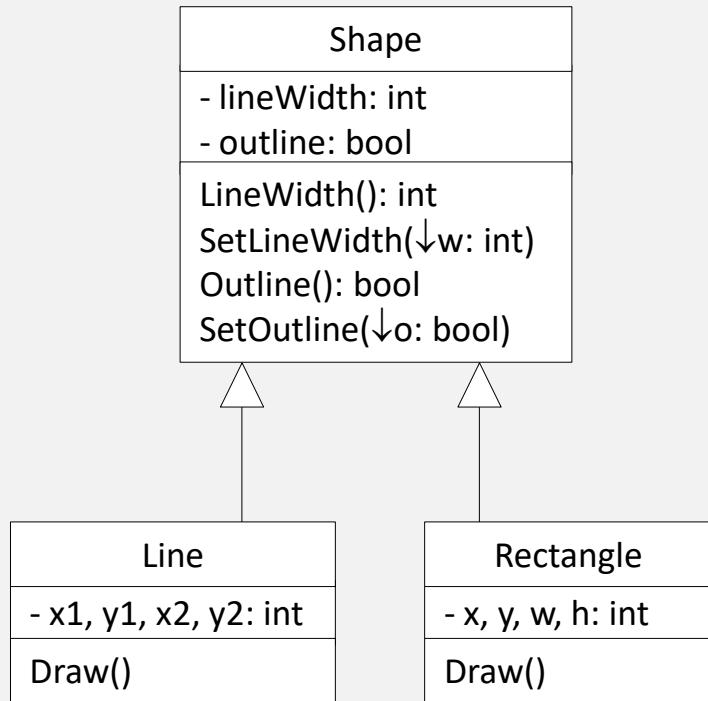
Methode Draw in abgeleiteter Klasse

```
ColoredRectangle.Draw()  
begin  
    if ink ≠ null then  
        SetFillColor(↓ink→Red() ↓ink→Green() ↓ink→Blue())  
    end -- if  
    FillRect(↓x() ↓y() ↓width() ↓Height())  
    super→Draw()  
end ColoredRectangle.Draw
```



Beispiel: Klassen für geometrische Objekte

- allgemeine Eigenschaften (Umriss, Linienstärke, ...) in Basisklasse
- Zeichenoperationen in abgeleiteten Klassen (Shape kann nicht gezeichnet werden)



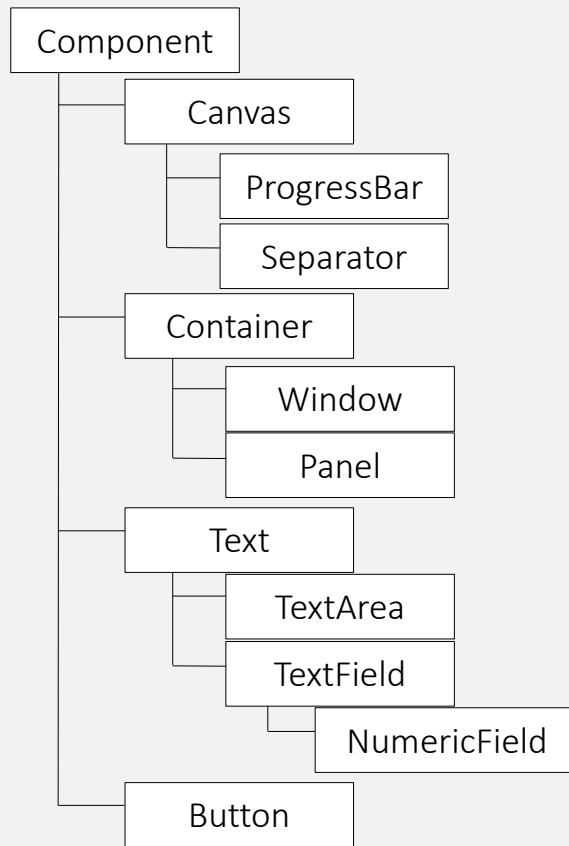
```
Shape.Outline(): bool
begin
    return outline
end Shape.Outline
```

```
Rectangle.Draw()
begin
    if Outline() then
        DrawRect(x, y, w, h)
    else
        FillRect(x, y, w, h)
    end -- if
end Rectangle.Draw
```

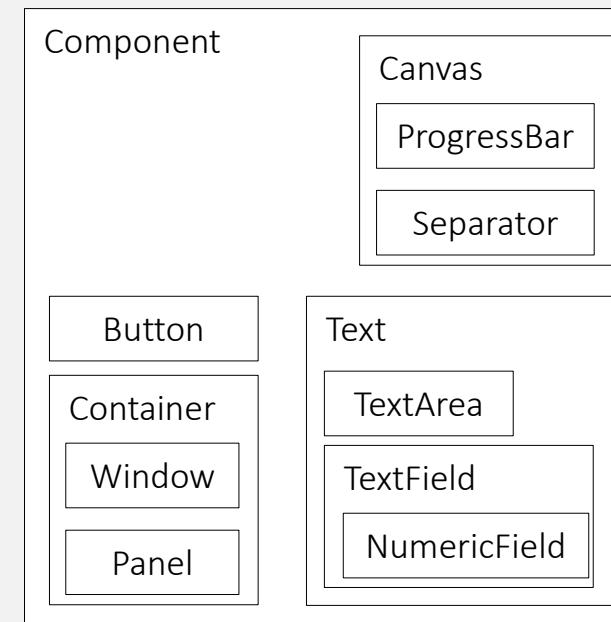
17.2.3 Klassenhierarchien

(einfache) Vererbung führt zu Klassenhierarchien

Baumdarstellung



Mengendiagramm



Abgeleitete Klassen sind im Allgemeinen spezieller als ihre Basisklassen

Vererbung

Vererbung wird benutzt um

- Codeverdopplungen bei ähnlichen abstrakten Datentypen zu vermeiden
- mit minimalem Aufwand neue Datentypen zu erzeugen, für die es schon Vorbilder gibt
- gemeinsame Schnittstellen für eine ganze Familie von Klassen zu definieren
- getesteten Code zu verwenden und zu erweitern, ohne ihn zu verändern

17.3 Polymorphismus

- Bisher (vor OOP) haben wir statisch typisierte Datenobjekte verwendet
- Eine Variable a vom Typ A kann nur A-Objekte aufnehmen
(= **monomorphe Variablen**)

```
var  
  a: A
```

```
var  
  i: int
```

```
var  
  s: string
```

- Oft wären auch **polymorphe Variablen** wünschenswert und das lässt sich durch die bisher eingeführten Konzepte nun realisieren

Polymorphe Variablen

- Durch Vererbung von Klassen entsteht eine „**ist-ein**“-Beziehung zwischen Objekten
- Eine Variable a vom Typ A kann deshalb Objekte der Klasse A und aller davon **abgeleiteten Klassen** aufnehmen

Polymorphismus in der OOP

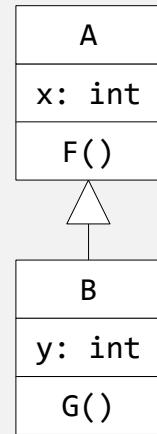
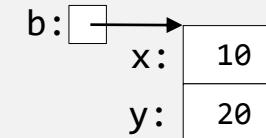
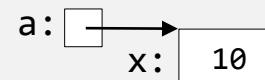
- In der objektorientierten Programmierung bedeutet Polymorphismus, dass mit Variablen vom Typ einer Basisklasse nicht nur Objekte dieser, sondern auch **Objekte aller direkt oder indirekt davon abgeleiteten Klassen** referenziert werden können
- Damit können solche Variablen auf **Objekte unterschiedlicher Gestalt** verweisen, weshalb wir in dieser Situation von Polymorphismus sprechen
- Diese Objekte haben allerdings eine Gemeinsamkeit: ihre Datentypen (Klassen) stammen **von derselben Basisklasse** ab
- Der Datentyp einer Variablen kann sich somit zur Laufzeit ändern. Wir sprechen daher vom **dynamischen Datentyp**
- Variablen im OO Sinne besitzen also einen **statischen** und einen **dynamischen Datentyp**

Beispiel: Polymorphe Variablen

Bisher

```
var  
  a: →A; b: →B
```

```
a := New(↓A ↓10)  
b := New(↓B ↓10 ↓20)
```

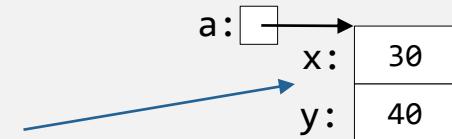


Neu:

- B erbt alle Eigenschaften und Operationen von A
- Objekte der Klasse B „können“ also alles, was Objekte der Klasse A „können“ (eventuell auch mehr)

```
a := New(↓B ↓30 ↓40)  
a→F()  ✓  a→G()  X
```

dynamischer Datentyp von a ist nun B (abgeleitet vom Typ A)



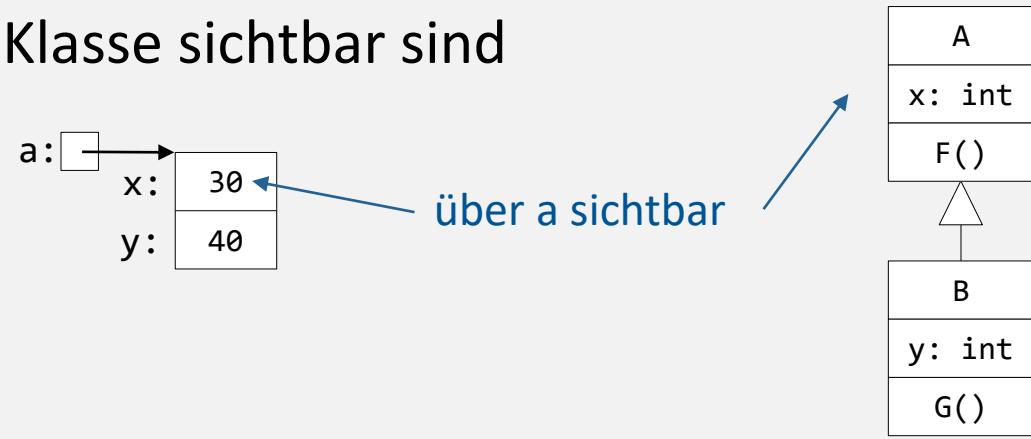
- über Variable a (vom stat. Typ A) kann nur auf Eigenschaften und Operationen des B-Objekts zugegriffen werden, die bereits in A definiert sind

Statischer und dynamischer Datentyp

Statischer Typ (*static type*)

- Wird in der Deklaration festgelegt und ist unveränderlich
- Bestimmt, welche Elemente der Klasse sichtbar sind

```
var  
    a: →A
```



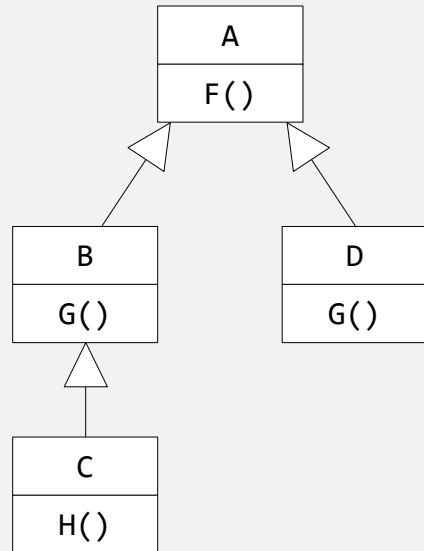
Dynamischer Typ (*dynamic type*)

- Ergibt sich durch die Zuweisung
- ```
a := New(↓B ↓30 ↓40)
```
- Kann sich durch eine neuerliche Zuweisung wieder ändern
- Bestimmt, welche Methode aufgerufen wird (dynamische Bindung)

## 17.3.1 Polymorphismusregeln für Zuweisungen

Objekte einer Klasse B dürfen einer Variable des Typs A zugewiesen werden, wenn B von A direkt oder indirekt abgeleitet wurde

Klassendiagramm:



Deklarationen:

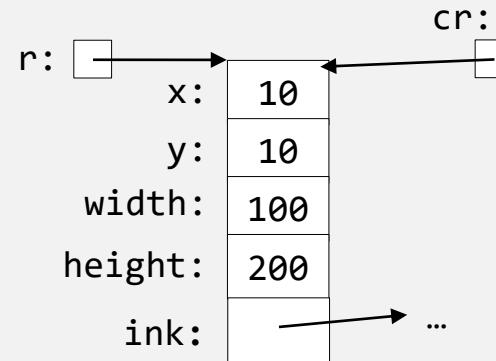
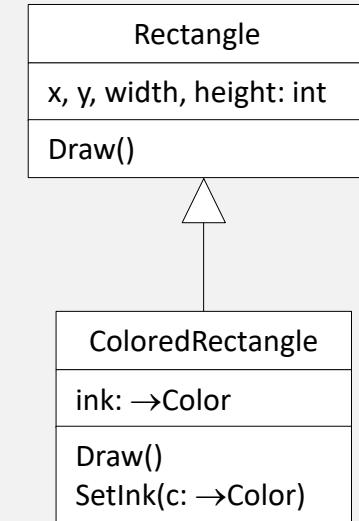
```
var
 a: →A; b: →B;
 c: →C; d: →D
```

|        |   |                                                                                 |
|--------|---|---------------------------------------------------------------------------------|
| a := b | ✓ | Variablen einer Wurzelklasse können Objekte jeder abgeleiteten Klasse aufnehmen |
| a := c | ✓ |                                                                                 |
| a := d | ✓ |                                                                                 |
| b := a | ✗ | B „kann mehr“ als A                                                             |
| b := c | ✓ |                                                                                 |
| b := d | ✗ | D-Objekte sind keine B-Objekte                                                  |
| c := a | ✗ | C „kann mehr“ als A und B                                                       |
| c := b | ✗ |                                                                                 |
| c := d | ✗ | C-Objekte sind keine D-Objekte                                                  |
| d := a | ✗ | D „kann mehr“ als A                                                             |
| d := b | ✗ | B- und C-Objekte sind keine D-Objekte                                           |
| d := c | ✗ |                                                                                 |

# Beispiel: Rechtecke

```
var
 r: →Rectangle
 cr: →ColoredRectangle
```

```
cr := New(↓ColoredRectangle ...)
r := cr
```



r→Draw() ✓  
r→SetInk() ✗

cr→Draw() ✓  
cr→SetInk() ✓

- Ein Objekt `r` (vom Typ `Rectangle`) kann nur die Nachricht `Draw()` interpretieren
- Ein Objekt `cr` (vom Typ `ColoredRectangle`) kann die Nachrichten `Draw()` und `SetInk()` interpretieren

## 17.3.2 Auswirkungen

- Der dynamische Datentyp einer Variable des statischen Datentyps X ist zur Übersetzungszeit nicht vorhersehbar

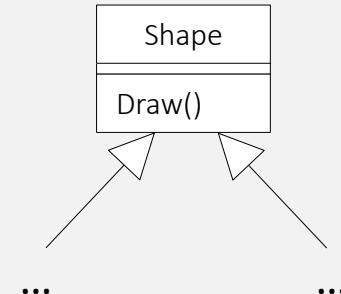
```
var item: →Shape
```

welches Objekt verbirgt sich hinter item?

- Programmteile, die mit Objekten der Klasse X arbeiten, können ohne Änderung auch mit Objekten von abgeleiteten Klassen arbeiten

```
const max = ...
var shapes: array [1:max] of →Shape
n: int

...
for i := 1 to n do
 shapes[i]→Draw()
end -- for
```



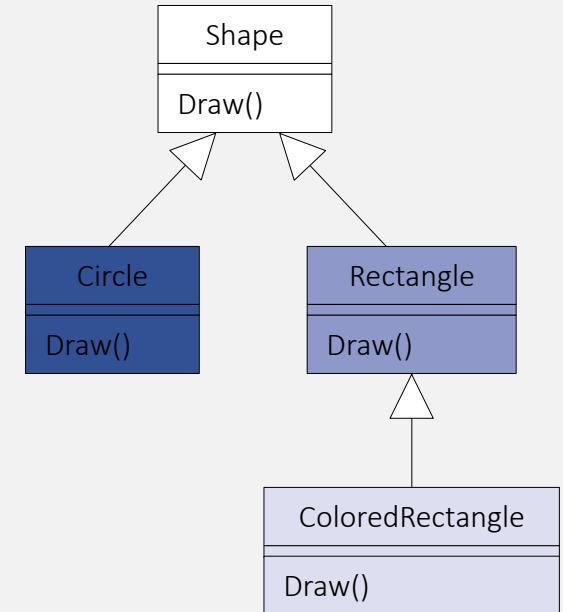
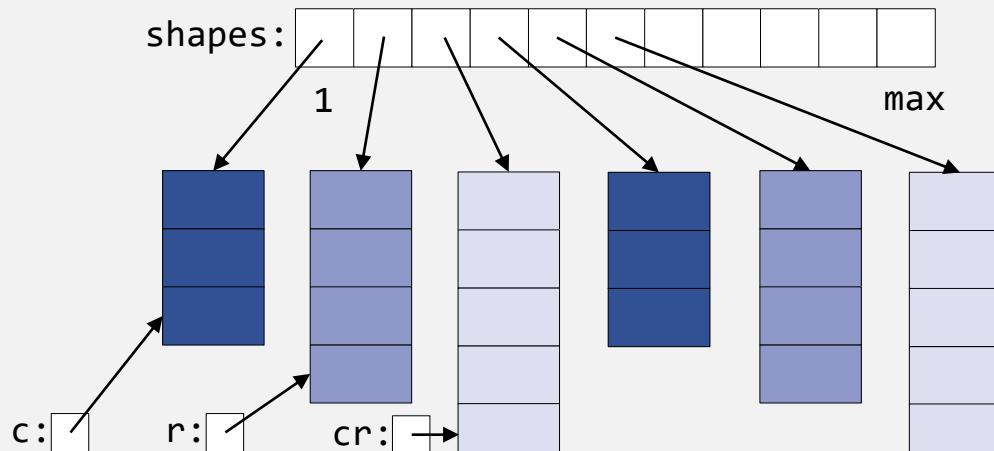
- Die Hinzunahme neuer Shapes kann ohne Programmänderung erfolgen

# Beispiel

## Beispiel: Sammlung geometrischer Objekte

```
var
 shapes: array [1:max] of →Shape
 c: →Circle; r: →Rectangle; cr: →ColoredRectangle
```

```
c := New(↓Circle); shapes[1] := c
r := New(↓Rectangle); shapes[2] := r
cr := New(↓ColoredRectangle); shapes[3] := cr
shapes[4] := New(↓Circle)
shapes[5] := New(↓Rectangle)
shapes[6] := New(↓ColoredRectangle)
```



Verwendung:

```
for i := 1 to 6 do
 shapes[i]→Draw()
end -- for
```

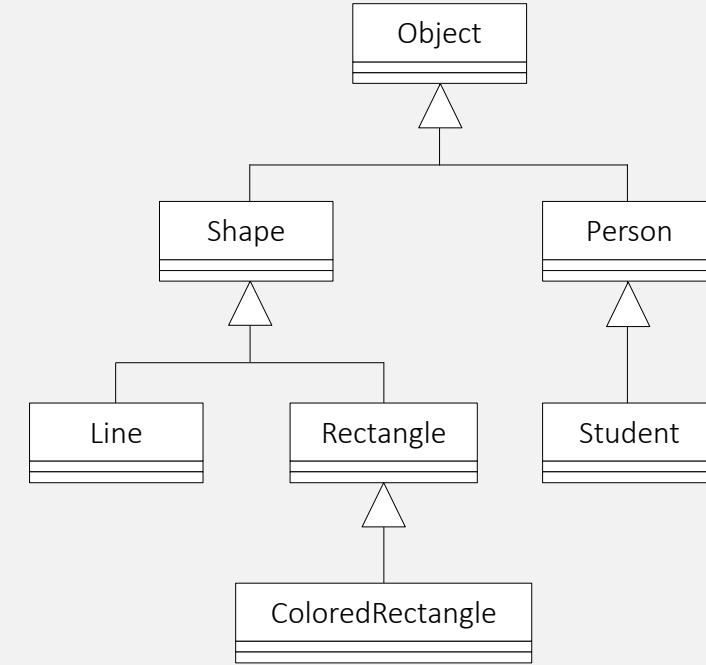
# Beispiel: Verallgemeinerte Objektsammlungen

Datenstrukturen (Listen, Bäume, ...) für Objekte beliebigen Typs

Alle Klassen/Typen müssen (direkt oder indirekt) von einer Wurzelklasse/Wurzeltyp abgeleitet werden

```
type
ObjectStack = class
private
 top: int
 data: array[1:max] of →Object
public
 Push(↓e: →Object)
 Pop(): →Object
 IsEmpty(): bool
end -- ObjectStack
```

```
var s: →ObjectStack
s := New(↓ObjectStack)
s→Push(↓New(↓Line))
s→Push(↓New(↓Person))
s→Push(↓New(↓Student))
```



## 17.4 Dynamische Bindung

---

Unter Bindung versteht man im Kontext von Programmiersprachen die **Verknüpfung des Aufrufs einer Operation** (einer Prozedur, Funktion oder Methode) mit der **auszuführenden Folge von Anweisungen**

In Abhängigkeit vom Zeitpunkt, wann die Bindung stattfindet, kann man zwei Ausprägungen unterscheiden:

- Bei nicht objektorientierten Programmiersprachen kann die Verknüpfung für Prozedur- und Funktionsaufrufe entweder **der Compiler** zur Übersetzungszeit (wenn das gesamte Programm in einer Übersetzungseinheit vorliegt) **oder der Binder (linker)** zur Bindezeit herstellen. In beiden Fällen erfolgt die Bindung vor der Programmausführung, weshalb diese Art von Bindung als **statische Bindung** bezeichnet wird.
- Bei objektorientierten Programmiersprachen gibt es (wegen des objekt-orientierten Paradigmas) auch Situationen, in denen die **Bindung** von Methodenaufrufen mit einem bestimmten Codestück **erst zur Laufzeit** durchgeführt werden kann, weshalb diese Art von Bindung als **dynamische Bindung** bezeichnet wird.

# Dynamische Bindung

---

- Bindung = Verknüpfung zwischen Senden einer Nachricht und Auswahl einer Methode
- Für monomorphe Variablen kann Methode zur Übersetzungszeit ausgewählt werden

```
var
 a: A
```

```
a.F()
```

hinter a kann sich nur ein A-Objekt verbergen  
daher muss Methode A.F() ausgeführt werden

- Bei polymorphen Variablen ist der dynamische Datentyp des Objekts zur Übersetzungszeit nicht bekannt
- Die „richtige“ Methode kann erst zur Laufzeit ausgewählt werden
- **Auswahl** hängt vom **dynamischen Datentyp des Objekts** ab, das sich hinter einer Variablen verbirgt

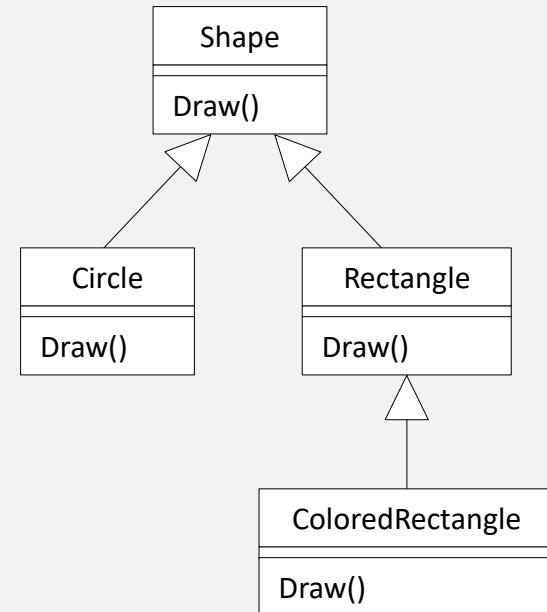
# Beispiel: Zeichnen einer Objektsammlung

- Damit Nachricht Draw() von Shape-Objekten „verstanden“ werden kann, muss Draw() in Shape definiert werden

```
for i := 1 to 6 do
 shapes[i]→Draw()
end -- for
```

- Dynamische Bindung

| Nachricht        | Methode                 |
|------------------|-------------------------|
| shapes[1]→Draw() | Circle.Draw()           |
| shapes[2]→Draw() | Rectangle.Draw()        |
| shapes[3]→Draw() | ColoredRectangle.Draw() |
| shapes[4]→Draw() | Circle.Draw()           |
| shapes[5]→Draw() | Rectangle.Draw()        |
| shapes[6]→Draw() | ColoredRectangle.Draw() |



Beachte: Shape.Draw() wird nie aufgerufen!

## 17.4.1 Statische vs. dynamische Bindung

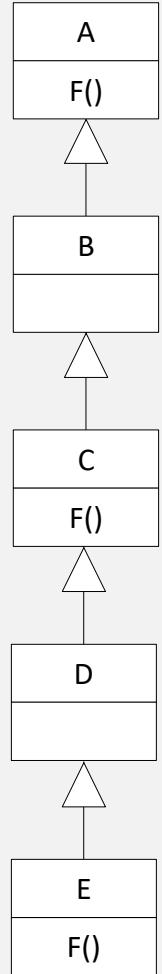
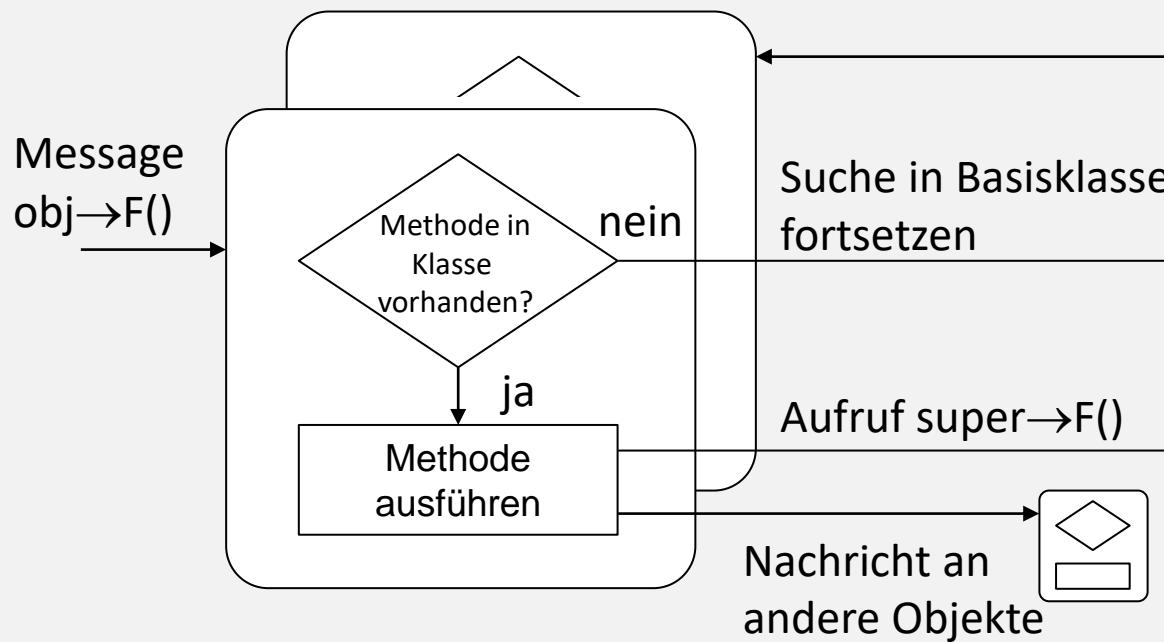
---

- Bei konventioneller Programmierung sieht man im Quelltext, welcher Code ausgeführt wird
- Das gilt bei objektorientierter Programmierung nicht mehr
- Dynamische Bindung wirkt sich auf die Laufzeit aus

|             | <b>konventionell</b>             | <b>objektorientiert</b>                                            |
|-------------|----------------------------------|--------------------------------------------------------------------|
| Einheit     | Prozedur/Funktion                | Methode                                                            |
| Aktivierung | Aufruf<br>$P()$ oder $v := F()$  | Nachricht<br>$obj \rightarrow P()$ oder $v := obj \rightarrow F()$ |
| Bindung     | durch Namen                      | durch Namen und Klasse des Empfängerobjektes                       |
| Zeitpunkt   | Übersetzung<br>Statische Bindung | Laufzeit<br>Dynamische Bindung                                     |

## 17.4.2 Dynamische Bindung durch Methodensuche

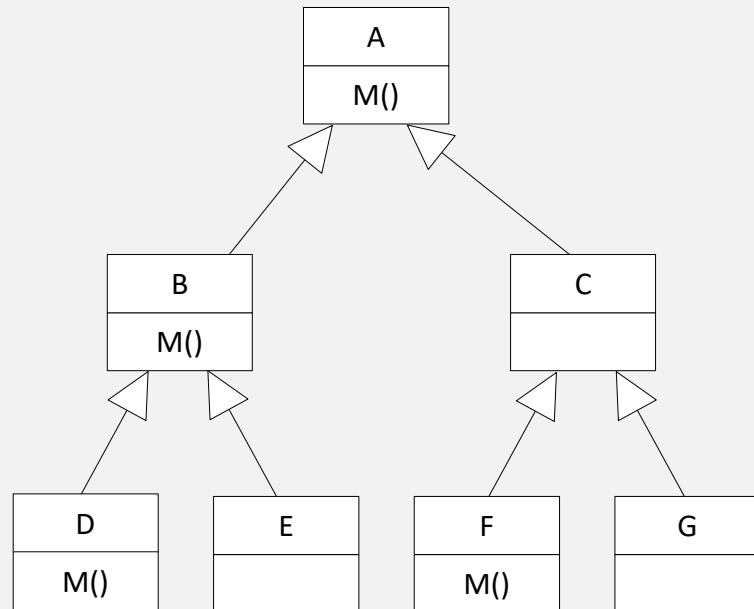
- Beginn bei Klasse des Empfänger-Objekts
- Ende bei direkter oder indirekter Basisklasse
- In typisierten Sprachen ist Methodensuche immer erfolgreich



# Methodensuche in einer Klassenhierarchie

Beispiel: Methode M() ist in den Klassen A, B, D und F implementiert

Klassendiagramm:



Methodensuche:

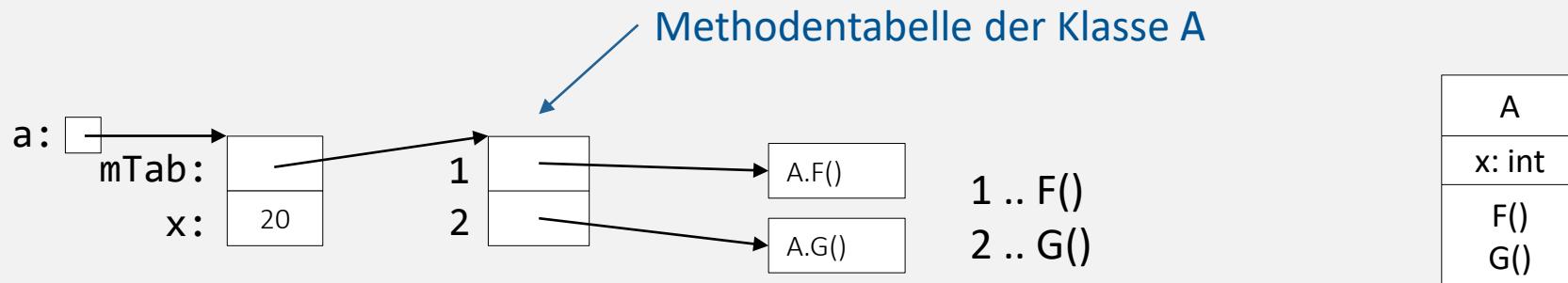
| Objekt  | Suchreihenfolge  | Aufruf    |
|---------|------------------|-----------|
| A,B,D,F | sofort in Klasse | A.M(),... |
| C       | C → A            | A.M()     |
| E       | E → B            | B.M()     |
| G       | G → C → A        | A.M()     |

=> Methodensuche ist aufwändig

## 17.4.3 Dynamische Bindung durch Methodentabelle

- Methoden werden nummeriert
- Adressen werden in **Methodentabelle** abgelegt
- jedes Objekt hat zusätzliche Variable mTab, die das Objekt mit der Methodentabelle verknüpft (mTab für Programmierer nicht zugänglich)
- in allen Objekten ein und derselben Klasse zeigt mTab auf ein und dieselbe Methodentabelle
- für jede Klasse gibt es eine Methodentabelle

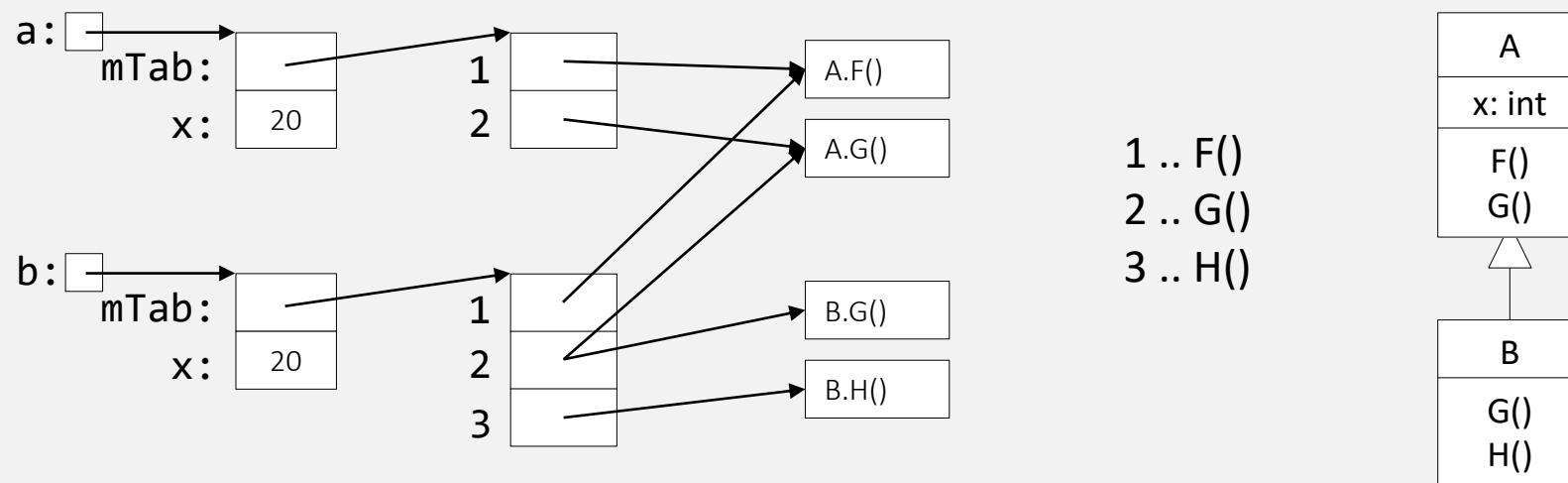
Beispiel:



# Methodentabelle für abgeleitete Klassen

- Für abgeleitete Klasse wird Methodentabelle der Basisklasse kopiert
- Bei hinzugefügten Methoden wird die Methodentabelle erweitert
- Einträge für überschriebenen Methoden werden überschrieben
- Verwendung von Methodentabellen bedeutet weniger Aufwand bei Methodensuche

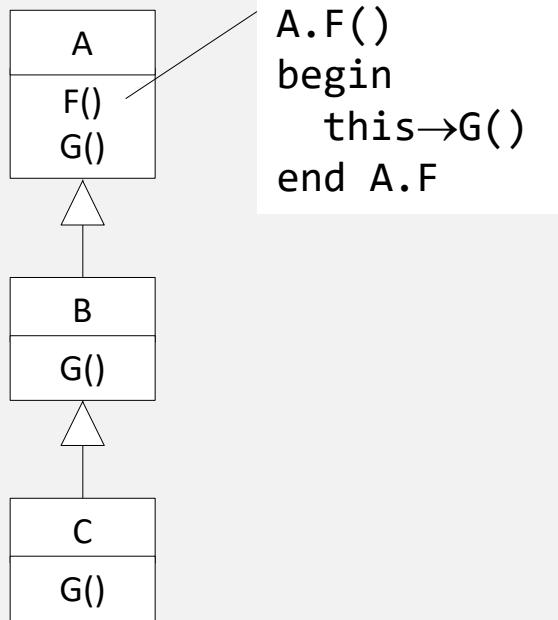
Beispiel:



## 17.4.4 Dynamische Bindung mit this

Nachrichten an this werden dynamisch gebunden

Klassendiagramm:



Aufrufe:

```
var
 b, c: →A
b := New(↓B)
c := New(↓C)
```

```
c→F()
→ A.F()
begin
 this→G()
end A.F
```

```
C.G()
begin ...
end C.G
```

```
b→F()
→ A.F()
begin
 this→G()
end A.F
```

```
B.G()
begin ...
end B.G
```

## 17.5 Abstrakte Klassen

---

- Zur Faktorisierung von Gemeinsamkeiten und Strukturierung ist es sinnvoll, abstrakte Klassen, d.h. Datentypen zu denen es keine konkreten Objekte gibt, zu verwenden
- Für Methoden solcher Klassen existiert u. U. keine sinnvolle Implementierung, Nachrichten sollen aber möglich sein

### Beispiel: Klasse Shape

- hinter shapes[i] verbirgt sich stets ein Objekt einer (konkreten) Klasse, die von der abstrakten Klasse Shape abgeleitet ist

```
var
 shapes: array [1:max] of →Shape
 n: int

for i := 1 to n do
 shapes[i]→Draw()
end -- for
```



# Deklaration und Eigenschaften

---

- Deklaration

```
type
 Shape = abstract class
 abstract Draw()
 end -- Shape
```

Eine Klasse ist abstrakt, wenn Sie mindestens eine abstrakte Methode hat oder mit `abstract` gekennzeichnet ist

Eine Methode ist abstrakt, wenn sie keine Implementierung hat

- Von abstrakten Klassen dürfen keine Objekte erzeugt werden

|                  |   |
|------------------|---|
| var s: →Shape    | ✓ |
| s := New(↓Shape) | ✗ |
| s→Draw()         | ✓ |

- Die Methoden für die Messages abstrakter Klassen werden in den von ihnen abgeleiteten Klassen implementiert (oder abgeleitete Klassen sind selbst abstrakt)
- Es können gemeinsame Schnittstellen für eine ganze Familie von Klassen realisiert werden

## 17.5.1 Realisierungsmöglichkeiten

---

### Sprachunterstützung (z.B. Java und C++)

Java:

```
abstract class Shape {
 abstract void Draw();
} // Shape
```

C++:

```
class Shape {
 void Draw() = 0;
} // Shape
```

### Nachbildung (z.B. Pascal)

```
TYPE
 Shape = OBJECT
 PROCEDURE Draw(); VIRTUAL;
 END; (* Shape *)
```

```
PROCEDURE Shape.Draw()
BEGIN
 WriteLn('Shape.Draw() is abstract');
 Halt(0);
END; (* Shape.Draw *)
```

## 17.5.2 Faktorisierung

---

- Zusammenfassen von Gemeinsamkeiten verwandter Klassen
- Gemeinsamkeiten werden in (abstrakten) Basisklassen definiert

**Beispiel:** Kreise, Rechtecke und Linien

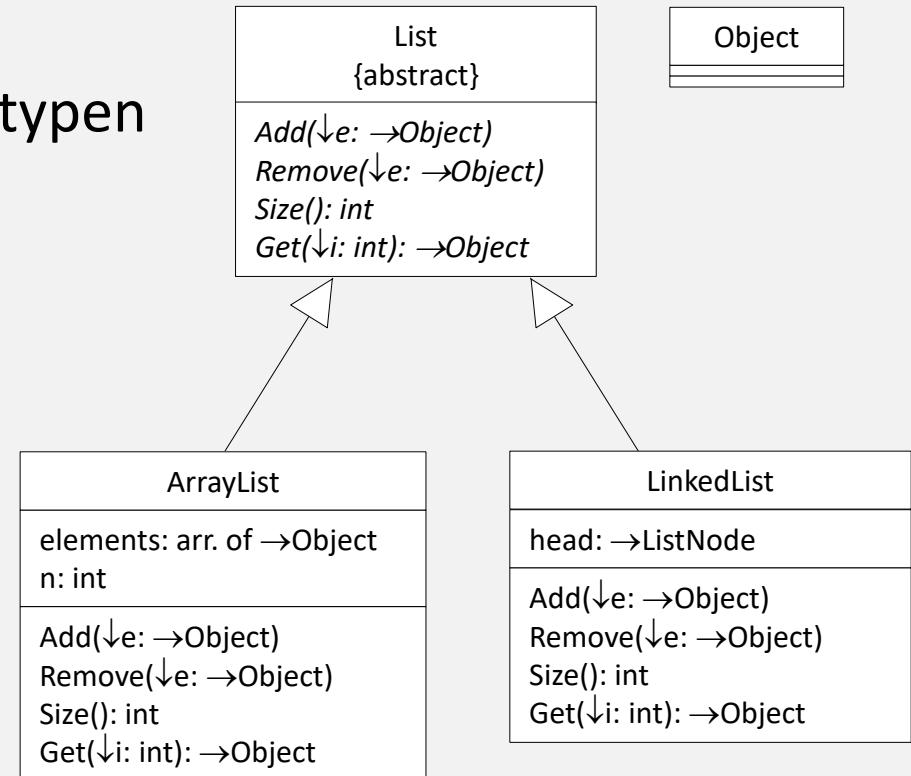
- haben ähnliche oder identische Eigenschaften u. Operationen, z.B.
  - Position und Ausdehnung
  - Zeichnen und Rotieren
  - Abfragen und Ändern von Position und Ausdehnung
- keine Klasse eignet sich als Basisklasse für eine andere
- Lösung:
  - Gemeinsamkeiten werden in abstrakter Basisklasse Shape definiert
  - abstrakte Messages für Operationen, die in Shape nicht (vollständig) implementiert werden können

## 17.5.3 Rein abstrakte Klassen (Schnittstellen)

- Faktorisierung kann zu Klassen führen, die ausschließlich abstrakte Operationen (Messages) definieren, d.h. ihre Schnittstellen haben
  - keine Methodenimplementierungen
  - keine Komponenten
- OO Realisierung von abstrakten Datentypen

**Beispiel:** Liste von Objekten

- gemeinsame Operationen:
  - Hinzufügen, Entfernen, ...
- versch. Implementierungsformen:
  - Feld (array), verkette Liste, ...

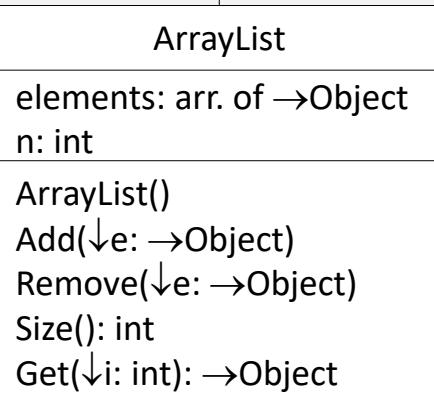
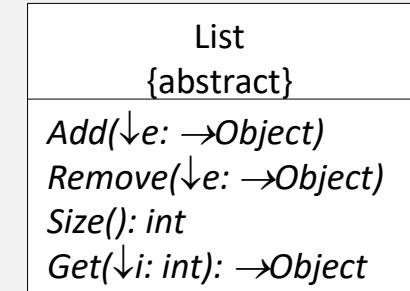


# Beispiel: Liste von Objekten

Deklarationen der (rein abstrakten) Klasse List und der Klasse ArrayList

```
type
 List = abstract class
 public
 abstract Add(↓e: →Object)
 abstract Remove(↓e: →Object)
 abstract Size(): int
 abstract Get(↓i: int): →Object
 end -- List
```

```
type
 ArrayList = class based on List
 private
 elements: array [1:max] of →Object
 n: int
 public
 ArrayList()
 override Add(↓e: →Object)
 override Remove(↓e: →Object)
 override Size(): int
 override Get(↓i: int): →Object
 end -- ArrayList
```



# Beispiel: Liste von Objekten

---

## Konstruktor und Methoden der Klasse ArrayList

```
ArrayList.ArrayList()
begin
 this→n := 0
end ArrayList.ArrayList
```

```
ArrayList.Add(↓e: →Object)
begin
 if n < max then
 n := n + 1
 elements[n] := e
 end -- if
end ArrayList.Add
```

```
ArrayList.Size(): int
begin
 return n
end ArrayList.Size
```

```
ArrayList.Get(↓i: int): →Object
begin
 if 1 ≤ i and i ≤ n then
 return elements[i]
 else
 return null
 end -- if
end ArrayList.Get
```

```
ArrayList.Remove(↓e: →Object)
begin
 -- siehe Vorlesung 1. Semester
end ArrayList.Remove
```

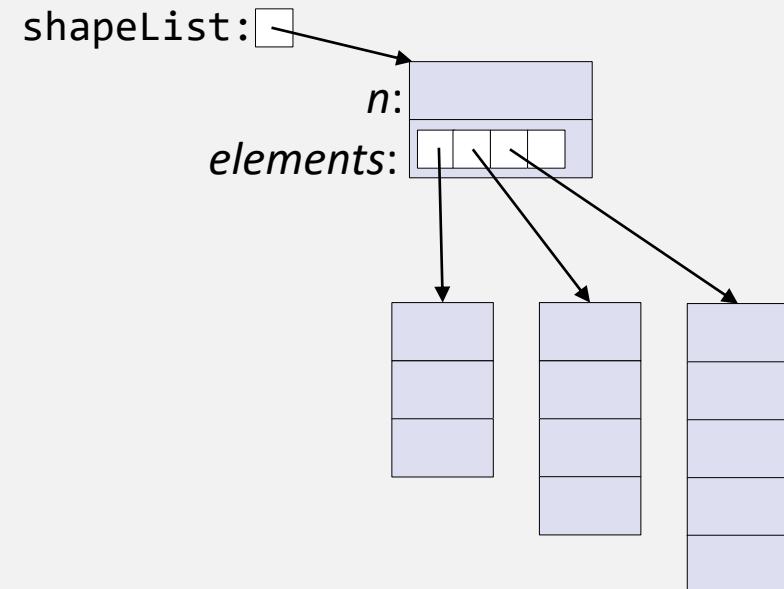
# Beispiel: Liste von Objekten

## Verwendung eines ArrayList-Objekts

```
var
 shapeList: →List

shapeList := New(↓ArrayList)
shapeList→Add(↓New(↓Circle ↓...))
shapeList→Add(↓New(↓Rectangle ↓...))
shapeList→Add(↓New(↓ColoredRectangle ↓...))
DrawShapes(↓shapeList)
```

```
DrawShapes(↓shapeList: →List)
 var obj: →Object; i: int
begin
 for i := 1 to shapeList→Size() do
 obj := shapeList→Get(↓i)
 -- draw obj
 end -- for
end DrawShapes
```



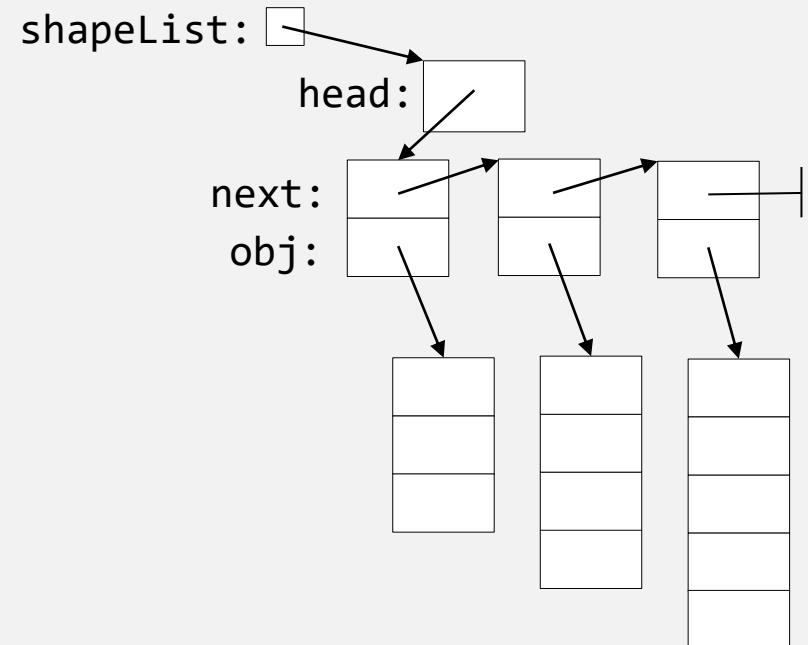
# Beispiel: Liste von Objekten

Verwendung: Austausch des ArrayList-Objekts durch LinkedList-Objekt

```
var
 shapeList: →List

shapeList := New(↓LinkedList)
shapeList→Add(↓New(↓Circle ↓...))
shapeList→Add(↓New(↓Rectangle ↓...))
shapeList→Add(↓New(↓ColoredRectangle ↓...))
DrawShapes(↓shapeList)
```

```
DrawShapes(↓shapeList: →List)
 var obj: →Object; i: int
begin
 for i := 1 to shapeList→Size() do
 obj := shapeList→Get(↓i) ←
 -- draw obj
 end -- for
end DrawShapes
```

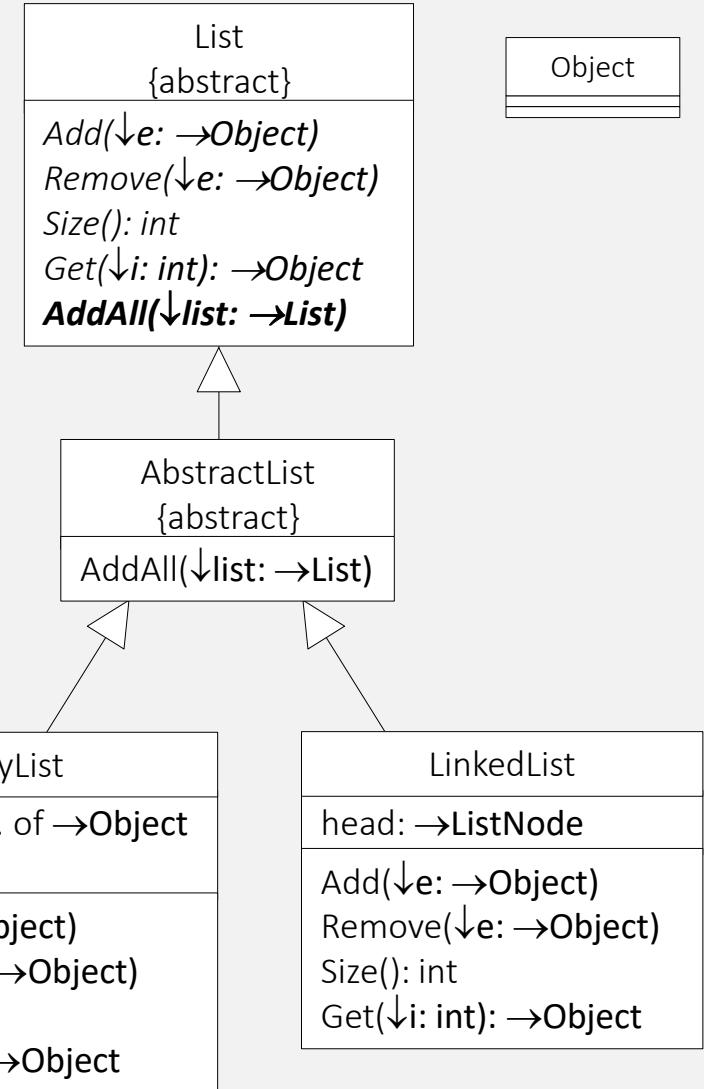


DrawShapes() arbeitet mit ArrayList-Objekten und LinkedList-Objekten

# Beispiel: Liste von Objekten

- Faktorisierung gemeinsamer Methoden in abstrakte Klasse "zwischen" Schnittstelle u. konkreten Klassen
- Beispiel: Methode AddAll()

```
AbstractList.AddAll(↓list: →List)
begin
 n := list→Size()
 for i := 1 to n do
 this→Add(↓list→Get(↓i))
 end -- for
end AbstractList.AddAll
```



- Typische Hierarchie
  - Schnittstelle
  - Abstrakte Basisklasse
  - Konkrete Klassen

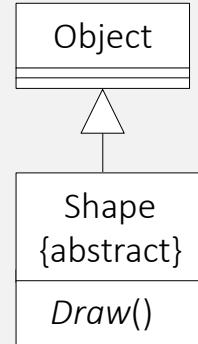
# 17.6 Klassentest und Klassengarantie

- Problem bei Verwendung allgemeiner Objektsammlungen

```
var
 objects: array [1:n] of →Object

for i := 1 to n do
 objects[i]→Draw()
end -- for
```

nicht möglich, wenn in Object  
kein Draw() definiert ist



- Lösung durch Klassentest und Klassengarantie

```
var
 obj: →Object; s: →Shape

 if obj→ isA Shape then
 s := Shape(obj)
 s→Draw()
 end -- if
```

Klassentest

Klassengarantie

- Klassentest prüft, ob Objekt hinter einer Variable vom Typ einer bestimmten Klasse oder einer davon abgeleiteten Klasse ist
- Klassengarantie ermöglicht dem Compiler, die Typkompatibilität zu prüfen

# Klassentest

---

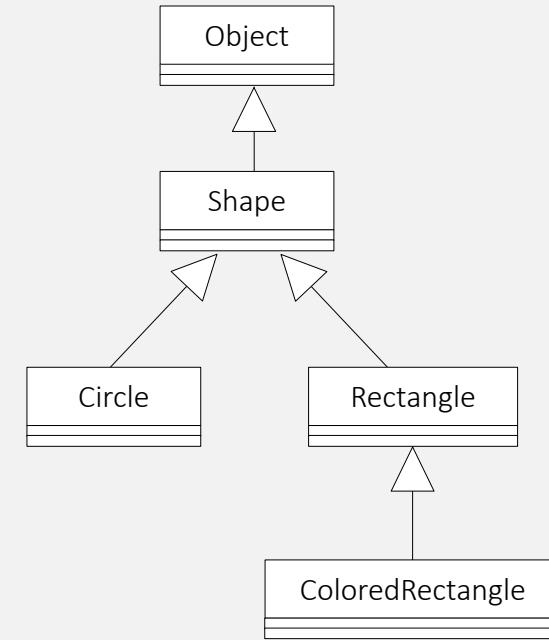
Klassentest prüft, ob Objekt hinter einer Variable vom Typ einer bestimmten Klasse oder einer davon abgeleiteten Klasse ist

`obj → isA ClassType`

liefert true, wenn obj vom dynamischen Typ ClassType (oder einer UnterkLASSE davon) ist

```
var
 obj: →Object
 obj := New(↓Rectangle)
```

|                                         |       |
|-----------------------------------------|-------|
| <code>obj → isA Shape</code>            | true  |
| <code>obj → isA Rectangle</code>        | true  |
| <code>obj → isA ColoredRectangle</code> | false |
| <code>obj → isA Circle</code>           | false |



# Klassengarantie

Klassengarantie ermöglicht dem Compiler, die Typkompatibilität zu prüfen

```
var
 s: →Shape; r: →Rectangle
 s := New(↓Rectangle)
```

statischer Typ: →Shape

```
r := Rectangle(s)
```

statischer Typ: →Rectangle

Wenn der dynamische Typ nicht Rectangle oder eine von Rectangle abgeleitete Klasse ist, wird zur Laufzeit eine Fehlermeldung ausgelöst

## Beispiele

```
r := Rectangle(s)
```

✓

```
c := Circle(s)
```

✗

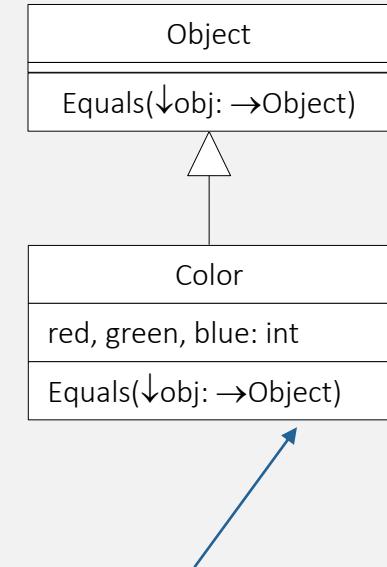
```
cr := ColoredRectangle(s)
```

✗

# Anwendungsbeispiel: Objekte vergleichen

- Message Equals in einer Wurzelklasse Object (gemeinsame Schnittstelle)
- Implementierung in Klasse Object (Objektidentität)

```
Object.Equals(↓obj: →Object): bool
begin
 return (this = obj)
end Object.Equals
```



- Abgeleitete Klassen können Methode überschreiben, z.B. um Color-Objekte zu vergleichen
- Verwendung z.B. in einer Methode Contains

```
ArrayList.Contains(↓obj: →Object): bool
begin
 for i := 1 to n do
 if elements[i]→Equals(↓obj) then
 return true
 end - if
 end -- for
 return false
end ArrayList.Contains
```

Achtung: Parametertyp darf beim Überschreiben der Methode Equals() nicht verändert werden

# Anwendungsbeispiel: Objekte vergleichen

## Überlegungen zur Implementierung der Methode Color.Equals:

```
var
 c1, c2, c3: →Color
 r1: →Rectangle

 c1 := New(↓Color ↓255 ↓0 ↓0)
 c2 := New(↓Color ↓255 ↓0 ↓0)
 c3 := New(↓Color ↓0 ↓255 ↓0)
 r1 := New(↓Rectangle ...)
```

|                  |       |
|------------------|-------|
| c1→Equals(↓c2)   | true  |
| c1→Equals(↓c3)   | false |
| c1→Equals(↓c1)   | true  |
| c1→Equals(↓r1)   | false |
| c1→Equals(↓null) | false |

```
Color.Equals(↓obj: →Object): bool
 var c: →Color
begin
 if obj ≠ null and obj→ isA Color then
 c := Color(obj)
 return red = c→red and green = c→green and blue = c→blue
 else
 return false
 end -- if
end Color.Equals
```

## 17.7 Vererbungsarten

---

### Anzahl der Basisklassen

- keine ... unabhängige Klassen
- eine ... **einfache Vererbung**
- mehrere ... **mehrfache Vererbung**

### Geschützte Vererbung

- Methoden dürfen überschrieben werden
- einige Methoden sind gegen überschreiben geschützt

### Eingeschränkte Vererbung

- Messages/Methoden werden vererbt/geerbt
- einige Messages/Methoden werden nicht vererbt/geerbt

## 17.7.1 Einfache Vererbung

---

### Prinzip

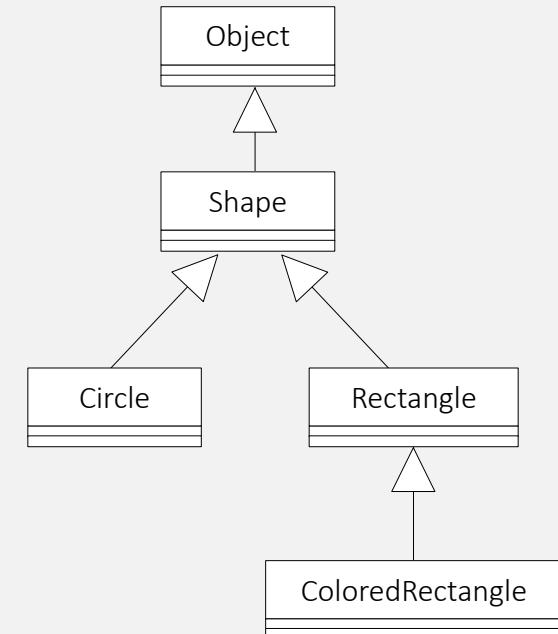
- Wenn von einer Basisklasse B eine Klasse C abgeleitet wird, erbt C alle Eigenschaften und Operationen, die B besitzt

### Anpassungsmöglichkeiten

- neue Datenkomponenten können hinzukommen
- neue Messages/Methoden können hinzukommen
- geerbte Methoden können überschrieben werden

### Konsequenzen

- Klassenhierarchie ist ein Baum
- einfach zu verstehen
- manchmal inadäquate Hierarchie
- Codeverdopplung manchmal unvermeidlich



## 17.7.2 Mehrfache Vererbung

---

In der realen Welt finden wir Artefakte, die Eigenschaften mehrerer anderer Artefakte in sich vereinigen

**Beispiel:** ein Schweizermesser ist

- ein Taschenmesser
- ein Flaschenöffner
- ein Korkenzieher
- eine Schere
- ...

Mehrfache Vererbung ist immer dann angebracht, wenn mit Objekten einer Klasse alle Operationen aus zwei oder mehreren Klassen möglich und sinnvoll sind

# Beispiele für mehrfache Vererbung

---

Ein Window ist ein

- Rectangle (geometrische Eigenschaften)
- TextOutputMedium (Textausgabe)

Ein TextEditor ist ein

- Window (Anzeigemöglichkeiten)
- Text (Inhalt, Manipulationsmöglichkeiten)
- EventHandler (Behandlung von Benutzereingaben)

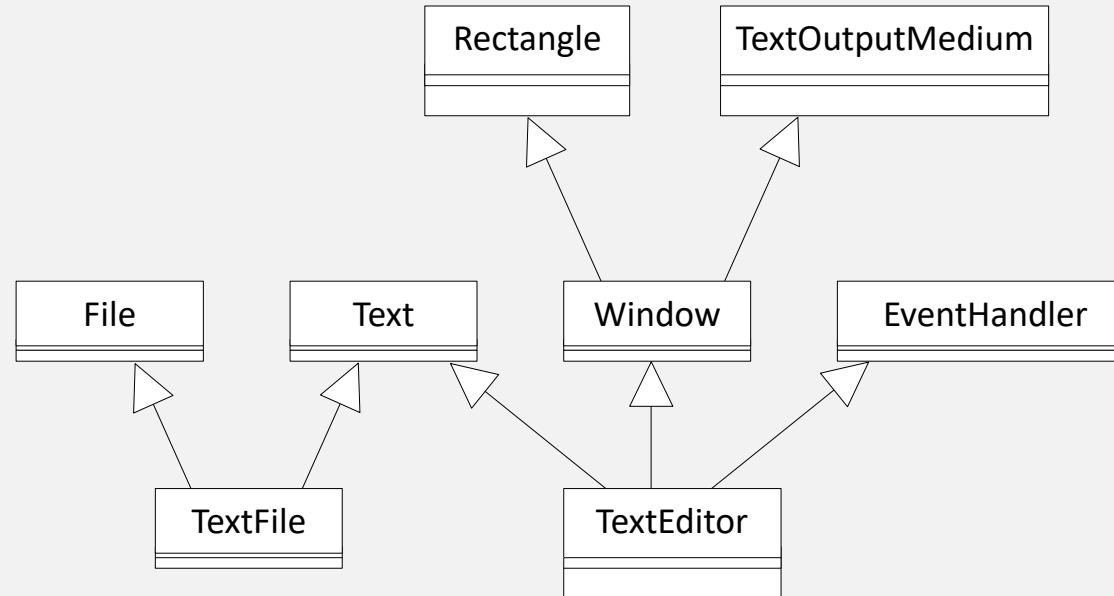
Ein TextFile ist ein

- Text (Inhalt)
- File (Operationen, externe Darstellung)

# Beispiele für mehrfache Vererbung

---

## Klassendiagramm



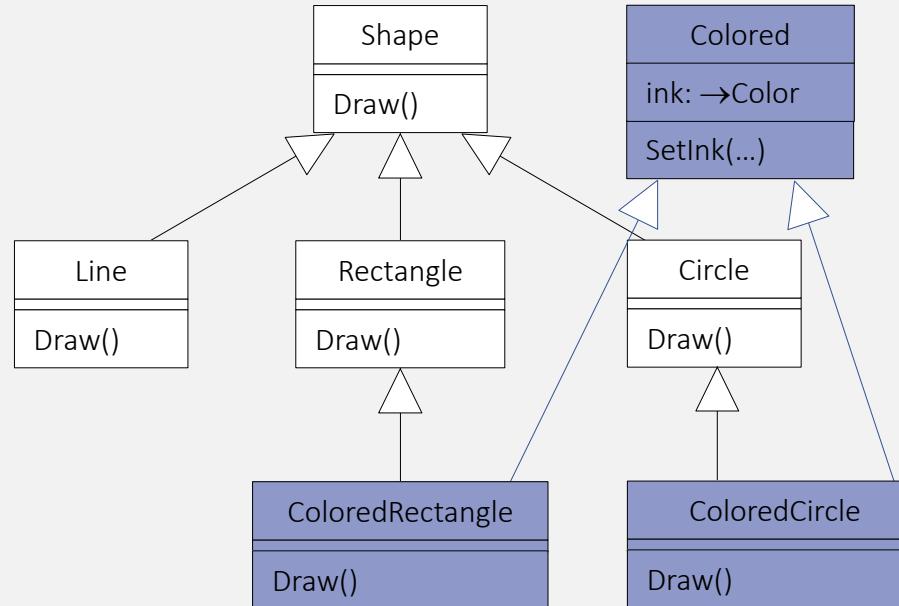
## Konsequenzen

- Vererbungsbeziehung ist ein gerichteter azyklischer Graph
- Realitätsnahe Modellierung möglich
- Namenskonflikte möglich

# Typische Verwendung

---

- Eine Haupthierarchie (z.B. geometrische Form)
- Eigenschaften werden von kleinen isolierten Klassen geerbt (z.B. Colored)
- diese Form wird als Mix-In bezeichnet



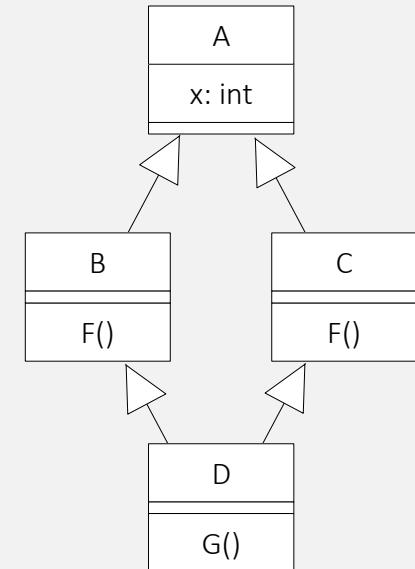
# Konsequenzen mehrfacher Vererbung

---

- Klassenbeziehung ist gerichteter azyklischer Graph
- Klassenbeziehungen sind komplexer und nicht so einfach zu verstehen wie Baum bei einfacher Vererbung
- Komplexe Zusammenhänge
- In der Praxis treten viele (sprachspezifische) Probleme auf

## Beispiel: C++

- in Methode D.G(): super→F() ruft welche Methode?
- für D-Objekte: gibt es x aus A zweimal?
- von A kann nur Default-Konstruktor aufgerufen werden



# Nachbildung mehrfacher Vererbung

---

Klassendefinition:

- Festlegung einer Primär-Basisklasse
- Hinzufügen der Messages der Sekundärklassen

Klassenimplementierung:

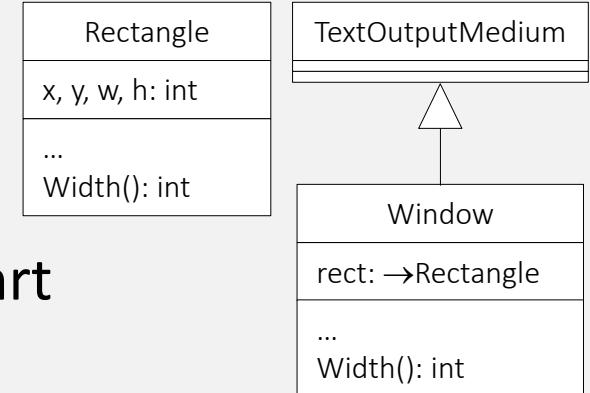
- Erben oder Überschreiben der Methoden der Primär-Basisklasse
- Kopieren der Datenkomponenten und Methoden der Sekundärklassen  
oder:
- Hinzufügen von Objekten der Sekundärklassen und Weiterreichen der Messages

# Nachbildung mehrfacher Vererbung

## Beispiel:

- Window wird von TextOutputMedium abgeleitet
- Statt einer weiteren Basisklasse Rectangle wird eine Komponente rect: →Rectangle eingeführt

```
Window.Window()
begin
 rect := New(↓Rectangle ...)
end Window.Window
```



- die Rectangle-Messages werden zusätzlich definiert und durch Weiterleiten an das Objekt rect implementiert (evtl. mit zusätzlichen Aktionen)

```
Window.Width(): int
begin
 return rect→Width()
end Window.Width
```

## 17.7.3 Geschützte Vererbung

---

Manche Messages müssen systemweit gleiche Bedeutung haben

Die entsprechenden Methoden werden an einer Stelle implementiert und dürfen aus Sicherheitsgründen nicht überschrieben werden

### Beispiel:

- Die Methode `Clone` ist in der Klasse `Object` definiert und erzeugt eine "flache" Kopie des Empfängerobjekts (d.h. mit diesem Objekt verbundene dynamische Datenstrukturen werden nicht mitkopiert):
- `Clone` darf nicht überschrieben werden, weil z.B. alle anderen Kopierverfahren darauf aufbauen
- Sprachunterstützung nützlich, um schwer zu lokalisierende Fehler wegen ungenügenden Systemverständnisses schon bei der Übersetzung zu erkennen

```
type
 Object = class
 final Clone(): →Object
 end -- Object
```

## 17.7.4 Eingeschränkte Vererbung

---

Manche Operationen sind nur in einer Klasse sinnvoll, nicht aber in davon abgeleiteten Klassen

Zwei Möglichkeiten:

- Einschränkung in der Basisklasse (ausschließen von M. von der Vererbung)
- Einschränkung in der abgeleiteten Klasse (ausschließen von Messages von der Ererbung)

Probleme:

- Zukünftige abgeleitete Klassen sind in Basisklassen kaum vorhersehbar
- Polymorphismus wird gestört

Beispiel:

- Rectangle dient als Basisklasse für Square (Spezialisierung)
- SetWidth und SetHeight sind nicht mehr sinnvoll;  
statt dessen neue Message SetSize

## 17.8 Klassenbibliotheken

---

- Darunter verstehen wir eine Menge von Klassen, zusammen mit ihrem Beziehungsgeflecht (ein Typsystem → Architekturdesign)
- Bei einfacher Vererbung durch eine oder mehrere Klassenhierarchien repräsentiert
- Bei mehrfacher Vererbung durch einen oder mehrere azyklische, gerichtete Graphen repräsentiert
- Einteilung von Klassen nach ihrem Verwendungszweck

## 17.8.1 Bedeutung von Klassenbibliotheken

---

- Sie bildet eine wichtige Basis für den Software-Entwicklungsprozess bei OOP
- Die Klassenbibliothek erweitert quasi die Programmiersprache
- Sprache tritt u.U. hinter die Klassenbibliothek zurück
- Die Klassenbibliothek repräsentiert nicht nur eine Menge von (abstrakten) Datentypen, sondern auch „Design“
- Ihr Umfang und ihre Ausprägung beeinflusst sowohl die Produktivität des Entwicklungsprozesses, als auch die Qualität des zu entwickelnden Systems

## 17.8.2 Umfang von Klassenbibliotheken

---

### Anzahl der Klassen

- Klein                       etwa 20 bis 100 Klassen
- Mittel                      etwa 100 bis 500 Klassen
- Groß                       mehr als 500 Klassen (heute ca. 5.000)

Je umfangreicher eine Klassenbibliothek, desto

- größer die Wahrscheinlichkeit, eine passende Klasse zu finden
- weniger eigener Code muss erzeugt werden
- geringer ist die Wahrscheinlichkeit von Programmfehlern
- größer ist der Einarbeitungs- und Lernaufwand

## 17.8.3 Einteilung von Klassen

---

Bibliotheken sind i.d.R. in Teile gegliedert, die bestimmte Aufgabengebiete abdecken

### (1) Standardklassen

- Gemeinsame Wurzelklasse Object mit gemeinsamer Schnittstelle für z.B.
  - Vergleich von Objekten (z.B. if  $c1 \rightarrow \text{Equals}(\downarrow c2)$  then...)
  - Berechnung eines Hash-Codes (z.B.  $h := \text{obj} \rightarrow \text{HashCode}()$ )
- Datentypen für elementare Datenobjekte wie Zahlen, Zeichen und Zeichenketten
- Objekte dieser Klassen "umhüllen" oft Werte von Standardtypen (wrapper)
- Datentypen für elementare strukturierte Datenobjekte wie Felder, etc.

## (2) Betriebssystemklassen

---

- Bilden Funktionen der Hardware und des Betriebssystems mit Objekten nach
- Kapseln Details des Betriebssystems und verbessern so die Portabilität darauf aufbauender Programme

### Beispiele

- Prozesse: Process, Semaphore, Thread
- Netze: Socket, InetAdress
- Dateien: File, Directory, InputStream

## (3) Klassen für Datenstrukturen (Container)

---

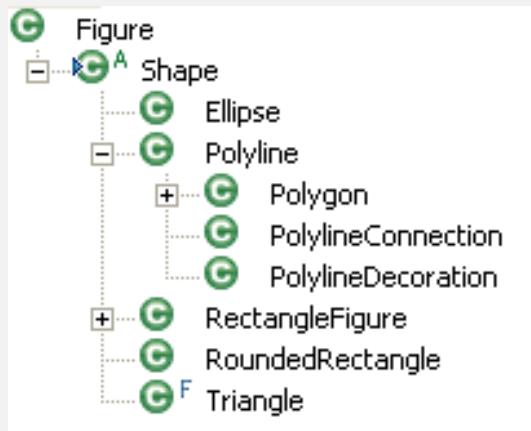
- Klassen für Objekte zur Aufnahme anderer Objekte (Behälterklassen, *container* oder *collection classes*)
- Typische Operationen: Einfügen, Löschen, Suchen und Sortieren
- Beispiele
  - Vector oder List
  - Set
  - Dictionary oder Map
- Behälterklassen setzen gemeinsame Wurzelklasse (Object) aller Klassen voraus

# (4) Klassen für grafische Objekte

---

Dienen der Bereitstellung von Eigenschaften und Operationen elementarer und spezieller Grafikobjekte wie Linien, geom. Figuren, etc.

**Beispiel:**

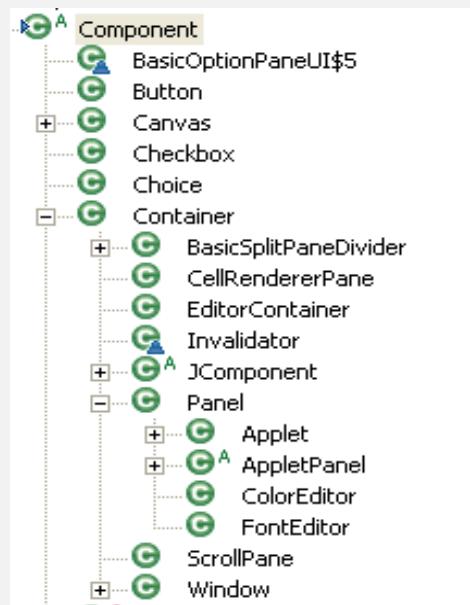


Eclipse Draw2D

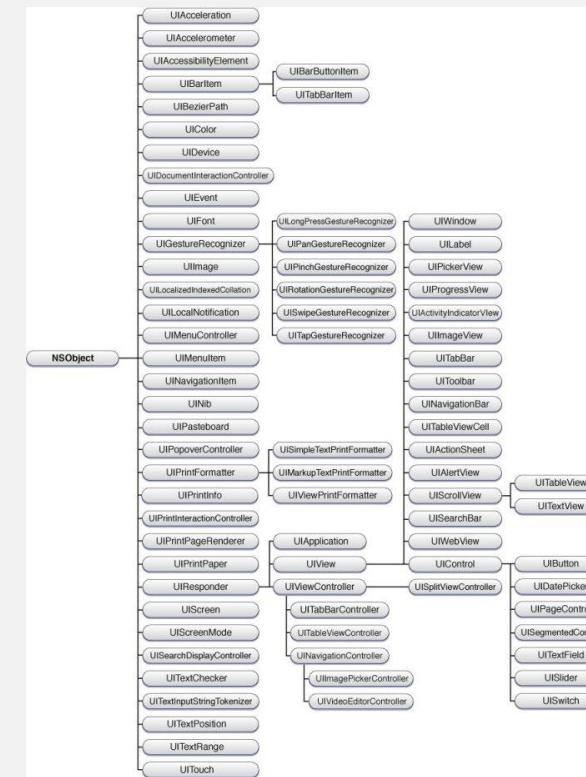
# (5) Benutzerschnittstellenklassen

Dienen der Bereitstellung von Objekten zur Konstruktion der Benutzungsschnittstelle interaktiver Programmsysteme  
*(graphical user interface, GUI)*

Beispiele:



Java AWT



UI Kit iOS

# (6) Anwendungsspezifische Klassen

---

Klassen zur Lösung von Problemstellungen bestimmter Aufgabengebiete (Domänen)

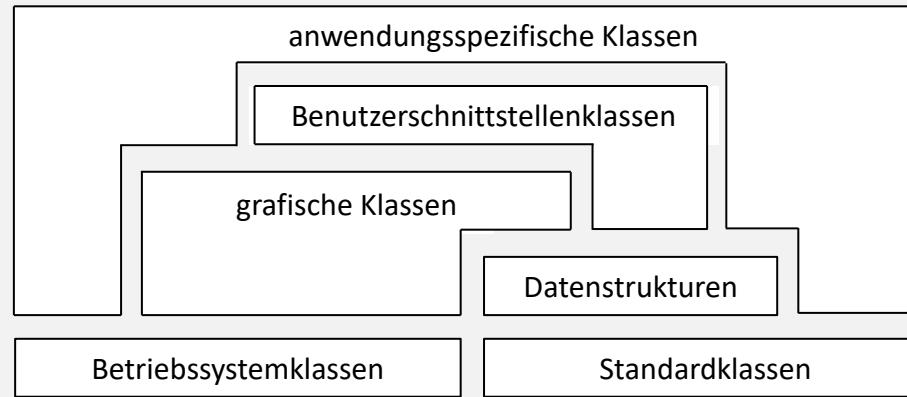
## Beispiele:

- Verarbeitung von XML-Dokumenten
- Einsatz relationaler Datenbanken
- Statistische Auswertungen
- Simulation
- Betriebswirtschaftliche Problemstellungen
- Prozessautomatisierung
- ...

## 17.8.4 Gliederung von Klassenbibliotheken

---

### Gliederung von Klassen nach Verwendungszweck



- Abhängigkeiten zwischen Teilen
- Kaum Teile einer Klassenbibliothek für sich verwendbar
- Verschiedene Klassenbibliotheken können nicht zu größeren vereinigt werden

## 17.8.5 Application Frameworks (AF)

---

Framework = Sammlung von Klassen für ein bestimmtes Anwendungsgebiet

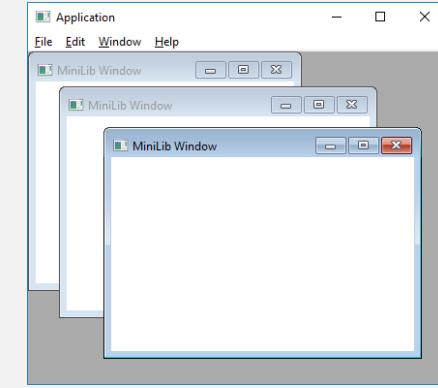
- **Application Frameworks** sind eine spezielle Ausprägung von Klassenbibliotheken, die typischerweise eine Klasse mit der Bezeichnung Application enthalten,
- Über die quasi eine „leere“, aber bereits ausführbare Anwendung zur Verfügung gestellt wird,
- Die vom Entwickler problemspezifisch an seine speziellen Bedürfnisse angepasst und darüber hinaus auch mit neuen Klassen (abgeleitet von der Klasse Application und anderen Klassen des AFs wie Window) angereichert werden kann.
- Bei Programmentwicklung auf der Basis von AFs geht man von außen nach innen vor (Outside-In-Appraoch)

# Typisches Programm unter Verwendung eines AFs

## "Leere" Applikation

```
var
 app: →Application

 app := New(↓Application)
 app→Run()
 app→Done()
 Dispose(↓app)
```



Ergänzung um anwendungsspezifische Teile durch Ableiten neuer Klassen und Überschreiben bestimmter Methoden

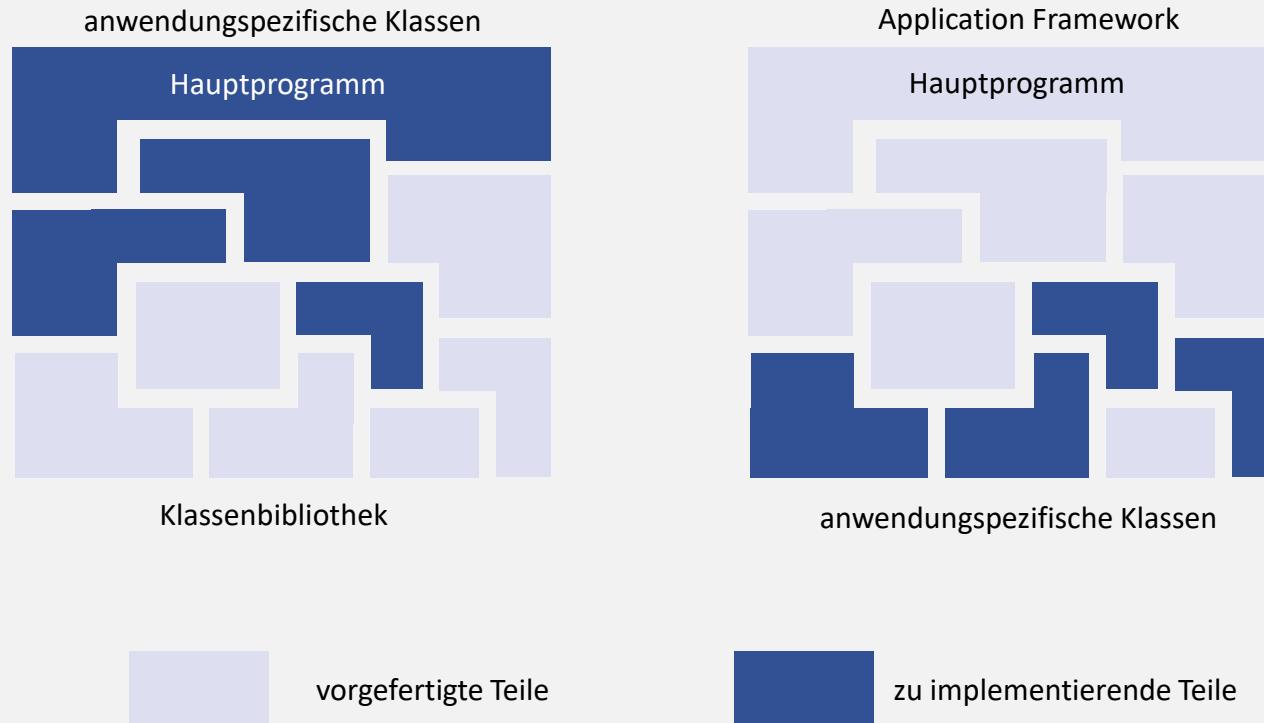
```
var
 app: →Application
 app := New(↓MiniExcelApplication)
 app→Run()
 app→Done()
 Dispose(↓app)
```

A screenshot of a Windows application window titled "MiniExcel". The main area displays a grid of numbers representing a multiplication table, starting from 1 and extending up to 12. The columns are labeled A through L, and the rows are labeled 1 through 13. The grid shows the product of each row and column index, such as 1x1=1, 1x12=12, 12x12=144, etc.

# Klassenbibliotheken / Application Frameworks

---

- Klassenbibliothek ist Fundament für Programmentwicklung
- Application Framework ist bereits lauffähiges Programm



## 17.8.6 Beispiele

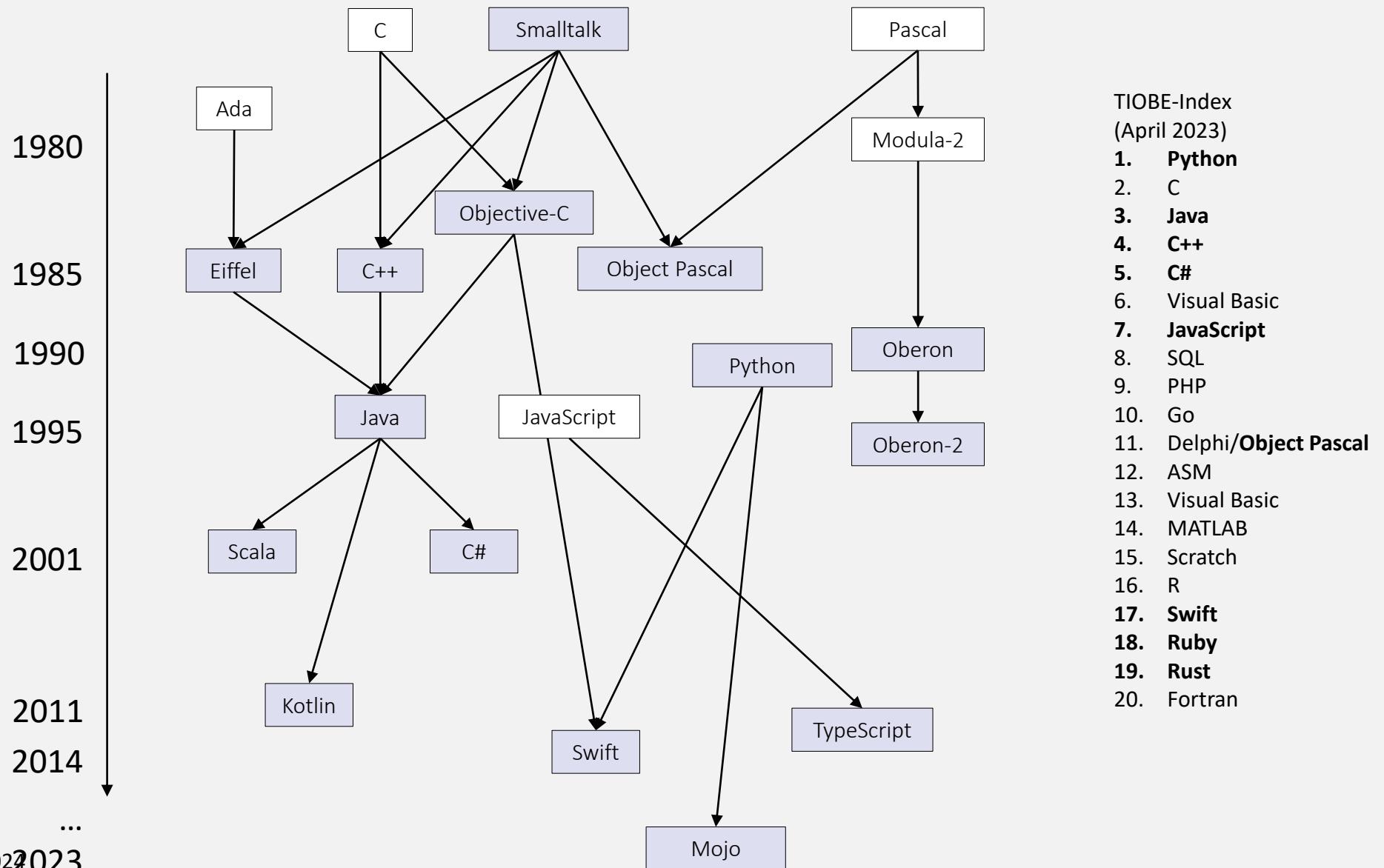
---

### Standardbibliothek ([Klassenbibliothek](#)) für Programmiersprache Java

| Version | Veröff. | Klassen | Methoden |
|---------|---------|---------|----------|
| 1.0     | 1996    | 212     | 1.545    |
| 1.1     | 1997    | 504     | 3.851    |
| 1.2     | 1998    | 1.520   | 15.060   |
| 1.3     | 2000    | 1.842   |          |
| 1.4     | 2002    | 2.991   |          |
| ...     |         |         |          |
| 1.7     | 2011    | 4.024   |          |
| 1.8     | 2014    | 4.240   |          |

Quelle: Robert Liguori u. Patricia Liguori: Java 8 Pocket Guide, O'Reilly Media, 2014.

# 17.9. Objektorientierte Programmiersprachen



# Charakteristika OO Programmiersprachen

---

| Sprache       | Eigenschaften                                                                                                                              |
|---------------|--------------------------------------------------------------------------------------------------------------------------------------------|
| Smalltalk     | einfache Syntax, rein objektorientiert, auch elementare Daten werden als Objekte aufgefasst                                                |
| Object Pascal | kleine Erweiterung von Pascal um die wichtigsten objektorientierten Sprachelemente                                                         |
| C++           | Weiterentwicklung von C, immer wieder erweitert, statische und dynamische Objekte, überladene Operatoren, Generizität, mehrfache Vererbung |
| Eiffel        | rein objektorientiert, mehrfache Vererbung, Zusicherungen, Vor- und Nachbedingungen, Ausnahmebehandlung                                    |
| Oberon        | Nachfolger von Modula-2, Typerweiterung als Mechanismus zur Implementierung von Vererbung                                                  |
| Oberon-2      | Erweiterung von Oberon um typgebundene Prozeduren                                                                                          |
| Self          | rein objektorientiert, Prototypen statt Klassen, Delegation statt Vererbung                                                                |
| Java          | Vereinfachung von C++, für verteilte Programme (z.B. im Internet) geeignet, Sicherheitsmechanismen                                         |
| C#            | Kopie von Java, einheitliches Typsystem, Attribute, Versionierung, statische Objekte                                                       |

# 17.10 Weitere Konzepte und Konstrukte der OOP

---

1. Generische Klassen
2. Persistente Objekte
3. Prototypen
4. Radikale Objektorientiertheit

## 17.10.1 Generische Klassen

---

Sollen die Möglichkeit bieten, einen Behälter (Container) so einzuschränken, dass er nur Objekte einer bestimmten Klasse (bzw. von davon abgeleiteten Klassen) aufnehmen kann

Lösungsansatz: parametrisierte Klassen (Klassenschablonen)

### Beispiel

- Element ist Platzhalter für konkreten Typ
- Mit Variablen/Parameter vom Typ Element ist nur Zuweisung erlaubt

```
type
 Stack<Element> = class
 data: array [1:max] of Element
 top: int
 Push(↓e: Element)
 Pop(): Element
 end -- Stack
```

```
Stack<Element>.Push(↓e: Element)
begin
 top := top + 1
 data[top] := e
end Stack<Element>.Push
```

# Verwendung

---

## Parametrisierung mit konkretem Elementtyp

```
var
 stack: Stack<→Shape> ← erzeugt neuen Typ
 s: →Shape
 r: →Rectangle
...
stack.Push(↓r) ← Zuweisung ohne Typzusicherung
s := stack.Pop()
s→Draw()
stack.Push(↓New(↓Person)) ← nicht möglich
```

Ohne Parametrisierung kann Klassenschablone nicht verwendet werden

```
var
 stack: Stack ← nicht möglich
```

# Wirkung

---

## Deklaration

```
var
 stack: Stack<→Shape>
```

entspricht

```
var
 stack: ShapeStack
```

mit der Definition

```
type
 ShapeStack = class
 data: array[1:max] of →Shape
 top: int
 Push(↓e: →Shape)
 Pop(): →Shape
 end -- ShapeStack
```

Durch die Parametrisierung werden – abstrakt betrachtet – mehrere Stack-Typen erzeugt

```
Stack<→Shape> ≠ Stack<→Rectangle> ≠ Stack<→Person>
```

Es reicht jedoch genau eine Implementierung

## 17.10.2 Persistente Objekte

---

Objekte werden dynamisch angelegt und zu verketteten Strukturen verbunden

Man braucht oft ein Mittel, einzelne Objekte oder ganze Objektstrukturen zur späteren Verwendung aufzubewahren (einzufrieren)

Möglichkeiten:

1. Aktivierung / Passivierung

Ein Objekt kann mit Hilfe einer Message "passiviert", d.h. in einem Archiv abgelegt werden. Die spätere "Aktivierung" (das Abholen aus dem Archiv) geschieht mit einer Message an die Klasse.

2. Persistenz-Deklaration

Objekte können bei ihrer Deklaration als "persistent" (= beständig) markiert werden und werden damit automatisch bei Bedarf aus dem Archiv geholt und bei Beendigung eines Programmes wieder dort abgelegt.

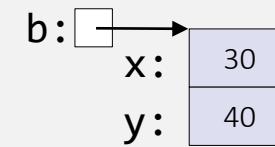
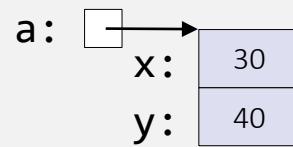
Ungelöstes Problem : Konsistenz der externen Aufzeichnungen bei Änderungen des Typsystems (und damit der Objektstruktur).

## 17.10.3 Prototypen

---

- Kein Klassenkonstrukt, statt dessen „Prototypen“
- Definition konkreter Objekte (interaktiv) mit Datenkomponenten und Methoden
- Neue Objekte werden durch Kopieren bestehender Objekte erzeugt

```
b := a→Copy()
```



- Häufig Delegation statt Vererbung
- Prototypbasierte Sprachen einfacher und flexibler als klassenbasierte Sprachen

## 17.10.4 Radikale Objektorientiertheit

---

Es gibt nur mehr Objekte und sonst nichts  
(alle Datenobjekte sind Objekte im OO Sinne)

Eliminierung von Variablen „konventioneller“ Datentypen

Vorteile

- Erweiterbarkeit der Standarddatentypen
- Gewinn an Allgemeinheit

Nachteile

- Effizienzverlust
- Operatoren werden als Nachrichten aufgefasst

$2+x$

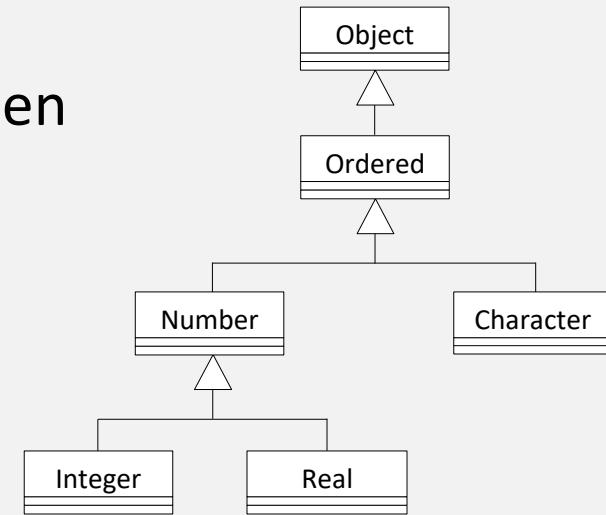
... Nachricht + an Objekt 2 mit Parameter x

$x+2$

... Nachricht + an Objekt x mit Parameter 2

$2+3*4$

... Auswertung von links nach rechts



# Anweisungen sind Objekte

---

Anweisungen können wie Objekte zugewiesen werden

## Vergleich

### Smalltalk

```
x>limit ifTrue: [x:=limit]
```

```
x<y
 ifTrue: [min:=x. max:=y]
 ifFalse:[min:=y. max:=x]
```

```
[x>0] whileTrue:
 [y := y*2. x := x-1]
```

### Pascal

```
IF x > limit THEN x := limit
```

```
IF x<y
 THEN BEGIN min:=x; max:=y END
 ELSE BEGIN min:=y; max:=x END
```

```
WHILE x>0 DO BEGIN
 y := y*2; x := x-1
END
```

# 17.11 Vor- und Nachteile der OOP

---

Objektorientierte Programmierung ist heute eine der wichtigsten Programmiertechniken

## Vorteile

- Vererbung, Klassenbibliotheken und Application Frameworks unterstützen die Wiederverwendung von bereits bestehenden Programmen und Programmkomponenten
- durch Wiederverwendung von Code werden Duplikate vermieden
- Polymorphie und dynamische Bindung helfen, umfangreiche Fallunterscheidungen zu vermeiden
- Abstrakte Klassen fördern die Schaffung von einheitlichen Schnittstellen und die Strukturierung von Softwarearchitekturen
- der Einsatz von Application Frameworks führt zu Systemen mit ähnlichen Benutzerschnittstellen und fördert so die Einheitlichkeit des Erscheinungsbildes von Programmen (Standardisierung)

# Vor- und Nachteile der OOP

---

## Nachteile

- Neben der Beherrschung einer objektorientierten Sprache, sind Kenntnisse des Aufbaus von Klassenbibliotheken u/o Application Frameworks erforderlich
- Dynamisch erzeugte Objekte erhöhen den Speicherbedarf
- Dynamische Bindung kann die Programmlaufzeit verlängern

---

# Algorithmen und Datenstrukturen

## Eine systematische Einführung in die

## Programmierung

## 17 Objektorientierter Systementwurf - Teil 4

Josef Pichler

Version 10.1, 2024

# 17.12 Grundlegendes zum objektorientierten Systementwurf

---

- Der objektorientierte Entwurf erfordert eine andere Denkweise als der aufgabenorientierte Entwurf
- Die Daten zusammen mit den darauf auszuführenden Operationen (Objekte) stehen im Mittelpunkt des Interesses
- Daher werden anstatt der Algorithmen zuerst die Abstraktionen von Objekten (Klassen) und die zwischen ihnen bestehenden Beziehungen modelliert

Typische Vorgehensweise:

1. Auffinden von geeigneten Klassen zur Lösung der Aufgabe
2. Festlegen, welche Operationen mit einzelnen Objekten möglich sein sollen
3. Festlegen, in welcher Beziehung diese Klassen und ihre Objekte zueinander stehen

# Grundlegendes zum objektorientierten Systementwurf

---

Aber wie?

- Entwerfen ist ein kreativer Prozess (analytische/intuitive/kritische Phase)
- ein guter Systementwurf erfordert eine gründliche Problemanalyse
- es gibt keine allgemein anerkannte Methode für guten Systementwurf

Es gibt **Richtlinien**

- für die Identifikation von Klassen, Datenkomponenten und Methoden
- für die Gestaltung von Klassenschnittstellen und Klassenbeziehungen

Es gibt **Entwurfsmuster**

- für die Gestaltung von Systemarchitekturen und guten Programmstrukturen

# Grundlegendes zum objektorientierten Systementwurf

---

In der Literatur findet man zahlreiche, oft sehr unterschiedliche Ansätze für den objektorientierten Entwurf

## Entwurfsmethode

- Eine **Methode** für den Entwurf objektorientierter Systeme besteht i.d.R. aus einem **systematischen Ansatz/Verfahren** für den Entwurfsprozess und einer **Notation**

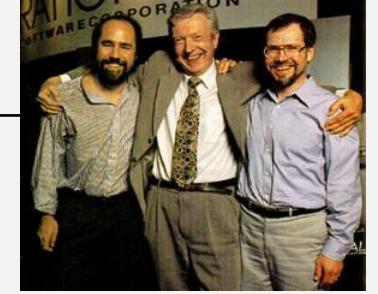
## Notation

- Als allgemein akzeptierte **Notation** für den Entwurf von objektorientierten Systemen hat sich **UML (Unified Modeling Language)** durchgesetzt



[https://de.wikipedia.org/wiki/Unified\\_Modeling\\_Language](https://de.wikipedia.org/wiki/Unified_Modeling_Language)

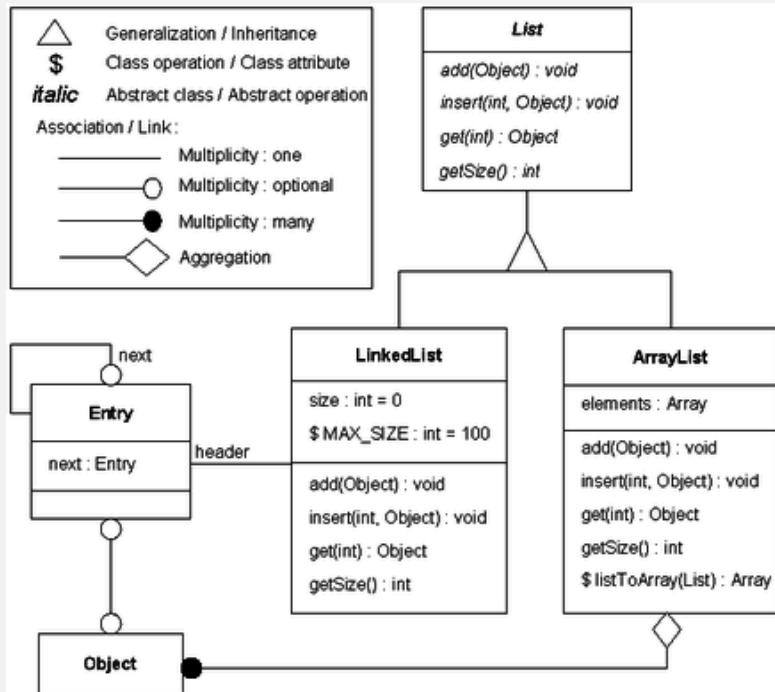
# 17.13 Unified Modeling Language (UML) – grober Überblick



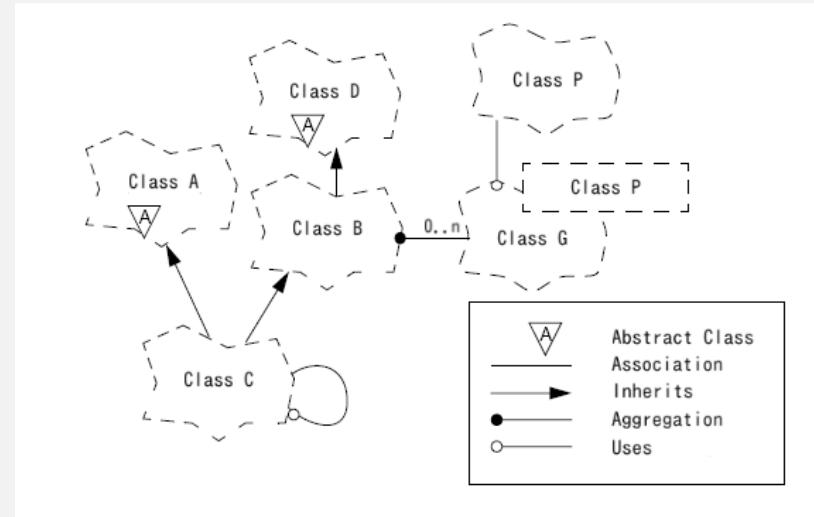
UML ist eine 1997 **standardisierte** Notation, die aus OMT (James Rumbaugh), OOSE (Ivar Jacobson) und den Methoden von Booch entstanden ist (die drei Amigos)

die drei Amigos

**Beispiele:** Klassen- und Objektdiagramme



Object-Modeling Technique (OMT)



Booch method

Bildquellen:

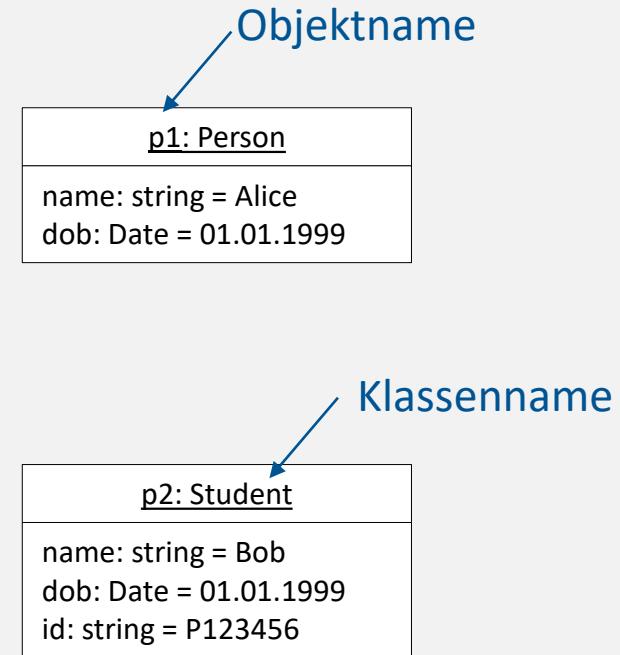
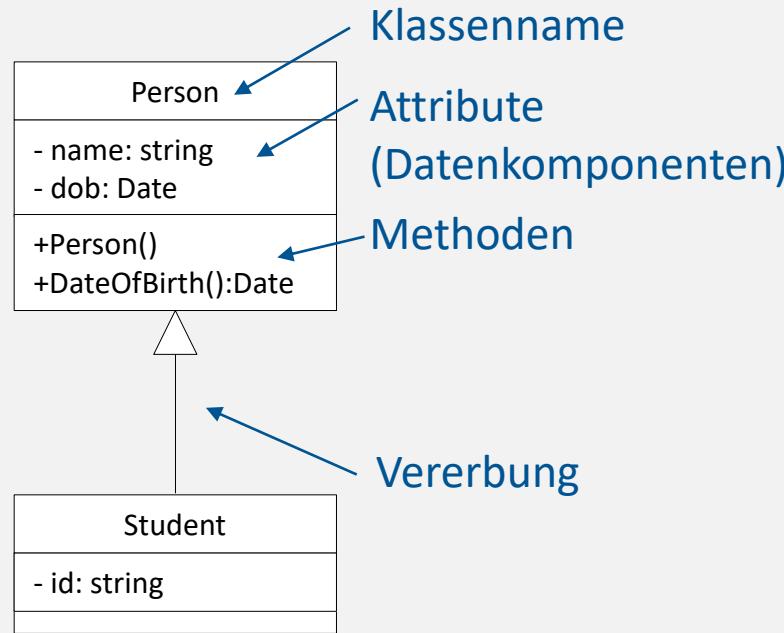
[https://de.wikipedia.org/wiki/Object-Modeling\\_Technique](https://de.wikipedia.org/wiki/Object-Modeling_Technique)

[https://en.wikipedia.org/wiki/Booch\\_method#/media/File:Booch-diagram.png](https://en.wikipedia.org/wiki/Booch_method#/media/File:Booch-diagram.png)

<https://www.oose.de/blogpost/20-jahre-uml>

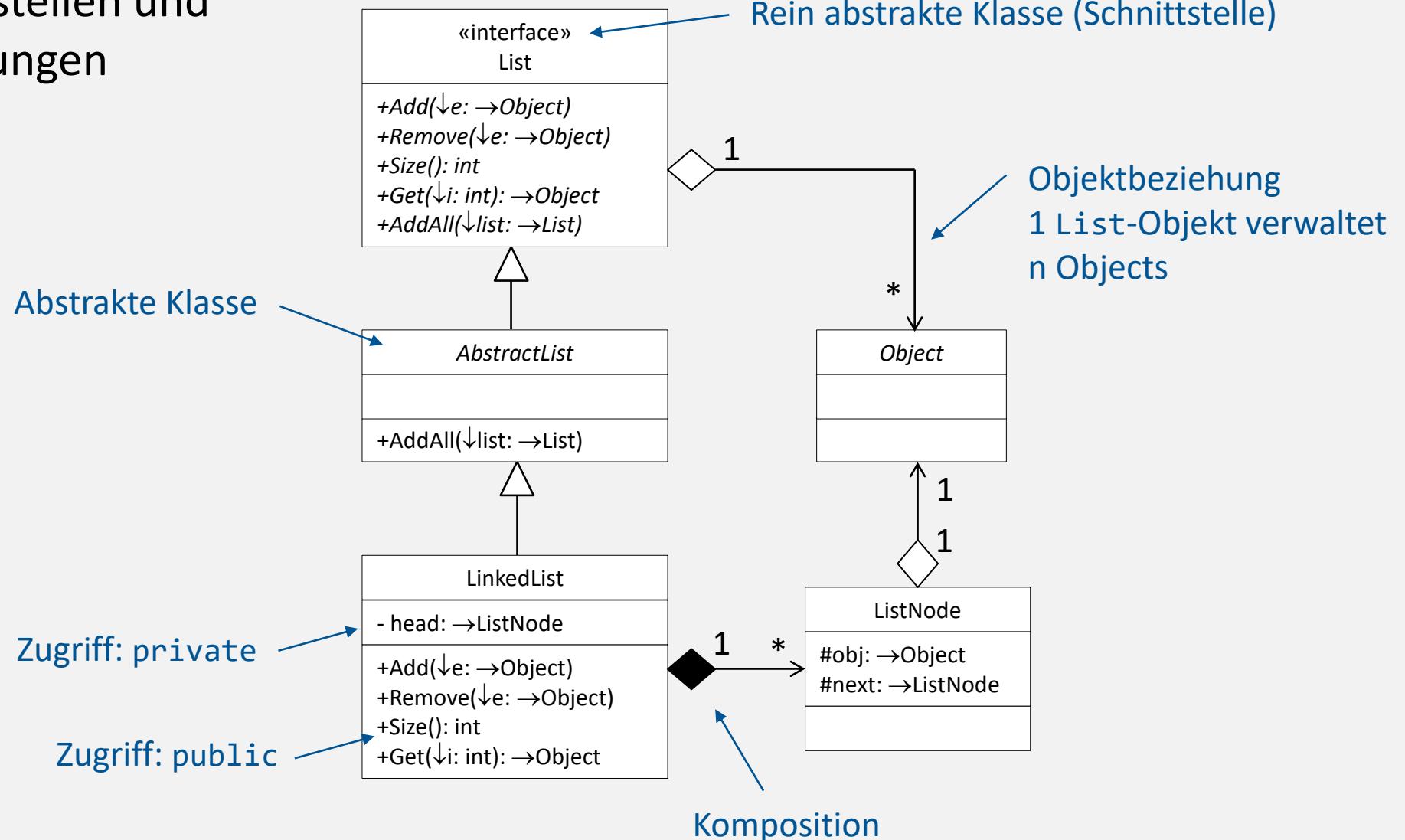
# Unified Modeling Language (UML) – grober Überblick

## Klassen- und Objektdiagramme



# Unified Modeling Language (UML) – grober Überblick

## Schnittstellen und Beziehungen



# Unified Modeling Language (UML) – grober Überblick

## Sequenzdiagramm

```
var a: →A
```

```
A.F()
begin
 b→F(↓this)
end A.F
```

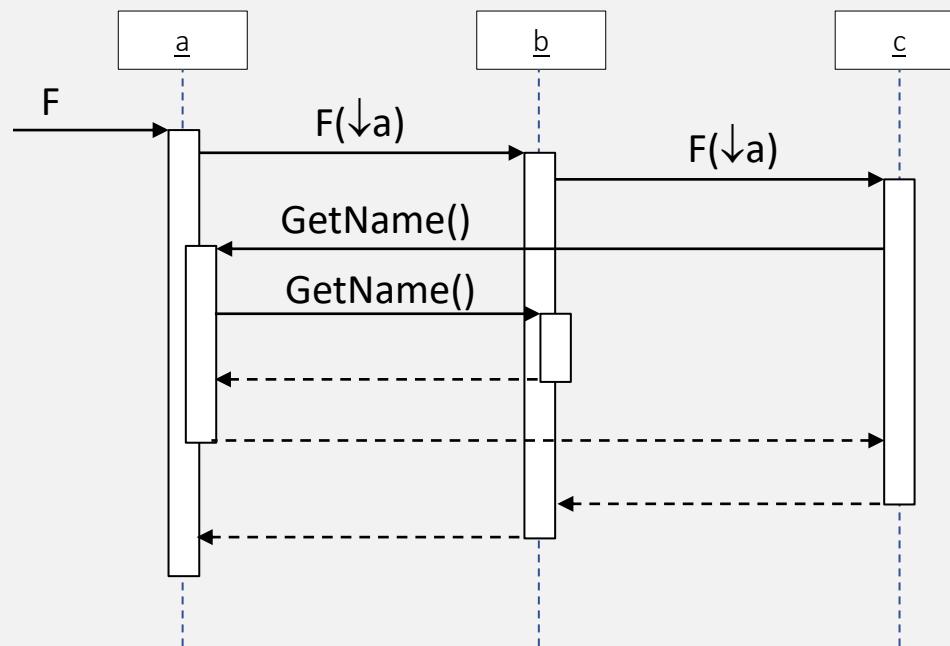
```
A.GetName(): string
begin
 return "A" + b→GetName()
end A.GetName
```

```
B.F(↓a: →A)
begin
 c→F(↓a)
end B.F
```

```
B.GetName(): string
begin
 return ...
end B.GetName
```

```
C.F(↓a: →A)
begin
 WriteLn(↓a→GetName())
end C.F
```

|   |                              |
|---|------------------------------|
| A | - b: B                       |
|   | + F()<br>+ GetName(): string |
| C | + F(↓a: A)                   |



## 17.14 Ein Basismodell für den OO Systementwurf

---

- Wir vertreten die Auffassung, dass die meisten der empfohlenen Methoden kaum mehr enthalten, als der gesunde Hausverstand vorgibt
- Die vorgeschlagenen Verfahren erwecken oft den Eindruck, dass, wenn man nur die richtige Notation benutzt, sich alles andere von selbst findet
- Der Entwurf großer Softwaresysteme ist und bleibt aber eine schwierige Aufgabe – auch wenn objektorientierte Techniken eingesetzt werden
- Wir können kein rundum befriedigendes Verfahren angeben und benutzen stattdessen ein **elementares Verfahren** zur Auffindung der für die Lösung **benötigten Klassen, Datenkomponenten und Methoden**, das an die Softwareentwurfsmethode von Abbott angelehnt ist

# Ein Basismodell für den OO Systementwurf

---

## Vorgehen (in Anlehnung an Abbott)

1. Verbale Spezifikation der Aufgabe aufstellen (besser noch ein Prototyp)
2. Klassen, Methoden und Attribute (Datenkomponenten) identifizieren
  - 2.1 Hauptwörter sind mögliche Kandidaten für Klassen
  - 2.2 Zeitwörter sind mögliche Kandidaten für Methoden
  - 2.3 Eigenschaftswörter sind mögliche Kandidaten für Attribute (Datenkomponenten)
3. Suche nach geeigneten Basisklassen bzw. Faktorisierung von Gemeinsamkeiten
4. Spezifikationen der Klassen aufstellen
5. Algorithmischer Entwurf der Methoden (ggf. mit schrittweiser Verfeinerung)

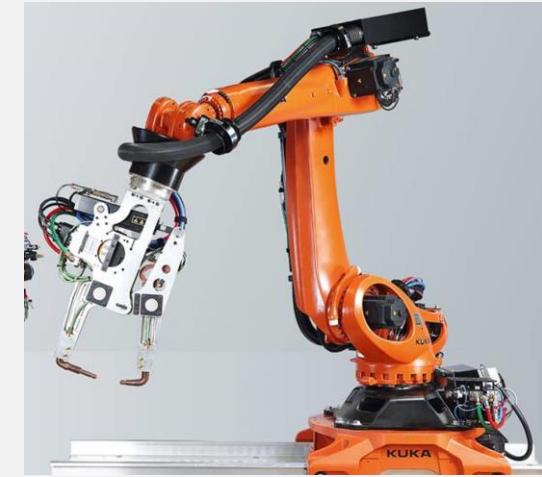
Tipp: Immer etwas allgemeiner bleiben als nötig (um zu möglichst wiederverwendbaren Klassen zu kommen)

# Ein Basismodell für den OO Systementwurf - Beispiel

---

## Aufgabe zur Illustration der Entwurfsmetapher

Ein Roboter soll auf einer geraden Linie von der Position p1 zur Position p2 zwei Karosserieteile durch Punktschweißung verbinden.



Bildquelle: <https://www.kuka.com/>

## Konventioneller Entwurf

- Man geht von den auszuführenden Aktionen aus, hier: „durch Punktschweißung verbinden“
- Ergebnis: eine Prozedur

```
JoinBySpotWelding(↓part1, part2: BodyPart ↓p1, p2: Position)
```

## Bekannte Probleme:

- breite Schnittstellen
- schlechte Wiederverwendbarkeit

# Ein Basismodell für den OO Systementwurf – OO Lösung 1

## Objektorientierter Entwurf – Vorgehen in Anlehnung nach Abbott

- Entwurfsschritt 1

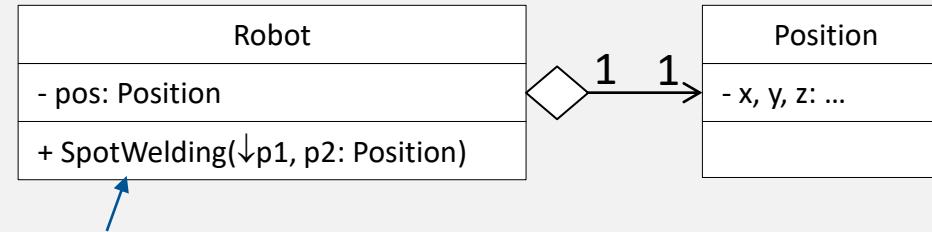
Ein Roboter soll auf einer geraden Linie von der Position p1 zur Position p2 zwei Karosserieteile durch Punktschweißung verbinden.

- Entwurfsschritte 2

Hauptwörter: Roboter, Linie, Position, Karosserieteil, Punktschweißung  
Zeitwörter: verbinden

- Entwurfsschritte 3 und 4

Ergebnis



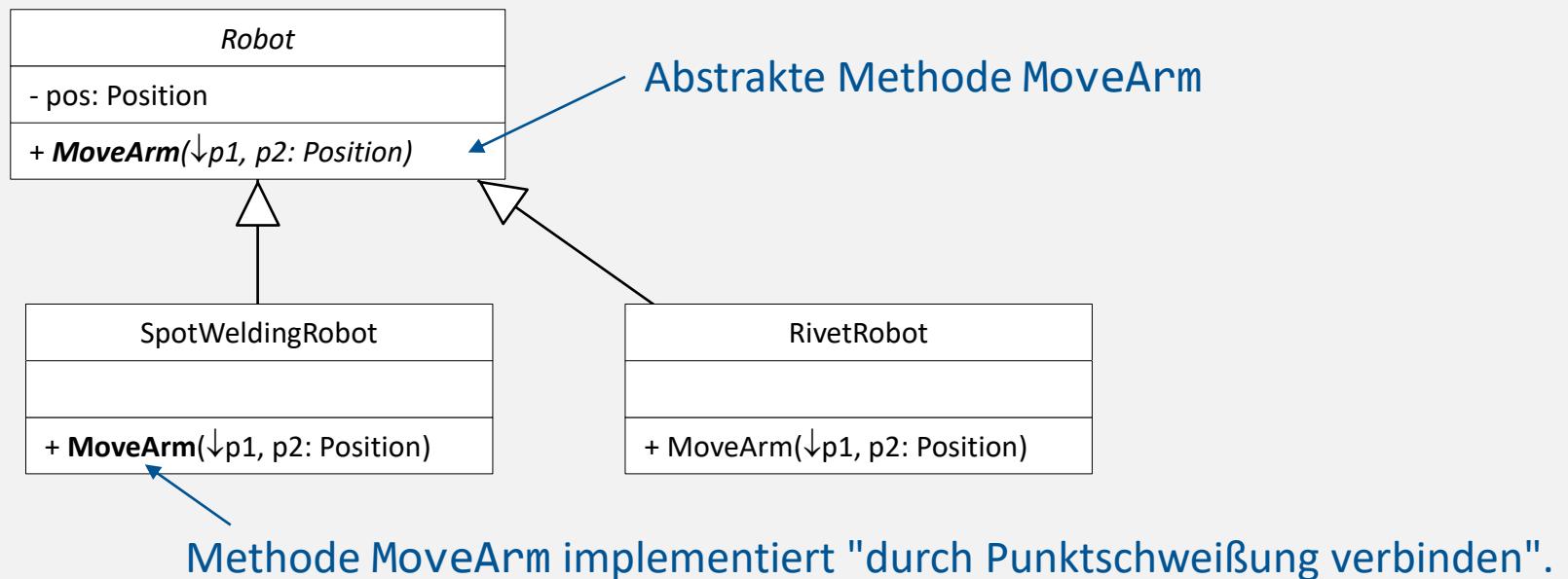
Methode `SpotWelding` implementiert "durch Punktschweißung verbinden".

- im Wesentlichen wie bei konventionellen Entwurf
- Erweiterungen, z.B. durch Nieten verbinden, bohren, lackieren, erfordern Änderungen in der Klasse Robot

# Ein Basismodell für den OO Systementwurf – OO Lösung 2

## Entwurfsschritte 3 und 4

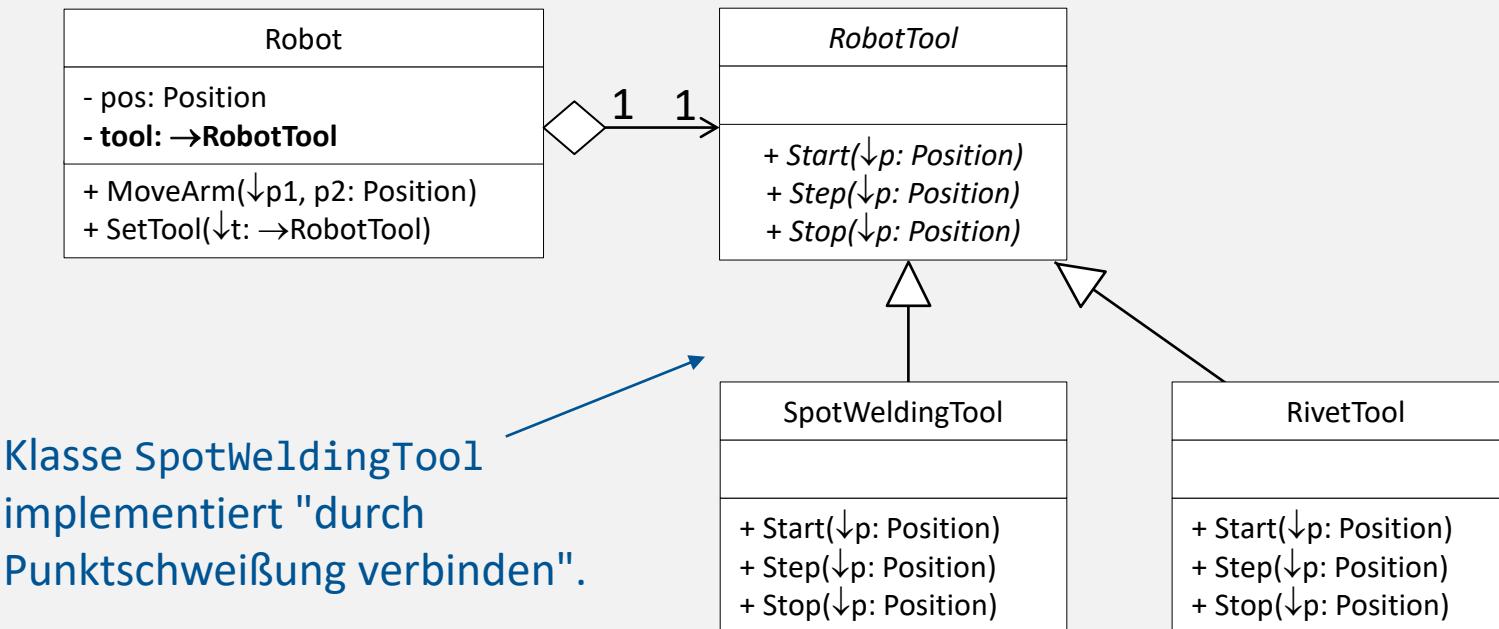
- Wunsch: Erweiterungen, ohne bestehende Klassen zu ändern
- Lösung: Erweiterung durch **Vererbung**



# Ein Basismodell für den OO Systementwurf – OO Lösung 3

## Entwurfsschritte 3 und 4

- Wunsch: Erweiterungen, ohne bestehende Klassen zu ändern
- Lösung: Erweiterung durch **Aggregation**  
"ein Roboter hat ein Werkzeug, z.B. für Punktschweißung".





## Vorschlag von ChatGPT

1. Verbale Spezifikation der Aufgabe aufstellen (besser noch ein Prototyp): Ein Roboter soll in der Lage sein, sich auf einer geraden Linie von einer Position zur anderen zu bewegen und dabei zwei Karosserieteile durch Punktschweißung zu verbinden.
2. Klassen, Methoden und Attribute (Datenkomponenten) identifizieren:
  - 2.1 Hauptwörter als Kandidaten für Klassen:
    - Roboter
    - Position
    - Karosserieteil
    - Punktschweißung
  - 2.2 Zeitwörter als Kandidaten für Methoden:
    - verbinden
    - bewegen
  - 2.3 Eigenschaftswörter und Hauptwörter im Genitiv als Kandidaten für Attribute (Datenkomponenten):
    - geradlinige Bewegung (Attribut von Roboter)
    - Position p1, Position p2 (Attribute von Roboter)
3. Suche nach geeigneten Basisklassen bzw. Faktorisierung von Gemeinsamkeiten: Hier könnte die Basisklasse "Maschine" sein, von der "Roboter" erbt.
4. Spezifikationen der Klassen aufstellen:
  - Klasse Roboter: Attribute sind Position (p1, p2) und eine Methode zum Bewegen und Verbinden.
  - Klasse Position: Attribute sind x- und y-Koordinaten.
  - Klasse Karosserieteil: Attribute könnten Art des Teils, Material, Größe usw. sein.
  - Klasse Punktschweißung: Attribute könnten Hitzeintensität, Dauer, usw. sein; Methoden könnten Durchführung der Schweißung usw. sein.
5. Algorithmischer Entwurf der Methoden (ggf. mit schrittweiser Verfeinerung): Hier könnte man einen einfachen Algorithmus entwerfen, der die Aktionen des Roboters, wie das Bewegen von Position p1 zu Position p2 und das Verbinden der Karosserieteile, steuert.

# Ein Basismodell für den OO Systementwurf

---

## Probleme

- Nicht alle Hauptwörter haben die Bedeutung von Klassen  
„Punktschweißung“, beschreibt ein Verfahren, könnte also auch zu einer Methode werden
- Zuordnung der Methoden zu Klassen kann nicht immer eindeutig vorgenommen werden
- Detailliertheitsgrad der Spezifikation hat großen Einfluss auf den daraus entstehenden Entwurf
- Gefahr zu vieler Klassen

→ Das Verfahren darf nicht blind angewandt werden.

→ Das Auffinden der relevanten Wörter setzt Erfahrung voraus.

# Ein Basismodell für den OO Systementwurf

---

## Weiteres Vorgehen

- Einzelne Klassen unabhängig von der Problemstellung entwickeln
- Beim Detailentwurf der Klassen kommt man auf neue Klassen (iterativ)

## Kategorisierung von Klassen

- Aus der Spezifikation abgeleitete anwendungsnahe Klassen (z.B. Robot)
- Mit der Spezifikation eng verbundene Attributklassen bzw. Klassen für die Manipulation von Datenkomponenten (z.B. Position)
- Für die Implementierung benötigte Klassen (z.B. Point)

## Vorteile

- Abstraktionslücke zwischen Problemwelt und Implementierung wird durch zunehmend konkretere Klassen geschlossen
- Unabhängige Planung der Funktionalität einzelner Klassen zwingt zu größerer Allgemeinheit und verbessert damit die Wiederverwendbarkeit

# Einbindung der Klassenbibliothek

---

## Wiederverwendung durch Benutzung

- Beim Detailentwurf und der Implementierung der Klassen werden Operationen und Attribute benötigt, die evtl. schon in der Klassenbibliothek zur Verfügung stehen

Zwei Möglichkeiten:

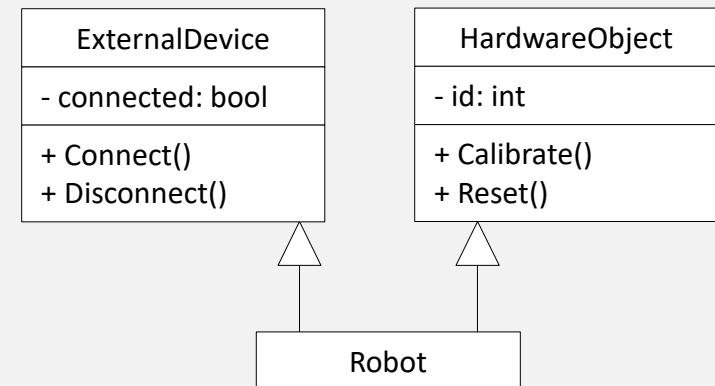
1. Spezifikation der gewünschten Klasse
  - Iterativer Prozess
  - führt wieder zu einer Spezifikation, auf die das Verfahren von Abbott erneut angewandt werden kann
  - Problem: Führt unter Umständen in die Irre, weil schon existierende Lösungen übersehen werden
2. Analyse der Klassenbibliothek
  - Aufbauender Prozess; führt zu Spezialisierungen bestehender Klassen
  - Problem: Overhead durch Ererben von zu viel Funktionalität

# Einbindung der Klassenbibliothek

---

**Wiederverwendung durch Ableitung:** Suche nach einer geeigneten Basisklasse

- Erstes Kriterium: „ist ein“-Relation
- Anforderungen:
  - muss bereits ähnliche Funktionen zur Verfügung stellen
  - soll keine Operationen enthalten, die auf Objekte der neuen Klasse nicht anwendbar sind
- Mehrdeutigkeitsproblem:
  - Robot kann ExternalDevice oder HardwareObject sein
  - Mögliche Lösung: mehrfache Vererbung

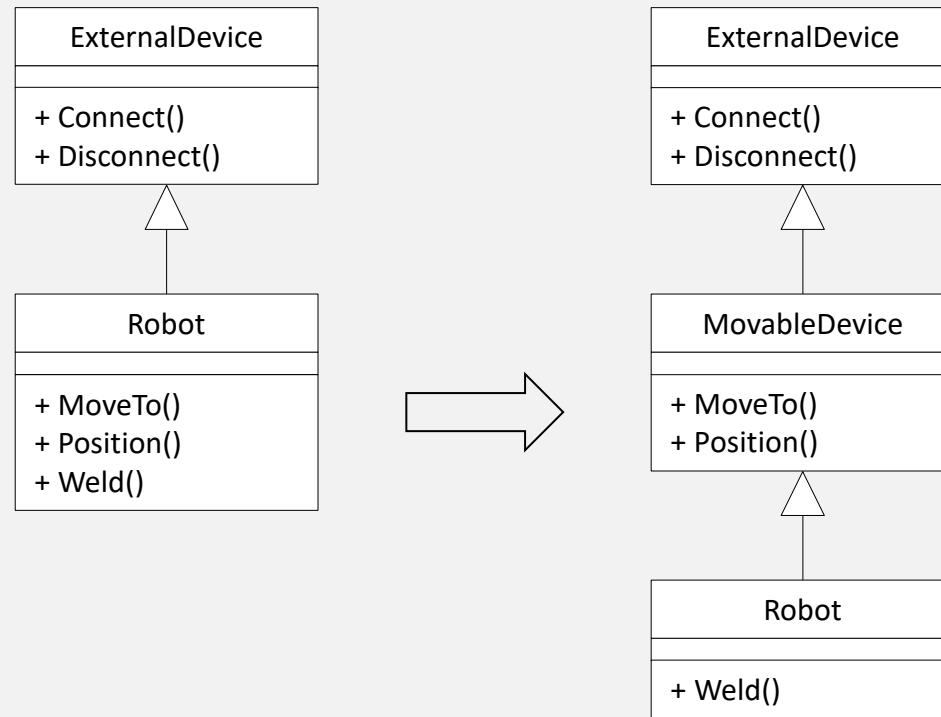


# Einbindung der Klassenbibliothek

**Wiederverwendung durch Ableitung:** Suche nach einer geeigneten Basisklasse

- Wichtig: Zu **große Abstraktionssprünge vermeiden** (Einführung von Zwischenschichten durch Kategorisierung der neuen Klasse und Herausheben allgemeinerer Operationen)

**Beispiel:**



# Top-down oder Bottom-up Vorgehensweise?

---

## Schnittstellenentwurf

- Beides möglich und sinnvoll
- Abbott: Top-down-Verfahren; gute Gliederung in problemnahe und implementierungsnahe Klassen
- Aufbau auf Klassenbibliothek: Bottom-up-Verfahren; optimale Wiederverwendung, setzt umfangreiche Klassenbibliothek voraus

## Algorithmenentwurf

- konventionell, d.h. top-down wie bisher
- Methoden können schrittweise verfeinert werden
- führen zu weiteren Methoden und Anforderungen an benutzte Klassen (größere Allgemeinheit und bessere Wiederverwendbarkeit)

## Alternative: Outside-In

- Ausgehend von einem allgemeinen Rahmen, der inkrementell um die gewünschte Funktionalität erweitert wird (Application Framework)

# Bedeutung der Schnittstellenbeschreibung

---

Bedeutung für konventionellen Entwurf

- Dokument zur Abgrenzung von Teilaufgaben bei Arbeitsteilung

Bedeutung für objektorientierten Entwurf

- Ausführliche Dokumentation der Schnittstelle einer Klasse, damit ihre Eignung für Wiederverwendung abgeschätzt werden kann
- Vergleichbar mit einem Datenblatt für ein elektronisches Bauelement

Mindest-Inhalt

- Klassendefinition
- für jede Message: Vorbedingungen, Wirkung und Nachbedingungen
- Hinweise auf benutzte Klassen und Effizienz

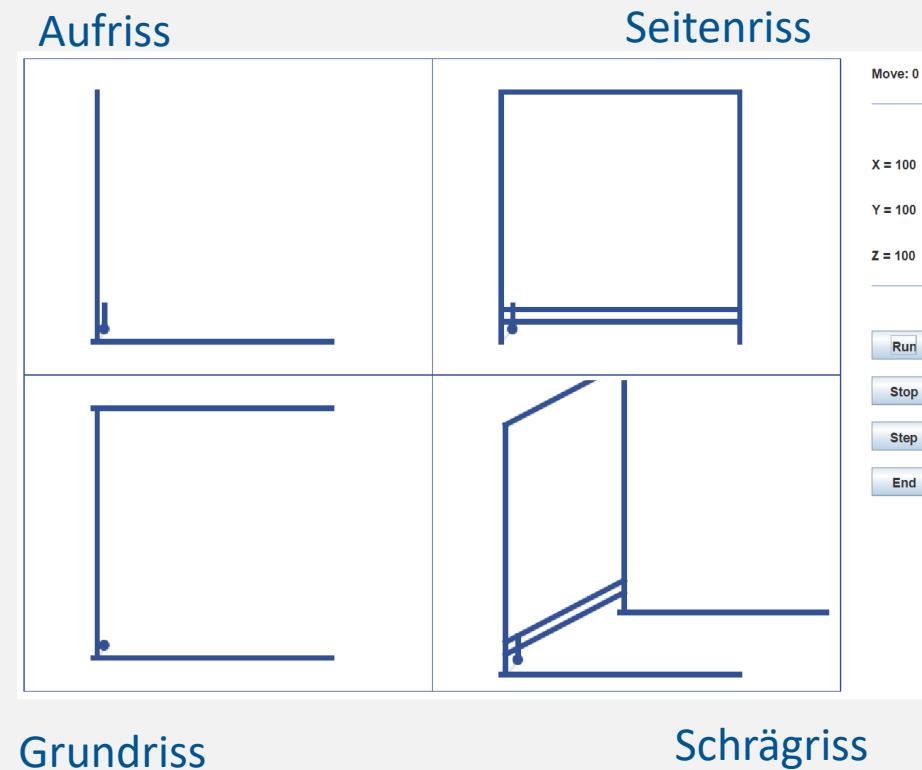
Trennung in Protokoll (syntaktische Beschreibung) und Semantik

Strikte Beschreibung von außen

# Entwurfsbeispiel: App. für 3D-Drucker

Zu Simulations- und Bedienungszwecken soll eine Anwendung für einen 3D-Drucker entwickelt werden

Illustration durch einen Prototyp:



# Entwurfsbeispiel: App. für 3D-Drucker

---

## Entwurfsschritt 1:

- Verbale Spezifikation der Aufgabe aufstellen bzw. Prototyp heranziehen

### Spezifikation

Ein 3D-Drucker, bestehend aus einem Fundament, einem Portal und einer horizontalen Gleitschiene mit einem Druckkopf, soll lineare Bewegungen ausführen können und nach jeder Teilbewegung im Aufriss, Grundriss, Seitenriss und Schrägriss gezeichnet werden.

Das Fundament besteht aus zwei Schienen, das Portal aus drei Balken (zwei vertikalen und einem horizontalen). Die horizontale Gleitschiene besteht aus zwei Balken und der Druckkopf soll durch einen Balken und eine Kugel (für die Düse) dargestellt werden.

Der zurückgelegte Weg soll durch einen Polygonzug in allen vier Sichten angezeigt werden.

# Entwurfsbeispiel: App. für 3D-Drucker

---

## Entwurfsschritt 2:

- Klassen, Methoden und Attribute (Datenkomponenten) identifizieren

### Spezifikation

Ein **3D-Drucker**, bestehend aus einem **Fundament**, einem **Portal** und einer horizontalen **Gleitschiene** mit einem **Druckkopf**, soll lineare Bewegungen ausführen können und nach jeder **Teilbewegung** im **Aufriss, Grundriss, Seitenriss** und **Schrägriss gezeichnet** werden.

Das Fundament besteht aus zwei **Schienen**, das **Portal** aus drei **Balken** (zwei vertikalen und einem horizontalen). Die horizontale Gleitschiene besteht aus zwei Balken und der Druckkopf soll durch einen Balken und eine **Kugel** (für die Düse) dargestellt werden.

Der zurückgelegte **Weg** soll durch einen **Polygonzug** in allen vier **Sichten** angezeigt werden.

# Entwurfsbeispiel: App. für 3D-Drucker

---

## Entwurfsschritt 2:

- Klassen, Methoden und Attribute (Datenkomponenten) identifizieren

### Klassenkandidaten

|              |                     |                   |                   |
|--------------|---------------------|-------------------|-------------------|
| 3D-Drucker   | Druckkopf           | Seitenriss        | Kugel             |
| Fundament    | <i>Teilbewegung</i> | <i>Schrägriss</i> | Weg               |
| Portal       | <i>Aufriss</i>      | Schienen          | <i>Polygonzug</i> |
| Gleitschiene | <i>Grundriss</i>    | Balken            | <i>Sichten</i>    |

### Methodenkandidaten

|                    |
|--------------------|
| Bewegung ausführen |
| gezeichnet werden  |

Annahme: Es ist keine Klassenbibliothek verfügbar, bzw. die benötigten Klassen können nicht aus bestehenden Klassen abgeleitet werden

# Entwurfsbeispiel: App. für 3D-Drucker

---

Entwurfsschritt 3: Suche nach geeigneten Basisklassen bzw. Faktorisierung von Gemeinsamkeiten

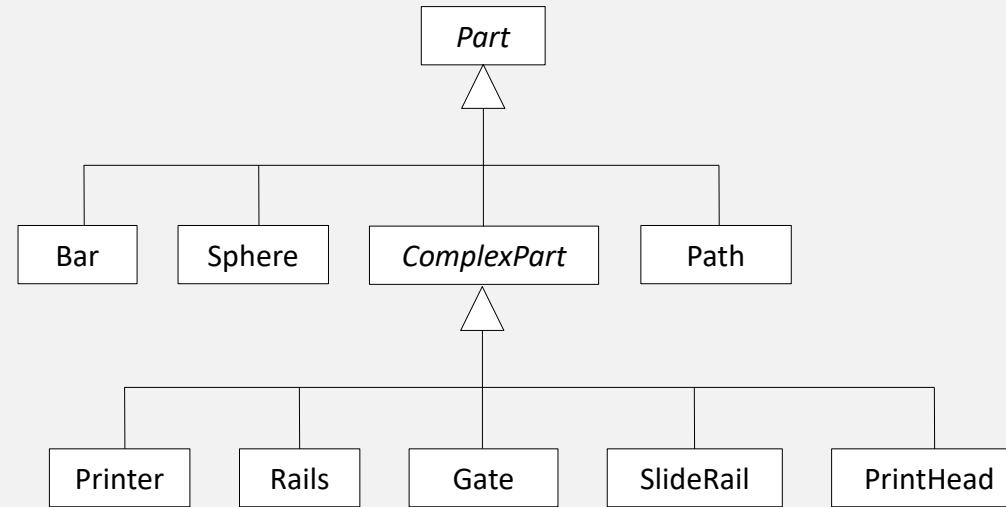
- Der erste Teil dieses Entwurfsschrittes entfällt, weil keine Klassenbibliothek existiert
- Faktorisierung von Gemeinsamkeiten
  - Alle Objekte müssen gezeichnet werden
  - Fast alle Objekte können bewegt werden
  - Überlegung
    - Alle Objekte können als Teile aufgefasst werden (Abstrakte Basisklasse Part)
  - Weitere Einteilung
    - Es gibt elementare Teile und komplexe Teile (d.h. wiederum aus anderen Teilen zusammengesetzte)
    - Bildung von abstrakten Klassen Part und ComplexPart

# Entwurfsbeispiel: App. für 3D-Drucker

---

Entwurfsschritt 4 und 5: Spezifikationen der Klassen aufstellen und Methoden entwerfen

(1) Klassenbeziehung (Vererbungshierarchie) festlegen



# Entwurfsbeispiel: App. für 3D-Drucker

---

## (2) Zuordnung von Messages und Komponenten

- Allgemeine Messages: können für alle Teile (Part) definiert werden

gezeichnet werden

DrawTop( $\downarrow r$ : Rectangle)

DrawFront( $\downarrow r$ : Rectangle)

DrawRight( $\downarrow r$ : Rectangle)

DrawReal( $\downarrow r$ : Rectangle)

Dispose()

Bewegung ausführen

MoveBy( $\downarrow dx$ ,  $dy$ ,  $dz$ : int)

- (noch) keine Komponentenkandidaten identifiziert!

# Entwurfsbeispiel: App. für 3D-Drucker

---

## (3) Konstruktoren festlegen

| Klasse      | Konstruktor                                         |
|-------------|-----------------------------------------------------|
| Part        | Part()                                              |
| ComplexPart | ComplexPart()                                       |
| Bar         | Bar(↓fromX, fromY, fromZ, toX, toY, toZ, thickness) |
| Sphere      | Sphere(↓centerX, centerY, centerZ, radius)          |
| Rails       | Rails(↓x0, y0, z0, length, distance)                |
| Gate        | Gate(↓x0, y0, z0, width, height)                    |
| SlideRail   | SlideRail(↓x0, y0, z0, width)                       |
| PrintHead   | PrintHead(↓x0, y0, z0)                              |
| Printer     | Printer(↓x0, y0, z0, width, height, length)         |
| Path        | Path(↓x0, y0, z0)                                   |

# Entwurfsbeispiel: App. für 3D-Drucker

---

## (4) Detailfragen zur Message-Festlegung

Wozu eine Message Dispose?

- Kenntnis der Implementierungssprache z.B. wenn kein Garbage Collector vorhanden, explizite Freigabe erforderlich

Warum MoveBy statt MoveTo?

- Objekte haben verschiedene Bezugskoordinaten
- relative Bewegung ist vom jeweiligen Koordinatensystem unabhängig

Wie reagieren Rails, Gate und SlideRail auf MoveBy?

- Schienen sind unbeweglich: „Leermethode“: MoveBy wird ignoriert  
„Übersetzung“:  $\text{MoveBy}(\downarrow dx \downarrow dy \downarrow dz)$    $\text{MoveBy}(\downarrow 0 \downarrow 0 \downarrow 0)$
- Portal kann sich nur in x-Richtung bewegen  
„Übersetzung“:  $\text{MoveBy}(\downarrow dx \downarrow dy \downarrow dz)$    $\text{MoveBy}(\downarrow dx \downarrow 0 \downarrow 0)$
- Gleitschiene kann sich nur in x- und z-Richtung bewegen  
„Übersetzung“:  $\text{MoveBy}(\downarrow dx \downarrow dy \downarrow dz)$    $\text{MoveBy}(\downarrow dx \downarrow 0 \downarrow dz)$

# Entwurfsbeispiel: App. für 3D-Drucker

---

## (5) Klassen ausspezifizieren (Schnittstellen)

```
type
 Part = abstract class
 public
 Part()
 Dispose()
 abstract DrawTop(↓r: Rectangle)
 abstract DrawFront(↓r: Rectangle)
 abstract DrawRigth(↓r: Rectangle)
 abstract DrawReal(↓r: Rectangle)
 abstract MoveBy(↓dx,dy,dz: int)
 end -- Part
```

```
type
 ComplexPart = abstract class
 based on Part
 private
 nParts: int
 partList: array [1:maxParts] of →Part
 public
 ComplexPart()
 -- inherited messages
 override Dispose()
 override DrawTop(↓r: Rectangle)
 override DrawFront(↓r: Rectangle)
 override DrawRigth(↓r: Rectangle)
 override DrawReal(↓r: Rectangle)
 override MoveBy(↓dx,dy,dz: int)
 -- new messages
 AddPart(↓p: →Part)
 GetPart(↓pNr: int): →Part
 RemovePart(↓pNr: int)
 end -- ComplexPart
```

# Entwurfsbeispiel: App. für 3D-Drucker

## (5) Klassen ausspezifizieren (Methoden entwerfen)

### Klasse ComplexPart

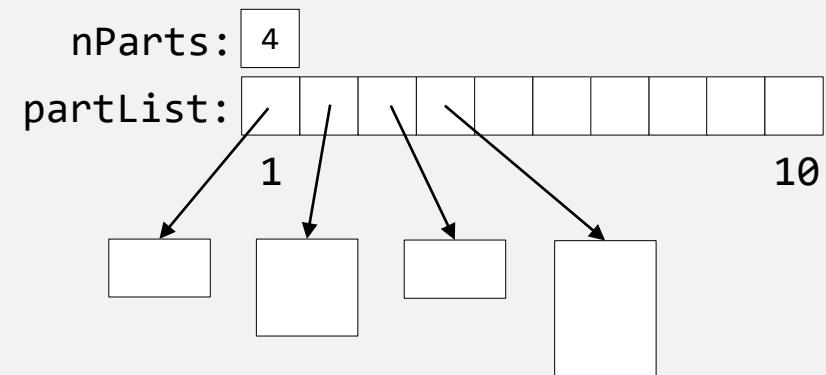
- Konstruktor

```
ComplexPart.ComplexPart()
begin
 super→Part()
 this→nParts := 0
end ComplexPart.ComplexPart
```

- Methode MoveBy

```
ComplexPart.MoveBy(↓dx: int ↓dy: int ↓dz: int)
var
 i: int
begin
 for i := 1 to nParts do
 partList[i]→MoveBy(↓dx ↓dy ↓dz)
 end -- for
end ComplexPart.MoveBy
```

Hinweis: Die dynamische Bindung wird ausgenutzt: Es ist unbedeutend, von welchen Klassen die Elemente im Feld partList sind. Siehe auch folgende Methoden.



# Entwurfsbeispiel: App. für 3D-Drucker

---

## ■ Zeichne Grundriss und Aufriss

```
ComplexPart.DrawTop(↓r: Rectangle)
 var i: int
begin
 for i := 1 to nParts do
 partList[i]→DrawTop(↓r)
 end -- for
end ComplexPart.DrawTop
```

```
ComplexPart.DrawFront(↓r: Rectangle)
 var i: int
begin
 for i := 1 to nParts do
 partList[i]→DrawFront(↓r)
 end -- for
end ComplexPart.DrawFront
```

Zeichnen des Seiten- und Schrägrisses analog dazu

## ■ Freigabe der Objekte

```
ComplexPart.Dispose()
 var i: int
begin
 for i := 1 to nParts do
 partList[i]→Dispose()
 end -- for
 super→Dispose()
end ComplexPart.Dispose
```

# Entwurfsbeispiel: App. für 3D-Drucker

---

## Entwurf der Klasse Bar

```
type
 Bar = class based on Part
 private
 pos1, pos2: Position
 thickness: int
 public
 Bar(↓fromX, fromY, fromZ, toX, toY, toZ, thickness: int)
 -- overwritten messages
 override DrawTop(↓r: Rectangle)
 override DrawFront(↓r: Rectangle)
 override DrawRigth(↓r: Rectangle)
 override DrawReal(↓r: Rectangle)
 override MoveBy(↓dx: int ↓dy: int ↓dz: int)
 -- new messages
 From(): Position
 To(): Position
 Thickness(): int
 SetThickness(↓thickness: int)
 end -- Bar
```

# Entwurfsbeispiel: App. für 3D-Drucker

---

## Entwurf der Klasse Bar

```
Bar.Bar(↓fromX, fromY, fromZ, toX, toY, toZ, thickness: int)
begin
 super→Part()
 pos1.x := fromX; pos1.y := fromY; pos1.z := fromZ
 pos2.x := toX; pos2.y := toY; pos2.z := toZ;
 this→thickness := thickness
end Bar.Bar
```

```
Bar.MoveBy(↓dx: int ↓dy: int ↓dz: int)
begin
 pos1.x := pos1.x+dx; pos1.y := pos1.y+dy; pos1.z := pos1.z+dz
 pos2.x := pos2.x+dx; pos2.y := pos2.y+dy; pos2.z := pos2.z+dz
end Bar.MoveBy
```

```
Bar.DrawTop(↓r: Rectangle)
begin
 "draw line from pos1 to pos2 with thickness within rectangle r"
end Bar.DrawTop
```

Hinweis: Die informelle Aktion ist leicht zu algorithmisieren. Die Hauptarbeit besteht darin, die "Weltkoordinaten" in Bildschirmkoordinaten umzurechnen.

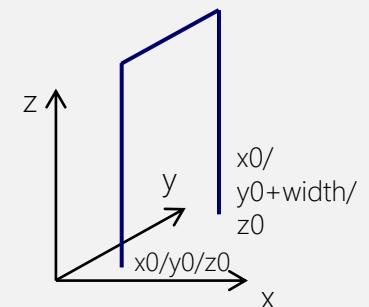


# Entwurfsbeispiel: App. für 3D-Drucker

## Entwurf der Klasse Gate

```
type
 Gate = class based on ComplexPart
 public
 Gate(↓x0, y0, z0, width, height: int)
 -- overwritten messages
 override MoveBy(↓dx: int ↓dy: int ↓dz: int)
 end -- Gate
```

```
Gate.Gate(↓x0, y0, z0, width, height: int)
 var b: →Bar
begin
 super→ComplexPart()
 b := New(↓Bar ↓x0 ↓y0 ↓z0 ↓x0 ↓y0 ↓z0+height)
 this→AddPart(↓b)
 b := New(↓Bar ↓x0 ↓y0+width ↓z0 ↓x0 ↓y0+width ↓z0+height)
 this→AddPart(↓b)
 b := New(↓Bar ↓x0 ↓y0 ↓z0+height ↓x0 ↓y0+width ↓z0+height)
 this→AddPart(↓b)
end Gate.Gate
```



# Entwurfsbeispiel: App. für 3D-Drucker

---

## Adaptierung der geerbten Methode MoveBy

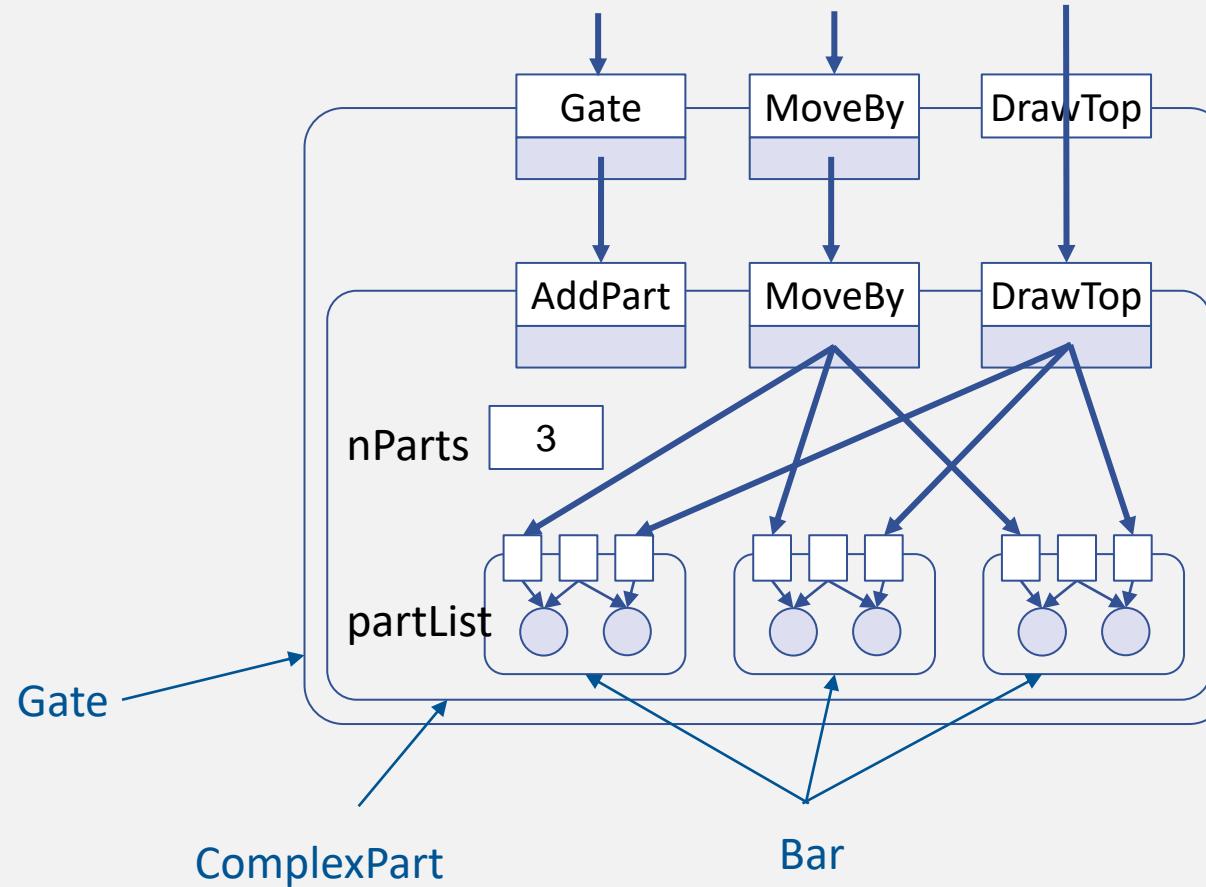
```
Gate.MoveBy(↓dx: int ↓dy: int ↓dz: int)
begin
 super→MoveBy(↓dx ↓0 ↓0)
end Gate.MoveBy
```

## Bemerkungen

- `Gate.MoveBy(...)` muss nur dafür sorgen, dass die geerbte Methode von `ComplexPart` mit modifizierten Parametern aufgerufen wird
- Außer Konstruktor und `MoveBy` müssen in `Gate` keine weiteren Methoden implementiert werden, weil die erforderliche Funktionalität schon von `ComplexPart` durch Vererbung zur Verfügung gestellt wurde
- Ähnliches gilt auch für `Rails` („Leermethode“ für `MoveBy`) und `RailSlide`

# Entwurfsbeispiel: App. für 3D-Drucker

Illustration der Konsequenzen unserer Entwurfsentscheidungen



# Entwurfsbeispiel: App. für 3D-Drucker

---

## Illustration der Konsequenzen unserer Entwurfsentscheidungen

- Vorstellung: Gate ist eine Hülle um ComplexPart, bzw. ComplexPart bildet den Kern von Gate
- MoveBy wird in Gate implementiert und mit veränderten Parametern an ComplexPart weitergeleitet, von wo aus die Message an die Elemente von partList verteilt wird
- DrawTop wird geerbt; die Methode von ComplexPart wird ausgeführt
- Konstruktor wird neu implementiert und greift über AddPart auf die in ComplexPart definierten Komponenten zu

# Entwurfsbeispiel: App. für 3D-Drucker

## Entwurf der Klasse Printer

```
type
 Printer = class based on ComplexPart
 private
 toolPos: Position
 public
 Printer(↓x0, y0, z0, width, height, length: int)
 override MoveBy(↓dx: int ↓dy: int ↓dz: int)
 ToolPosition(): Position
 end -- Printer
```

Aktualisieren der  
Werkzeugposition

```
Printer.Printer(↓x0, y0, z0, width, height, length: int)
 var rails, gate, slide, head: →Part
begin
 super→ComplexPart()
 rails := New(↓Rails ↓x0 ↓y0 ↓z0 ↓length ↓width)
 this→AddPart(↓rails)
 gate := New(↓Gate ↓x0 ↓y0 ↓z0 ↓width ↓height)
 this→AddPart(↓gate)
 slide := New(↓SlideRail ↓x0 ↓y0 ↓y0 ↓width)
 this→AddPart(↓slide)
 head := New(↓PrintHead ↓x0 ↓y0 ↓z0)
 this→AddPart(↓head)
end Printer.Printer
```

# Entwurfsbeispiel: App. für 3D-Drucker

## Entwurf der Klasse Printer

```
Printer.MoveBy(↓dx: int ↓dy: int ↓dz: int)
 super→MoveBy(↓dx ↓dy ↓dz)
 toolPos.x := toolPos.x + dx
 toolPos.y := toolPos.y + dy
 toolPos.z := toolPos.z + dz
end Printer.MoveBy
```

Wiederverw. der Funktionalität aus ComplexPart

Hinzufügen neuer Funktionalität

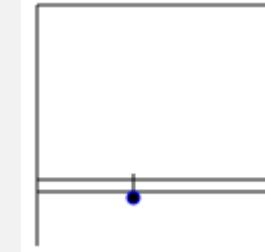
## Verwendung der Klasse Printer

```
var
 p: →Printer

p := New(↓Printer ↓0 ↓0 ↓0 ↓2000 ↓2000 ↓2000)
p→DrawRight(↓...)
```

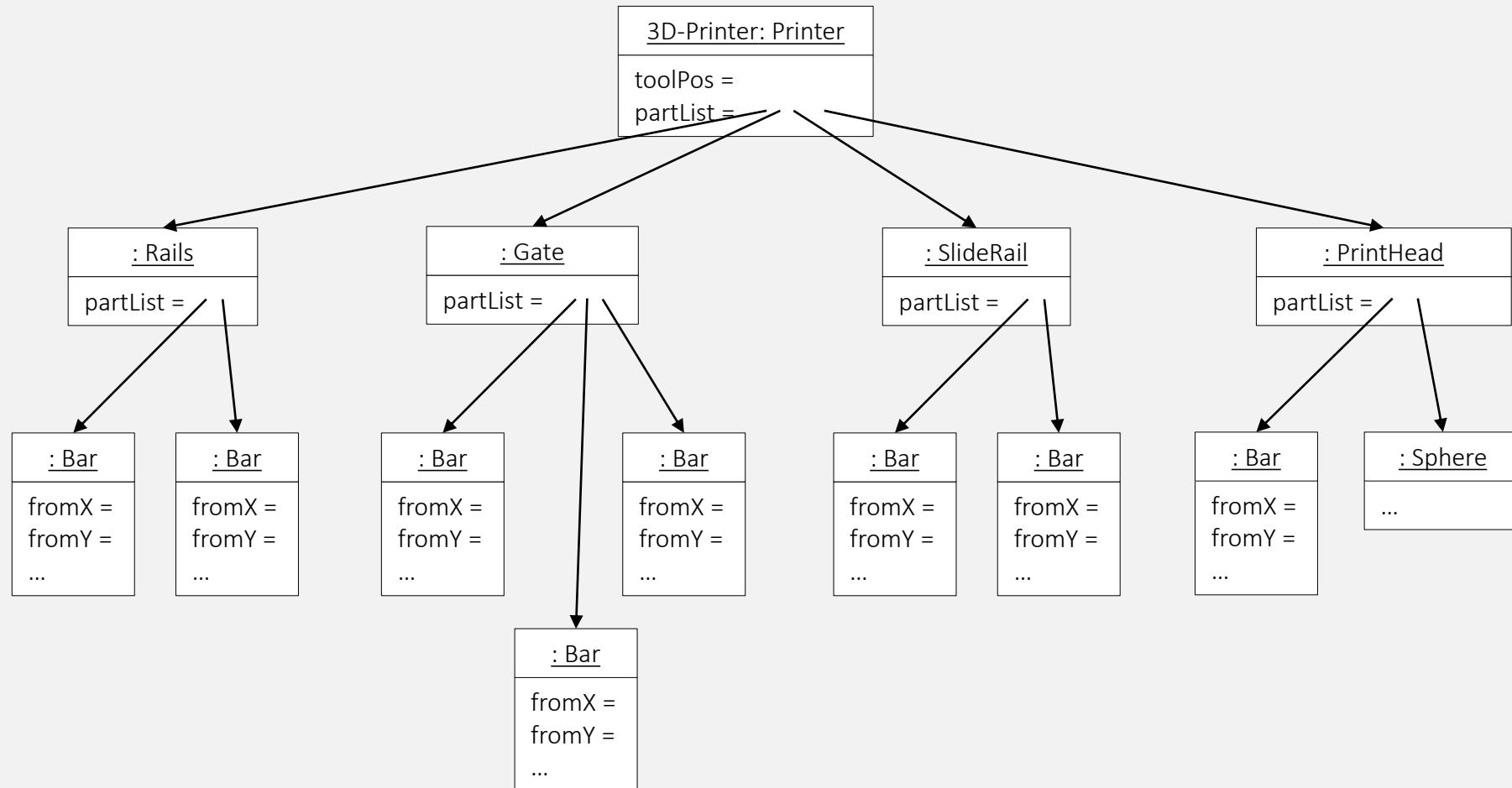


```
p→MoveBy(↓500 ↓800 ↓400)
p→DrawRight(↓ ...)
```



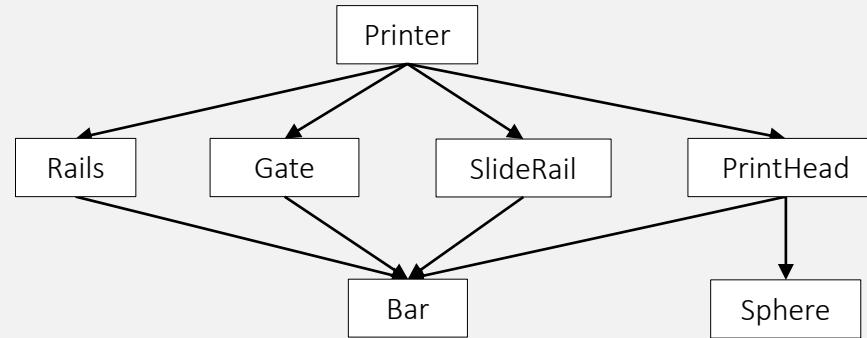
# Entwurfsbeispiel: Objektbaum

---

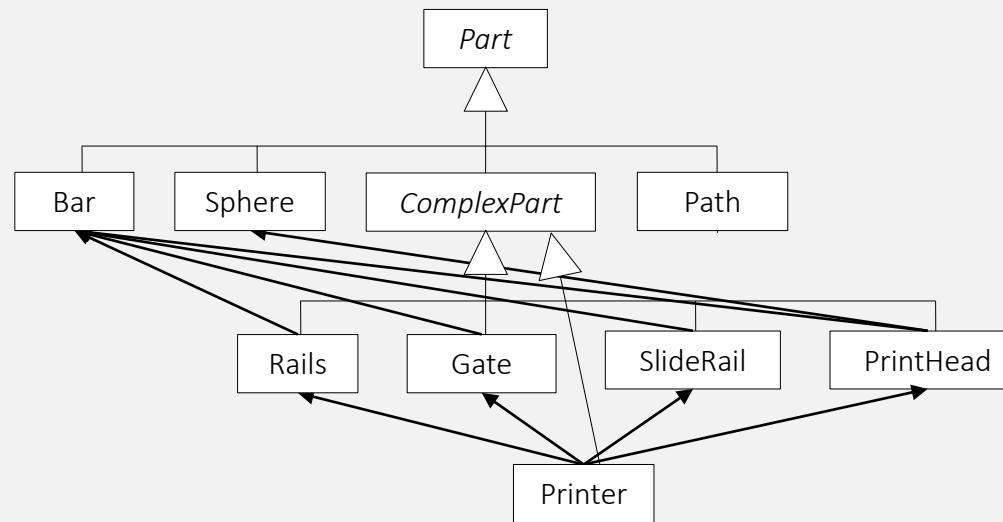


# Entwurfsbeispiel: Beziehungsgeflecht des Typsystems

„Besteht aus“ – Relation



Gemeinsame Darstellung von Vererbungs- („ist-ein“-) und „besteht-aus“-Beziehung



Oft sehr komplexe Zusammenhänge

# Aufgabe: Klassenhierarchie für arithmetische Ausdrücke

---

Quelle: Aufgabe 13.4 (Grammatik objektorientiert modelliert, Seite 533, Lehrbuch AuD Pomberger u. Dobler, 2008).

Gegeben ist folgende Grammatik für einfache arithmetische Ausdrücke:

Expr = Term { '+' Term }.

Term = Fact { '\*' Fact }.

Fact = number | '(' Expr ')'.  
.

Elemente dieser Ausdrücke sind Operanden (Zahlen und geklammerte Ausdrücke) sowie binäre Operatoren (die auf jeweils zwei Operanden angewandt werden).

Entwickeln Sie eine Hierarchie von Klassen, deren Objekte es gestatten, arithmetische Ausdrücke in Form von Binärbäumen darzustellen, auszuwerten und in Infix-Notation auszugeben.

Geben Sie zumindest die Klassendeklarationen an, und versuchen Sie dann, die Methodenimplementierungen auszuarbeiten.

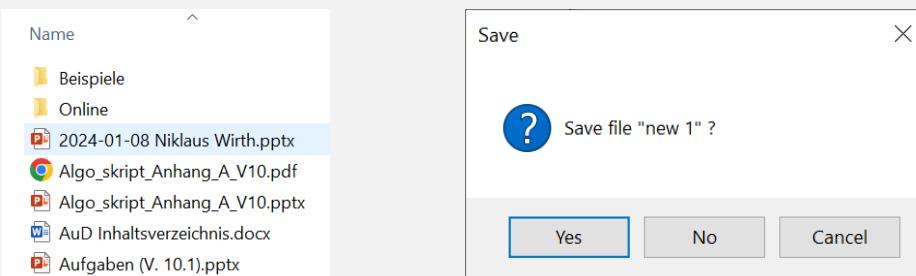
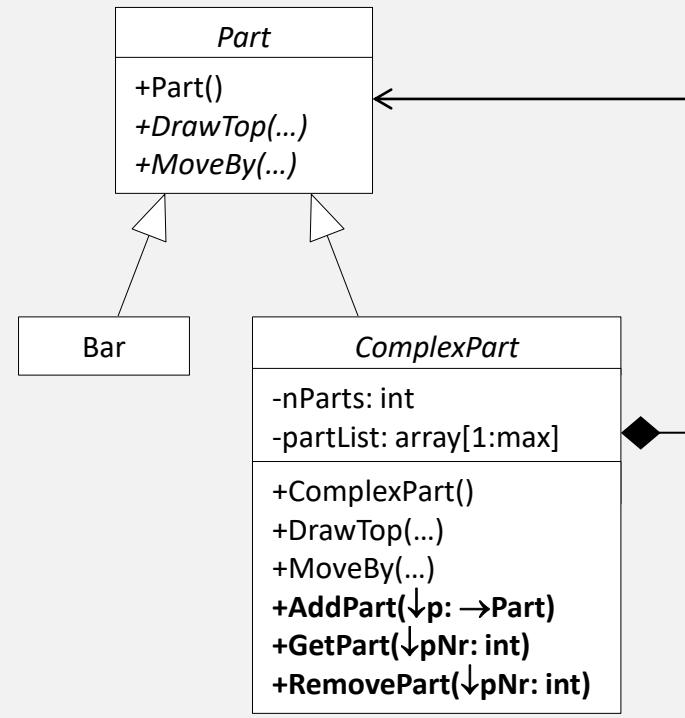
# 17.15 Entwurfsmuster

Gutes Design lernt man durch Erfahrung

**Beispiel:** Einfache und komplexe Teile eines 3D-Druckers (Seite 32- 33)

- Es gibt elementare Teile und komplexe Teile
- Bildung von abstrakten Klassen Part und ComplexPart

Die hier verwendete Idee, um einfache und komplexe Objekte gleich zu behandeln, finden wir häufig.

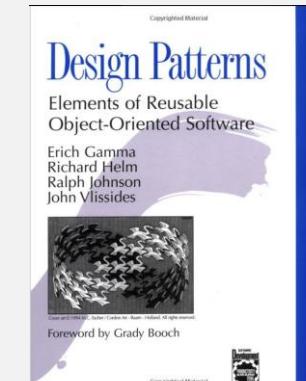
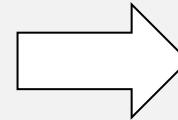
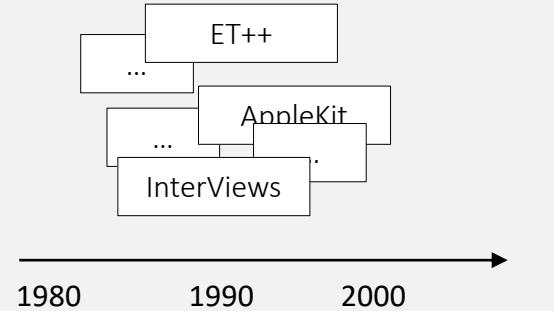


# 17.15 Entwurfsmuster

Gutes Design lernt man durch Erfahrung

Es gibt **Entwurfsmuster**

- für die Gestaltung von Systemarchitekturen und guten Programmstrukturen



1994

- Katalog mit 23 Entwurfsmuster
- Erzeugungsmuster (engl. *creational patterns*)
- Strukturmuster (engl. *structural patterns*)
- Verhaltensmuster (engl. *behavioral patterns*)

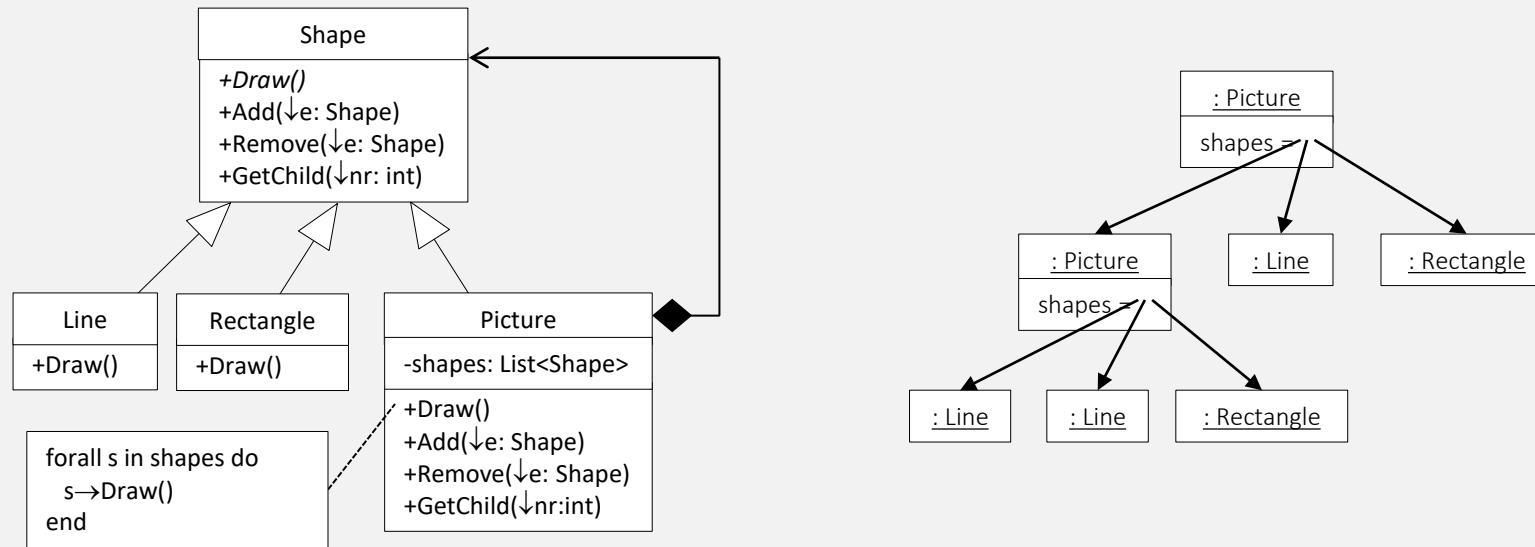
# Strukturmuster: Composite

## Zweck

- Bilde Teil-Ganzes-Hierarchien von Objekten
- Behandle elementare und komplexe Objekte gleich

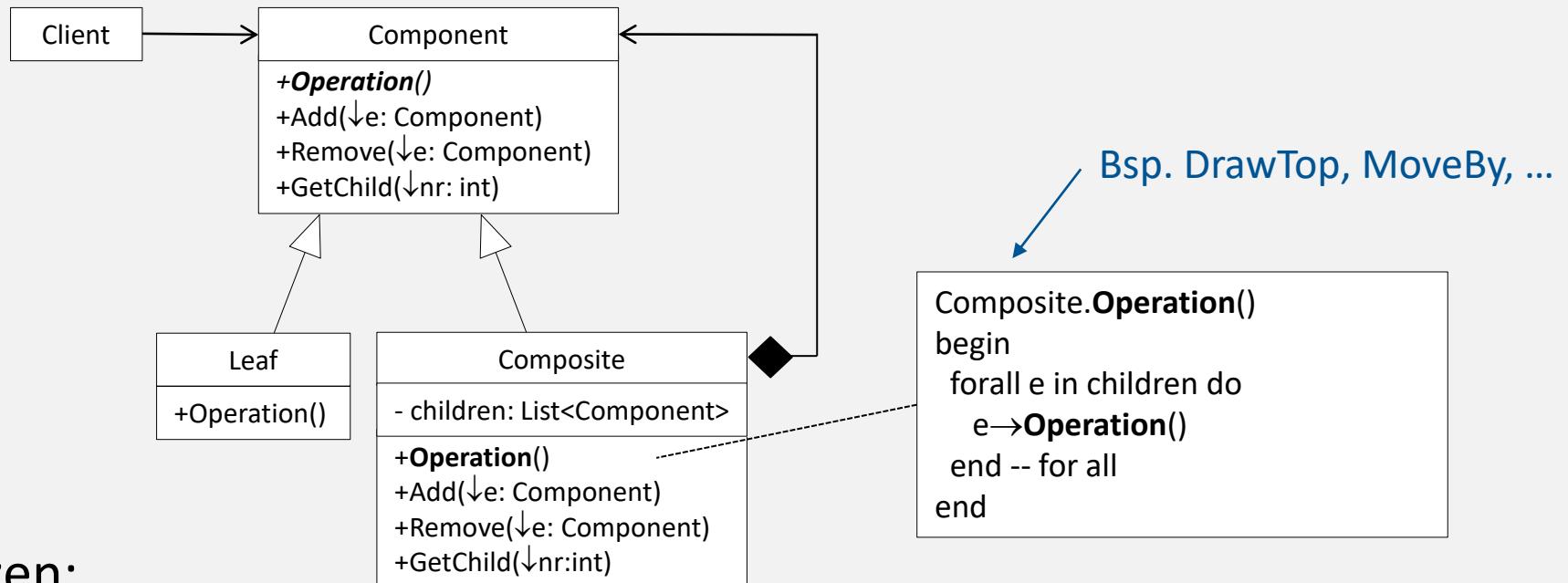
## Motivation

- Grafikeditor mit verschiedenen grafischen Objekten (Linie, Rechteck, ...), die zu Bildern zusammengefasst werden können



# Strukturmuster: *Composite*

## Struktur



## Konsequenzen:

- Verwendung (aus Client-Sicht) wird einfach, neue Komponenten können einfach hinzugefügt werden
- Implementierungsdetails: Verwaltung der Kindknoten
  - Implementierung in der Klasse Composite
  - Schnittstelle in Component- oder Composite-Klasse?

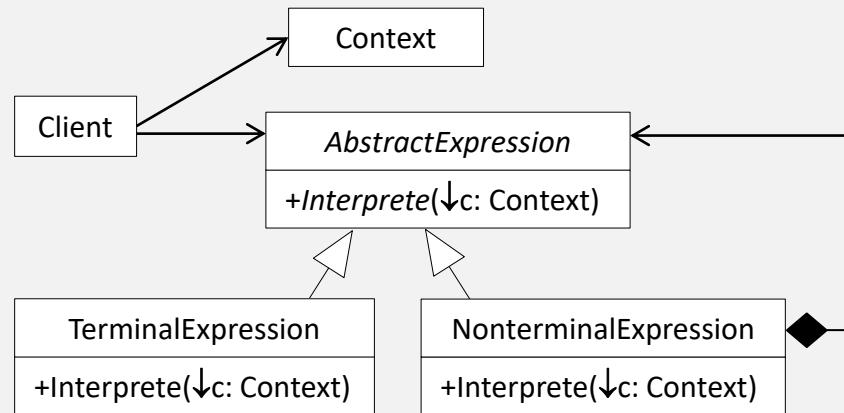
# Verhaltensmuster: *Interpreter*

---

## Zweck

- Definiere eine Repräsentation für die Grammatik einer formalen Sprache und einen Interpreter, um Sätze dieser Sprache zu interpretieren

## Struktur



### Objekte der Klassen

**TerminalExpression** und  
**NonterminalExpression** bilden einen  
sogenannten abstrakten Syntaxbaum.

## Bestandteile

**AbstractExpression** deklariert die Operation **Interprete**, die für alle Knoten im abstrakten Syntaxbaum implementiert wird.

**TerminalExpression** implementiert die Operation für ein Terminalsymbol (TS) der Sprache. Ein Objekt für jedes TS.

**NonterminalExpression** implementiert die Operation für ein Nonterminalsymbol.

**Context** stellt Informationen (z.B. VariablenSpeicher) bereit.

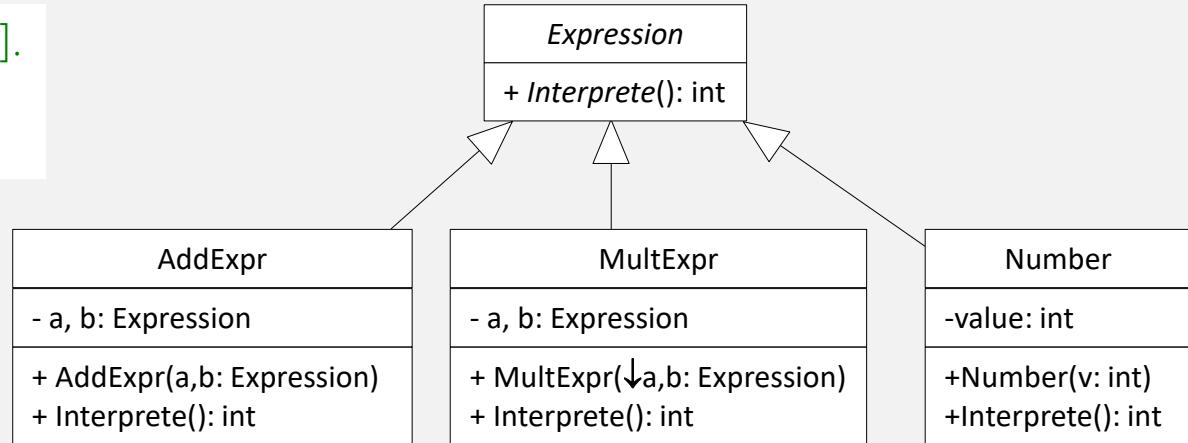
# Verhaltensmuster: Interpreter

## Anwendung

AddExpr = MultExpr [ "+" MultExpr ].

MultExpr = Fact [ "\*" Fact ].

Fact = number | '(' AddExpr ')'.



## Methoden

```
AddExpr.AddExpr(↓a ↓b: →Expression)
begin
 this→a := a
 this→b := b
end AddExpr.AddExpr
```

```
AddExpr.Interprete(): int
begin
 return a→Interprete() + b→Interprete()
end AddExpr.Interprete
```

```
Number.Number(↓v: int)
begin
 this→value := v
end Number.Number
```

```
Number.Interprete(): int
begin
 return value
end Number.Interprete
```

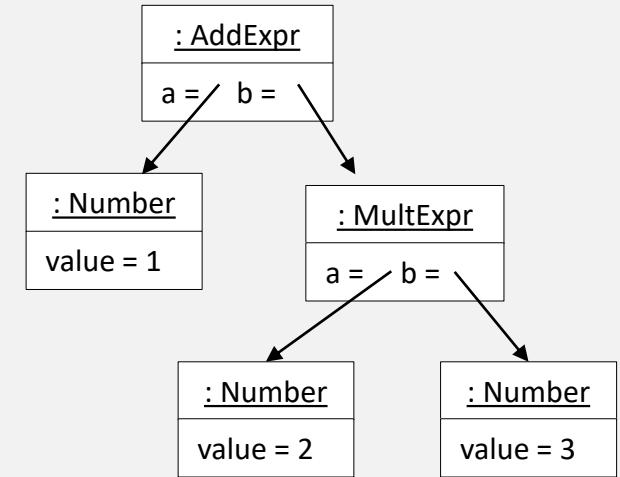
# Verhaltensmuster: *Interpreter*

## Objektdiagramm für Ausdruck $1 + 2 * 3$

```
var
 n1, n2, n3: →Number
 add: →AddExpr
 mul: →MultExpr
 v: int

 n1 := New(↓Number ↓1)
 n2 := New(↓Number ↓2)
 n3 := New(↓Number ↓3)
 mul := New(↓MultExpr ↓n2 ↓n3)
 add := New(↓AddExpr ↓n1 ↓mul)

 v := add→Interprete()
```



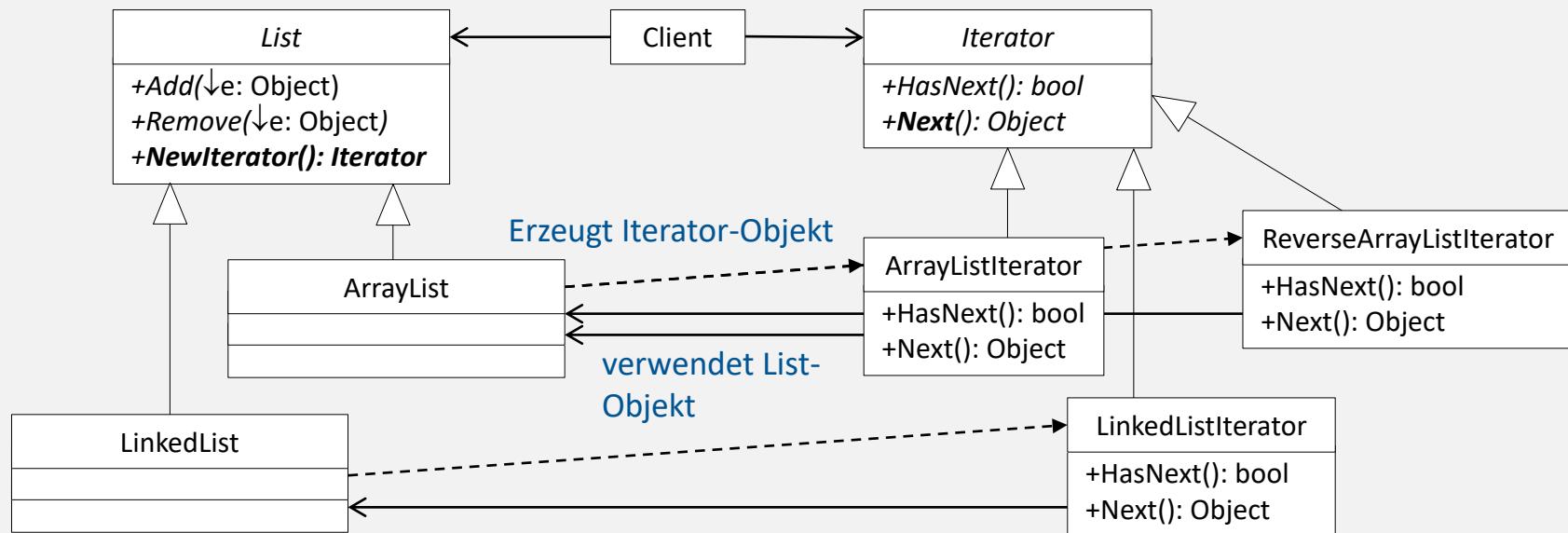
# Verhaltensmuster: Iterator

## Zweck

- Möglichkeit, auf die Elemente eines Behälters sequenziell zuzugreifen, ohne interne Darstellung (Datenstruktur) offenzulegen

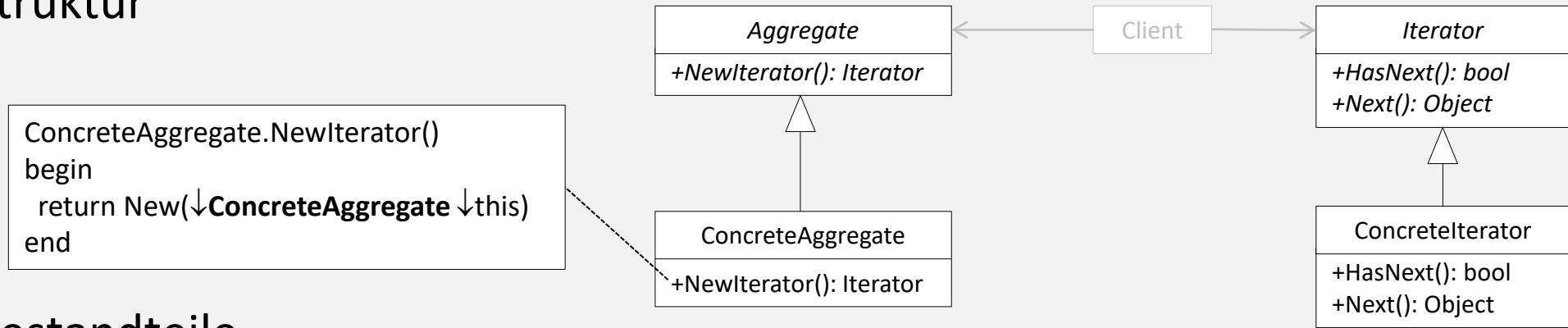
## Motivation

- Zugriff auf Elemente eines Behälters, ohne interne Darstellung offenzulegen
- Behälter auf verschiedene Weise durchlaufen (z.B. vorwärts oder rückwärts)



# Verhaltensmuster: Iterator

## Struktur



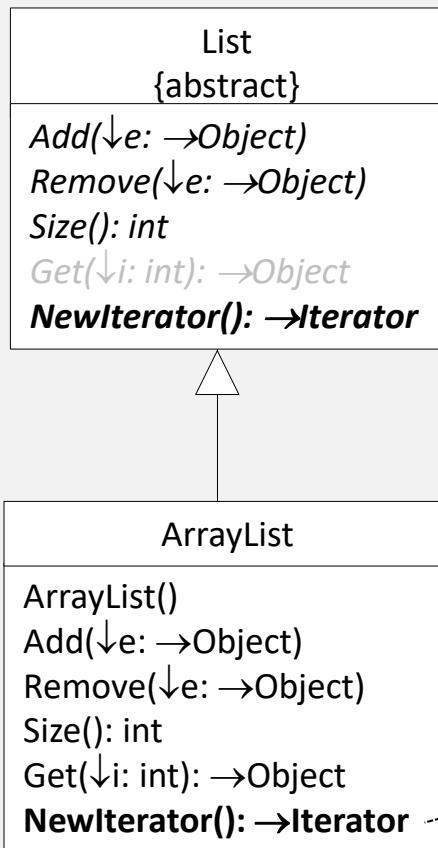
## Bestandteile

- **Iterator** definiert eine Schnittstelle für den Zugriff auf Elemente (`Next`) und das Durchlaufen von Elementen (`HasNext`)
- **ConcretelIterator** implementiert **Iterator** und verfolgt die aktuelle Position beim Durchlaufen des Behälters (**Aggregats**)
- **Aggregate** definiert eine Schnittstelle für das Erzeugen eines Iterator-Objekts
- **ConcreteAggregate** erzeugt und liefert konkrete Iterator-Objekte

# Verhaltensmuster: *Iterator*

---

## Anwendung: Iterator für Liste von Objekten



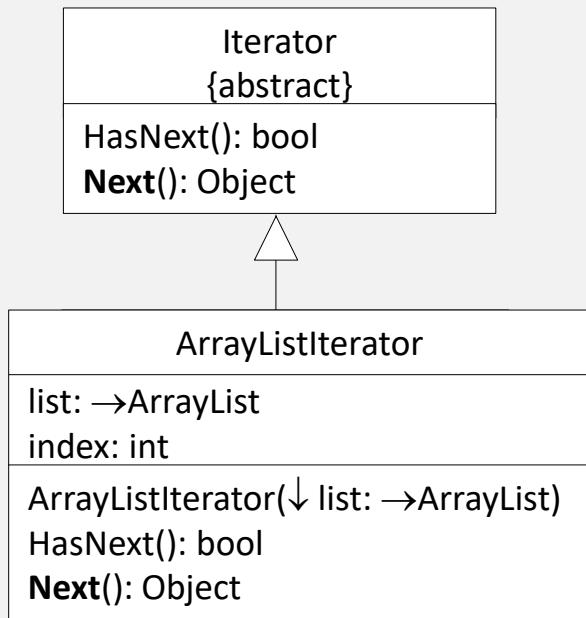
Anmerkung: Zugriff auf Elemente mit Iterator ist allgemeiner als mit Zugriffsoperation `Get(i: int): Object`  
`ArrayList.Get(...)` ...  $O(1)$   
`LinkedList.Get(..)` ...  $O(n)$

```
ArrayList.NewIterator()
begin
 return New(ArrayListIterator this)
end ArrayList.NewIterator
```

# Verhaltensmuster: *Iterator*

---

## Anwendung: Iterator für Liste von Objekten



```
ArrayListIterator.ArrayListIterator(↓list: →ArrayList)
begin
 this→list := list
 this→index := 1
end ArrayListIterator.ArrayListIterator
```

```
ArrayListIterator.HasNext(): bool
begin
 return index ≤ list→Size()
end ArrayListIterator.HasNext
```

```
ArrayListIterator.Next(): →Object
var
 obj: →Object
begin
 obj := list→Get(↓index)
 index := index + 1
 return obj
end ArrayListIterator.Next
```

# Verhaltensmuster: Iterator

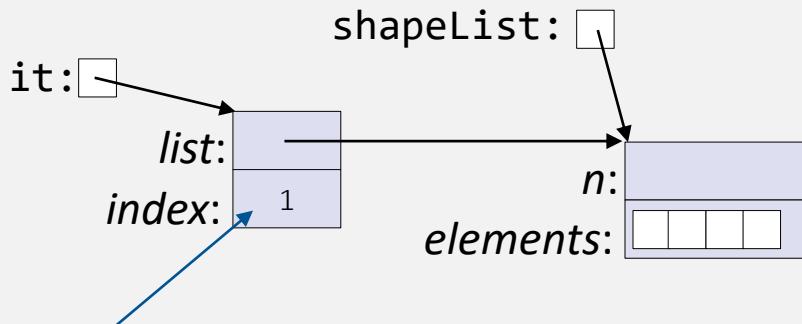
ohne Iterator

```
DrawShapes(↓shapeList: →ArrayList)
 var obj: →Object; i: int
begin
 for i := 1 to shapeList→Size() do
 obj := shapeList→Get(↓i)
 Shape(obj)→Draw()
 end -- for
end DrawShapes
```

Aktuelle Position wird in DrawShape gehalten

mit Iterator

```
DrawShapes(↓shapeList: →List)
 var obj: →Object; it: →Iterator
begin
 it := shapeList→NewIterator()
 while it→HasNext() do
 obj := shapeList→Next()
 Shape(obj)→Draw()
 end -- for
 Dispose(↓it)
end DrawShapes
```



Aktuelle Position wird in Iterator gehalten

## 17.16 Design by Contract

---

- Das Basismodell für den objektorientierten Systementwurf, die Entwurfsmuster und die UML helfen, einen Entwurf zu erstellen.
- Wie können wir sicherstellen, dass die entworfene Software korrekt ist?

### Design by Contract

- Eine Software-Engineering-Methode zur Spezifikation, Entwicklung und Verifizierung von objektorientierter Software
- Verifizierung erfolgt zur Laufzeit durch Prüfungen basierend auf Verträgen (engl. *contracts*)
- Eingeführt von Bertrand Meyer in der Programmiersprache Eiffel
- Contract-Bibliotheken für einige Programmiersprachen (z.B. Java und C#) verfügbar

# Software-Korrektheit

---

**Beispiel:** Ist folgende Anweisung korrekt?

```
x := y + 1
```

**Korrekt** für die Spezifikation “Stellen Sie sicher, dass x und y unterschiedliche Werte haben“.

**Inkorrekt** in Bezug auf die Spezifikation “Stellen Sie sicher, dass x einen negativen Wert hat“.

- Korrektheit ist ein relativer Begriff - relativ zu einer Spezifikation

Korrektheitsformel

Vorbedingung

Nachbedingung

- Die Spezifikation basiert auf der Korrektheitsformel  $\{P\} C \{Q\}$
- Bedeutung: Jede Ausführung von C, die in einem Zustand beginnt, in dem P gilt, wird in einem Zustand enden, in dem Q gilt

**Beispiele**

```
{x ≥ 9} x := x + 5 {x ≥ 13}
```

**Korrekt** aber nicht strengste Nachbedingung

```
{x ≥ 9} x := x + 5 {x ≥ 14}
```

**Korrekt** und strengste Nachbedingung

```
{x ≥ 9} x := x + 5 {x ≥ 15}
```

**Inkorrekt**

# Zusicherungen: Vor- und Nachbedingungen

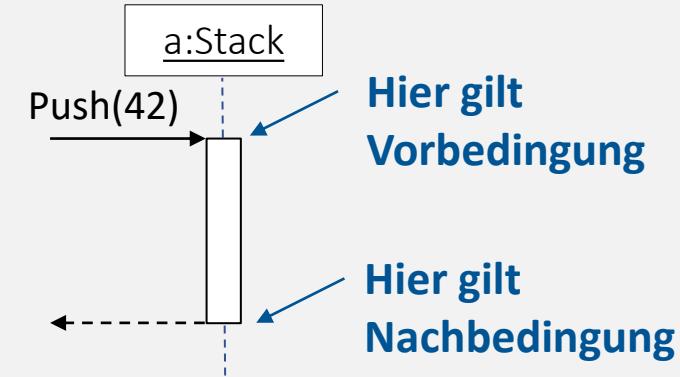
Semantische Spezifikation einer Methode durch zwei mit der Methode verbundenen Bedingungen:

## Vorbedingung

- Die Vorbedingung gibt die Eigenschaften an, die immer erfüllt sein müssen, wenn die Methode aufgerufen wird
- Ein korrektes System wird niemals einen Aufruf in einem Zustand ausführen, der die Vorbedingung der aufgerufenen Methode nicht erfüllt

## Nachbedingung

- Die Nachbedingung gibt die Eigenschaften an, die die Methode garantiert, wenn sie zurückkehrt
- Garantie: die Methode liefert einen Zustand, der spezifizierte Eigenschaften erfüllt, vorausgesetzt, sie wurde mit erfüllter Vorbedingung aufgerufen



# Vor- und Nachbedingungen

## Beispiel: Eine Klasse Stack

```
Stack.Push(\downarrow e: →Object)
require
 not IsFull()
begin...
ensure
 not IsEmpty()
 Top() = e
 Count() = old Count() + 1
end
```

Push darf nicht aufgerufen werden, wenn Stack voll ist

Nach Push darf Stack nicht leer sein, das „oberste“ Element ist e und die Anzahl der Elemente wurde um 1 erhöht

```
Stack.Pop()
require
 not IsEmpty()
begin...
ensure
 not IsFull
 Count() == old Count() - 1
end
```

Pop darf nicht auf einen leeren Stack angewandt werden

Nach Pop darf Stack nicht voll sein und seine Anzahl der Elemente wurde um eins verringert.

Hinweis. Die Notation **old** e mit dem Ausdruck e, bezeichnet den Wert, den e beim Aufruf der Methode hatte.

```
type
Stack = class
public
 Stack()
 Push(\downarrow e: →Object)
 Pop()
 Top(): →Object
 IsEmpty(): bool
 IsFull(): bool
 Count(): int
end -- Stack
```

# Vor- und Nachbedingungen

---

Vorbedingung bezieht sich nur auf die Parameter der Methode und Datenkomponenten des Objekts

Nachbedingung kann sich beziehen auf:

- Parameter der Methode und Datenkomponenten des Objekts (wie Vorb.)
- Parameter der Methode und Datenkomponenten des Objekts im Vorzustand, d.h. zum Zeitpunkt des Methodeneintritts (**old e**)
- den Rückgabewert der Methode (**result**)

```
X.Swap(↓↑a: int ↓↑b: int)
begin
 t := a
 a := b
 b := t
ensure
 a = old b
 b = old a
end
```

```
Account.Withdraw(↓amount: int): int
require
 amount > 0
begin
 balance := balance - amount
 return balance
ensure
 balance = old balance - amount
 result = balance
end
```

# Verträge: Rechte und Pflichten

---

Die Vorbedingung bindet den Klienten:

- sie definiert die Bedingungen, unter denen ein Aufruf der Methode legitim ist
- sie ist eine **Verpflichtung für den Klienten** und ein **Vorteil für die Klasse**

Die Nachbedingung bindet die Klasse:

- sie definiert die Bedingungen, die von der Methode beim Zurückkehren sichergestellt werden müssen
- sie ist ein **Vorteil für den Klienten** und eine **Verpflichtung für die Klasse**

Beispiel

|               | <b>Verpflichtung</b>                                                                                        | <b>Vorteil</b>                                                                                |
|---------------|-------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------|
| <b>Klient</b> | (Vorbedingung erfüllen)<br>Push(x) nur bei einem nicht vollen Stack aufrufen                                | (aus Nachbedingung)<br>Stack wurde aktualisiert, ist nicht leer und Anzahl ist um 1 erhöht    |
| <b>Klasse</b> | (Nachbedingung erfüllen)<br>Stack-Darstellung aktualisiert, die Anzahl um 1 erhöht und Stack ist nicht leer | (aus Vorbedingung)<br>einfachere Verarbeitung dank der Annahme, dass der Stack nicht voll ist |

# Verträge: Nicht-Redundanz-Prinzip

Unter keinen Umständen darf die Methodenimplementierung die Vorbedingung der Methode prüfen.

## Design by Contract

```
sqrt(↓x: float): float
require
 x >= 0
begin
 -- compute sqrt
end
```



```
sqrt(↓x: float): float
require
 x >= 0
begin
 if x < 0 then
 -- handle the error
 else
 -- compute sqrt
 end -- if
end
```



## Defensives Programmieren

```
sqrt(↓x: float): float
require
 true
begin
 if x < 0 then
 -- handle the error
 else
 -- compute sqrt
 end -- if
end
```

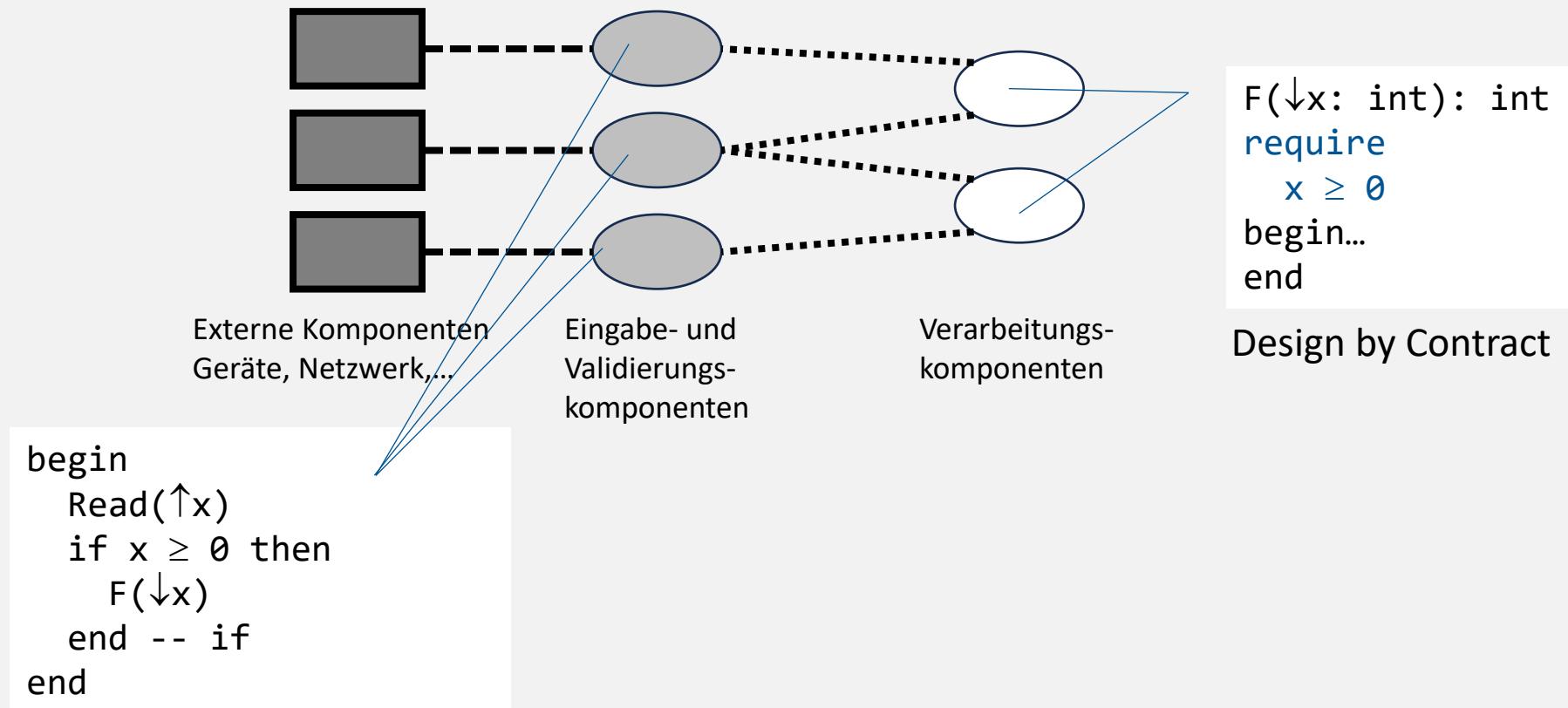


Die Vorbedingung require true wird man in der Praxis nicht explizit anschreiben.  
Keine Vorbedingung impliziert require true

# Verträge: nicht zur Eingabeprüfung verwenden

Zusicherungen (*assertions*) sind kein Mechanismus zur Eingabeüberprüfung.

- Verträge gelten zwischen einer Methode (der Klasse) und einer anderen Methode (ihrem Aufrufer), also Software-zu-Software-Kommunikation



Defensives Programmieren

# Verträge: Zusicherungen sind keine Ablaufstrukturen

---

- Zusicherungen sind keine Ablaufstrukturen (if ... then ... else)
- Sie drücken Korrektheitsbedingungen aus

**Beispiel:** Wenn Sqrt eine Vorbedingung  $x \geq 0$  hat, ist ein Aufruf mit  $x < 0$  kein Sonderfall sondern ein klarer Programmierfehler.

## Regel 1

- Eine Verletzung einer Zusicherung zur Laufzeit ist die Manifestation eines **Fehlers in der Software**.

## Regel 2

- Eine Verletzung einer Vorbedingung ist die Manifestation eines **Fehlers beim Aufrufer der Methode**
- Eine Verletzung der Nachbedingung ist die Manifestation eines **Fehlers in der Methode**

# Arbeitsweise mit Zusicherungen

Beispiel: Ein Stack, der mit einem Array implementiert wird.

```
const
 max: int = ...
type
 Stack = class
 capacity: int
 count: int
 repr: array [1:max] of →Object
 ...
 end -- Stack
```

```
Stack.Stack(↓n: int)
require
 0 ≤ n and n ≤ max
begin
 capacity := n
 count := 0
ensure
 capacity = n
 IsEmpty()
end
```

```
Stack.Push(↓e: →Object)
require
 not IsFull()
begin
 count := count + 1
 repr[count] := e
ensure
 not IsEmpty()
 count = old count + 1
 repr[count] = e
end
```

```
Stack.Pop()
require
 not IsEmpty()
begin
 count := count - 1
ensure
 not IsFull()
 count = old count - 1
end
```

# Arbeitsweise mit Zusicherungen

Beispiel: Ein Stack, der mit einem Array implementiert ist.

```
Stack.Top(): →Object
require
 not IsEmpty()
begin
 return repr[count]
end
```

```
Stack.IsEmpty()
begin
 return (count = 0)
ensure
 result = (count = 0)
end
```

```
Stack.IsFull ()
begin
 return (count = capacity)
ensure
 result = (count = capacity)
end
```

```
const
 max: int = ...
type
 Stack = class
 capacity: int
 count: int
 repr: array [1:max] of →Object
 ...
 end -- Stack
```

# Arbeitsweise mit Zusicherungen

Entwurf von Vorbedingungen: tolerant oder anspruchsvoll?

- Anspruchsvoll: Verantwortung den Klienten zuweisen  
(Bedingung erscheint als Vorbedingung)
- Tolerant: Verantwortung der Klasse zuweisen  
(Bedingung erscheint in einer bedingten Anweisung)

Bis zu einem gewissen Grad ist dies eine Frage der persönlichen Wahl

Es gibt jedoch starke Argumente für den anspruchsvollen Stil

```
Sqrt(↓x: float): float
begin
 if x < 0 then
 -- handle the error
 else
 -- compute sqrt
 end
end
```

```
Stack.Pop()
begin
 if IsEmpty() then
 WriteLn(↓"error...")
 else
 count := count - 1
 end
end
```

fehlender Kontext für eine ordnungsgemäße  
Fehlerbehandlung (im Falle einer allgemeinen Routine)

# Arbeitsweise mit Zusicherungen

---

**Reasonable Precondition Principle:** Jede Vorbedingung muss die folgenden Anforderungen erfüllen:

- die Vorbedingung erscheint in der offiziellen Dokumentation, die an die Entwickler\*innen verteilt wird
- es ist möglich, die Notwendigkeit der Vorbedingung ausschließlich anhand der Spezifikation zu beschreiben

**Precondition Availability Rule:**

- Jedes Feature in der Vorbedingung einer Methode muss für jeden Klienten verfügbar sein, dem die Methode/Klasse zur Verfügung steht

# Klasseninvarianten

---

- eine Klasseninvariante ist die Bedingung, die während der gesamten Lebensdauer der Objekte der Klasse gilt
- eine Klasseninvariante muss von allen Methoden der Klasse eingehalten werden

**Beispiel:** Eine Klasse Stack, die unter Verwendung eines Array implementiert ist.

```
const
 max: int = ...
type
 Stack = class
 capacity: int
 count: int
 repr: array [1:max] of →Object

 invariant 0 ≤ count and count ≤ capacity
 invariant IsEmpty() = (count = 0)
 invariant (count > 0) ==> repr[count] = Top()
 end -- Stack
```

# Klasseninvarianten

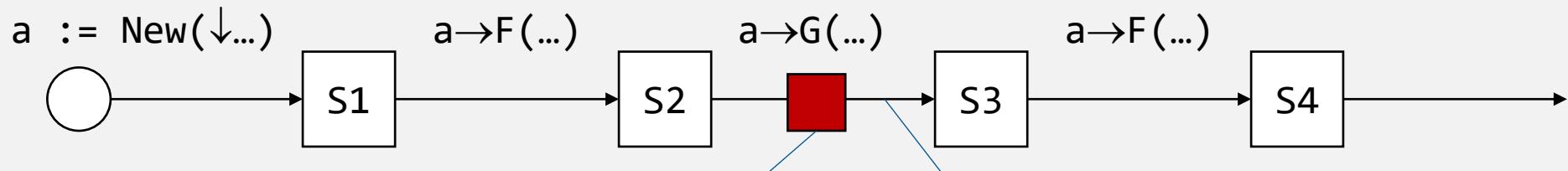
---

- eine Klasseninvariante gilt vor und nach jedem Methodenaufruf
- die Klasseninvariante wird automatisch zur Vor- und Nachbedingung jeder Methode hinzugefügt

Die Invariante ist die charakteristische Eigenschaft der Zustände, die von außen beobachtbar sind:

- der Zustand, der aus der Erstellung eines Objekts resultiert ( $S_1$ )
- die Zustände unmittelbar vor und nach einem Aufruf der Form  $a \rightarrow F(\dots)$

Die Lebensdauer eines Objekts:



In einem Zwischenzuständen während der Ausführung der Methode  $G(\dots)$  ist die Invariante nicht gültig

das ist in Ordnung, solange  $G(\dots)$  die Invariante vor dem Beenden wiederherstellt

# Wann ist eine Klasse korrekt?

---

Mit Vorbedingungen, Nachbedingungen und Invarianten können wir nun genau definieren, was es bedeutet, dass eine Klasse korrekt ist.

- Korrektheitsformel  $\{pre\} body \{post\}$
- Objektinvariante  $inv$   
 $\{pre \text{ and } inv\} body \{post \text{ and } inv\}$

Eine Klasse C ist in Bezug auf ihre Zusicherungen dann und nur dann korrekt, wenn

- für jeden gültigen Satz von Argumenten  $x_p$  für einen Konstruktor p gilt:  
 $\{\text{default}_C \text{ and } pre_p(x_p)\} body_p \{post_p(x_p) \text{ and } inv\}$
- für jede öffentliche Methode m und jede Menge gültiger Argumente  $x_m$  gilt:  $\{pre_m(x_m) \text{ and } inv\} body_m \{post_m(x_m) \text{ and } inv\}$

# Schreibweise von Zusicherungen (kleiner Ausschnitt)

---

In Zusicherungen können beliebige Bedingungen verwendet werden

- Bedingungen wie in Programmiersprachen

```
x > 0 and y > 0
```

```
a→Equals(↓b)
```

```
a = this→b
```

- Implikation  $F \Rightarrow G$ , z.B. in Klasseninvarianten

```
p ==> q
```

```
a ≠ null ==> a→Equals(↓b)
```

- Spezielle Operatoren in Nachbedingungen

```
old e
```

```
result = e
```

# Aufgabe

---

**Beispiel:** Invarianten und Methodenverträge für die Klasse Person.

```
type
 Person = class
 private
 name: string
 age: int
 isFemale: bool
 father: →Person
 mother: →Person
 public
 Person(↓name: string ↓age: int ↓isFemale: bool)
 CelebrateBirthday()
 BecomeParentTo(↓child: →Person)
 end -- class
```

```
Person.BecomeParentTo(↓child: →Person)
begin
 if isFemale then
 child→mother := this
 else
 child→father := this
 end -- if
end Person.BecomeParentTo
```

```
Person.CelebrateBirthday()
begin
 age := age + 1
end Person.CelebrateBirthday
```

```
Person.Person(↓name: string ↓age: int ↓isFemale: bool)
begin
 this→name := name
 this→age := age
 this→isFemale := isFemale
 this→father := null
 this→mother := null
end Person.Person
```

# Methoden ohne Nebeneffekte (Pure Functions)

- Methoden mit Seiteneffekten verändern den Zustand der Objekte
- Ausdrücke in Spezifikationen dürfen keine Auswirkungen auf das Objekt / auf das Programm haben
- Methoden, die in Spezifikationen aufgerufen werden, dürfen keine Nebeneffekte haben

```
type
 X = class
 ...
 Sign(↓x: int): int
 Abs(↓x: int): int
end -- X
```

```
X.Abs(↓x: int): int
begin
 if x < 0 return -x
 else return x end
end

X.Sign(↓x: int): int
begin
 if x = 0 then return 0
 elseif x > 0 then return 1
 else return -1 end
end
```

# Vererbung

---

In abgeleiteten Klassen können Methoden überschrieben werden  
Auswirkung auf die Zusicherungen

- Abgeleitete Klassen haben unterschiedliche Klasseninvarianten
- Überschriebene Methoden können unterschiedliche Vor- und Nachbedingungen haben

Relation zu den Vor- und Nachbedingungen der Basisklasse?

- Richtlinie: Eigenschaften, die von einem Objekt der Klasse T erwartet werden, sollten auch für ein Objekt einer von T abgeleiteten Klasse S gelten

# Klasseninvarianten einer abgeleiteten Klasse

---

## Beispiel.

```
type
 Table = class
 capacity: int
 count: int
 invariant count ≤ capacity
 Get(↓key: string): →Object
 Put(↓key: string ↓e: →Object)
 end -- Table
```

## Spezifikation der Methode Put

```
Table.Put(↓key: string ↓elem: →Object)
require
 “table not full”: count < capacity
ensure
 “new element in table”: Get(↓key) = e
 “count updated”: count = old count + 1
end
```

# Klasseninvarianten einer abgeleiteten Klasse

---

## Beispiel.

```
type
 HashTable = class based on Table
 end -- Table
```

- Jede Eigenschaft, die von Table erwartet wird, müssen Objekte der Klasse HashTable ebenfalls erfüllen
- Falls o den Typ HashTable hat, dann muss Invariante der Klasse Table für das Objekt o gelten (Invariante HashTable => Invariante Table)
- Neue Invariante in abgeleiteter Klasse, um den Füllfaktor der Hashtabelle zu begrenzen

```
type
 HashTable = class based on Table
 invariant count < capacity div 3
 end -- Table
```

# Methodenspezialisierung

---

Falls die Klasse HashTable die Methode Put überschreibt, dann kann...

- die neue Vorbedingung **schwächer** sein und
- die neue Nachbedingung **stärker** sein

Der Rufer der Methode

- **garantiert** nur die Vorbedingung der Klasse Table
- **erwartet** die Nachbedingung der Klasse Table

Beispiel

```
var
 t: →Table

 t := New(↓HashTable ↓100)
 t→Put(↓"Arnie" ↓New(↓Person ...))
```

# Anforderung der Methodenspezialisierung

---

Falls die Klasse B von der Klasse A abgeleitet ist und Methode M überschreibt, ist die Methode M in der Klasse B eine korrekte Methodenspezialisierung, wenn gilt

- Vorbedingung in A.M impliziert Vorbedingung B.M
- Nachbindung B.M impliziert Nachbedingung A.M

## Beispiel.

```
Table.Put(\downarrow key: string \downarrow elem: Object)
require
 count < capacity
 ...
 ...
```

```
HashTable.Put(\downarrow key: string \downarrow elem...)
require
 count < capacity div 3
 ...
 ...
```

- Die Vorbedingung `count < capacity div 3` ist keine korrekte Methodenspezialisierung, da sie nicht von `count < capacity` impliziert wird

# OCL (Object Constraint Language)

---

- ergänzt Modellierungssprachen (UML), hybride Sprache
- formale Sprache für die Definition von Zusicherungen
- fügt graphischen (UML-)Modellen präzisierte Semantik hinzu

**Beispiel** Invariante in OCL.

```
context Person
 inv: mother <> null implies mother.isFemale
```

**Beispiel** Methodenvertrag in OCL.

```
context Person::CelebrateBirthday()
 pre:
 mother <> null implies self.age < father.age - 1 and
 father <> null implies self.age < mother.age - 1
 post:
 self.age = self.age@pre + 1
```

# Bewertung

---

Design by Contract ist ein pragmatischer Ansatz, um Klassen Code nah zu spezifizieren

- ein Vertrag auf der Basis von Zusicherungen definiert genau, was die Klasse leistet, und erleichtert damit eine korrekte Nutzung und Wiederverwendung
- die Klasse lässt sich auch leichter testen, weil sich aus den Zusicherungen Testfälle für Normal- und Sonderfälle direkt ableiten lassen
- das Prüfen von Zusicherungen zur Laufzeit erleichtert Fehlersuche

# 17.17 Quantitative Beurteilung des OO-Konstruktionsansatzes

---

(im Vergleich mit konventioneller Programmierung)

## Auswirkungen auf den Umfang des Programmcodes

|                                   |             |
|-----------------------------------|-------------|
| zu schreibender Code <sup>1</sup> | 25 – 50 %   |
| Gesamter Programmcode             | 120 – 300 % |

<sup>1</sup> Ausnutzung von Klassenbibliotheken und Application Frameworks, abhängig vom Umfang und Sortiment von Halbfabrikaten

## Auswirkungen auf die Laufzeit

|                                  |              |
|----------------------------------|--------------|
| Methodenaufruf                   | 105 – 150 %  |
| Zugriff auf Datenkomponenten     | 100 – 1000 % |
| Automatische Speicherbereinigung | 105 – 150 %  |
| Gesamtlaufzeit                   | 80 – 120 %   |

# Quantitative Beurteilung des OO-Konstruktionsansatzes

---

## Auswirkungen auf den Speicherbedarf

|                    |             |
|--------------------|-------------|
| Programmcode       | 120 – 200 % |
| Methodentabellen   | 101 – 110 % |
| Dynamische Objekte | 100 – 120 % |

## Auswirkungen auf die Entwicklungszeit

|                  |           |
|------------------|-----------|
| Entwicklungszeit | 10 – 70 % |
|------------------|-----------|

# Quantitative Beurteilung des OO-Konstruktionsansatzes

---

## Auswirkungen auf den Projektverlauf

- + Planbarkeit
- + Organisationsaufwand
- + Entwurf und Implementierung
- + Arbeitsteilung
- + Prototyping
- Einarbeitung
- Verwaltung von Klassenbibliotheken
- Kostenbewertung

# Quantitative Beurteilung des OO-Konstruktionsansatzes

---

## Zwei Fallbeispiele

### (1) Implementierung eines Prototyping-Werkzeuges

|                       | Konventionell<br>(M2) | Objektorientiert<br>(C++) |
|-----------------------|-----------------------|---------------------------|
| Neu entw. Code (LOC)  | 12500                 | 3300                      |
| Prozeduren/Methoden   | 230                   | 171                       |
| Lines per Proz./Meth. | 48                    | 19                        |
| Wiederv. Code (LOC)   | 2200                  | 6800                      |

# Quantitative Beurteilung des OO-Konstruktionsansatzes

---

## Zwei Fallbeispiele

### (2) Implementierung einer verteilten Prozesssteuerung

| Systemkomponenten      | rein objektorientiert   |                     |                 |
|------------------------|-------------------------|---------------------|-----------------|
|                        | projektinvar.<br>Klasse | projektb.<br>Klasse | Fremd-<br>fabr. |
| Prozesskoord./-strukt. | 17                      | 11                  | 60              |
| Kommunikation          | 24                      | 0                   | 60              |
| Ben.Schnittst./Leitst. | 19                      | 13                  | 100             |
| Steuerungskomp.        | 1                       | 10                  | 60              |
| Datenhaltung           | 5                       | 8                   | 60              |
| Datenstruktur          | 21                      | 17                  | 60              |
|                        | 87                      | 59                  | 160             |

# 17.18 Ein Metamodell für die OO Systemkonstruktion

---

