Contents

1 Allgemeine Konzepte

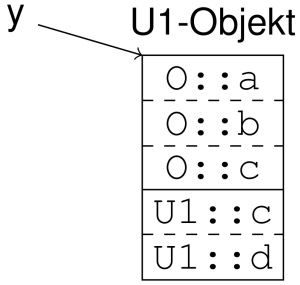
1.1 TOC

- TOC
- Vokabeln
- Dynamische Bindung
  - Upcasts und dynamische Bindung
  - this-Zeiger
- Subobjekte
- v-table
- Abschalten dynamischer Bindung
- Type Casts
- Objektbegriff
- Tücken der dynamischen Bindung

1.2 Vokabeln

- Objekt
  - context abhängig, meistens = “Bezugsobjekt”
- Bezugsobjekt
  - = alle subobjekte
  - ist struct im Speicher
  - 1 pro Instanziierung
  - Größe und relative Positionen innerhalb stehen zur compile time fest
  - Absolute Position im Speicher steht zur run time fest
- statischer Typ
  - Typ zur compile time
- dynamischer Typ

  - Typ zur run time
- up cast/down cast
  - ändern des statischen Types (bei C++ und Java)
- virtual
  - immer, wenn dem Stoustrup kein Wort eingefallen ist
- smart
  - immer wenn etwas einfaches kompliziert gemacht wird
- verdeckt
  - bei dynamisch: = überschrieben
  - bei statisch: = versteckt
- Objektlayout
  - Diagram den Objektes im Speicher. Mit Subobjekte. Optional mit Attributen



  -

1.3 Dynamische Bindung

Member:

- immer statisch

Funktionen:

- öffentlich: immer dynamisch
- privat: immer statisch

Zugriff auf verdeckte Member:

- bei statischer Bindung:
  - upcast dann Zugriff
- bei dynamischer Bindung:
  - C++:
    - `::`
    - "*scope operator*"
    - in Methoden: `K::c()`
    - außerhalb: `o.K::c()`
  - Java:
    - `super`
    - direkte Oberklasse

        * in Methoden: `super.c()`
        * außerhalb: verboten
    - `super` und `::` schalten dynamische Bindung ab

1.3.1 Upcasts und dynamische Bindung

Upcasts schalten *nicht* die dynamische Bindung ab!

und downcasts auch nicht (vermutlich)

1.3.2 this-Zeiger

ganz normaler pointer mit statischer Typ = aktuelle Klasse.

⇒ es gilt dynamische Bindung

1.4 Subobjekte

subobjekte sind der Grund, weshalb statische Bindung geht.

- pro Objekt ein supobjekt pro (geerbter) Klasse
- Subobjekt besteht aus:
    - vpointer (bei Einfachvererbung reicht 1 pro Gesamtobjekt)
    - eigene Member
- Objektlayout:
    - oben: geerbte Subobjekte
    - unten: Subobjekt der eigenen Klasse
- Zugriff über statische offsets zur compile time

1.5 v-table

v-table sind Grund weshalb dynamische Bindung geht.

- global eine v-table pro Klasse
- jedes Bezugsobjekt (bzws. jeden Subobjekt bei Mehrfachvererbung) hat einen v-pointer, der auf die v-table seiner Klasse zeigt
- enthält Einsprungadressen der Methoden:
    - 1 Eintrag pro Methode
    - redefinierte Methoden überschreiben geerbte Methoden
- Position in vtable wird zur compile time bestimmt
- Position des v-pointer: immer gleich im Subobjekt (i.d.R ganz am Anfang)

- Zugriff zur run time:

```
load [x + OFFSET VPTR], reg0
load [reg0 + OFFSET g], reg1
call reg1
```

1.6 Abschalten dynamischer Bindung

bei folgendem wird immer statisch gebunden:

- private methoden
- attribute
- `super` (java only)

was ist mit `::`? (z.B. `o.K::f()`) (Vorsicht: `((K)o).f()` ist was anderes)

In den Fällen wird bereits zur compile Zeit die Adresse der Funktion eingetragen ohne den Umweg über die v-table zu gehen.

1.7 Type Casts

```
class O { int a, b; }
class U extends O { int b, c; }

O x = new U()
```

Durch casts wird *nur* ein anderes Subobjekt ausgewählt

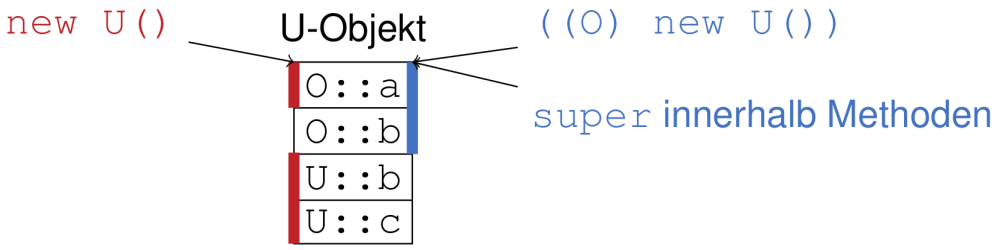


Figure 1: 1 Objekt, 2 Typen

–> member werden sichtbar, Methoden aber nicht.

down cast: unsicher, schlechter stil

1.8 Objektbegriff

Ziel von Objektorientierung:

- Geheimnisprinzip
- Lokalitätsprinzip

Klassifikation von OO-Sprachen

- Typisierung
  - statisch: c++, java
  - dynamisch: self, smalltalk
- Methodenaufrufe
  - echte Klassen: c++, java
  - objektbasiert ohne Klassen: javascript
- Abstraktion
  - statisch Schnittstellen
  - Zugriffsrechte für Klassen

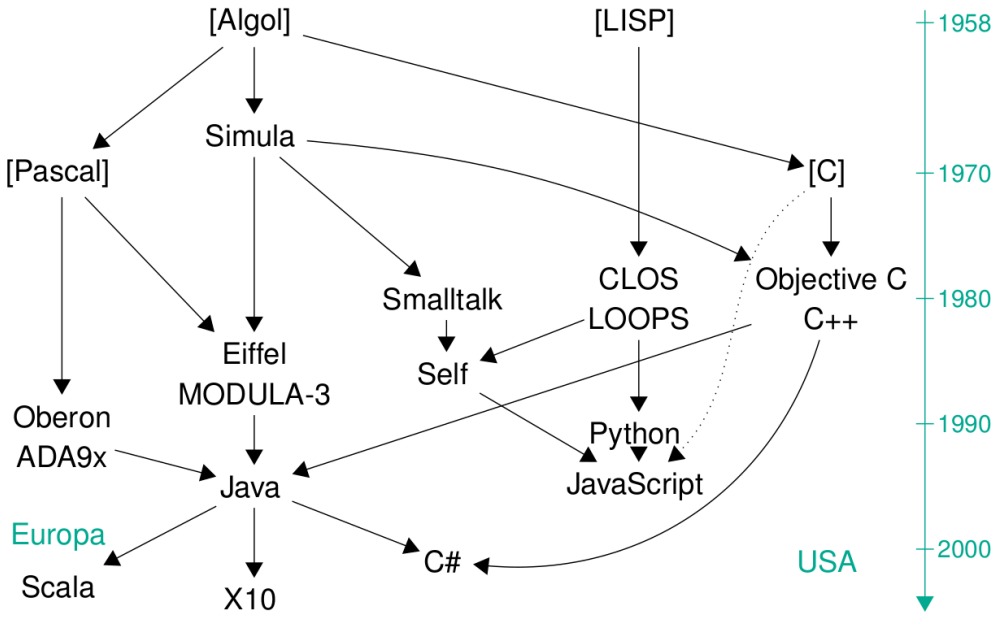


Figure 2: Historische Entwicklung

1.9 Tücken der dynamischen Bindung

Dynamische Bindung greift auch bei Rekursion!

d.h. wenn man versucht mit `super.m()` eine Rekursive Methode `m` im parent aufzurufen, dann kann es sein, dass die zweite Iteration wieder in der cildklasse ist, auch wenn die erste Iteration im parent war. (Vorraussetzung: `m` wird in childklasse überschrieben)

Dynamische Bindung im Konstruktor

Im Funktionen, die im parent Konstrutor aufgerufen werden, können in Childklase überschrieben werden.

- Kontruktoren von der Childklasse mussen immer den Kontruktor vom parent aufrufen.

2 Invarianten und sichere Vererbung

2.1 TOC

- TOC
- Motivation
- Verhaltenskonformanz
  - Vokabeln
  - Realisierung Verhaltenskonformanz
- Weitere Verhaltensbeziehungen
  - Spezialisierung
  - Verhaltenskovarianz
  - Verhaltenskontravarianz
- Inheritance is not Subtyping

2.2 Motivation

```
class U extends O
```

Typkonformanz: Jedes U ist auch als O verwendbar, denn es hat mindestens die selben Members im O-Subobjekt

Problem: U kann O -Methode so umdefinieren, dass sie etwas völlig anderes macht

Stärkere Forderung:

Subtyping: Klientencode funktioniert auch mit U statt O

Andere Namen:

- **Verhaltenskonformanz**
- Klientencode-Wiederverwendung
- Liskov'sches Substitutionsprinzip [LW94]
- Inclusion Polymorphism

2.3 Verhaltenskonformanz

Für Klienten sichere Vererbung erfordert Verhaltenskonformanz

Aus Sicht des Methoden**verhaltens** ist jedes U -Objekt auch ein O -Objekt.

→ Man kann O durch U ersetzen und garantieren, dass nicht kaputt geht

Anwendung: Implementierung von Oberlasse oder Interface

2.3.1 Vokabeln

Klasseninvariante: Wird pro Klasse definiert. Gilt vor und nach jedem Methodenaufruf.

Precondition: Wird pro Methode definiert. Muss gelten, damit die Methode aufgerufen werden darf.

→ *Das, was die Methode braucht*

Postcondition: Wird pro Methode definiert. Gilt nachdem die Methode durchgelaufen ist.

→ *Das, was die Methode tut*

2.3.2 Realisierung Verhaltenskonformanz

Klasseninvariante von Unterklasse U ist stärker als von Oberklasse O:

$$INV(U) INV(O)$$

stärkere Vorbedingung in Oberklasse:

→ *Oberklasse verlangt mehr*

$$PRE(O.m) PRE(U.m)$$

stärkere Nachbedingung in Unterklasse:

→ *Unterklasse leistet mehr*

$$POST(U.m) POST(O.m)$$

→ Unterklasse verlangt weniger und leistet mehr

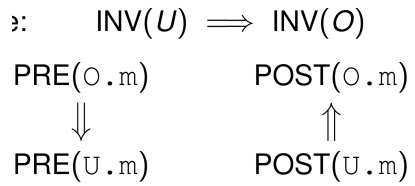


Figure 3: alternative Darstellung

2.4 Weitere Verhaltensbeziehungen

2.4.1 Spezialisierung

= *Gegenteil von Verhaltenskonformanz*

Problem: Verhaltenskonformanz in der Praxis selten, weil schwierig umzusetzen

Häufiger: Spezialisierung. (leistet auch etwas, jedoch was ganz anderes)

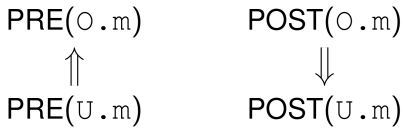


Figure 4:

Anwendung: Implementierung für Spezialfälle

2.4.2 Verhaltenskovarianz

Implikation in Vererbungsrichtung

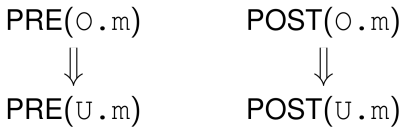


Figure 5:

2.4.3 Verhaltenskontravarianz

Implikation entgegen Vererbungsrichtung

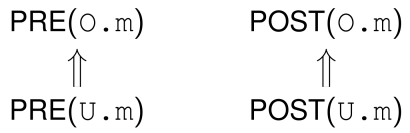


Figure 6:

2.5 Inheritance is not Subtyping

Inheritance → Spezialisierung

Subtyping → Verhaltenskonformanz

→ Trennung von Klassen und Typen

Java: Interfaces für Subtyping, Klassenvererbung für Inheritance

3 Mehrfachvererbung

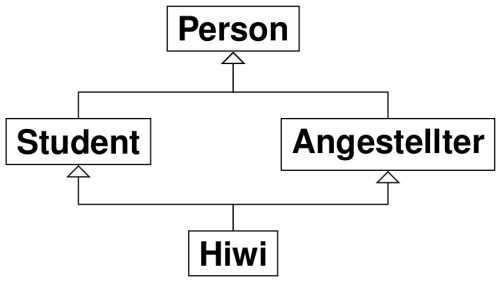


Figure 7:

3.1 TOC

- TOC
- Mehrfachvererbung in C++
  - Nicht-virtuelle Vererbung in C++
  - virtuelle Vererbung in C++
- Subobjektgraphen
  - Konstruktion von Subobjektgraphen
- Static Lookup
  - Formale Definition
- Dynamic lookup
- Static lookup vs Dynamic lookup

3.2 Mehrfachvererbung in C++

C++ erlaubt Klassen-Mehrfachvererbung

Beispielanwendung: GUI-Fenster mit Rand und Menüleiste

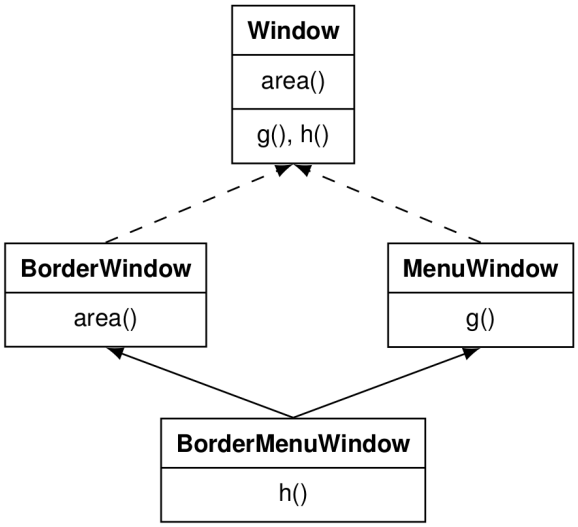


Figure 8:

```
class Window {
    virtual int area() { ... }
    virtual void g() { ... }
    virtual void h() { ... }
};
class BW : public virtual Window {
    virtual int area() { ... }
};
class MW : public virtual Window {
    virtual void g() { ... }
};
class BMW : public BW, public MW {
    virtual void h() { ... }
};


BMW* bmw = new BMW(); // bmw ist Zeiger auf BMW -Objekt
MW* mw = bmw; // Impliziter Upcast
mw->area(); // Ruft BW: : area( ) auf !
```

3.2.1 Nicht-virtuelle Vererbung in C++

- Standardvererbungstyp
- Durchgezogene Linie im Klassendiagramm
- nur von historischer Bedeutung

  `class BW : public W { ... };`

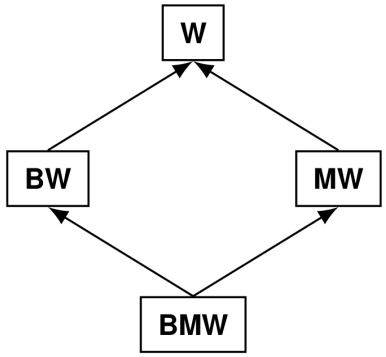


Figure 9: Klassendiagramm mit nicht-virtueller Vererbung

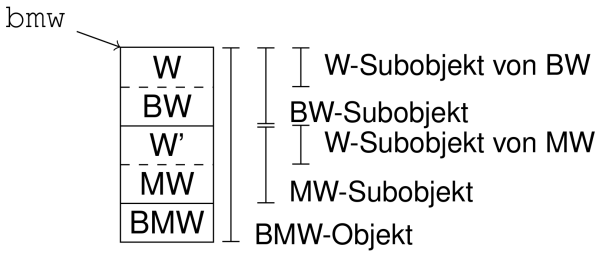


Figure 10: Subobjekte im Speicher. Mit Pointer `bmw` auf Hauptobjekt

3.2.2 virtuelle Vererbung in C++

- Schlüsselwort `virtual`
- Gestrichelte Linie im Klassendiagramm
- Unterklassenobjekt enthält Zeiger auf Oberklassenobjekt
- in der Praxis verwendet

  class BW : public virtual W { … };

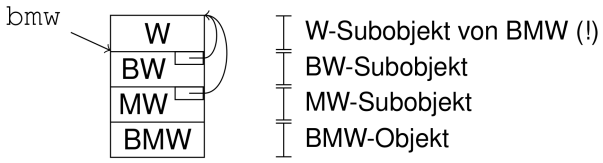


Figure 11: Virtuelle Vererbung im Speicher

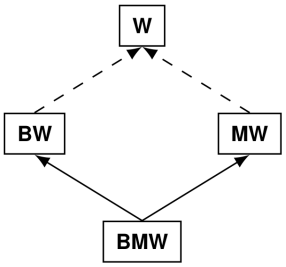


Figure 12: Klassendiagramm mit virtueller Vererbung

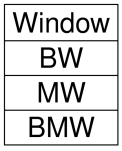


Figure 13: Objektlayout bei virtueller Vererbung

3.3 Subobjektgraphen

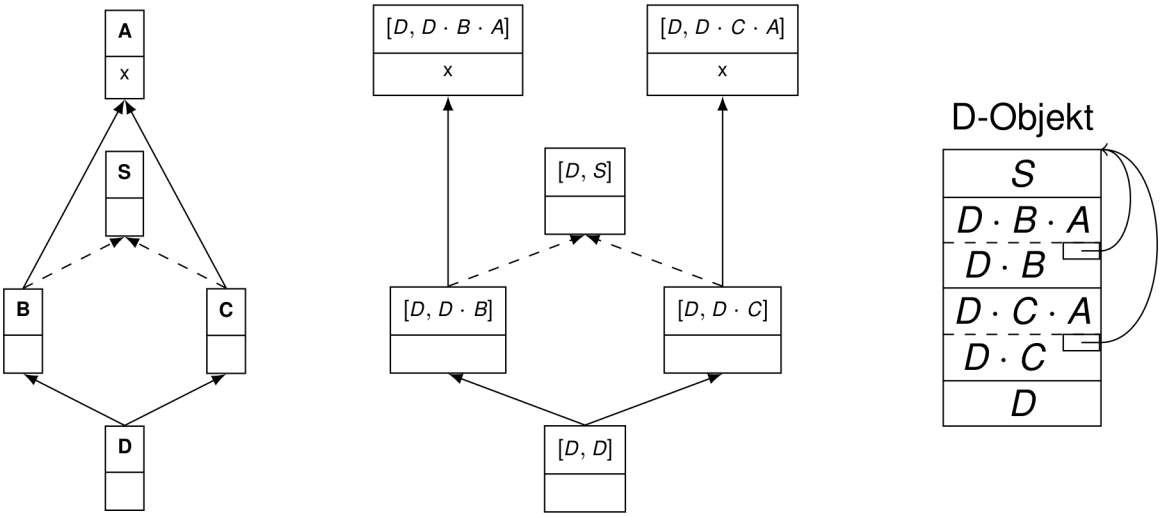


Figure 14: *links*: Klassendiagramm, *mitte*: Subobjektgraph, *rechts*: Objektlayout

Warum: Bei nicht-virtueller Vererbung entstehen komplexe Subobjekte, die z.T Duplikate enthalten

- Knoten [C, C · B · A]:
  - Subobjekte
  - "*Das C B A Subobjekt eines C-Objektes*"
- Kanten $S \to S'$ (durchgezogen)
  - *default Vererbung*
  - S' ist direkt in S enthalten
- Kanten $S \dashrightarrow S'$ (gestrichelt)
  - *virtuelle Vererbung*
  - S hat pointer auf S'

3.3.1 Konstruktion von Subobjektgraphen

Gegeben: Klassenhierarchie mit virtueller und nicht-virtueller Vererbung

Gesucht: Zugehöriger Subobjektgraph

Definitionen:

Symbol	Bedeutung
$X <_N Y$	X erbt **direkt** und **nicht-virtuell** von Y

Symbol	Bedeutung
$X <_V Y$	X erbt **direkt** und **virtuell** von Y
$X < Y$	X erbt **direkt** von Y
$X <^* Y$	X erbt **indirekt** von Y, reflexive transitive Hülle von $<$
$X \sqsubset_p Y$	Pfad der Länge p von X nach Y, $\sqsubset_1$ ist Kante

3.3.1.1 Ermitteln der Knoten

- $[X, X]$ ist *Gesamtobjekt*
- $[X \mathring{u} X \mathring{u} Z]$ ist Subobjekt
  - Wenn $[X, \mathring{u} Y]$ Subobjekt
  - und $Y <_N Z$
- $[X, Z]$
  - wenn $[X,]$ Subobjekt
  - und $_Y$:
    - $X <^* Y$
    - und $Y <_V Z$

3.3.1.2 Ermitteln der Kanten

$\sqsubset_1$: Kante im Graph (sowohl getrichelt als auch durchgezogen)

- $[X,] \sqsubset_1 [X, \mathring{u} Y]$
  - *durchgezogen*
- $[X, \mathring{u} Y] \sqsubset_1 [X, Z]$
  - wenn $Y <_V Z$
  - *gestrichelt*

3.4 Static Lookup

Gegeben: Klasse `C` in Klassenhierarchie, Membername `m`

Gesucht: „Speziellstes“ Subobjekt eines `C`-Objekts, in dem `m` deklariert ist

Problem: Klasse `C` alleine reicht nicht. Man muss bei Subobjekt anfangen zu suchen.

`lookup( ,m) = ':`

- `:` Ausgangssubobjekt
- `m`: gesuchte Methode
- `'`: gesuchtes Subobjekt
- Beispiel: `lookup([C,C],x) = [C,C·B]`

Falls mehreindeutig: Wähle Dominantes Subobjekt. Falls immernoch mehreindeutig: `return`

Dominanzrelation: Das dominante Subobjekt ist spezieller. D.h. es steht gibt im Subobjektgraphen einen Pfad vom dominanten Supobjekt zum dominiertem Subobjekt.

3.4.1 Formale Definition

$$lookup(, m) = min(Defs(, m))$$

Bemerkung:

- $min()$ ist das kleinste Subobjekt bzgl. $\sqsubset$
- Historisch wurde min () auch „größtes“ Subobjekt genannt, und max statt min geschrieben, weil es physikatlisch am größten ist

Definitionen:

- $= min(S)$:
  - ist dominantes Subobjekt aus Subobjekt-Menge S
  - d.h. $\exists$ Pfad von zu jedem anderen Subobjekt in S
  - Bsp. $min(\{[D, D\mathring{u}C\mathring{u}A], [D, D]\}) = [D, D]$
- $Member(C)$:
  - die Menge aller in Klasse C deklarierten Member
- $ldc([C, \mathring{u}A]) = A$
  - A ist „least derived class“
  - = letzter Teil in Subobjektschreibweise: [C,C · B · **A**]
- $Defs(, m)$
  - $= \{' \sqsupseteq |mMember(ldc('))\}$
  - Menge aller erreichbaren Subobjekte, die Member m enthalten

3.5 Dynamic lookup

$$dynBind(, f) = min(Defs([mdc(), mdc()], f))$$

mit:

- $mdc([C, \mathring{u}A]) = C$
  - „most derived class“
  - = erster Teil in Subobjektschreibweise: [C,**C** · B · A]

3.6 Static lookup vs Dynamic lookup

Lemma

$$dynBind(, f) = lookup([mdc(), mdc()], f)$$

Dynamische Bindung ist wie statischer Lookup angewendet auf den dynamischen Typ

4 Implementierung von Mehrfachvererbung

Reminder aus Abschnitt 1:

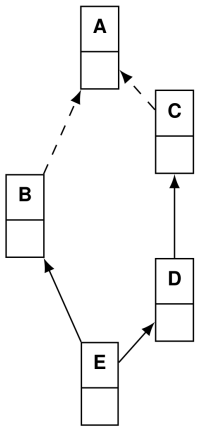


Figure 15:

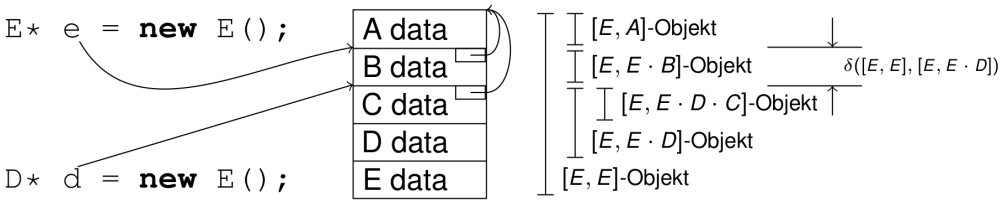


Figure 16: C++-Objektlayout

4.1 TOC

- TOC
- C++ Type Casts
  - nicht-virtuelle Mehrfachvererbung
  - virtuelle Mehrfachvererbung
  - Ausnahmen
- Implementierung vtables
  - Mehrfachvererbung mit vtables
- **(Z)** - Mehrfachvererbung in Java

4.2 C++ Type Casts

- nicht-virtuelle Einfachvererbung Nullcode
- nicht-virtuelle Mehrfachvererbung Verschiebung des Zeigers
- virtuelle Mehrfachvererbung Verfolgen des Subobjektzeigers

4.2.1 nicht-virtuelle Mehrfachvererbung

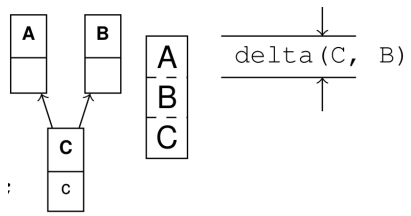


Figure 17: nicht-virtuelle Mehrfachvererbung

```
class C : A, B { int c; };
B* pb; C* pc;

pb = pc;
```

wird zu:

```
pb = (B*) (((char *) pc) + delta(C, B));
```

`delta(C, B)`

- Offset des `B`-Subobjekts in einem `C`-Subobjekt
- zur compile time bekannt
- `delta(C, B) = sizeof(A)`

4.2.2 virtuelle Mehrfachvererbung

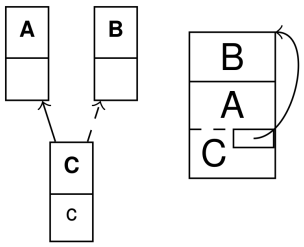


Figure 18: virtuelle Mehrfachvererbung

```
class C : A, virtual B { int c; };
B* pb; C* pc;

pb = pc;
```

wird zu:

```
pb = *((B**) (((char *) pc) + offset(B_ptr)));
```

- `offset(B_ptr)`
  - Offset des `B`-Subobjekt**zeigers** in `C`-Objekt
  - Frage: zur Übersetzungszeit bekannt?

4.2.3 Ausnahmen

- Nullzeiger werden nicht verschoben
  - Zur Laufzeit: if null return null

4.3 Implementierung vtables

- Enthält Einsprungadressen für Methoden
- pro Klasse eine statische vtable "*vtbl*" bei Einfachvererbung
- jedes Objekt enthält Zeiger auf vtbl seiner Klasse
- Indizes für Methodennamen klassenhierarchieweit eindeutig
  - wird vom compiler vorgegeben
- beim Methodenaufruf zusätzlicher Indirektionsschritt:
  - `C* pc = new C(); pc->h(42);` wird zu `(*(pc->vptr[2]))(pc, 42)`

4.3.1 Mehrfachvererbung mit vtables

- eine `vtable` pro Subobjekt-Typ
- jedes Subobjekt enthält Zeiger auf vtbl seines Subobjekt-Typs
- jede Zeile enthält zusätzlich `delta`

Problem: Methoden müssen ihren statischen Typ kennen, um `this` korrekt casten zu können.

Lösung: pro Methode wird `delta` ebenfalls in `vtable` abgelegt. Damit kann dann beim Methodenaufruf gecastet werden.

Alternative zu Deltas: Thunks

Statt Deltas in vtable generiert Compiler bei Bedarf weitere Methode (Thunk), die this -Pointer entsprechend verschiebt.

Ein Thunk ist sozusagen ein Wrapper

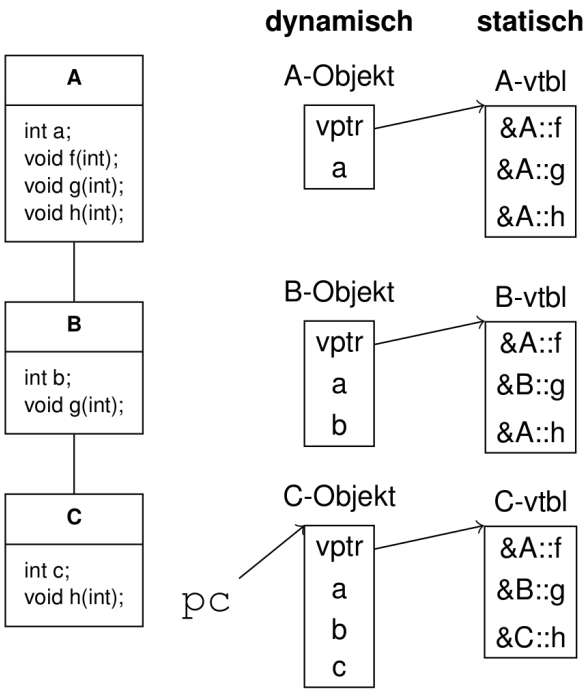


Figure 19: vtables und Klassehierarchie

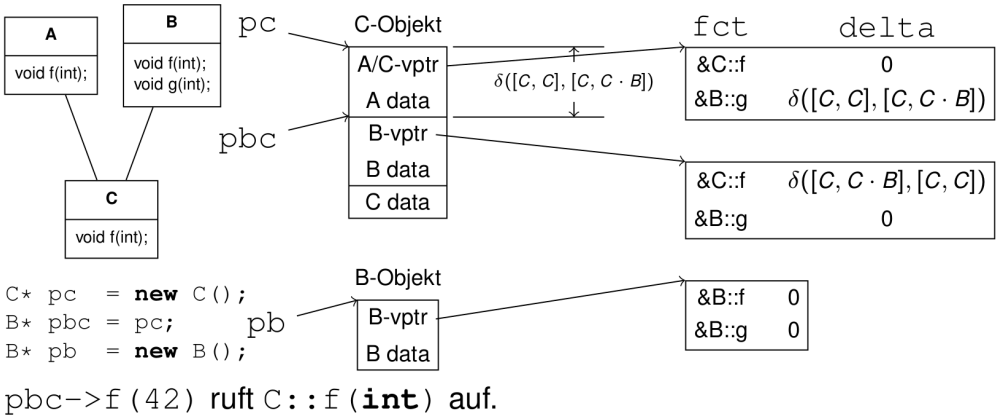


Figure 20: Mehrfachvererbung mit vtables

4.4 (Z) - Mehrfachvererbung in Java

via interfaces

ab javu 8: Default-Methoden. Damit können auch Methoden mehrfachvererbt werden

5 weitere features von Objektorientierung

5.1 TOC

- TOC
- Überladungen
  - lookup
  - aufruf
  - Smart-Pointer
- Innere Klassen
  - statische innere Klasse:
  - Dynamische innere Klassen
- Generics
  - Typschranken
  - Type Erasure
  - Vererbung bei generischen Klassen
  - Wildcards

5.2 Überladungen

Jede überladene Variante hat eigenen vtable-Eintrag

Übliche Anwendung: Überladener Konstruktor

5.2.1 lookup

- Java:
  - static lookup direkt mit Signatur
- C++:
  - static lookup mit Namen und bei Mehrdeutigkeit vergleich der Signatur
  - bei mehreren Kandidaten: wähle spezifischste Methode

“spezifischer”: Alle parameter sind Unterklassen

Java findet andere Methode als c++

5.2.2 aufruf

ganz normale Methodenaufruf mit dynamischer Bindung

Problem: Funktionen, die in Kindklasse gleich heißen, aber andere Signatur haben, sehen aus als würden sie uberschreiben, tun sie aber nicht

5.2.3 Smart-Pointer

Realisiert durch Überladung des `->` operators

5.3 Innere Klassen

Klassendefinition innerhalb einer anderen Klassendefinition

übliche Anwendung: Iteratoren

5.3.1 statische innere Klasse:

```
* statisch
* Zugriff nur auf statische Member der Äußeren Klasse
* Äußeren Klasse kann auf nur statische Member zugreifen
* Erstellung: `new AußenKlasse.InnenKlasse()`
    * Warum?

class A {
    int x = 42;
    private static int y = B.u; // Okay
    public static class B {
        private static int u = 17; // Okay
        int z = x; // Fehler
    }
}
A.B b = new A.B();
```

5.3.2 Dynamische innere Klassen

- dynamische innere Klasse darf auf nicht-statische Member der äußeren Klasse zugreifen, auh private
- Jede Instanz von innerer Klasse hat impliziten Verweis auf Instanz der äußeren Klasse, ansprechbar über Outer.this

class C { private int x = 42; private static int y = private int w = 17; class D { int z = x + y; // Okay int w = C.this.w; // Okay } } C c = new C(); C.D d = c.new D();

5.4 Generics

Ziel: bessere Wiederverwendbarkeit in Libaries

- Java, C#: Bounded Polymorphism:
  - Parameter muss Unterklasse einer gewissen Klasse sein
  - Verfügbarkeit von Methoden kann auch in generischen Klassen statisch geprüft werden
- Eiffel, C++: Parameterklasse kann mit irgendwas instanziiert werden
  - generische Klasse nicht isoliert typcheckbar, nur konkrete Instanzen
  - Bibliotheken nicht statisch typprüfbar

5.4.1 Typschranken

Ab java 1.5

Am Anfang von Klassen oder Methoden steht eine Typschranke und innerhalb der Klaase/Methode kann der Typ dann wie eine nermale Klaase benutzt werden.

5.4.2 Type Erasure

implementierung in Java

Zur compile time alle verwendeten Varianten von Methoden vom compiler angelegt. Im Byte cod sieht man dann nur noch die Überladungen, und nichs mehr von den generics

5.4.3 Vererbung bei generischen Klassen

Vererbung ganz normal

Typschranken dürfen spezieller werden

5.4.4 Wildcards

für generics im Methodenparameter

6 Tyrannei der dominanten Dekomposition

6.1 TOC

- TOC
- Warum Dekomposition
- bisherige Arten von Dekomposition
- Lösungsansätze

6.2 Warum Dekomposition

Ziel: Typsicherheit und Lokalitätsprinzip sind schwer zu veneinbaren.

Problem: Bei Erweiterungen in verschiedenen Hierarchien: Eine Hierarchie dominiert die andere

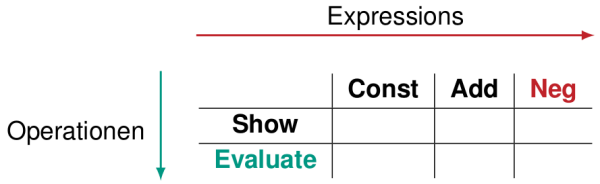


Figure 21: Man kann nur schwer erweiterbare Expsennions *und* Operationen haben

6.3 bisherige Arten von Dekomposition

Objektorientierte Dekomposition:

```
* Operationen als dynamisch gebundene Methoden
* Neuer Datentyp kein Problem
* Neue Operation erfordert Änderung aller Expression-Klassen
*   **Datentypen dominieren Operationen**
```

Funktionale Dekomposition:

```
* Operationen als Visitor-Objekte
* Neue Operation kein Problem
* Neuer Datentyp erfordert Änderung aller Visitor-Klassen
*   **Operationen dominieren Datentypen**
```

6.4 Lösungsansätze

- Multimethoden
  - z.B. in MultiJava
- Traits, Mixins und abstrakte Typmember
  - z.B. in Scala
- Virtuelle Klassen
  - z.B. in Beta

7 Programmanalyse

Ziel:

- Optimierung:
  - wenig Speicher
  - entfernen von dynamischer Bindung
- Verständnis & debugging

Beispiel:

```
class A { void f() { ... } }
class B extends A { void f() { ... } }
A p = new B();
p.f();
```

`A.f()` wird neimals aufgerufen und kann entfernt werden

und `B::f()` kann statisich gebunden werden

7.1 TOC

- TOC
- Eigenschaften
  - Fluss-Sensitivität
  - Kontext-Sensitivität
- Rapid Type Analysis
  - Call Graph
  - Umgang mit dynamischer Bindung
  - Reduktion des Call-Graphs
  - RTA als Constraint-Problem
  - Fazit
- Points-To-Analyse
  - Points-To-Graph
  - nach Anderson
  - nach Stengaard

7.2 Eigenschaften

- Statische Analyse: ohne Kompilat auszuführen
- Whole-Program-Analyse: Gesamtes Programm (inklusive Klassenhierarchie) bekannt

7.2.1 Fluss-Sensitivität

Flusssensitive Programmanalysen

- Beachten Reihenfolge der Anweisungen
- Analyseergebnis pro Programmpunkt
- präziser und aufwendiger als Fluss-insensitive Analysen

Fluss**in**sensitive Programmanalysen

- Berechnen Analyseergebnis pro Programm
- gleiches Ergebnis für alle Programmpunkte
- Sind unpräziser, aber schneller
- Ergebnis für p im Beispiel: p kann auf o 1 oder o 2 zeigen

Beispiel:

```
Object p = new Object(); // o 1
Object q = new Object(); // o 2
p = q;
```

7.2.2 Kontext-Sensitivität

Kontextsensitive Programmanalysen

- Beachten den Aufrufkontext einer Methode
- Berechnen Analyseergebnis pro Aufrufkontext und pro Methode
- Ergebnis im Beispiel:
  - Bei Aufruf aus f() kann x nur auf o 1 zeigen
  - Bei Aufruf aus g() kann x nur auf o 2 zeigen
- Sind präziser und aufwendiger

Kontext**in**sensitive Programmanalysen

- Ignorieren den Aufrufkontext einer Methode
- Berechnen Analyseergebnis pro Methode
- Ergebnis im Beispiel:
  - x kann auf o 1 oder o 2 zeigen
- Sind unpräziser, aber schneller

Beispiel:

```
Object foo(Object x) { return x; }
void f() { Object p = new Object(); /* o 1 */ foo(p); }
void g() { Object q = new Object(); /* o 2 */ foo(q); }
```

7.3 Rapid Type Analysis

Einfach und schnell, aber wirkungsvoll

Ziele: Entfernen toter Members, Eingrenzen dynamischer Bindung

Idee: Wenn nie ein C-Objekt erzeugt wird, dann kann auch nie eine C-Methode aufgerufen werden! (außer von Unterklassen von C)

7.3.1 Call Graph

Knoten: Alle Methoden `C::m(P1, ..., Pn)`, inklusive main()-Methode, Konstruktoren und Überladungen

Kanten: Kante von `C::f()` nach `D::g()` `C::f()` hat einen Aufruf `d.g()` mit d vom statischen Typ `D`

```
1  class A {
2    void f(int x) { }
3    void g(A p) { p.f(42); }
4  }
5  class B extends A {
6    void f(int x) { }
7  }
8  void main() {
9    A p = new A();
10   p.g(p);
11 }
```

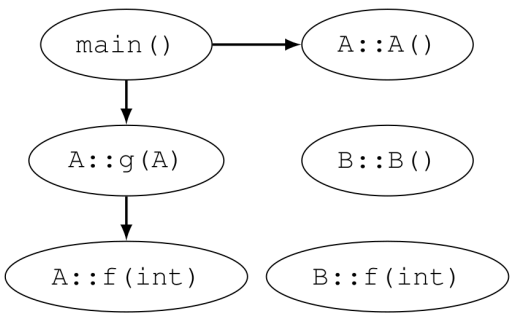


Figure 22: *links*: Beispielprogramm, *rechts*: zugehöriger call graph

Achtung bei dynamischer Bindung: Alle potentiellen Ziele des Aufrufs `d.g()` müssen berücksichtigt werden, also auch alle `E::g()` für alle Unterklassen `E` von `D`

7.3.2 Umgang mit dynamischer Bindung

Konservative Approximation:

Es wird nie ein möglicher Aufruf vergessen, aber manche Aufrufe werden fälschlich als möglich angenommen.

Menge der Aufrufziele ist manchmal zu groß, aber nie zu klein!

7.3.3 Reduktion des Call-Graphs

Problem: Durch dyn. Bindung viele unnötige Kanten im Call-Graph

Lösung: finde Konstruktoren, die **nie** aufgerufen werden und entferne alle Methoden dieser Klasse

```
1  class A {
2    void f(int x) { }
3    void g(A p) { p.f(42); }
4  }
5  class B extends A {
6    void f(int x) { }
7  }
8  void main() {
9    A p = new A();
10   p.g(p);
11 }
```

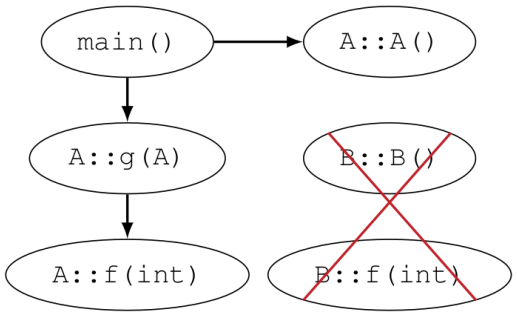


Figure 23: Klasse, die niemals instanziiert wird, kann entfernt werden

Algorithmus:

1. starte mit Call-Graph (, der dynamische Bindung berücksichtigt)
2. markiere alle Kanten zu virtuellen Methoden als *verboten*
3. markiere alle Kanten **von** `main()` **zu** Konstruktoren als *erlaubt*
4. while (keine neuen Kanten):
    5. Starte bei `main()` und markiere Knoten, die über *erlaubte* Kanten erreichbar sind
    6. Wenn markierter Knoten Konstruktor von Klasse C ist, markiere alle Kanten als erlaubt, die die **von** markierten Knoten **zu** Methoden der VTable von C zeigen als *erlaubt*
5. Entferne alles, was von `main()` aus nicht über erlaubte Kanten erreichbar isr

Achtung: VTable von C kann auch ererbte Methoden enthalten Methoden der Oberklasse können erlaubt werden, obwohl nie ein Oberklassenobjekt erstellt wird

Beispiel:

7.3.4 RTA als Constraint-Problem

RTA auch als Constraint-Problem formulierbar:

```
1  class A {
2    int f() { return 42; }
3  }
4  class B extends A {
5    int f() { return 17; }
6    int f(A x) { return x.f(); }
7  }
```

```
void main() {
  B p = new B();
  int r1 = p.f(p);
  int r2 = p.f();
  A q = p;
  int r3 = q.f();
}
```

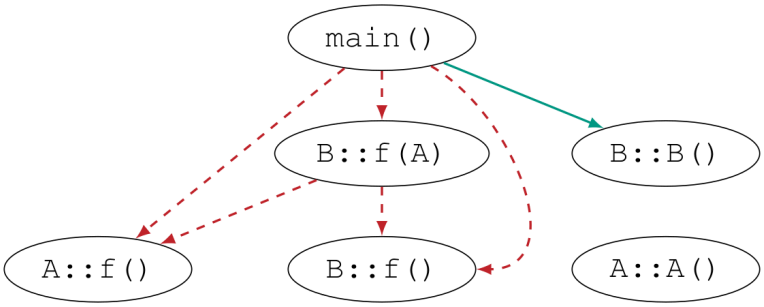


Figure 24: Initialisierung

- Menge R: Menge der „lebendigen“ Methoden
- Menge S: Menge der „lebendigen“ Klassen

7.3.5 Fazit

- sehr schnell
  - da fluss- und kontext-insensitiv
  - Overhead bei Kompilierdauer $< 5\%$
- Sehr effektiv bei Klassenbibliotheken
- sehr ungenau:
  - von vieln Objekten wird **nicht** jede Methode aufgerufen
  - Lösung Zeiger-Analyse (nächstes Kapitel)

7.4 Points-To-Analyse

Ziel: Bestimme für jeden Zeiger p , auf welche Objekte er zeigen könnte

Points-To-Menge

- *= “PT-Menge”*

```
1  class A {
2    int f() { return 42; }
3  }
4  class B extends A {
5    int f() { return 17; }
6    int f(A x) { return x.f(); }
7  }
```

```
void main() {
  B p = new B();
  int r1 = p.f(p);
  int r2 = p.f();
  A q = p;
  int r3 = q.f();
}
```

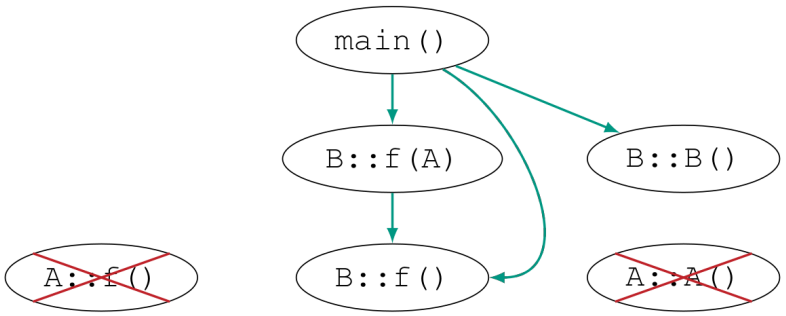


Figure 25: Ergebnis

$$\frac{}{\texttt{main} \in R} \qquad \frac{}{\texttt{Main} \in S} \qquad \frac{M \in R \qquad \textbf{new}\ \texttt{C()} \in body(M)}{C \in S}$$

$$\frac{M \in R \qquad e.m(x) \in body(M) \qquad C \leq staticType(e) \qquad C \in S \qquad lookup(C, m) = M'}{M' \in R}$$

Figure 26: Inferenzregeln zur Bestimmung der Mengen R, S

- $$PT(p) = \{o_1, o_2, , o_n\}$$
- Menge der **Objektrepräsentanten** auf die **dieser pointer** zeigen könnte

Objektrepräsentanten

- Problem: Potentiell werden unendlich viele Objekte erstellt (z.B. wegen `new` in Schleife)

betrachte nur einen Repräsentanten von jedem `new` im Programm

fun fact: Exakte Points-To-Analyse ist sogar mit Repräsentantentrick unentscheidbar [Ramalingam 1994]

7.4.1 Points-To-Graph

- Knoten
  - Objektrepräsentanten: Durchnummerieren o_i
  - Zeigervariable (pointer p): Name der Variablen
- Kanten
  - Kante po_i , wenn p bei irgendeiner Ausführung des Programms auf o_i zeigen *könnte*

Triviallösung: Jeder Zeiger zeigt auf alle Objektrepräsentanten.

mehr Berechnungsaufwand für bessere Präzision

7.4.2 nach Anderson

Idee: nich Zuweisung $p = q$ kann alles auf p zeigen, was vonher auf q zeugen konnte.

$$p = qPT(q)PT(p)$$

damit bekommt man ein Mengenungleichungssystem

```
Object p = new Object(); // {o1}   PT(p)
Object q = p;   // PT(p)   PT(q)
Object r = q;   // PT(q)   PT(r)
p = r;  // PT ( r )   PT(p)
Object s = new Object(); // {o2}   PT(s)
r = s;  // PT(s)   PT(r)
```

beachte: $q = ppq$ (Veriablen sind vertauscht)

7.4.2.1 Lösen des Mengenungleichungssystems

gegeben:

- Objektrepräsentanten $o_1, , o_n$
- points-to Mengen $M_1, , M_m$
- Ungleichungen als Liste
  - $M_i \subseteq N_i$
  - $\{o_i\} \subseteq N_i$

gesucht:

- Kleinste Mengen, die Ungleichungen erfüllen

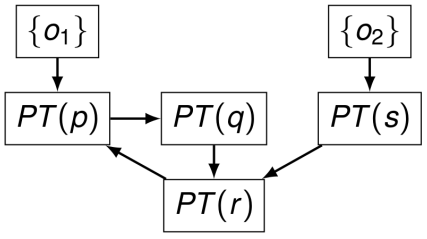
Algorithmus "*Fixpunktiteration*":

1. (optional) Male Ungleichungen als gerichteten Graphen
2. initialisiere alle Mengen als leere Menge
3. repeat until no more change:
   3. wähle Ungleichungen (= Kante im Graphen) $M_i \subseteq N_i$
   4. setze in Ergebnisliste $M_i = M_i N_i

beachte: Kanten müssen wiederholt werden

Optimierung:

- beginne mit Objektrepräsentanten (weil alle anderen =)
- Zyklen möglichst spät



$$PT(p) = \emptyset$$
$$PT(q) = \emptyset$$
$$PT(r) = \emptyset$$
$$PT(s) = \emptyset$$

```
1  void main() {
2    Object p = new Object(); // {o1} ⊆ PT(p)
3    Object q = p;            // PT(p) ⊆ PT(q)
4    Object r = q;            // PT(q) ⊆ PT(r)
5    p = r;                   // PT(r) ⊆ PT(p)
6    Object s = new Object(); // {o2} ⊆ PT(s)
7    r = s;                   // PT(s) ⊆ PT(r)
8  }
```

Figure 27: *links*: Ungleichungsgraph, *rechts*: Ergebnisliste am Anfang, *unten*: Beispielprogramm

Behandlung von Attributen:

$$
\begin{aligned}
PT(p) &= \{o_1, o_2\} \\
PT(q) &= \{o_1, o_2\} \\
PT(r) &= \{o_1, o_2\} \\
PT(s) &= \{o_2\}
\end{aligned}
$$

Figure 28: Ergebnisliste am Ende

Attribute sind pointer

Berechne PT-Menge für jedes Attribut jedes Objektrepräsentanten:

Beispiel (Einfachvorkettete Liste)

```
class C { Object data; C next; }

C c = new C(); // o1 mit PT-Mengen für o1.data, o1.next
c.data = new Object(); // o2
c.next = new C();   // o3 mit PT-Mengen für o3.data, o3.next
C c2 = c.next;
```

Behandlung Funktionen

Variablen:

jede Funktion mit Signatur $RA.f(p)$ verwendet neue, implizite pointer:

- $this_f^A$
- ret_f^R
- p_f^A (falls Name p nicht eindeutig)

Behandlung dynamischer Bindung

Problem: Beim Aufruf ist eine Methode ist nicht klar, welche Methode das ist, weil man das Bezugsobjekt nicht kennt

Lösung: Verwende jedes Bezugsobjekt in $PT(o)$

Wenn $PT(o)$ wächst, kommen hier Methoden dazu

7.4.3 nach Stengaard

Idee: Mache Graph ungenauer, aber dafür kleiner

$$p = qPT(p) = PT(q)$$

Effiziens (mit union-find): $O(n\mathring{u}(n)), (n := Programmgre)$

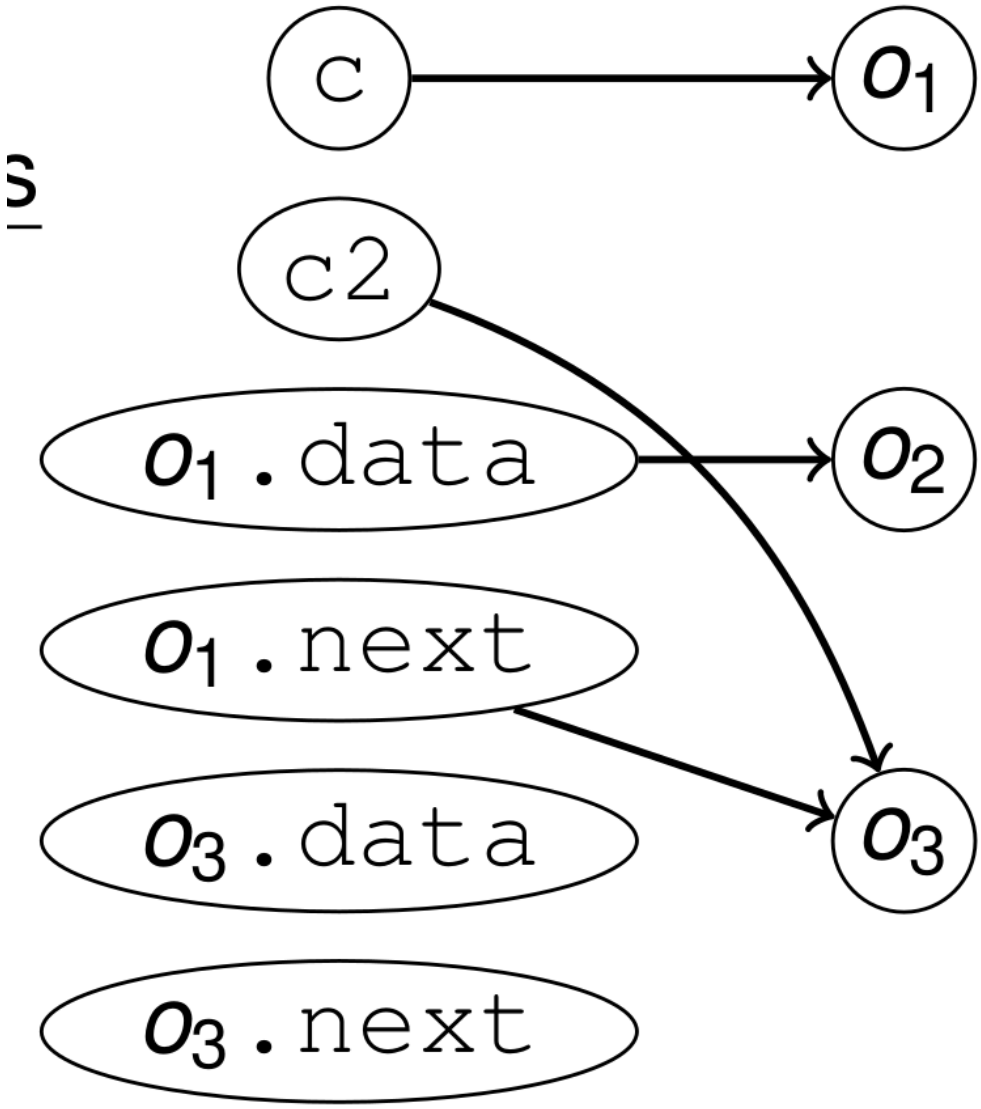


Figure 29: PT-Graph

8 Typsysteme für Objektorientierung

Anm. d. Red.: Kommt zwar vermutlich in Klausur dran, aber brauche ich in meinem Programmieralltag nicht. Daher wird hier nur sehr oberflächlich darauf eingegangen.

8.1 TOC

- TOC
- Basis
- Erweiterungen des Typsystems

8.2 Basis

8.3 Erweiterungen des Typsystems

Bisher: Objekte als Records mit Subtyping und veränderbarem Zustand

Erweiterungen:

- this -Zeiger für Methoden
- Generische Klassen
  - Was ist der Typ einer generischen Klasse?
  - Kann man zwischen generischen Klassen oder ihren Instanzen Subtypen definieren?
- Rekursive Typen
  - Wie kann man einen Typ in sich selbst wiederverwenden?
  - `class Node { int data; Node next; }`
- Abstrakte Klassen
  - Was ist der Typ einer abstrakten Klasse?
  - Wie garantiert man, dass es eine Implementierung gibt?

9 mitschrieb-foo

Preview: https://rawgit.com/wotanii/mitschrieb-foo/master/build.html

Dies ist meine Prüfungsvorbereitung für

Fortgeschrittene Objektorientierung bei *Prof. Snelting* am *KIT*

Für Korrekturen und Hinweise bis zum 2.10.2017 bin ich dankbar. Danach interessiert es mich nicht mehr.

Schwerpunkt meiner Vorbereitung ist alles, was:

- praxisrelevant ist
- sicher in der Klausur drann kommt

Dementsprechend ziele ich auf eine 2.3, rechne jedoch mit eine 3.3 als Endnote.

Am Ende werde ich das hier ausrdicken und mit in tie Klausur nehmen

9.1 Kompilierung

Das markdown kompiliere ich mit `pandoc` mit folgenden Parametern:

```
-f markdown-raw_tex+tex_math_single_backslash --mathjax --smart
--standalone --normalize
```

und das Ergebnis schaue ich mir mit Firefox (ggf. mit Addon *Owl - Dark Background*) an.

Bonus: dieses CSS: https://gist.github.com/killercup/5917178

9.2 Workflow

Atom Texteditor mit Plugins:

- markdown-preview-enhanced
- markdown-img-paste

wobei ich markdown-img-paste erweitert habe, sodass eine default width von 500 gosetzt wird.

Wenn ich nur lesen will, benutze ich dieses Nemo-Plugin: https://github.com/wotanii/nemo-action-pandoc2ff