Reengineering von Klassenhierarchien mittels Begriffsanalyse (Concept Analysis)

Reengineering von Klassenhierarchien

Lernziele:

- Beispiel für weitere Anwendung von Begriffsanalyse
- Reengineering von Klassenhierarchien unter Berücksichtigung der tatsächlichen Benutzung der Hierarchien
- Reengineering objektorientierter Software

Kontext

- 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

```
class T { public:
  AttributeType1 attr1; // data member T::attr1
  int f ();             // non-virtual function member T::f()
  virtual g ();         // virtual function member T::g()
};
class NT : T {public:    // class derived from T
  AttributeType2 attr2;  // yet another attribute
  AttributeType1 attr1;  // hides T::attr1
  virtual g() {          // re-definition
    attr1 = ...;         //  this is NT::attr1
};
```

C++ Primer

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

  nt.attr1 = ...;

  nt.g();      // static binding; call to NT::g()
  t->f ();     // static binding; call to T::f()
  t->g ();     // dispatching call to NT::g()
```


C++ Primer

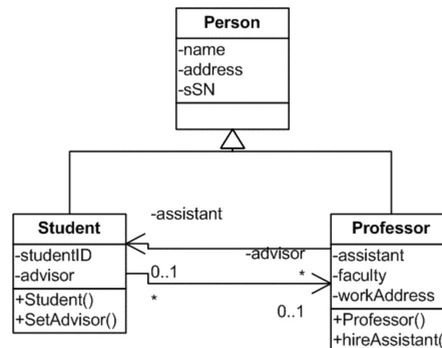
Logische Sicht

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

Implementierungssicht

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

Beispiel




```
Student::Student (sn, sa, si)
{ name = sn; address = sa; studentID = si; }
Student::setAdvisor (Professor *p)
{ advisor = p; }
```

```
Professor::Professor (n, f, wa)
{ name = n; faculty = f;
  workAddress = wa; assistant = 0;}
Professor::hireAssistant (a) {
{ assistant = a; }
```


Beispiel Verwender-Code

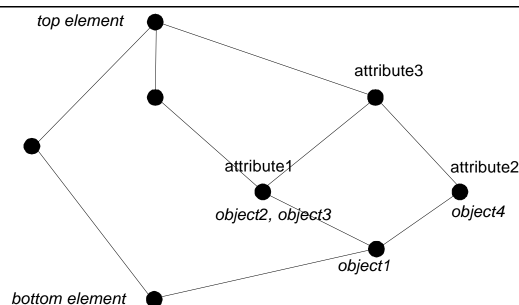
```
void main () {
  String s1name, p1name;
  Address s1addr, p1addr;

  Student *s1 = new Student
(s1name, s1addr, 123456);
    /* Student 1 */
  Professor *p1 = new Professor
(p1name, Mathematics, p1addr);
   /* Professor 1 */

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

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

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

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

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

Begriffsanalyse (Wiederholung)

- Kontext = (Objekte, Attribute, Relation)
- Konzept = (Objektmenge O, Attributmenge A), wobei jedes o ∈ O alle Attribute in A aufweist gemäß zugrundeliegender Relation und O und A maximal groß sind
- Partielle Ordnung:
  $(O1, A1) \leq (O2, A2) \Leftrightarrow O1 \subseteq O2 \Leftrightarrow A1 \supseteq A2$
- Menge der Konzepte und Relation ≤ ergeben einen vollständigen Begriffsverband

Interpretation des Begriffsverbands



Ein Objekt hat alle Attribute oberhalb seines Auftretens
- *object1* hat *attribute1, attribute2, attribute3*
- *object2, object3* und *object4* haben *attribute3* gemeinsam

Ein Attribut findet sich in allen Objekten unterhalb seines Auftretens
- *attribute1* findet sich in *object1, object2, object3*
- *attribute1* und *attribute2* finden sich beide bei *object1*

Anwendung der Begriffsanalyse

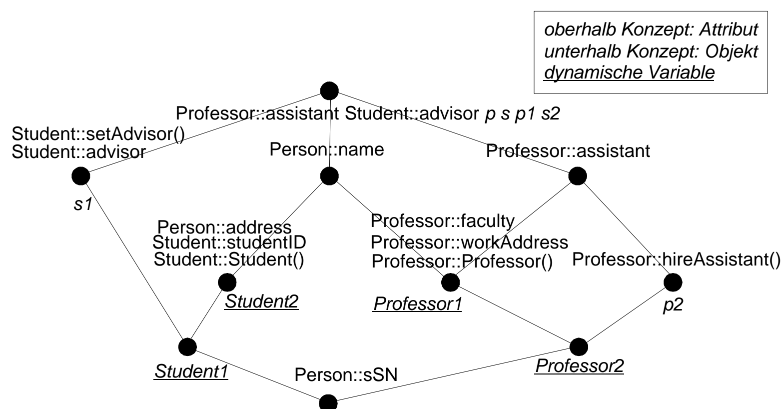
Begriffsanalyse kann wie folgt auf Verwendungen einer Klassenhierarchie angewandt werden:

- Objekte: Variablen (inklusive Parametern und Data Members)
- Attribute: Data und Function Members
- Relation: beschreibt für jede Variable, welche Members ihr Typ voraussetzt

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

- ⇨ Konzept = potentielle Klasse mit den Attributen als Members
- ⇨ Oberbegriffe = Basisklassen

Begriffsverband für das Beispiel



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 Betreuter (*Student1*) und solche ohne (*Student2*)

Konventionen

- Variablen beinhalten Parameter im Folgenden
- v, w, ... stehen für Variablen eines Klassentyps
- p, q, ... stehen für Zeiger auf Variablen eines Klassentyps
- $\mathcal{P}$ steht für ein Programm
- TypeOf ($\mathcal{P}$, e) steht für den Typ eines Ausdrucks *e* in $\mathcal{P}$
- A::f steht für den this-Zeiger der Methode A::f()

Objekte der Begriffsanalyse

- Variablen eines Klassentyps

  ClassVars ($\mathcal{P}$) = { v | v ist eine Variable in $\mathcal{P}$ und TypeOf ($\mathcal{P}$, v) = C, wobei C eine Klasse in $\mathcal{P}$ ist}

- Zeiger auf Variablen eines Klassentyps

  ClassPtrVars ($\mathcal{P}$) = {p | p ist eine Variable in $\mathcal{P}$, TypeOf ($\mathcal{P}$, *p) = C, wobei C eine Klasse in $\mathcal{P}$ ist}

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

Attribute der Begriffsanalyse

Attribute sind Klassen-Members:

- Data Members
- Function Members
  - **Deklaration** = Signatur
  - **Definition** = Signatur + Implementierung
  - Unterschied ist relevant für Aufrufe virtueller Methoden:
    `p->my_virtual_function ()`
    nur die Deklaration von `my_virtual_function ()` muss im Klassentyp von `*p` enthalten sein
  - keine Unterscheidung für nicht-virtuelle Methoden, da statisch eindeutig feststeht, welche Operation aufgerufen wird

Attribute der Begriffsanalyse

MemberDcls (*P*) =
{ dcl(C::m) | m ist ein Data Member
oder eine virtuelle Methode der Klasse C }

MemberDefs (*P*) =
{def(C::m) | m ist eine virtuelle
oder nicht-virtuelle Methode der Klasse C }

Objekte und Attribute (Beispiel)

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

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

ClassVars ($\mathcal{P}$1) = {a, b, c}
ClassPtrVars ($\mathcal{P}$1} = {ap, A::f, A::g, B::g, C::f}
MemberDecls ($\mathcal{P}$1) = {dcl(A::f), dcl(A::g), dcl(A::x), dcl(B::g), dcl(B::y), dcl(C::f), dcl(C::z)}
MemberDefs ($\mathcal{P}$1) = {def(A::f), def(A::g), def(B::g), def(C::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 V*ariable* beinhaltet deshalb auch Parameter und class-related Members im Folgenden

Relation

- 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
- Relation R (v, *dcl* (A::m)) gilt g.d.w. die Deklaration von *m* im (Verwendungs-)Typ von v enthalten ist
- Relation R (v, *def* (A::m)) gilt g.d.w. die Definition von *m* im (Verwendungs-)Typ von v enthalten ist
- Relation R kann als Tabelle T repräsentiert werden:
  - Zeilen: Variablen
  - Spalten: Deklarationen und Definitionen von Members

Relation

- 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:

  points-to ($\mathcal{P}$) = {<p, v> | p $\in$ ClassPtrVars ($\mathcal{P}$) und
  v $\in$ ClassVars ($\mathcal{P}$) und
  p zeigt potenziell auf v}

Points-To-Information

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

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

Points-To (P1) =
{<ap, a>, <ap, b>, <ap, c>,
<A::f, a>, <A::f, b>, <C::f, c>,
<A::g, a>, <B::g, b>, <B::g, c>}

Member-Zugriffe

Relation R wird durch Member-Zugriffe induziert (*m* kann Data oder Function Member sein):

- $v.m \Rightarrow <m, v> \in$ MemberAccess ($\mathcal{P}$)
- $p\text{->}m \Rightarrow <m, *p> \in$ MemberAccess ($\mathcal{P}$)
- $p\text{->}m \wedge <p, x> \in$ Points-To ($\mathcal{P}$) $\wedge$ m ist eine virtuelle Methode $\Rightarrow <m, x> \in$ MemberAccess ($\mathcal{P}$)

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

Reengineering 23

Member-Zugriff (1)

v.m / p->m

$<m, y> \in$ MemberAccess ($\mathcal{P}$)

$m \in$ DataMembers ($\mathcal{P}$)

$X \equiv$ static-lookup (TypeOf ($\mathcal{P}$ y), m)

(y, dcl (X::m)) $\in$ *T*

Reengineering 24

Member-Zugriff (2)

v.m() / p->m()

<m, y> $\in$ MemberAccess ($\mathcal{P}$)

m $\in$ NonvirtualMethods ($\mathcal{P}$)

X $\equiv$ static-lookup (TypeOf ($\mathcal{P}$ y), m)

(y, def (X::m)) $\in$ *T*

Member-Zugriff (3)

p->m()

<m, y> $\in$ MemberAccess ($\mathcal{P}$)

m $\in$ VirtualMethods ($\mathcal{P}$)

y $\equiv$ *p

p $\in$ ClassPtrVars ($\mathcal{P}$)

X $\equiv$ static-lookup (TypeOf ($\mathcal{P}$ y), m)

(y, dcl (X::m)) $\in$ *T*

Member-Zugriff (4)

v.m()

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

Reengineering 27

`this`-*Zeiger*

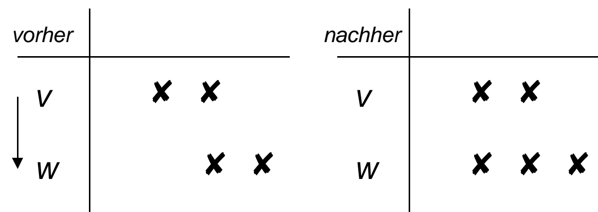
Für jeden Function Member C::f():

- 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 Konzepten des Begriffsverbandes sein; es ist aber klar, dass C::f zu C::f() gehören muss
- technischer Trick: (C::f, *def* (C::f)) $\in$ *T*
  - ➥ C::f und C::f() sind stets in demselben Konzept / derselben Klasse

Reengineering 28

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



Reengineering 29

Zuweisungen

- $v = w \Rightarrow \langle v, w\rangle \in \text{Assignments}(\mathcal{P})$
- $p = \&w \Rightarrow \langle *p, w\rangle \in \text{Assignments}(\mathcal{P})$
- $*p = w \Rightarrow \langle *p, w\rangle \in \text{Assignments}(\mathcal{P})$
- $v = *q \Rightarrow \langle v, *q\rangle \in \text{Assignments}(\mathcal{P})$
- $*p = *q \Rightarrow \langle *p, *q\rangle \in \text{Assignments}(\mathcal{P})$
- $p = q \Rightarrow \langle *p, *q\rangle \in \text{Assignments}(\mathcal{P})$

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

Reengineering 30

Zuweisungen

- Parameterübergabe wird wie Zuweisung behandelt:
```
f (T t) {...};
f (a);
```
  wird behandelt als:
```
f (t := a);
```
- im Falle von indirekten Aufrufen wird zusätzlich Points-To-Information verwendet:
```
p -> f(a);
```
  wird behandelt als:
```
x.f (t := a);
```
  für alle <p, x> $\in$ Points-To ($\mathcal{P}$)

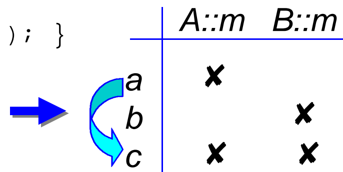
Dominance

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

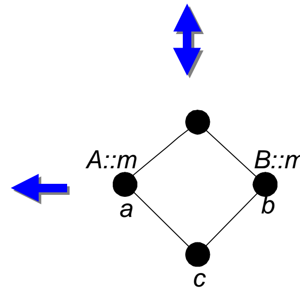
```
class A {int i;
         m (); };
class B : public A { int i;
              m() { i=0;} // B::i
};
B b;
b.m (); // B::m ()
```

Dominance/Hiding

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




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



Erhaltung von Dominance/Hiding

(x, dcl(A::m)) ∊ T, (x, dcl (B::m)) ∊ T
A ist eine transitive Basisklasse von B

dcl(B::m) ® dcl (A::m)

(x, dcl(A::m)) ∊ T, (x, def (B::m)) ∊ T
A=B oder A ist eine transitive Basisklasse von B

def(B::m) ® dcl (A::m)

(x, def(A::m)) ∊ T, (x, def (B::m)) ∊ T
A ist eine transitive Basisklasse von B

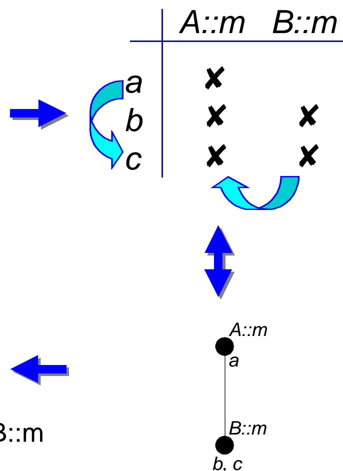
def (B::m) ® def (A::m)

(x, def(A::m)) ∊ T, (x, dcl (B::m)) ∊ T
A ist eine transitive Basisklasse von B

dcl(B::m) ® def (A::m)

Dominance/Hiding

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




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


Schlussfolgerungen aus dem Begriffsverband

- Members im Bottom-Element des Verbands:
  - nicht verwendet (*Person::sSN*), sofern keine Objekte im Bottom-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 Top-Element des Verbands
  - Variablen, über die kein Member verwendet wird (*s*), sofern keine Attribute im Top-Element sind
- Data Members in keinem Begriff ≥ 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