Reengineering von Klassenhierarchien mittels Begriffsanalyse

9 Analyse und Restrukturierung von Vererbungshierarchien
- Motivation
- C++ Kurzüberblick
- Beispiel Klassenhierarchie
- Objekte
- Attribute
- Relation
- Member-Zugriffe
- this-Zeiger
- Zuweisungen
- Dominance und Hiding
- Schlussfolgerungen

Reengineering von Klassenhierarchien mittels Begriffsanalyse (Concept Analysis)

- Lernziele
  - Beispiel für weitere Anwendung von Begriffsanalyse
  - Reengineering von Klassenhierarchien unter Berücksichtigung der tatsächlichen Benutzung der Hierarchien
  - Reengineering objektorientierter Software
- Kontext
  - Snelting und Tip (2000)
  - Übergang Codierung / Entwurf

Motivation

Notwendigkeit für das Reengineering von Klassenhierarchien:

- Entwurf einer Klassenhierarchie ist schwierig und kann Schwächen aufweisen
  - Anwendungsbereich am Anfang noch nicht ganz verstanden
  - zukünftige Verwendungsweise der Klassenhierarchie noch unklar
- Ad-Hoc-Erweiterungen erhöhen die Entropie

C++ Primer I

Klasse

```
class T {
  public:
    // data member T::attr1
    AttributeType1 attr1;

    // non-virtual function member T::f()
    int f (void);

    // virtual function member T::g()
    virtual void g (void);
};
```

C++ Primer II

Unterklasse

```
// class derived from T
class NT : public T {
  public:
    // hides T::attr1
    AttributeType1 attr1;
    // yet another attribute
    AttributeType2 attr2;

    // (inlined) re-definition
    virtual void g(void) {
      attr1 = ...;          //  this is NT::attr1
};
```

C++ Primer III

Verwendungen

```
class NT nt ;
class T *t = &nt;

nt.attr1 = ...;

// static binding; call to NT::g()
nt.g();

// static binding; call to T::f()
t->f ();

// dispatching call to NT::g()
t->g ();
```

C++ Primer IV

Logische Sicht

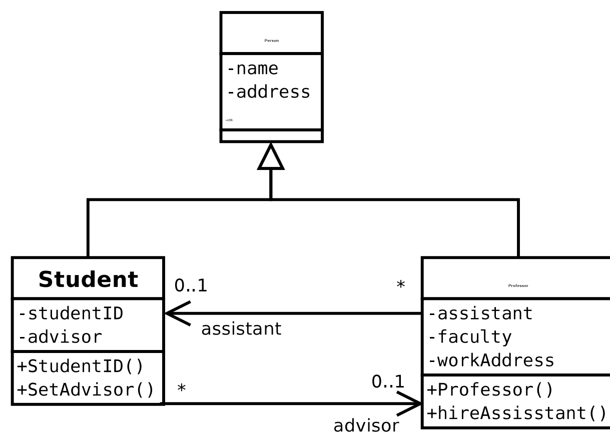
```
int T::f (void) {
  attr1 = ...;
}

T t;
t.f();
```

Implementierungssicht

```
int f (T *this) {
  this->attr1 = ...;
}
T t;
f(&t);
```

Beispiel: Klassenhierarchie I



Beispiel: Klassenhierarchie II

```
// Implementierung von Student
Student::Student (sn, sa, si) {
  name = sn; address = sa;
  studentID = si;
}

Student::setAdvisor (Professor *p) { advisor = p; }

// Implementierung von Professor
Professor::Professor (n, f, wa) {
  name = n; faculty = f;
  workAddress = wa; assistant = 0;
}

Professor::hireAssistant (Student *s)
  { assistant = s; }
```

Beispiel: Verwendung der Klassenhierarchie I

```
void main (void) {
  String s1name, p1name;
  Address s1addr, p1addr;

  /* Student 1 */
  Student *s1 = new Student (s1name, s1addr, 123456);

  /* Professor 1 */
  Professor *p1 = new Professor
                    (p1name, Mathematics, p1addr);

  s1->setAdvisor (p1);
}
```

Beispiel: Verwendung der Klassenhierarchie II

```
void main (void) {
  String s2name, p2name;
  Address s2addr, p2addr;

  /* Student 2 */
  Student *s2 = new Student (s2name, s2addr, 123457);

  /* Professor 2 */
  Professor *p2 = new Professor (p2name, Biology, p2addr);

  p2->hireAssistant (s2);
}
```

Anwendung der Begriffsanalyse

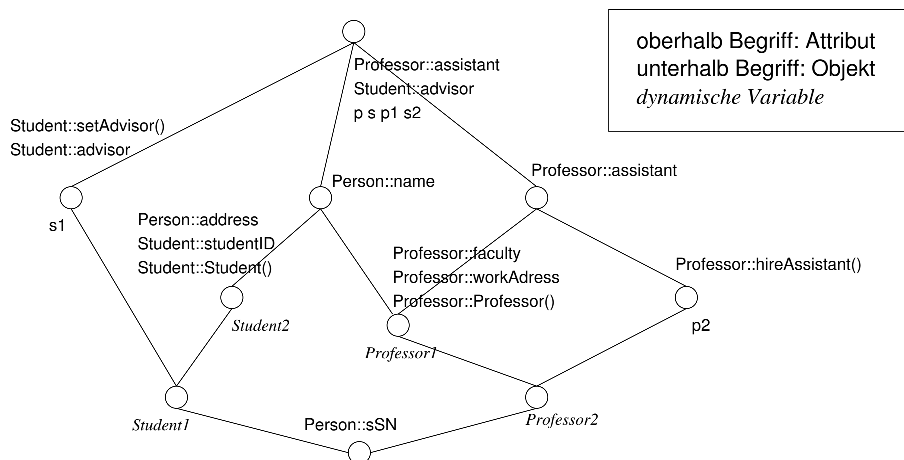
Formaler Kontext:

- Objekte: Variablen (inklusive Parametern und Data Members), durch die auf die Klassen zugegriffen wird
- Attribute: Data und Function Members
- Relation: beschreibt für jede Variable, welche Members ihr Typ voraussetzt

Ein Begriff kann als Klasse aufgefasst werden: Alle Objekte im Begriff haben alle Attribute gemeinsam, d.h. setzen alle Member voraus:

- Begriff = potentielle Klasse mit den Attributen als Members
- Oberbegriffe = Basisklassen

Begriffsverband für das Beispiel



oberhalb Begriff: Attribut
unterhalb Begriff: Objekt
dynamische Variable

Beobachtungen

- `Person::sSN` wird nirgendwo verwendet.
- `Person::address` wird nur von Studenten benutzt; bei Professoren wird statt dessen `Professor::workAddress` verwendet.
- Keine Members werden mittels Parameter `s` und `p` sowie von Data Members `advisor` und `assistant` benutzt.
- Der Konstruktor von Student initialisiert `advisor` nicht.
- Es gibt Professoren, die Studenten einstellen (Professor2) und solche ohne Studenten (Professor1).
- Es gibt Studenten mit Betreuer (Student1) und solche ohne (Student2).

Konventionen

Im Folgenden:

- Variablen umfassen auch Parameter
- $v, w, \ldots$ stehen für Variablen eines Klassentyps
- $p, q, \ldots$ stehen für Zeiger auf Variablen eines Klassentyps
- P steht für ein Programm
- $TypeOf(P, e)$ steht für den Typ eines Ausdrucks e in P
- $A :: f$ steht für den `this`-Zeiger der Methode $A :: f()$

Objekte der Begriffsanalyse

- Variablen eines Klassentyps

$$ClassVars(P) = \left\{ v \;\middle|\; \begin{array}{l} v \text{ ist eine Variable in } P \text{ und} \\ TypeOf(P, v) = C, \\ \text{wobei } C \text{ eine Klasse in } P \text{ ist} \end{array} \right\}$$

- Zeiger auf Variablen eines Klassentyps

$$ClassPtrVars(P) = \left\{ p \;\middle|\; \begin{array}{l} p \text{ ist eine Variable in } P\text{, und} \\ TypeOf(P, *p) = C, \\ \text{wobei } C \text{ eine Klasse in } P \text{ ist} \end{array} \right\}$$

N.B.: *ClassPtrVars* beinhaltet den `this`-Zeiger

Attribute der Begriffsanalyse I

Attribute sind Klassen-Members:

- Data Members
- Function Members
  - **Deklaration** = Signatur, $dcl(A :: g)$
  - **Definition** = Signatur + Implementierung, $def(A :: g)$
  - Unterschied ist relevant für Aufrufe virtueller Methoden: für

    ```
    p->my_virtual_function ()
    ```

    muss nur die Deklaration von

    ```
    my_virtual_function ()
    ```

    im Klassentyp von `*p` enthalten sein
  - keine Unterscheidung für nicht-virtuelle Methoden, da statisch eindeutig feststeht, welche Operation aufgerufen wird

Attribute der Begriffsanalyse II

$$MemberDcls(P) = \left\{ dcl(C :: m) \,\middle|\, \begin{array}{l} m \text{ ist ein Data Member oder} \\ \text{eine virtuelle Methode der Klasse } C \end{array} \right\}$$

$$MemberDefs(P) = \left\{ def(C :: m) \,\middle|\, \begin{array}{l} m \text{ ist eine virtuelle Methode oder} \\ \text{nicht-virtuelle Methode der Klasse } C \end{array} \right\}$$

Beispiel: Objekte und Attribute I

```
class A { public:
  virtual int f (void) {return g();};
  virtual int g (void) {return x;};
  int x;
};

class B : public A {public:
  virtual int g (void) {return y;}
  int y;
};

class C : public B {public:
  virtual int f (void)
  {return g () + z;};
  int z;
}
```

Beispiel: Objekte und Attribute II

```
void main (void) {
  A a; B b; C c; A *ap;
  if (...) {ap = &a;}
  else {if (...) { ap = &b;}
    else {ap = &c;}
  }
  ap->f ();
}
```

$ClassVars(P_1) = \{\texttt{a}, \texttt{b}, \texttt{c}\}$

$ClassPtrVars(P_1) = \{\texttt{ap}, \texttt{A} :: \texttt{f}, \texttt{A} :: \texttt{g}, \texttt{B} :: \texttt{g}, \texttt{C} :: \texttt{f}\}$

$$MemberDecls(P_1) = \left\{ \begin{array}{l} dcl(\texttt{A} :: \texttt{f}), dcl(\texttt{A} :: \texttt{g}), dcl(\texttt{A} :: \texttt{x}), dcl(\texttt{B} :: \texttt{g}), \\ dcl(\texttt{B} :: \texttt{y}), dcl(\texttt{C} :: \texttt{f}), dcl(\texttt{C} :: \texttt{z}) \end{array} \right\}$$

$MemberDefs(P_1) = \{def(\texttt{A} :: \texttt{f}), def(\texttt{A} :: \texttt{g}), def(\texttt{B} :: \texttt{g}), def(\texttt{C} :: \texttt{f})\}$

Data Members mit Klassentyp

- Data Members können über Variablen eines Klassentyps verwendet werden und sind somit Attribute im Sinne der Begriffsanalyse
- Data Members können aber auch selbst von einem Klassentyp sein (class-related Data Members)
  ```
  class Professor;
  class Student { Professor *advisor; };
  ```
- andere Members können über class-related Data Members verwendet werden
  ```
  Student *s;
  (s->advisor)->hireAssistant (s);
  ```
- class-related Data Members sind sowohl Attribute als auch Objekte im Sinne der Begriffsanalyse
- der Begriff *Variable* beinhaltet deshalb auch Parameter und class-related Members im Folgenden

Relation I

- Die Relation zwischen Objekten (Variablen) und Attributen (Members) drückt aus, dass der Klassentyp einer Menge von Variablen eine gewisse Menge von Members voraussetzt
- Objekte: Variablen
  Attribute: Deklarationen *dcl* und Definitionen *def* von Members
- $(v, dcl(A :: m)) \in \mathcal{I}$ gilt g.d.w. die Deklaration von m im (Verwendungs-)Typ von v enthalten ist
- $(v, def(A :: m)) \in \mathcal{I}$ gilt g.d.w. die Definition von m im (Verwendungs-)Typ von v enthalten ist
- Relation $\mathcal{I}$ kann als Tabelle repräsentiert werden:
  - Zeilen: Variablen
  - Spalten: Deklarationen und Definitionen von Members

Relation II

- Die Relation ergibt sich aus
  - Member-Zugriffen
  - this-Zeigern
  - Zuweisungen (führen zu Implikationen)
  - Dominance/Hiding von Namen (führen zu Implikationen)
- Polymorphismus in C++ ist nur über Zeiger möglich ⇒ Points-To-Information ist nötig:

$$\textit{points-to}(P) = \left\{ \langle p, v \rangle \;\middle|\; \begin{array}{l} p \in \textit{ClassPtrVars}(P) \text{ und} \\ v \in \textit{ClassVars}(P) \text{ und} \\ p \text{ zeigt potenziell auf } v \end{array} \right\}$$

Points-To-Information I

```
class A { public:
  virtual int f (void) {return g();};
  virtual int g (void) {return x;};
  int x;
};

class B : public A {public:
  virtual int g (void) {return y;}
  int y;
};

class C : public B {public:
  virtual int f (void)
  {return g () + z;};
  int z;
}
```

Points-To-Information II

```
void main (void) {
  A a; B b; C c; A *ap;
  if (...) {ap = &a;}
  else {if (...) { ap = &b;}
    else {ap = &c;}
  }
  ap->f ();
}
```

$$
\begin{aligned}
\textit{points-to}(P_1) = \{ & \langle ap, a\rangle, \langle ap, b\rangle, \langle ap, c\rangle, \langle A :: f, a\rangle, \langle A :: f, b\rangle, \\
& \langle C :: f, c\rangle, \langle A :: g, a\rangle, \langle B :: g, b\rangle, \langle B :: g, c\rangle\}
\end{aligned}
$$

Member-Zugriffe

Relation $\mathcal{I}$ wird durch Member-Zugriffe induziert (m kann Data oder Function Member sein).
$MemberAccess(P)$ ist:

1. `v.m` $\Rightarrow \langle m, v\rangle \in MemberAccess(P)$
2. `p->m` $\Rightarrow \langle m, *p\rangle \in MemberAccess(P)$
3. `p->m` und $\langle p, x\rangle \in Points\text{-}To(P)$ und m ist virtuelle Methode $\Rightarrow \langle m, x\rangle \in MemberAccess(P)$

```
p = &x;
p->m();              ⇒ x.m()
```

Im Beispiel

$$MemberAccess(P) = \{\langle x, A :: g\rangle, \langle y, B :: g\rangle, \langle z, C :: f\rangle, \langle g, A :: f\rangle, \langle g, C :: f\rangle, \langle f, ap\rangle, \langle f, a\rangle, \langle f, b\rangle, \langle f, c\rangle, \langle g, a\rangle, \langle g, b\rangle, \langle g, c\rangle\}$$

Tabelleneinträge für Memberzugriff I

- `v.m` oder `p->m`

$$\frac{\langle m, y\rangle \in \mathit{MemberAccess}(P) \quad m \in \mathit{DataMembers}(P) \quad X \equiv \mathit{static\text{-}lookup}(\mathit{TypeOf}(P, y), m)}{(y, \mathit{dcl}(X :: m)) \in \mathcal{I}}$$

- `v.m()` oder `p->m()`

$$\frac{\langle m, y\rangle \in \mathit{MemberAccess}(P) \quad m \in \mathit{NonvirtualMethods}(P) \quad X \equiv \mathit{static\text{-}lookup}(\mathit{TypeOf}(P, y), m)}{(y, \mathit{def}(X :: m)) \in \mathcal{I}}$$

Tabelleneinträge für Memberzugriff II

- `p->m()`

$$\frac{\begin{array}{c}\langle m, y\rangle \in \mathit{MemberAccess}(P) \\ m \in \mathit{VirtualMethods}(P) \\ y \equiv *p \\ p \in \mathit{ClassPtrVars}(P) \\ X \equiv \mathit{static\text{-}lookup}(\mathit{TypeOf}(P, y), m)\end{array}}{(y, \mathit{dcl}(X :: m)) \in \mathcal{I}}$$

- `v.m()`

$$\frac{\begin{array}{c}\langle m, y\rangle \in \mathit{MemberAccess}(P) \\ m \in \mathit{VirtualMethods}(P) \\ y \equiv v \\ v \in \mathit{ClassVars}(P)\ X \equiv \mathit{static\text{-}lookup}(\mathit{TypeOf}(P, y), m)\end{array}}{(y, \mathit{def}(X :: m)) \in \mathcal{I}}$$

this-Zeiger

Für jeden Function Member $C :: f()$:

- Spalte $def(C :: f())$
  - $C :: f()$ kann aufgerufen werden ($C :: f()$ ist ein Attribut)

`this`-Zeiger

Für jeden Function Member $C :: f()$:

- Spalte $def(C :: f())$
  - $C :: f()$ kann aufgerufen werden ($C :: f()$ ist ein Attribut)
- Zeile $C :: f$
  - $C :: f()$ kann andere Members verwenden über seinen `this`-Zeiger ($C :: f$ ist ein Objekt)

this-Zeiger

Für jeden Function Member $C :: f()$:

- Spalte $def(C :: f())$
  - $C :: f()$ kann aufgerufen werden ($C :: f()$ ist ein Attribut)
- Zeile $C :: f$
  - $C :: f()$ kann andere Members verwenden über seinen `this`-Zeiger ($C :: f$ ist ein Objekt)
- $C :: f$ und $C :: f()$ könnten in verschiedenen Begriffen des Begriffsverbandes sein; es ist aber klar, dass $C :: f$ zu $C :: f()$ gehören muss

`this`-Zeiger

Für jeden Function Member $C :: f()$:

- Spalte $def(C :: f())$
  - $C :: f()$ kann aufgerufen werden ($C :: f()$ ist ein Attribut)
- Zeile $C :: f$
  - $C :: f()$ kann andere Members verwenden über seinen `this`-Zeiger ($C :: f$ ist ein Objekt)
- $C :: f$ und $C :: f()$ könnten in verschiedenen Begriffen des Begriffsverbandes sein; es ist aber klar, dass $C :: f$ zu $C :: f()$ gehören muss
- technischer Trick: $(C :: f, def(C :: f)) \in \mathcal{I}$
  - $\Rightarrow$ $C :: f$ und $C :: f()$ sind stets in demselben Begriff und damit derselben Klasse

Zuweisungen

- `v = w` ist nur dann gültig wenn $TypeOf(v)$ eine Basisklasse von $TypeOf(w)$ ist
- folglich muss jede Deklaration oder Definition, die im Typ von `v` vorkommt, auch im Typ von `w` vorkommen
- dies kann erzwungen werden, indem die Zeile von `v` die Zeile von `w` impliziert

(vorher)			
v	×	×	
w		×	×

⇒

(nachher)			
v	×	×	
w	×	×	×

Zuweisungen: Zeilenimplikation

- `v = w` $\Rightarrow \langle v, w \rangle \in \mathit{Assignments}(P)$
- `p = &w` $\Rightarrow \langle *p, w \rangle \in \mathit{Assignments}(P)$[5]
- `*p = w` $\Rightarrow \langle *p, w \rangle \in \mathit{Assignments}(P)$
- `v = *q` $\Rightarrow \langle v, *q \rangle \in \mathit{Assignments}(P)$
- `*p = *q` $\Rightarrow \langle *p, *q \rangle \in \mathit{Assignments}(P)$
- `p = q` $\Rightarrow \langle *p, *q \rangle \in \mathit{Assignments}(P)$

$$\frac{\langle x, y \rangle \in \mathit{Assignments}(P)}{x \rightarrow y}$$

[5] N.B.: Das * bei *p in *Assignments* kennzeichnet lediglich Zeiger; es soll nicht das Objekt, auf das der Zeiger zeigt, denotieren.

Zuweisungen: Parameterübergabe

- Parameterübergabe wird wie Zuweisung behandelt:
```
f (T t) {...}
f (a);
```
wird behandelt als:
```
f (t = a);
```

Zuweisungen: Parameterübergabe

- Parameterübergabe wird wie Zuweisung behandelt:
  ```
  f (T t) {...}
  f (a);
  ```
  wird behandelt als:
  ```
  f (t = a);
  ```
- analog für Methodenaufrufe:
  ```
  T::m () {...}
  o.m ();
  ```
  wird behandelt als:
  ```
  *T::m = &o;
  ```

Zuweisungen: Parameterübergabe

- Parameterübergabe wird wie Zuweisung behandelt:
  `f (T t) {...}`
  `f (a);`
  wird behandelt als:
  `f (t = a);`
- analog für Methodenaufrufe:
  `T::m () {...}`
  `o.m ();`
  wird behandelt als:
  `*T::m = &o;`
- `p -> f();` // f ist `virtual` → dynamisch gebunden
  Points-To-Information wird berücksichtigt:
  Für alle x mit $\langle p, x\rangle \in \mathit{Points\text{-}To}(P)$:
  $\langle *T :: f, *p\rangle \in \mathit{Assignments}(P)$, wobei T der statische Typ von x ist.

Dominance

- Ein Name $B :: f$ dominiert einen Namen $A :: f$, falls die Klasse B (transitiv) abgeleitet ist von A.
- Der dominierende Name überdeckt den dominierten:

```
class A {
  int i;
  void m (void);
};

class B : public A {
  int i;
  void m (void) { i=0; } // B::i
};

B b;
b.m (); // B::m ()
}
```

Dominance und Hiding

```
class A {void m (void);}
class B : public A
  {void m (void);}
A a; B b; B c;
a.m(); b.m(); c.m();
a = c; // Implikation
```

Dominance und Hiding

```
class A {void m (void);}
class B : public A
  {void m (void);}
A a; B b; B c;
a.m(); b.m(); c.m();
a = c; // Implikation
```

	$A :: m$	$B :: m$
a	×	
b		×
c		×

Dominance und Hiding

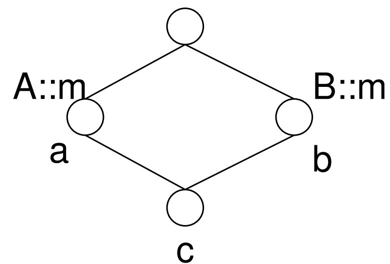
```
class A {void m (void);}
class B : public A
  {void m (void);}
A a; B b; B c;
a.m(); b.m(); c.m();
a = c; // Implikation
```

	$A :: m$	$B :: m$
a	×	
b		×
c	×	×

Dominance und Hiding

```
class A {void m (void);}
class B : public A
  {void m (void);}
A a; B b; B c;
a.m(); b.m(); c.m();
a = c; // Implikation
```

	$A :: m$	$B :: m$
a	×	
b		×
c	×	×

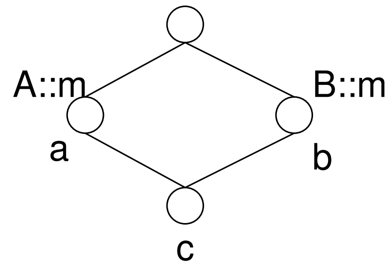


Dominance und Hiding

```
class A {void m (void);}
class B : public A
  {void m (void);}
A a; B b; B c;
a.m(); b.m(); c.m();
a = c; // Implikation
```

	$A::m$	$B::m$
a	×	
b		×
c	×	×

```
class A {void m (void);}
class B {void m (void);}
class C : public A, B {};
A a; B b; C c;
... c.m();// mehrdeutig!
```



Erhaltung von Dominance und Hiding I

$$\frac{(x, dcl(A :: m)) \in \mathcal{I}, (x, dcl(B :: m)) \in \mathcal{I} \quad A \text{ ist eine transitive Basisklasse von } B}{dcl(B :: m) \rightarrow dcl(A :: m)}$$

$$\frac{(x, dcl(A :: m)) \in \mathcal{I}, (x, def(B :: m)) \in \mathcal{I} \quad A = B \text{ oder } A \text{ ist eine transitive Basisklasse von } B}{def(B :: m) \rightarrow dcl(A :: m)}$$

Erhaltung von Dominance und Hiding II

$$\frac{(x, def(A :: m)) \in \mathcal{I}, (x, def(B :: m)) \in \mathcal{I} \quad A \text{ ist eine transitive Basisklasse von } B}{def(B :: m) \rightarrow def(A :: m)}$$

$$\frac{(x, def(A :: m)) \in \mathcal{I}, (x, dcl(B :: m)) \in \mathcal{I} \quad A \text{ ist eine transitive Basisklasse von } B}{dcl(B :: m) \rightarrow def(A :: m)}$$

Dominance und Hiding, nochmal

```
class A {void m (void);}
class B : public A
  {void m (void);}
A a; B b; B c;
a.m(); b.m(); c.m();
a = c; // Implikation
```

Dominance und Hiding, nochmal

```
class A {void m (void);}
class B : public A
  {void m (void);}
A a; B b; B c;
a.m(); b.m(); c.m();
a = c; // Implikation
```

	$A :: m$	$B :: m$
a	×	
b		×
c		×

Dominance und Hiding, nochmal

```
class A {void m (void);}
class B : public A
  {void m (void);}
A a; B b; B c;
a.m(); b.m(); c.m();
a = c; // Implikation
```

	$A :: m$	$B :: m$
a	×	
b		×
c	×	×

Dominance und Hiding, nochmal

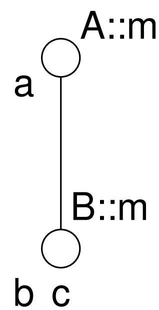
```
class A {void m (void);}
class B : public A
  {void m (void);}
A a; B b; B c;
a.m(); b.m(); c.m();
a = c; // Implikation
```

	$A :: m$	$B :: m$
a	×	
b	×	×
c	×	×

Dominance und Hiding, nochmal

```
class A {void m (void);}
class B : public A
  {void m (void);}
A a; B b; B c;
a.m(); b.m(); c.m();
a = c; // Implikation
```

	$A :: m$	$B :: m$
a	×	
b	×	×
c	×	×

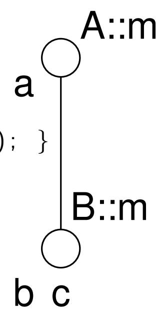


Dominance und Hiding, nochmal

```
class A {void m (void);}
class B : public A
  {void m (void);}
A a; B b; B c;
a.m(); b.m(); c.m();
a = c; // Implikation
```

	$A :: m$	$B :: m$
a	×	
b	×	×
c	×	×

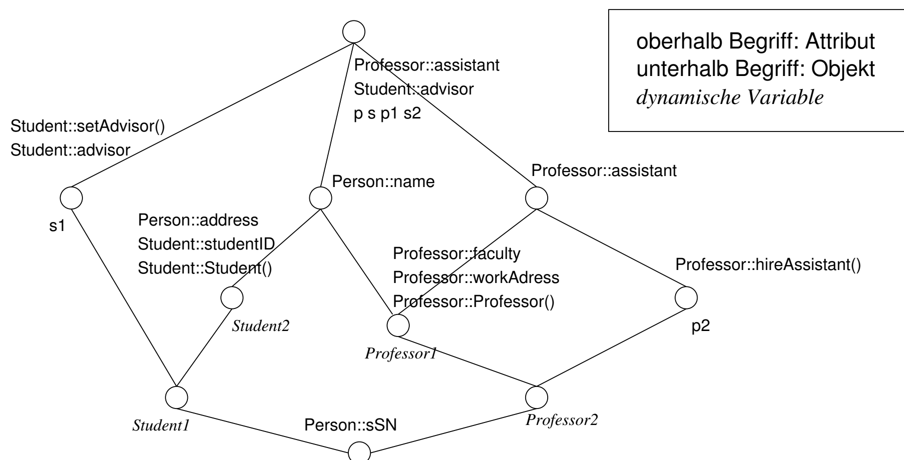
```
class A {void m (void); }
class B : public A {void m (void); }
A a; B b; B c;
... c.m (); // eindeutig B::m
```



Schlussfolgerungen aus dem Begriffsverband

- Members im **0**-Element des Verbands:
  - → nicht verwendet (`Person::sSN`), sofern keine Objekte im **0**-Element sind
- Members oberhalb von einigen, aber nicht aller abgeleiteten Klassen einer Klasse B:
  - → werden nicht von allen abgeleiteten Klassen verwendet (`Person::address`)
- Variablen im **1**-Element des Verbands:
  - → Variablen, über die kein Member verwendet wird (`s`), sofern keine Attribute im **1**-Element sind
- Data Members in keinem Begriff $\geq$ des entsprechenden Konstruktors:
  - → potenziell undefiniert (`Student::advisor`)
- Variablen des Klassentyps C treten an verschiedenen Stellen im Verbund auf:
  - → Instanzen der Klasse C verwenden verschiedene Teilmengen von Members von C (`Student1`, `Student2`)
  - → Kandidat für Restrukturierung

Begriffsverband für das Beispiel



oberhalb Begriff: Attribut
unterhalb Begriff: Objekt
dynamische Variable