

Vorlesung Software-Reengineering

Prof. Dr. Rainer Koschke

Arbeitsgruppe Softwaretechnik
Fachbereich Mathematik und Informatik
Universität Bremen

Wintersemester 2005/06

Überblick I

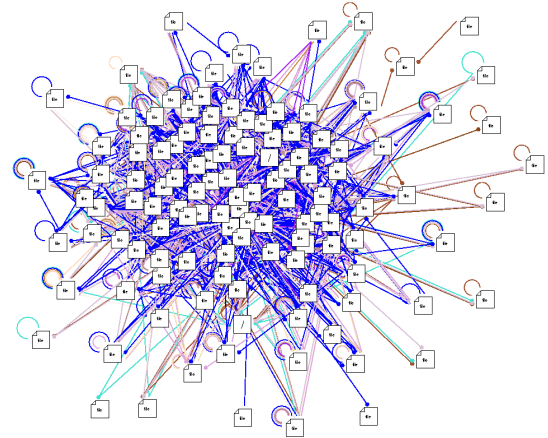
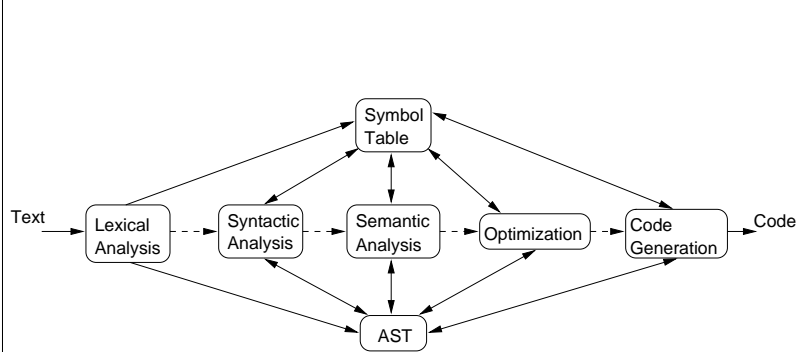
- 1 Strukturelle Architektursichten

1 Strukturelle Architektursichten

- Reflektionsmethode
- Software-Clustering
- Wiederholungsfragen

Lernziele

- strukturelle Sicht von Software-Architektur kennen
- strukturelle Sicht rekonstruieren
 - hypothesengesteuert
 - automatisches Clustering



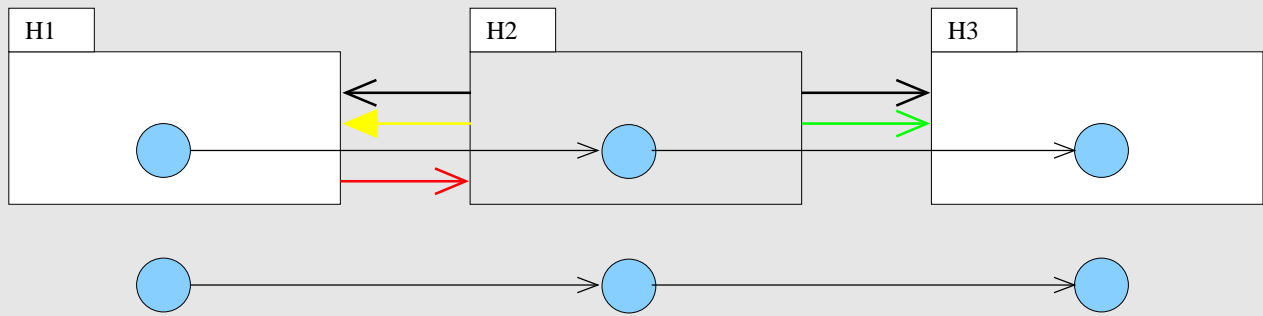
Reflektionsmethode von Murphy u. a. (1995, 2001)

Ziele der Reflektionsmethode:

- hypothesengetriebene Architekturrekonstruktion
- Architekturvalidierung

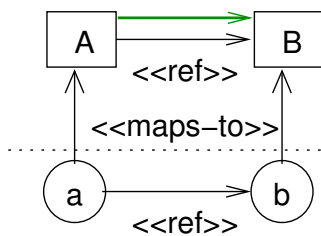
- ① Stelle Architekturmodell auf
- ② Extrahiere Implementierungsmodell
- ③ Bilde Modelle aufeinander ab
- ④ Berechne Reflektionsmodell
- ⑤ Verfeinere/korrigiere

Example



Reflektionsmethode

convergence



hypothesized
module
view

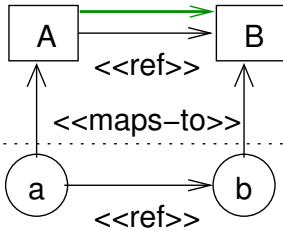
concrete
module
view

$$\begin{aligned}
 \text{propagated-ref}(A, B) &\Leftrightarrow \exists(a, b \in M) : (\text{ref}(a, b) \\
 &\quad \wedge \text{maps-to}(a) = A \\
 &\quad \wedge \text{maps-to}(b) = B)
 \end{aligned}$$

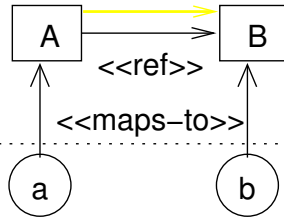
$$\text{convergence}(A, B) \Leftrightarrow \text{ref}(A, B) \wedge \text{propagated-ref}(A, B)$$

Reflektionsmethode

convergence



absence

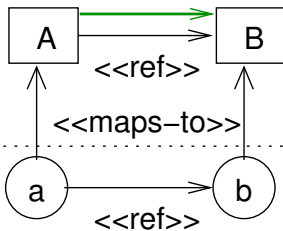


hypothesized
module
view
concrete
module
view

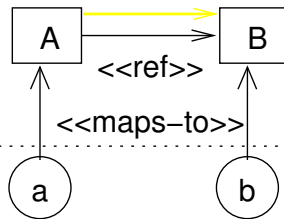
$$absence(A, B) \Leftrightarrow ref(A, B) \wedge \neg propagated-ref(A, B)$$

Reflektionsmethode

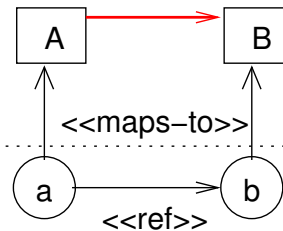
convergence



absence



divergence

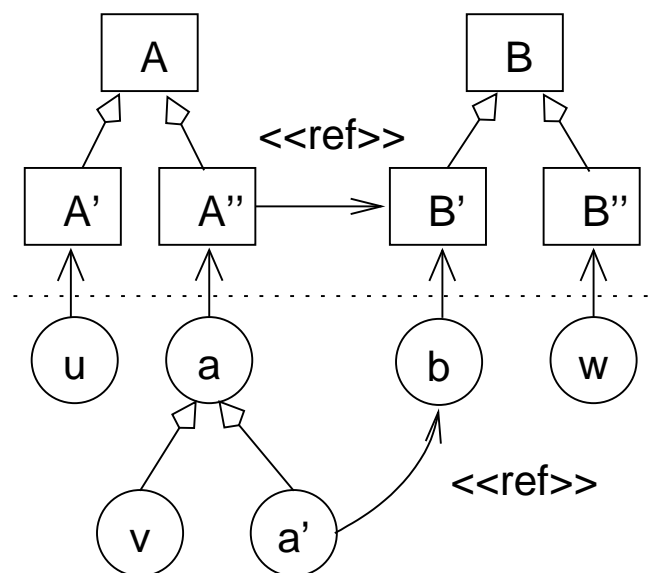


hypothesized
module
view
concrete
module
view

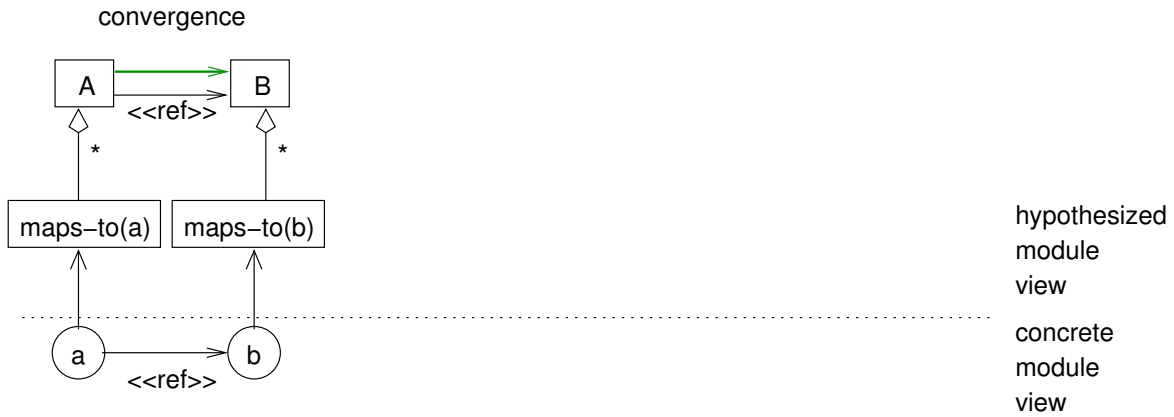
$$divergence(A, B) \Leftrightarrow \neg ref(A, B) \wedge propagated-ref(A, B)$$

- große Systeme werden hierarchisch in Subsysteme zerlegt
 - die originale Reflektionsmethode erlaubt keine hierarchische hypothetische Sicht
- Erweiterung von Koschke und Simon (2003)

Hierarchische Reflektionsmethode: Inkonsistentes Modell?

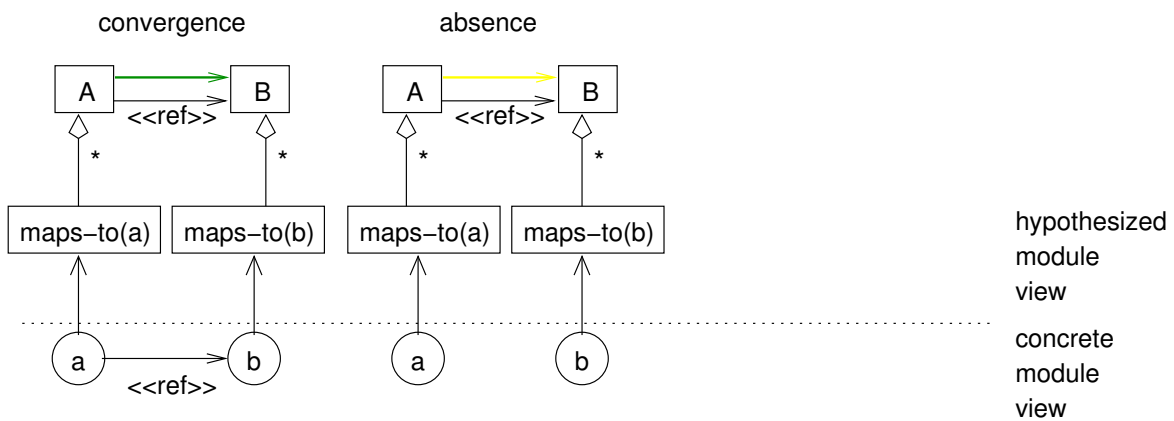


Hierarchische Reflektionsmethode



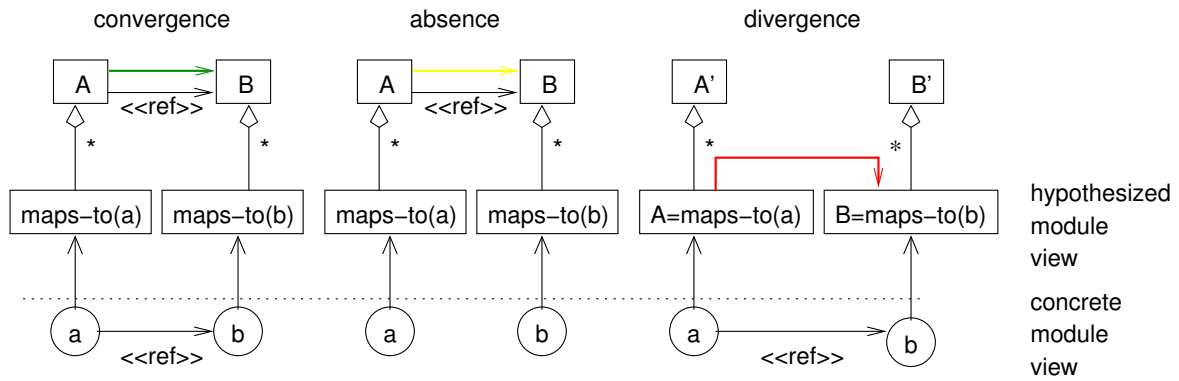
$$\begin{aligned}
 \text{propagated-ref}^\uparrow(A, B) &\Leftrightarrow \exists(a, b \in M) : (\text{ref}(a, b) \\
 &\quad \wedge \text{partof}^*(\text{maps-to}(a), A) \\
 &\quad \wedge \text{partof}^*(\text{maps-to}(b), B)) \\
 \text{convergence}(A, B) &\Leftrightarrow \text{ref}(A, B) \wedge \text{propagated-ref}^\uparrow(A, B)
 \end{aligned}$$

Hierarchische Reflektionsmethode



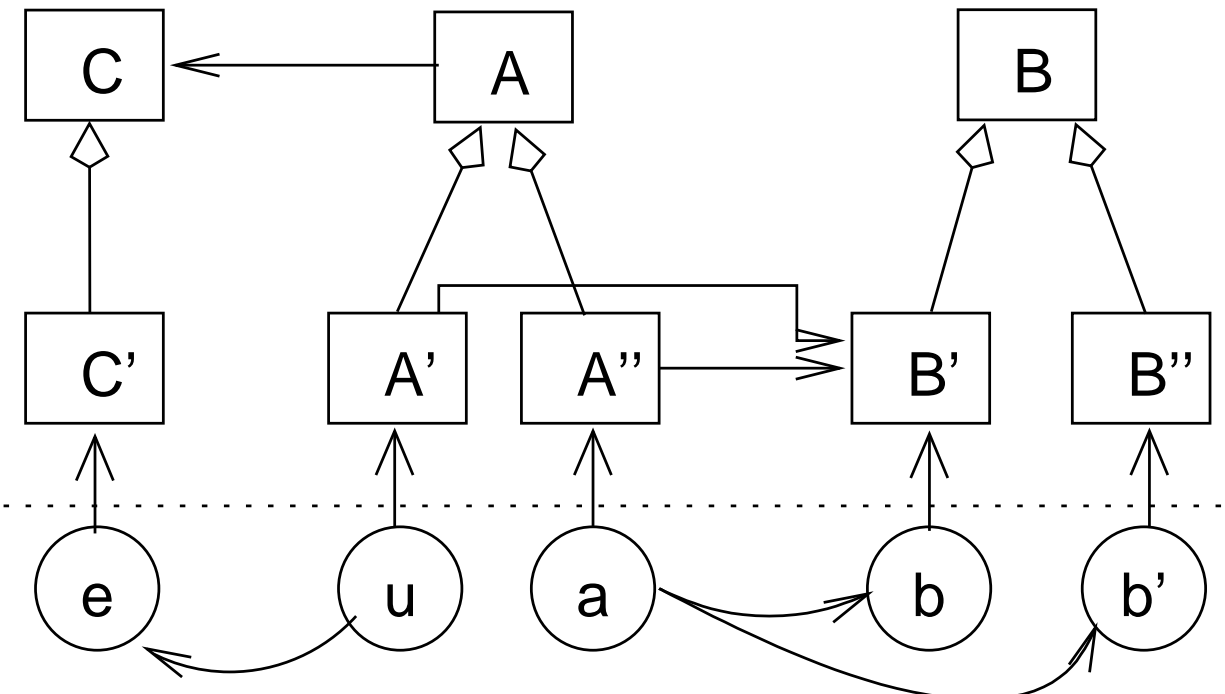
$$\text{absence}(A, B) \Leftrightarrow \text{ref}(A, B) \wedge \neg \text{propagated-ref}^\uparrow(A, B)$$

Hierarchische Reflektionsmethode

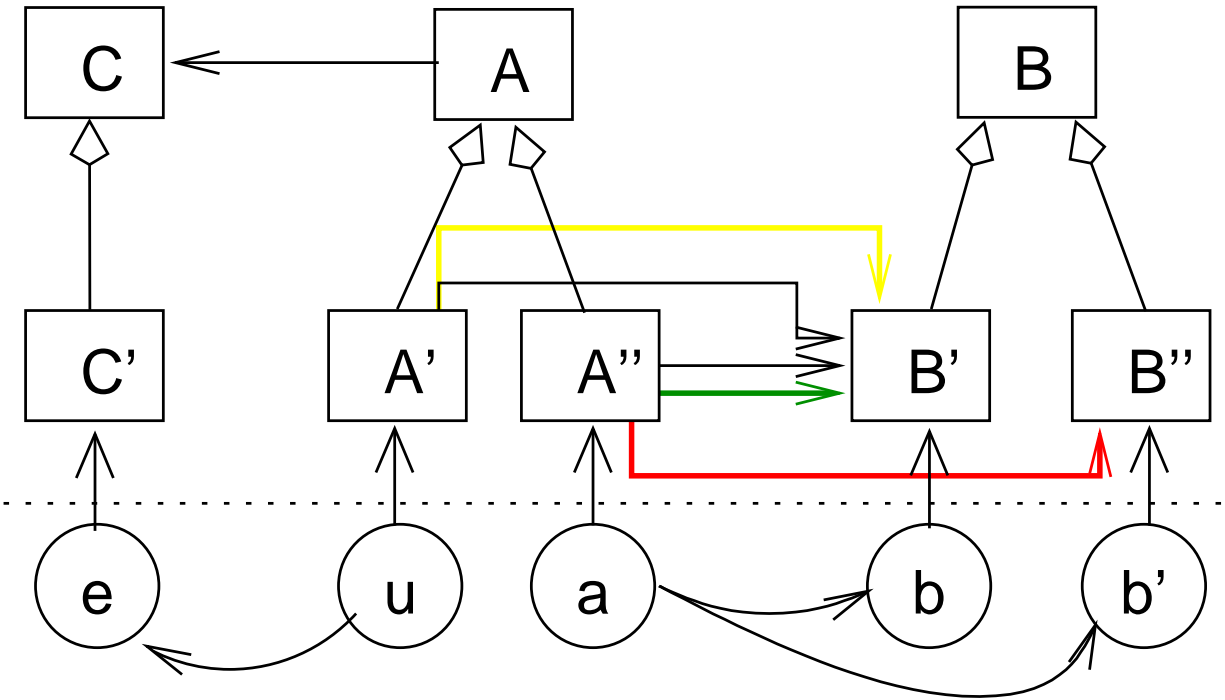


$$\begin{aligned}
 \text{divergence}(A, B) \Leftrightarrow & \neg \exists (A', B') : (\text{partof}^*(A, A') \\
 & \wedge \text{partof}^*(B, B') \\
 & \wedge \text{ref}(A', B')) \wedge \text{propagated-ref}(A, B)
 \end{aligned}$$

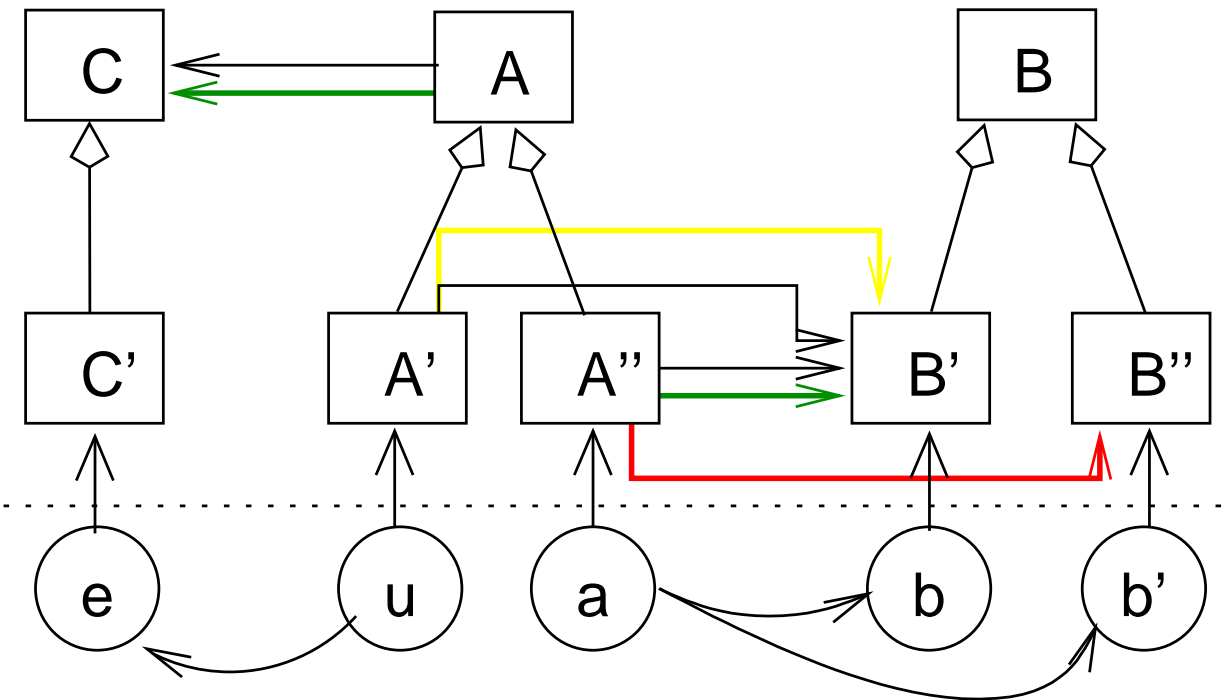
Beispiel

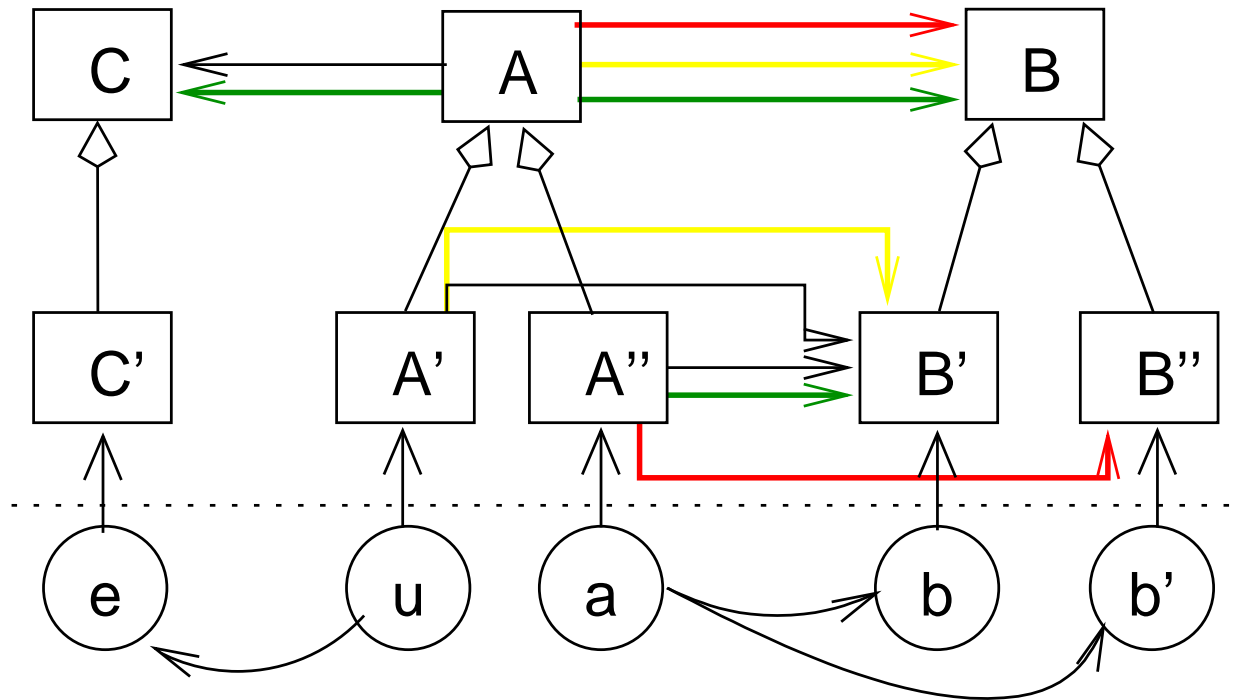


Beispiel



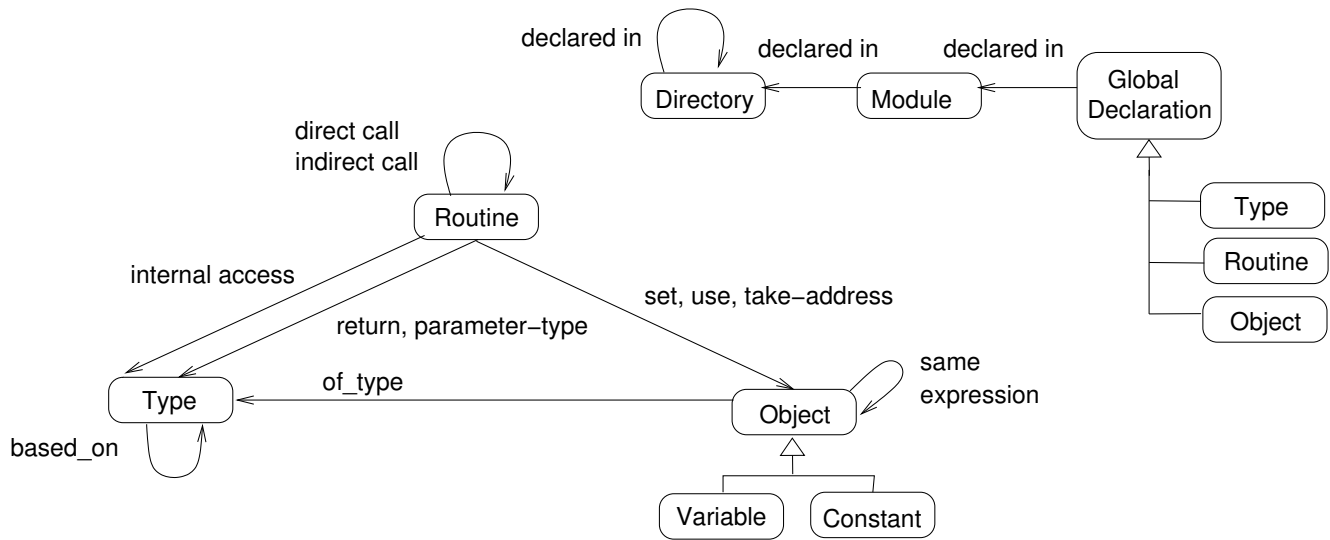
Beispiel



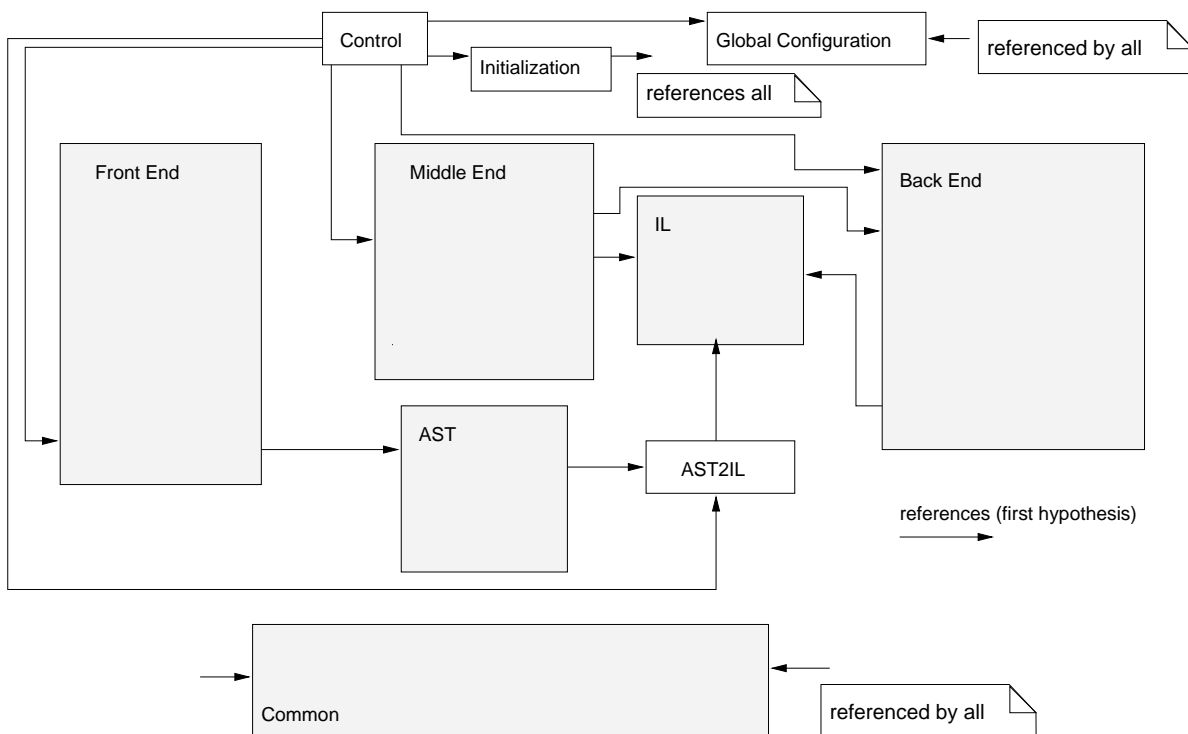


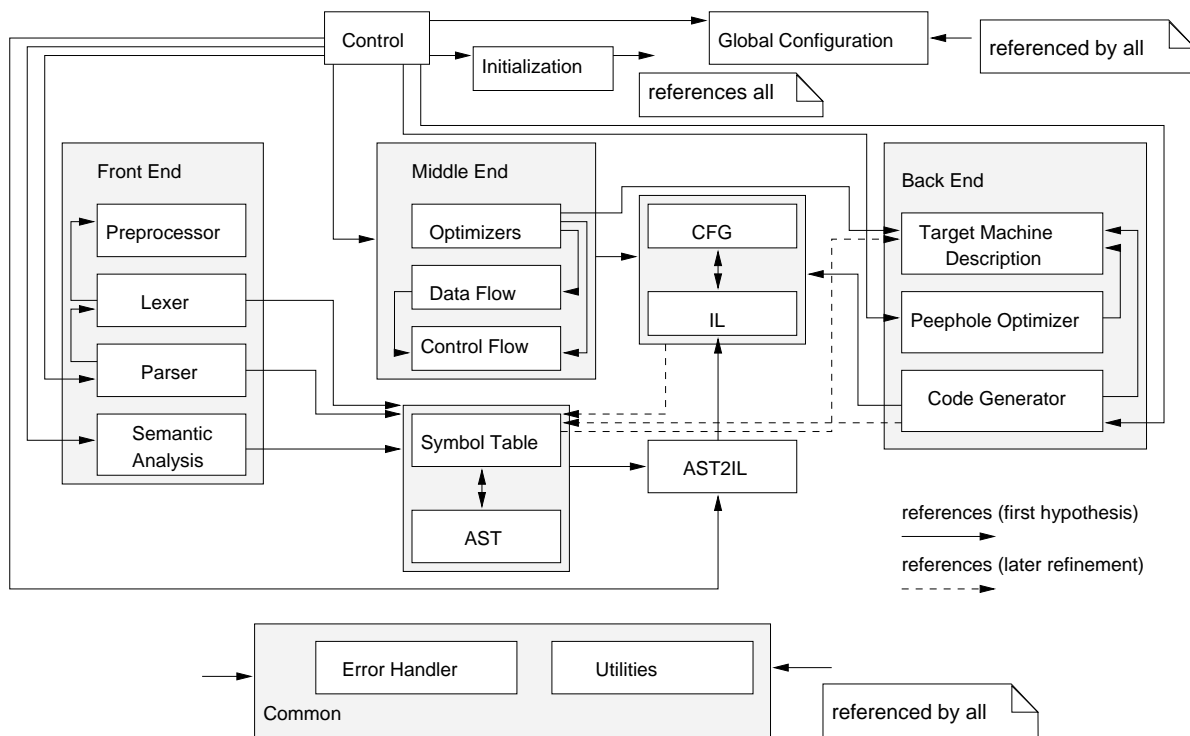
Fallstudie für C-Compiler (Koschke und Simon 2003)

System	KLOC	C-Units	Multi-Plattform	Aufwand
sdcc	100	49	ja	6 h
cc1	500	156	ja	8+ h



Compiler-Architektur (grobe Modulsicht)





Resultate für *sdcc*

- Abbildung von Dateien relativ klar
- viele Divergenzen in der ersten Iteration
- Verfeinerung (5 zusätzliche Iterationen):
 - 45 globale Deklarationen nicht auf die konzeptionelle Komponente abgebildet, auf die Datei abgebildet wurde, die sie deklariert
 - die meisten davon auf *Global Declarations*
 - einige übersehene Abhängigkeiten in der hypothetischen Sicht

Architekturverletzungen:

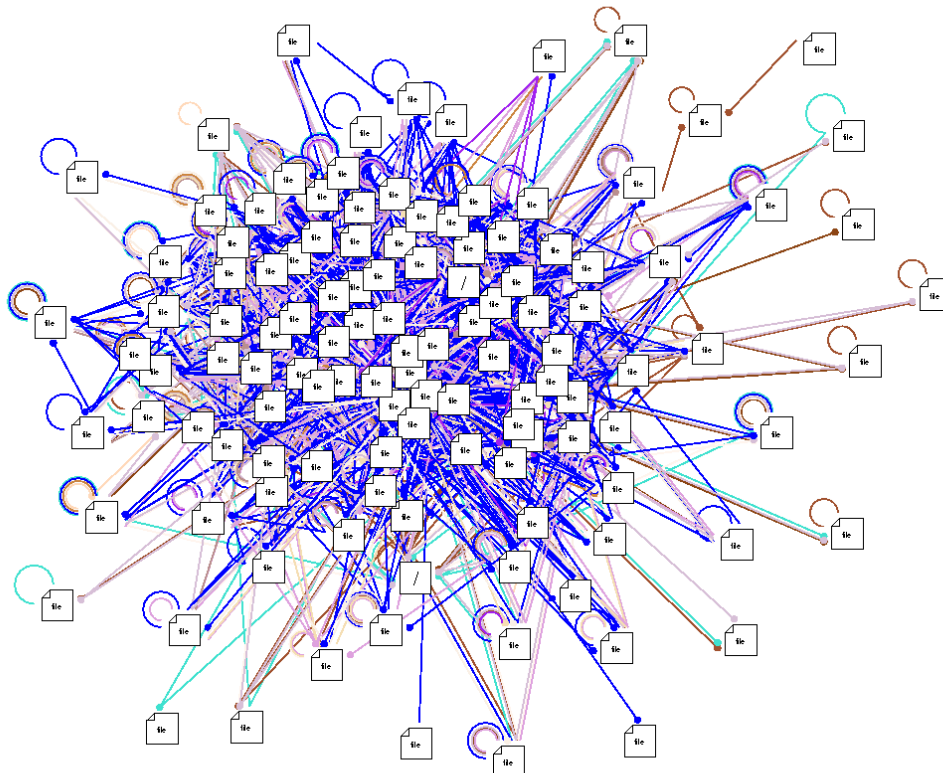
- Symboltabelle referenziert Parser
 - Blocknummer von Deklarationen und Zeilennummer
- Back-End referenziert Parser
 - globale Variablen, die Kellergröße für Aktivierungsblöcke verwalten

Architekturmuster:

- Optimierung referenziert Back-End
 - um plattformabhängige Parameter zu erhalten
 - mittels Funktionszeigern
- “anonyme” Abhängigkeit

- nur Units abgebildet (keine Verfeinerung für globale Deklarationen)
- Front-End ist gut strukturiert and lose gekoppelt an Middle-/Back-End
- Back-End ist einfach identifizierbar
- Middle-End ist ein “big ball of mud”

Modulabhängigkeiten von *cc1*



Architekturverletzungen:

- Middle-End referenziert Preprozessor
 - benutzt Hash-Tabelle des Preprozessors
 - (es gibt mindestens drei verschiedene Hash-Tabellen in *cc1*)
- viele, viele Divergenzen. . .

Gesammelte Erfahrungen

- Extraktion:
 - Funktionszeigeranalyse sehr wichtig
- Verfeinerungen der Modelle:
 - alle Modelle (hypothetisches und konkrete Sicht, Abbildung)
- Einfluss der Strukturiertheit des analysierten Systems:
 - Kohäsion der Units erleichtert Abbildung enorm

- hierarchisches hypothetisches Modell ist notwendig für große Systeme
- hypothetisches Modell und Abbildung müssen manuell erstellt werden
- Voraussetzungen:
 - hypothetische Sicht erfordert Wissen über Anwendungsbereich und potentielle Architektur
- Weitere notwendige Erweiterungen:
 - Abbildung von Relationen
 - Kontroll- und Datenflussabhängigkeiten statt nur Referenzen

Hypothesengetriebene manuelle Zuordnung versus unüberwachte Klassifikation

- Reflektionsmethode beginnt mit Architekturhypothese
 - die Abbildung konkreter auf logischer Komponenten ist manuell
 - bei 1,5 MLOC müssen leicht 10.000 Routinen, zumindest hunderte von Dateien abgebildet werden
- umgekehrter Ansatz: automatische Gruppierung zusammengehöriger Elemente

Kennzeichen:

Software-Elemente werden automatisch auf der Basis ihrer Ähnlichkeit gruppiert¹ bzw. auf Basis ihrer Unähnlichkeit unterschieden.

Software-Clustering ist unüberwachte Klassifikation.

Fragen:

- Welche Software-Elemente werden gruppiert?
- Wie ist die Ähnlichkeit zwischen Software-Elementen definiert?
- Auf welche Weise wird gruppiert (welcher Algorithmus wird verwendet)?

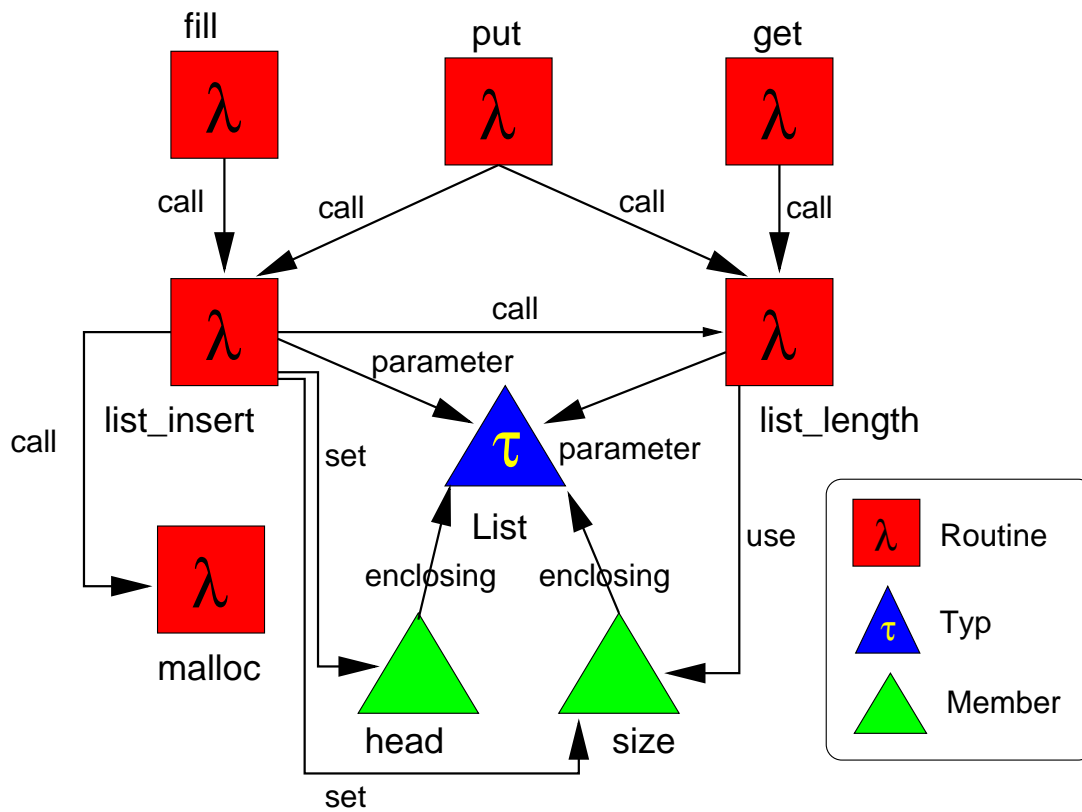
¹Wir folgen hier einer allgemeinen Darstellung von Wiggert (1998). Eine alternative ausführliche Zusammenstellung und Klassifikation von Methoden und Techniken findet man insbesondere bei Koschke (2000)

Gegenstand des Software-Clusterings

Gruppiert werden. . .

- globale Deklarationen in C
- Attribute und Methoden in objektorientierten Programmen
- Klassen in objektorientierten Programmen
- Module und Pakete
- Dateien
- vereinzelt sogar einzelne Anweisungen und Ausdrücke
- . . .

Ähnlichkeit zwischen zwei Elementen



2006-01-29

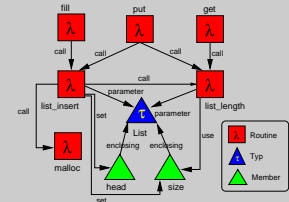
Vorlesung Software-Reengineering

└─ Strukturelle Architektursichten

└─ Software-Clustering

└─ Ähnlichkeit zwischen zwei Elementen

Ähnlichkeit zwischen zwei Elementen



Ähnlichkeitsdefinition kann sich auf

- Beziehungen zwischen Elementen und
- Eigenschaften jedes einzelnen Elements

stützen.

Arten von Eigenschaften (abhängig von Skalen):

- Nominalskala
- Ordinalskala
- Intervallskala
- Rationalskala
- Absolutskala

Binäre Eigenschaften I

Binäre Eigenschaften sind spezielle nominale Eigenschaften, die zweiwertig repräsentiert werden können.

Symmetrische binäre Eigenschaften:

- zwei Kategorien
- z.B. Routine versus Variable.

Asymmetrische binäre Eigenschaften:

- Eigenschaften ist vorhanden (1)
 - z.B. Routine ist Blatt im Aufrufgraph
- Eigenschaft ist nicht vorhanden (0)
 - z.B. Routine ruft andere Routinen auf

Binäre Eigenschaften II

Beispiel für asymmetrische binäre Eigenschaften:
Routine liest/schreibt Variable.

set/use	v1	v2	v3	v4	v5	v6
r1	×	×				×
r2	×		×			×
r3	×					

Binäre Eigenschaften III

Assoziationskoeffizienten:

		Element j		
		1	0	
Element i	1	a	b	$a + b$
	0	c	d	$c + d$
		$a + c$	$b + d$	$n = a + b + c + d$

- Werden 0-0-Übereinstimmungen gewertet, und wenn ja, wie?
- Gewichtung von Übereinstimmungen und Differenzen?

Binäre Eigenschaften IV

set/use	v1	v2	v3	v4	v5	v6
r1	×	×				×
r2	×		×			×
r3	×					

$$\begin{array}{c}
 r2 \\
 \begin{array}{cc}
 & 1 & 0 \\
 r1 \begin{array}{l} 1 \\ 0 \end{array} & \begin{array}{|c|c|} \hline 2 & 1 \\ \hline 1 & 2 \\ \hline \end{array} & \begin{array}{l} 2+1 \\ 1+2 \end{array} \\
 & 2+1 & 1+2 & n = 2+1+1+2
 \end{array}
 \end{array}$$

$$\begin{array}{c}
 r3 \\
 \begin{array}{cc}
 & 1 & 0 \\
 r1 \begin{array}{l} 1 \\ 0 \end{array} & \begin{array}{|c|c|} \hline 1 & 2 \\ \hline 0 & 3 \\ \hline \end{array} & \begin{array}{l} 1+2 \\ 0+3 \end{array} \\
 & 1+0 & 2+3 & n = 1+2+0+3
 \end{array}
 \end{array}$$

Binäre Eigenschaften V

simple matching coefficient:

$$simple(i, j) = \frac{a + d}{n}$$

$$simple(r1, r2) = \frac{2 + 2}{2 + 1 + 1 + 2} = \frac{2}{3}$$

$$simple(r1, r3) = \frac{1 + 3}{1 + 2 + 0 + 3} = \frac{2}{3}$$

Binäre Eigenschaften VI

Jaccard coefficient:

$$Jaccard(i, j) = \frac{a}{a + b + c}$$

$$Jaccard(r1, r2) = \frac{2}{2 + 1 + 1} = \frac{1}{2}$$

$$Jaccard(r1, r3) = \frac{1}{1 + 2 + 0} = \frac{1}{3}$$

Binäre Eigenschaften VII

Sorensen-Dice coefficient:

$$Sorensen-Dice(i, j) = \frac{2 \times a}{2 \times a + b + c}$$

$$Sorensen-Dice(r1, r2) = \frac{2 \times 2}{2 \times 2 + 1 + 1} = \frac{4}{5}$$

$$Sorensen-Dice(r1, r3) = \frac{2 \times 1}{2 \times 1 + 2 + 0} = \frac{1}{2}$$

$$\text{Sorensen-Dice}(i, j) = \frac{2 \times a}{2 \times a + b + c}$$

$$\text{Sorensen-Dice}(r1, r2) = \frac{2 \times 2}{2 \times 2 + 1 + 1} = \frac{4}{5}$$

$$\text{Sorensen-Dice}(r1, r3) = \frac{2 \times 1}{2 \times 1 + 2 + 0} = \frac{1}{2}$$

Eine Fallstudie von Saeed u. a. (2003) zeigte, dass:

- Sorensen-Dice und Jaccard verhalten sich gleich → die zusätzliche Gewichtung gemeinsamer Eigenschaften bringt keinen erkennbaren Vorteil.
- Simple matching coefficient erscheint ungeeignet (was man intuitiv erwarten würde, weil er der Abwesenheit von Eigenschaften viel Bedeutung zumisst; im Falle von Software-Elementen sind die meisten Eigenschaften der oben genannten Art üblicherweise solche, die die beiden Element beide nicht besitzen).

Intervallskala

$$\begin{pmatrix} x_{1,1} & x_{1,2} & \dots & x_{1,f} & \dots & x_{1,p} \\ x_{2,1} & x_{2,2} & \dots & x_{2,f} & \dots & x_{2,p} \\ \dots & \dots & \dots & \dots & \dots & \dots \\ x_{n,1} & x_{n,2} & \dots & x_{n,f} & \dots & x_{n,p} \end{pmatrix}$$

mit $x_{i,j}$: Wert der Eigenschaft j für Element i .

Beispiele:

- Anzahl Aufrufe
- McCabe-Komplexität
- Länge in Codezeilen
- Bezeichnerlänge
- ...

→ siehe Klonerkennung...

$$\begin{pmatrix} x_{1,1} & x_{1,2} & \dots & x_{1,f} & \dots & x_{1,p} \\ x_{2,1} & x_{2,2} & \dots & x_{2,f} & \dots & x_{2,p} \\ \dots & \dots & \dots & \dots & \dots & \dots \\ x_{n,1} & x_{n,2} & \dots & x_{n,f} & \dots & x_{n,p} \end{pmatrix}$$

Erster Schritt: Normalisiere (standardisiere) die Werte:

$$s_f = \frac{1}{n} \times (|x_{1,f} - m_f| + |x_{2,f} - m_f| + \dots + |x_{n,f} - m_f|)$$

wobei $m_f = \frac{1}{n}(x_{1,f} + x_{2,f} + \dots + x_{n,f})$.

Z-Score:

$$z_{i,f} = \frac{x_{i,f} - m_f}{s_f}$$

Ähnlichkeit für Intervallskala I

Minkowski-Distanz:

$$d(i, j) = \sqrt[q]{|z_{i,1} - z_{j,1}|^q + |z_{i,2} - z_{j,2}|^q + \dots + |z_{i,p} - z_{j,p}|^q}$$

wobei $q > 0$.

Falls $q = 1$, dann ist d die Manhattan-Distanz:

$$d(i, j) = |z_{i,1} - z_{j,1}| + |z_{i,2} - z_{j,2}| + \dots + |z_{i,p} - z_{j,p}|$$

Ähnlichkeit:

$$sim(i, j) = 1 - d(i, j)$$

Falls $q = 2$, dann ist d der Euklidische Abstand:

$$d(i, j) = \sqrt{|z_{i,1} - z_{j,1}|^2 + |z_{i,2} - z_{j,2}|^2 + \dots + |z_{i,p} - z_{j,p}|^2}$$

Eigenschaften:

- $d(i, j) \geq 0$
- $d(i, i) = 0$
- $d(i, j) = d(j, i)$
- $d(i, j) \leq d(i, k) + d(k, j)$

Weitere Alternativen (gewichtete Distanz, parametrische Produktmoment-Korrelation) siehe Literatur.

Clustering-Algorithmen

Varianten:

- graphentheoretische Algorithmen
- Konstruktionsalgorithmen
- Optimierungsalgorithmen
- Hierarchische Agglomeration

Clustering von Knoten in einem Graph (binäre Relation)

Beispiele

- Starke Zusammenhangskomponenten (Zyklen)
- Dominanzanalyse
- Einfache Zusammenhangskomponenten (isolierte Teilgraphen)

Einfache Zusammenhangskomponenten

Stecke jeden Knoten in eigene Menge

```
foreach edge (s,t) in graph loop  
  union (find(s),find(t))  
end loop
```

Gruppierung der Knoten in einem Durchlauf auf der Basis ihrer Eigenschaftsvektoren

- geographische Techniken
- Suche nach dichten Ansammlungen

Optimierungsalgorithmen

Erstelle initiale Partition mit n Gruppen

repeat

bestimme Kristallisationskerne² (KK)

gruppiere jedes Element zum ähnlichsten KK

until keine Elemente mehr re-gruppiert werden können

²Kristallisationskern ist oft Zentroid

Hierarchische Agglomeration

Partitioniere n Element in n Gruppen

Berechne $n \times n$ -Ähnlichkeitsmatrix

repeat

finde das Paar (a,b) mit maximaler Ähnlichkeit

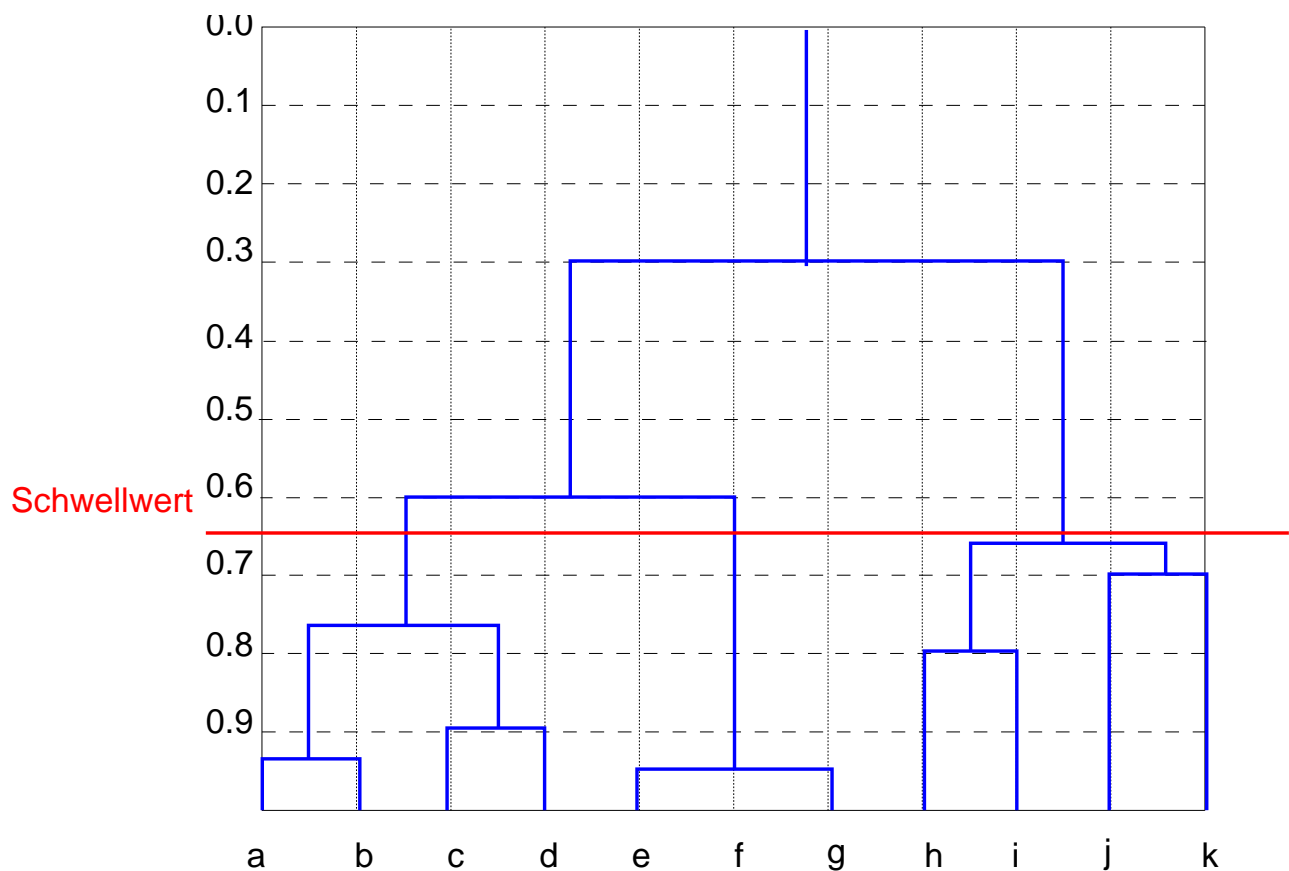
merge (a,b)

aktualisiere Ähnlichkeitsmatrix

until keine Elemente mehr re-gruppiert werden können

Resultat: Dendrogramm

Dendrogramm



Wie wird Ähnlichkeit zwischen ganzen Gruppen bestimmt?

- Single-Linkage

$$\text{single-link}(C, A \cup B) = \max(\text{sim}(C, A), \text{sim}(C, B))$$

- Complete-Linkage

$$\text{complete-link}(C, A \cup B) = \min(\text{sim}(C, A), \text{sim}(C, B))$$

- Unweighted Pair-Group-Average

$$\text{avg-link}(C, A \cup B) = \frac{|A| \times \text{sim}(C, A) + |B| \times \text{sim}(C, B)}{|A| + |B|}$$

Wiederholungs- und Vertiefungsfragen I

- Wie geht man bei der Reflektionsmethode vor?
- Wie sind Konvergenzen, Divergenzen und Abwesenheit genau definiert?
- Was sind ihre wesentlichen Kennzeichen?
- Wo liegen ihre Schwierigkeiten?
- Was ist Software-Clustering?
- Wozu wird Software-Clustering verwendet?
- Wie kann Ähnlichkeit zwischen zwei Elementen definiert werden?
- Wie hängt dies mit den Skalen zusammen?
- Wie kann Ähnlichkeit für ganze Mengen von Elementen definiert werden?
- Welche Algorithmen kann man für das Clustering benutzen?
- Welche prinzipiellen Schritte sieht Symphony vor?
- Was soll die Trennung von Blickwinkeln (Viewpoints) und Sichten (Views) bewirken?

- 1 **Koschke 2000** KOSCHKE, Rainer:
Atomic Architectural Component Recovery for Program Understanding and
Universität Stuttgart, Deutschland, Dissertation, 2000
- 2 **Koschke und Simon 2003** KOSCHKE, Rainer ; SIMON, Daniel:
Hierarchical Reflexion Models. In: Working Conference on Reverse
Engineering, IEEE Computer Society Press, November 2003, S. 36–45
- 3 **Murphy u. a. 1995** MURPHY, Gail C. ; NOTKIN, David ; SULLIVAN,
Kevin: Software Reflexion Models: Bridging the Gap Between Source
and High-Level Models. In: Proc. of the Third ACM SIGSOFT
Symposium on the Foundations of Software Engineering, ACM Press,
1995, S. 18–28
- 4 **Murphy u. a. 2001** MURPHY, Gail C. ; NOTKIN, David ; SULLIVAN,
Kevin J.: Software Reflexion Models: Bridging the Gap between Design
and Implementation. In: IEEE Computer Society Transactions on
Software Engineering 27 (2001), April, Nr. 4, S. 364–380

- 5 **Saeed u. a. 2003** SAEED, M. ; MAQBOOL, O. ; BABRI, H.A. ;
HASSAN, S.Z. ; SARWAR, S.M.: Software Clustering Techniques and the
Use of Combined Algorithm. In: European Conference on Software
Maintenance and Reengineering, IEEE Computer Society Press, 2003,
S. 301–310
- 6 **Wiggert 1998** WIGGERT, T. A.: Using Clustering Algorithms in
Legacy Systems Remodularization. In: Working Conference on Reverse
Engineering, IEEE Computer Society Press, 1998, S. 33–43. – Nice
introduction to software clustering. Does not introduce a new clustering
technique, but gives an overview on the clustering techniques compiled
from literature on general clustering (not software related).