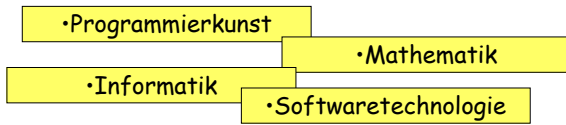


## Abstrakte Datentypen

Implementation  
 Prototypen  
 Spezifikation  
 Formale Beschreibungen  
 Abstrakte Modelle



## ABSTRAKTE DATENTYPEN

Beobachtung: Unterschiedliche Realisierungen  
 (Darstellung, Algorithmen, Fehlerbehandlung, ...)  
 für gleiche Aufgaben

Beispiel: Prolog-Arithmetik für natürliche Zahlen

– Mittels Deklarationen

```

    add(0, X, X).
    add(s(X), Y, s(Z)) :- add(X, Y, Z).

    mult(0, X, 0).
    mult(s(X), Y, s1) :- mult(X, Y, s11), add(s11, Y, s1).

    pot(s(X), Y, s1) :- pot(X, Y, s11), mult(s11, Y, s1).
    
```

– Mittels Prolog-Arithmetik

## Mögliche Implementationen

```

    ?- nachfolger(a, b).
    ?- nachfolger(a, Nachfolger).
    ?- nachfolger(Vorgänger, b).
    ?- nachfolger(Vorgänger, Nachfolger).
    
```

```

    nachfolger(X, Y) :- integer(X), !, Y is X+1.
    nachfolger(X, Y) :- integer(Y), !, X is Y-1.
    nachfolger(0, 1).
    nachfolger(X, Y) :- nachfolger(U, V), X is U+1, Y is V+1.
    
```

Fehlermöglichkeiten z.B.:

- Weder Zahl noch Variable
- Negative Zahl

## Mögliche Implementationen

Argumente: Natürliche Zahl, Variable, sonstiges

```

    natural(X) :- integer(X), X >= 0.

    nachfolger(X, Y) :- natural(X), natural(Y), !, Y is X+1.
    nachfolger(X, Y) :- natural(X), var(Y), !, Y is X+1.
    nachfolger(X, Y) :- var(X), natural(Y), !, X is Y-1.
    nachfolger(0, 1) :- var(X), var(Y).
    nachfolger(X, Y) :- var(X), var(Y), nachfolger(U, V), X is U+1, Y is V+1.
    nachfolger(X, Y) :- write('nicht im Bereich der natürlichen Zahlen').
    
```

(es gibt noch weitere Ausnahmen ....)

• „Exceptions“: Programmiersprache/-umgebung stellt Mittel für Ausnahmebehandlung bereit

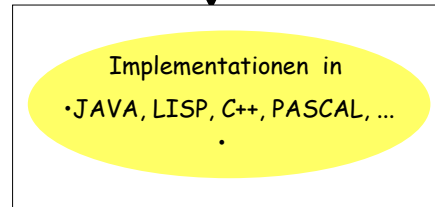
## Weitere Implementationen

Konvertiere Dezimalzahl  $n$  in  $s(\dots s(o)\dots)$

`nachfolger(X, s(X))`

Konvertiere  $s(\dots s(o)\dots)$  in Dezimalzahl  $n$

## Weitere Implementationen



**Abstrakte Datentypen:**

- Daten-Formate und Prozeduren in allgemeiner (mathematischer) Form
- Lösung von konkreten Details (Datenkapselung)

## Abstrakte Datentypen

### Mathematische Modelle:

- Strukturen (Definitionen)
- Algorithmen
- Komplexitätsaussagen

### •Listen

- Sortierverfahren

### •Relationen

- Prädikatenlogik

### Datenkapselung:

- Interface
- Laufzeiteigenschaften

### •Datenbank

### •Prolog-Modul

### •Java-Klasse

Implementation in ... von ... Version ...

## Abstrakte Datentypen

### Verallgemeinerung primitiver Datentypen

- boolean, integer, char, ...

### Strukturen mit Operationen

- Mengen
- Relationen
- Liste (endliche Folge)
- Keller
- Warteschlangen
- Graph
- Baum
- ...

Mathematisches Modell  
Algorithmen

Abstrakter Datentyp  
→ Funktionalität

Datenstruktur  
→ Programm

## Liste

### Definitionen:

Länge  $n$ : Anzahl der Elemente

Liste als Funktion:

$$L: \{1, \dots, n\} \rightarrow M \quad M = \text{Menge der (möglichen) Elemente}$$

Liste als Aufzählung:  $[L(1), L(2), \dots, L(n)]$

### Geordnete Listen, z.B.:

- Zahlen in aufsteigender Reihenfolge
- Wörter in lexikographische Anordnung

- Geordnete Liste bzgl. Ordnungsrelation  $R \subseteq M \times M$

Liste  $L$  ist *geordnet bzgl.  $R$* ,

falls gilt:  $\forall i, j \in \{1, \dots, n\}: i < j \rightarrow R(L(i), L(j))$

## Liste

### Operationen:

Zugriff auf Elemente :

- Suchen, Einfügen, Löschen von Elementen

Bearbeitung von Listen :

- Verketteten
- Reorganisation, Sortieren, ...

Sortierverfahren erzeugen eine Reihenfolge der Listenelemente entsprechend einer vorgegebenen Ordnungsrelation  $R$ .  
Unterschiedliche Sortierverfahren haben unterschiedliche Komplexität.

- Quick-Sort
- Selection-Sort
- Merge-Sort
- Heap-Sort
- Bubble-Sort
- ...

## Listen

### Implementation als Datenstruktur:

Organisation bzgl. Speicherorganisation

Felder

Referenzen

- verkettete Liste: Referenz auf Nachfolger-Element
- Doppelt verkettete Liste

Rekursiv (Prolog)

Jeweils mit Methoden für Einfügen, Verketteten, ...

Geschichte Wahl  
der Datenstruktur

## Listen-Operationen: member

Iterative Implementation für verkettete Liste

```
boolean lookup(List L, Element x)
```

```
E = L;
while (true)
{ if (E == NIL) {return false;}
  if (E.Value == x) {return true;}
  E = E.Next;
}
```



## Listen-Operationen: member

Rekursive Implementation für rekursive Liste

```
boolean lookup(List L, Element x)
{ if ( Head(L) == x ) { return true;}
  if ( Tail(L) == NIL ) { return false;}
  return lookup(Tail(L),x);
}
```

```
member(X, [X | T]) :- !.
member(X, [_ | T]) :- member(X, T).
```

## Listen-Operationen: append

Iterative Implementation für verkettete Liste

```
Liste concatenate(Liste L1, Liste L2)
{ E = L1;
  while (E.Next != NIL) {E=E.Next;}
  E.Next = L2; return L1;
}
```

Rekursive Implementation für rekursive Liste

```
append([], L, L).
append([X | L1], L2, [X | L3]) :- append(L1, L2, L3).
```

## Streichen von Elementen

```
delete(X, [], []).
delete(X, [X | L], Deleted) :- delete(X, L, Deleted).
delete(X, [_ | L], [Y | Deleted]) :- X \= Y, delete(X, L, Deleted).
```

Streicht **alle** Vorkommen von X aus L.

## Differenz-Listen

Andere Repräsentation von Listen:

```
Liste als Anfangsstück [l1,...,ln]
einer längeren Liste [l1,...,ln | r1,...,rm],
vermindert um den Rest [r1,...,rm].
```

Schreibweise z.B. [l<sub>1</sub>,...,l<sub>n</sub>, r<sub>1</sub>,...,r<sub>m</sub>] - [r<sub>1</sub>,...,r<sub>m</sub>]

```
Rest [r1,...,rm] dabei beliebig,
z.B. auch [l1,...,ln | []] - []
allgemeinste Form: [l1,...,ln | R] - R
```

```
conv_DiffList_to_List(A-[ ],A).
```

```
conv_List_to_DiffList([ ],L-L).
```

```
conv_List_to_DiffList([X|L1], [X|L2]-L) :- conv_List_to_DiffList(L1, L2-L).
```

## Differenz-Listen

```
Liste als Anfangsstück [l1,...,ln]
einer längeren Liste [l1,...,ln | r1,...,rm],
vermindert um den Rest [r1,...,rm].
```

Verkettung von Differenz-Listen in einem Schritt:

```
append_dl(A-B, B-C, A-C).
```

A:  $a_1, \dots, a_n, b_1, \dots, b_m, c_1, \dots, c_s$   
 $\frac{A - B}{A - C}$

B:  $b_1, \dots, b_m, c_1, \dots, c_s$   
 $\frac{B - C}{B - C}$

C:  $c_1, \dots, c_s$

```
?- append_dl([a,b,c|R1] - R1, [1,2|R2] - R2, L).
```

```
L = [a, b, c, 1, 2|_G173] - G173
```

## Differenz-Listen

Umkehrung einer Liste:

```
naive_reverse([ ],[ ]) . Quadratische Zeit
naive_reverse([H | T], L) :- naive_reverse(T,L1), append(L1,[H], L).
```

```
reverse_dl([ ], L - L) . Lineare Zeit
reverse_dl([H | T], L - S) :- reverse_dl(T,L - [H|S]).
```

```
reverse(X, Y) :- reverse_dl(X,Y - []).
```

Listen-Subtraktor S  
als Akkumulator

```
reverse_acc([ ],R, R).
reverse_acc([H | T], Acc, R) :- reverse_acc(T, [H | Acc], R).
reverse(L,R) :- reverse_acc(L,[],R).
```

## Quicksort

```
quicksort([H | T], Sorted) :-  
  partition(T, H, L, B),  
  quicksort(L, SortedL),  
  quicksort(B, SortedB),  
  append(SortedL, [H|SortedB], Sorted).  
quicksort([], []).
```

```
partition([A | M], H, [A | Ls], Bs) :- A <= H, partition(M, H, Ls, Bs).  
partition([A | M], H, Ls, [A | Bs]) :- A > H, partition(M, H, Ls, Bs).  
partition([], H, [], []).
```

## Quicksort mit Differenz-Liste

```
quicksort_dl([H | T], Sorted - S) :-  
  partition(T, H, L, B),  
  quicksort_dl(L, SortedL - [H|B]),  
  quicksort_dl(B, B - S).  
quicksort_dl([], L - L).
```

```
quicksort(X, Y) :- quicksort_dl(X, Y - []).
```

Verwendet wie vorher:

```
partition([A | M], H, [A | Ls], Bs) :- A <= H, partition(M, H, Ls, Bs).  
partition([A | M], H, Ls, [A | Bs]) :- A > H, partition(M, H, Ls, Bs).  
partition([], H, [], []).
```

## Komplexität: Laufzeit

- Zeit-Messungen abhängig von Eingabewerten  
"worst case" vs. "average case"
- o *Benchmarks*:  
Vergleich von Algorithmen/Programmen
  - o *Profiling*:  
Analyse des Zeitverhaltens im Programm
  - o *Programm-Analyse*:  
Anzahl "elementarer" Anweisungen, Abschätzungen für  
Schleifen-Durchläufe bzw. Verzweigungen

**90-10-Regel:**  
90% der Laufzeit in 10% des Codes verbraucht

## Komplexitäts-Betrachtungen

- Programm  $\pi$  auf Maschine  $M$  benötigt  
 $m$  Operationen und  $n$  Byte Speicher  
andere Maschine  $M'$  simuliert  $M$ :  
pro Operation von  $M$  benötigt  $M'$  maximal:  
 $a$  Operationen und  $b$  Byte Speicher  
 $\Rightarrow \pi$  auf  $M'$  benötigt maximal  
 $a \cdot m$  Operationen und  $b \cdot n$  Byte Speicher
- $\Rightarrow$  Abstraktion von konkreter Maschine  
(Unterschied: konstanter Faktor  $a$  bzw.  $b$ )  
Bezug auf universelle Maschine, z.B. TURING-Maschine

## Komplexitäts-Betrachtungen

Allgemeine Aussagen bezogen auf **Problem-Umfang**:  
Anzahl der Variablen, Länge eines Ausdrucks, ...

**O-Notation** (vgl. Landau-Symbole)  
obere Schranke für Größen-Ordnung von Funktionen:

$T_\pi(n)$  sei Komplexität von  $\pi$  bei Problemgröße  $n$ .  
Programm  $\pi$  hat Komplexität  $O(f(n))$ ,  
falls eine feste (!) Konstante  $c$  existiert,  
so daß für **fast alle** Problemgrößen  $n$  gilt:  $T_\pi(n) \leq c \cdot f(n)$

„fast alle“ = alle außer endlich viele  
(d.h. alle ab einer festen Zahl  $n_0$ )

## Zeit-Komplexität

- Erfüllbarkeit eines AK-Ausdrucks mit  $n$  Variablen:  
Zeit-Komplexität  $O(2^n)$  Wertberechnungen  
*Branch-and-bound*-Verfahren für Suche nach kürzestem Weg  
in einem Graphen mit  $n$  Knoten und mit  $m$  Kanten:  
Zeit-Komplexität  $O(n^2)$   
andere Abschätzung mit  $r = \text{Max}(m, n)$ :  
Zeit-Komplexität  $O(r \cdot \log(n))$
- Suche in einer geordneten Liste der Länge  $n$ :  
Zeit-Komplexität  $O(\log(n))$
- Sortieren einer geordneten Liste der Länge  $n$ :  
Zeit-Komplexität  $O(n \cdot \log(n))$
- Quicksort:  $O(n^2)$ , im Mittel  $O(n \cdot \log(n))$
- `append(L1, L2, L)` lineare Zeit bzgl. Länge von  $L1$
- `naive_reverse(L, R)` quadratische Zeit bzgl. Länge von  $L$

## Exponentielle vs. polynomiale Komplexität

n	$n^2$	$n^3$	$2^n$
10	100	1000	1024 ( $\sim 10^3$ )
100	10000	1000000	$\sim 10^{30}$
1000	1000000	1000000000	$\sim 10^{300}$

bei Komplexität  $2^n$ :  
Steigerung der Rechenleistung um Faktor 1000  
ermöglicht Steigerung der Problemgröße von  $n$  auf  $n+10$

## Speicher vs. Zeit

Suche in einer geordneten Liste der Länge  $n$ :  
Zeit-Komplexität  $O(\log(n))$

mit spezieller Datenstruktur („Index“)

für Zugriff auf Liste, z.B. als binäre Baum

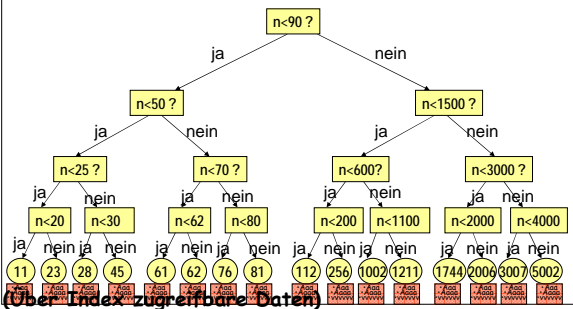
Allgemein:

Speicher-Zeit-„Trade-Off“

[11,23,28,45, 61,62,76,81, 112,256,1002,1211, 1744,2006,3007,5002]

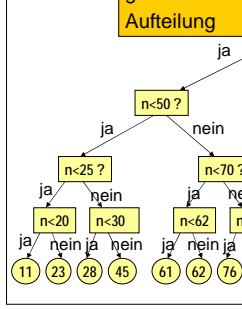
## Indexstruktur „Binärer Suchbaum“

[11,23,28,45, 61,62,76,81, 112,256,1002,1211, 1744,2006,3007,5002]



## Indexstruktur

[11,23,28,45,



$\log(n)$  Tests  
bei günstig gewählter Aufteilung

```

if n < 90 then
  if n < 50 then
    if n < 25 then return 11
    else return 23;
  else
    if n < 30 then return 28
    else return 45;
else
  if n < 70 then
    if n < 62 then return 61
    else return 62;
  else
    if n < 80 then return 76
    else return 81;
else
  if n < 1500 then
    if n < 600 then
      if n < 200 then return 112
      else return 256;
    else
      if n < 1100 then return 1002
      else return 1211;
  else
    if n < 3000 then
      if n < 2000 then return 1744
      else return 2006;
    else
      if n < 4000 then return 3007
      else return 5002;
  
```

## Keller (Stapel, Stack, LIFO)

Liste  $K = [K(1), K(2), \dots, K(n)]$  mit beschränktem Zugriff

### Operationen:

pop: liefert oberstes Element  $K(1)$   
entfernt oberstes Element:  $K' = [K(2), \dots, K(n)]$   
(Fehler bei leerem Keller)

push(x): speichert Element  $x$  als oberstes Element:  
 $K' = [x, K(1), K(2), \dots, K(n)]$  (Fehler bei vollem Keller)

isEmpty: "true", falls Keller leer

isFull: "true", falls Keller voll

clear: Keller leeren

## Keller

Anwendungen

Suchverfahren in Graphen (später mehr dazu):

Tiefe-Zuerst-Suche, Back-Tracking;

Speicherung der offenen Knoten (Liste OPEN)

Textverarbeitung: „rückgängig“-Befehle

Syntax-Analyse von Programmen

Auswertung von Ausdrücken

Laufzeitkeller

## Auswertung von Ausdrücken

Darstellung in  
umgekehrt polnischer Notation  
(postfix-Darstellung)

$$x + y * (z - x) + 12 * x \\ = x y z x - * + 12 x * +$$

```
push(x)
push(y)
push(z)
push(x)
push(pop - pop)
push(pop * pop)
push(pop + pop)
push(12)
push(x)
push(pop * pop)
push(pop + pop)
```

## Prozedur-Keller

Prozedur-Segmente enthalten:

Resultat

Parameter

Lokale Variable

Rücksprung-Adresse

```
int fak(int n)
{ if (n==0) { return 1;}
  else { return n * fak(n-1); }
}
```

Push bei Aufruf einer Prozedur

Pop bei Beendigung einer Prozedur

## Keller: Einsatz für Handlungsplanung

Push:

Planschritte in umgekehrter Reihenfolge ablegen.

Pop:

Nächsten Planschritt bearbeiten:

- Einfache Aktion ausführen bzw.
- (Verfeinerten) Teilplan zur Umsetzung des Planschritts einkellern

## Warteschlangen (Queue, FIFO)

Liste  $Q = [ \alpha(1), \alpha(2), \dots, \alpha(n) ]$  mit beschränktem Zugriff

**Operationen:**

**dequeue:** liefert erstes Element  $Q(1)$   
entfernt erstes Element:  $Q' = [ \alpha(2), \dots, \alpha(n) ]$   
(Fehler bei leerer Schlange)

**enqueue(x) :**  
speichert Element  $x$  als letztes Element:  
 $Q' = [ \alpha(1), \alpha(2), \dots, \alpha(n), x ]$   
(Fehler bei voller Schlange)

isEmpty, isFull, clear ...

## Warteschlangen (Queue, FIFO)

**Implementation:**

- verkettete Liste
- "Ring-Array" mit Positionsreferenz
- front: aktuell erstes Element
- rear: aktuell letztes Element

**Anwendung:**

- Suchverfahren in Graphen
- Breite-Zuerst-Suche:
- Speicherung der offenen Knoten (Liste *OPEN*)

## Graphen

**Definitionen**

Ein (*gerichteter*) Graph ist ein Paar  $G = [ V, E ]$  mit einer Menge  $V$  von *Knoten* ("vertex", "node") und einer Menge  $E \subseteq V \times V$  von Kanten ("edge"):

"Die Kante  $[v_1, v_2]$  führt von  $v_1$  nach  $v_2$ ."

Allgemeiner (Mehrfachkanten):

$G = [ V, E, f ]$  mit  $f: E \rightarrow V \times V$  (Inzidenz-Funktion)

2. Bei einem *ungerichteten* Graphen  $G = [ V, E ]$  ist die Relation  $E$  symmetrisch:

$[v_1, v_2] \in E \leftrightarrow [v_2, v_1] \in E$

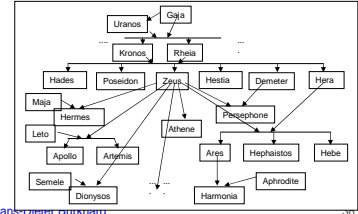
## Graphen

- Knoten-beschrifteter Graph:** 4-Tupel  $G = [V, E, A, \alpha]$  mit  $[V, E]$  ist ein Graph,  $A$  ist eine Menge (von Beschriftungen),  $\alpha$  ist ein Funktion  $\alpha: V \rightarrow A$
- Kanten-beschrifteter Graph:** 4-Tupel  $G = [V, E, B, \beta]$  mit  $[V, E]$  ist ein Graph,  $B$  ist eine Menge (von Beschriftungen),  $\beta$  ist ein Funktion  $\beta: E \rightarrow B$
- Beschrifteter Graph:** 6-Tupel  $G = [V, E, A, \alpha, B, \beta]$  mit  $[V, E, A, \alpha]$  ist ein Knoten-beschrifteter Graph,  $[V, E, B, \beta]$  ist ein Kanten-beschrifteter Graph.

## Darstellungsformen

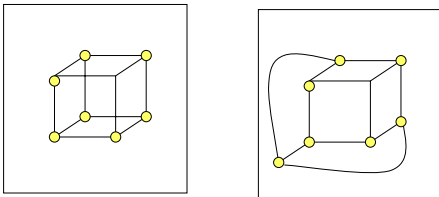
Graphisch:

Knoten (z.B.) als Kreise (mit Knotennamen)  
Kanten als Bögen bzw. Pfeile (mit Kantennamen)  
Beschriftungen an Kreisen bzw. Bögen



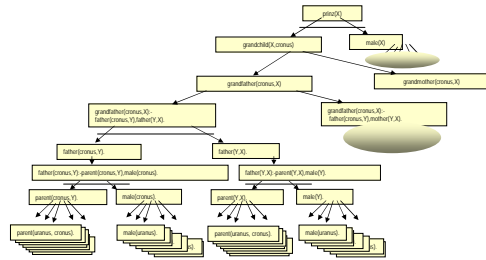
## Darstellungsformen

Planarer Graph: 2-D-Darstellung ohne Kreuzung von Kanten

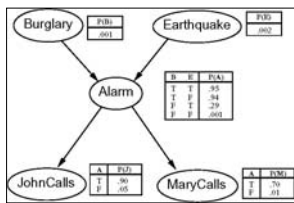


## Beweisbaum

2 Sorten von Knoten: „bipartiter Graph“  
an Und-Verzweigungen bzw. Oder-Verzweigungen

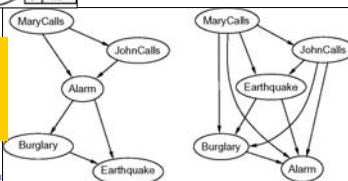


## Kausale Abhängigkeiten: Belief-Netze

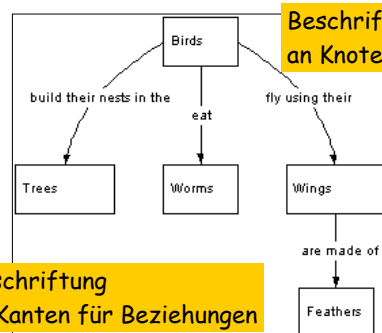


Beschriftung  
an Knoten für bedingte  
Wahrscheinlichkeiten

Unterschiedliche  
Ansätze für  
Abhängigkeiten

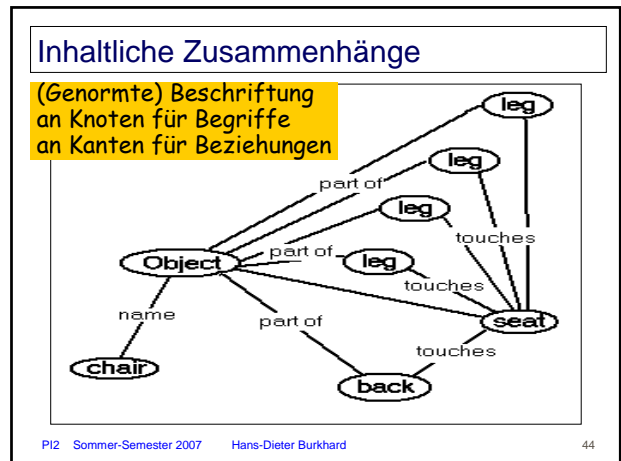
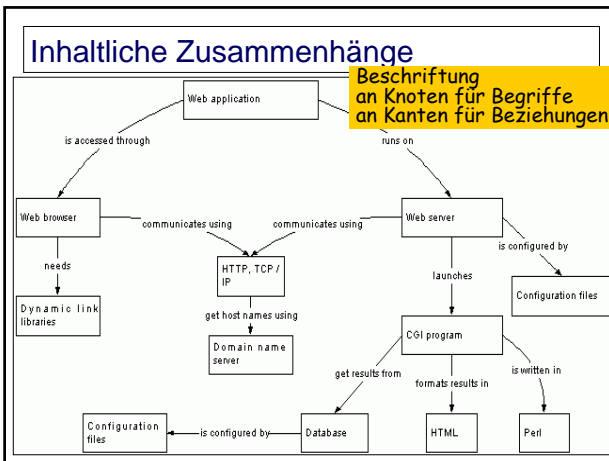


## Sprachliche Zusammenhänge: Semantische Netze



Beschriftung  
an Knoten für Begriffe

Beschriftung  
an Kanten für Beziehungen



### UML: Unified Modeling Language

- Graphische Notationen für Objektorientierte Modellierung
- Hilfsmittel für Erzeugung von Code
- Hilfsmittel für Dokumentation und Test

Graphische Darstellungen für ...

- Anwendungsfall-Beschreibung (use-case)
- Klassen/Objekte (Struktur)
  - Vererbung
  - Assoziationen: Beziehungen
  - Aggregationen: Bestandteile („Container“)
- Sequenzdiagramm (Ablauf von Interaktionen)
- Zustandsdiagramm (Ablauf innerhalb eines Objekts)
- Aktivitätsdiagramm (Zusammenarbeit) ... und vieles mehr

