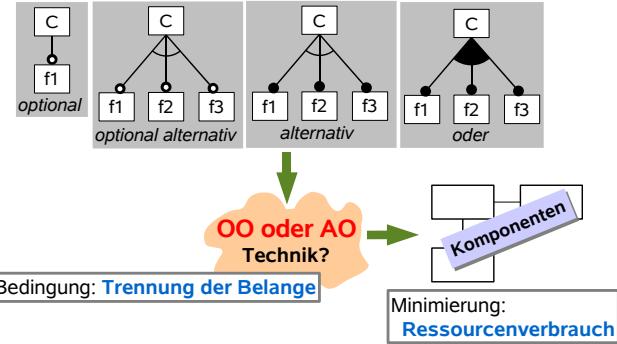


### Trennung der Belange mittels OO oder AO Entwurf

1

### Umsetzung von Variabilität

... in wiederverwendbaren Plattformkomponenten auf Basis von objekt- und aspektorientierten Sprachmitteln (OO und AO).



### Was macht Objektorientierung aus?

Klassifizierung nach P. Wegner [1]:

*object-oriented* = *data abstraction*  
+ *abstract data types*  
+ *type inheritance*

- wesentliches Alleinstellungsmerkmal: **Vererbung**
- im Fall von C++ sind zu untersuchen:
  - einfache/mehrstufige Vererbung
  - virtuelles Vererben
  - dynamisches Binden

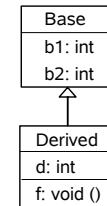


© 2006 Olaf Spinczyk

3

### (Einfach-)Vererbung

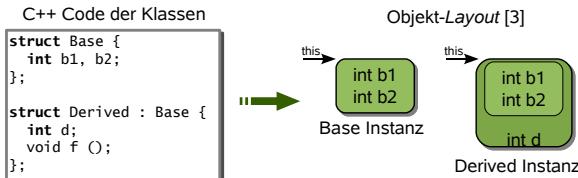
- eine abgeleitete Klasse erbt von einer Basisklasse
  - geerbt werden Attribute, Methoden, ...
- statt einer Instanz der Basisklasse kann immer auch eine Instanz der abgeleiteten Klasse verwendet werden
  - gilt nicht umgekehrt!
  - möglichst kompatibles Objekt-Layout
  - Liskov'sches Substitutionsprinzip [2]
- Methoden können hinzugefügt oder überdefiniert werden



© 2006 Olaf Spinczyk

4

## (Einfach-)Vererbung – Ressourcen (1)



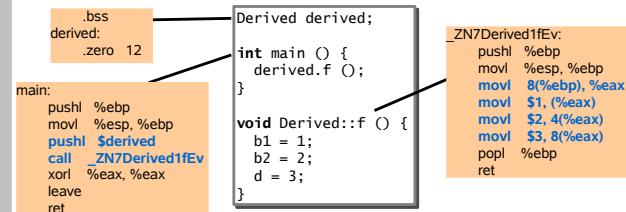
- die Attribute der Basisklasse liegen im Speicher am Anfang des Objekts
- keine Zeigeranpassung bei Typumwandlung von `Base*` in `Derived*` oder `Derived*` in `Base*` nötig



© 2006 Olaf Spinczyk

5

## (Einfach-)Vererbung – Ressourcen (2)



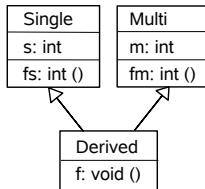
- Methoden erhalten den `this`-Pointer als unsichtbaren ersten Parameter
- Zugriff auf eigene Attribute und Basisklassenattribute kosten gleich viel.
- Kein *Overhead* durch (Einfach-)Vererbung.



© 2006 Olaf Spinczyk

6

## Mehrfachvererbung



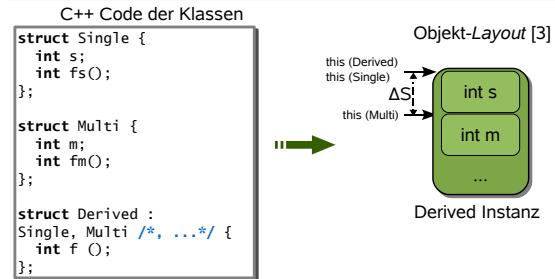
- eine abgeleitete Klasse erbt von **mehreren Basisklassen**
  - eine auf dem „Einfachvererbungspfad“
  - 1-N auf dem „Mehrfachvererbungspfad“
- Vererbungshierarchie ist keine Baumstruktur mehr
- mehrfaches Erben von der selben Klasse möglich!



© 2006 Olaf Spinczyk

7

## Mehrfachvererbung – Ressourcen (1)



- die Attribute der Basisklassen liegen nacheinander im Speicher am Anfang des Objekts
- bei der Typumwandlung von `Derived*` in einen Zeiger auf eine Klasse im Mehrfachvererbungspfad muss ein *Offset* addiert werden



© 2006 Olaf Spinczyk

8

## Mehrfachvererbung – Ressourcen (2)

```
void Derived::f () {
    fs();
    fm();
}
```

```
ZN7Derived1fEv:
pushl %ebp
movl %esp, %ebp
pushl %ebx
movl 8(%ebp), %ebx
pushl %ebx
call _ZN6Single2fsEv
addl $4, %ebx
pushl %ebx
call _ZN5Multi2fmEv
popl %eax
movl -4(%ebp), %ebx
popl %edx
leave
ret
```

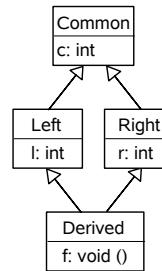
- Beim Aufruf einer Methode der Klasse im Einfachvererbungspfad kann der *this*-Pointer einfach durchgereicht werden
- Beim Zugriff auf *Multi* muss *this* angepasst werden (+ 4)
- Bei *inline*-Methoden tritt das Problem nicht auf
- Geringer *Overhead* bei Mehrfachvererbung

© 2006 Olaf Spinczyk

9

## Virtuelle Vererbung

- durch virtuelle Vererbung wird vermieden, dass eine mehrfach geerbte Basis mehr als einmal instanziert wird.
- Speicherplatz im Objekt wird eingespart
- Mehrdeutigkeiten bei der Namensauflösung werden vermieden
- Wo werden die Instanzen der virtuellen Basisklasse *Common* abgelegt?

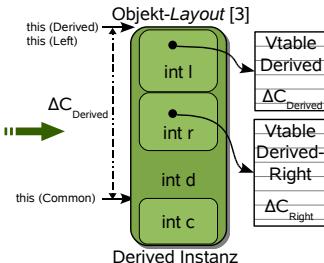


© 2006 Olaf Spinczyk

10

## Virtuelle Vererbung – Ressourcen (1)

```
struct Common { int c; };
struct Left : virtual Common {
    int l;
};
struct Right : virtual Common {
    int r;
};
struct Derived : Left, Right {
    int d;
    void f () {};
};
```



- die Attribute virtueller Basisklassen liegen am Ende
- der Objekttyp-spezifische *Offset* macht die Typkonvertierung kompliziert
- (mindestens) eine virtuelle Funktionstabelle wird benötigt!

© 2006 Olaf Spinczyk

11

## Virtuelle Vererbung – Ressourcen (2)

```
ZN7Derived1fEv:
pushl %ebp
movl %esp, %ebp
movl 8(%ebp), %eax
movl (%eax), %edx
movl -12(%edx), %edx
movl $1, (%edx,%eax)
movl $2, 4(%eax)
movl $3, 12(%eax)
movl $4, 16(%eax)
popl %ebp
ret
```

- Der Zugriff auf ein Attribut einer virtuellen Basisklasse ist erheblich komplizierter
- Dazu kommen noch (in diesem Beispiel):
  - 90 Byte für Tabellen
  - Konstruktor-Code zum Initialisieren der Vtable Zeiger
- Deutlicher *Overhead* bei virtueller Vererbung!
- insbesondere, wenn die beteiligten Klassen sonst keine Vtable benötigen würden

© 2006 Olaf Spinczyk

12

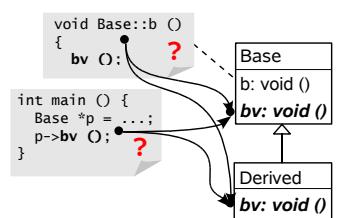
## Dynamisches Binden

- dynamisches Binden erfolgt bei **virtuellen Funktionen**
  - C++ Schlüsselwort **virtual**
- die Zielfunktion eines Aufrufs wird dabei zur Laufzeit ermittelt
  - wäre `bv()` nicht virtuell, würde in den Beispielen immer `Base::bv()` ausgeführt werden
  - ob `Base::bv()` oder `Derived::bv()` ausgeführt wird, hängt vom Objekttyp (nicht vom Zeigertyp) ab
  - da `Base::b()` sowohl auf `Base` als auch `Derived` Objekten ausgeführt werden kann, muss der Objekttyp ermittelt werden
  - da nicht immer zur Übersetzungszeit bestimmt werden kann, worauf `p` in `main()` zeigt, muss auch hier der Typ ermittelt werden

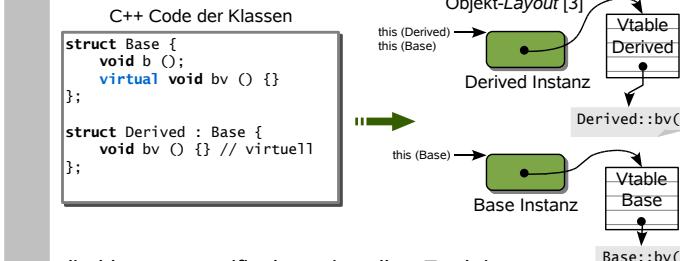


© 2006 Olaf Spinczyk

13



## Dynamisches Binden – Ressourcen (1)



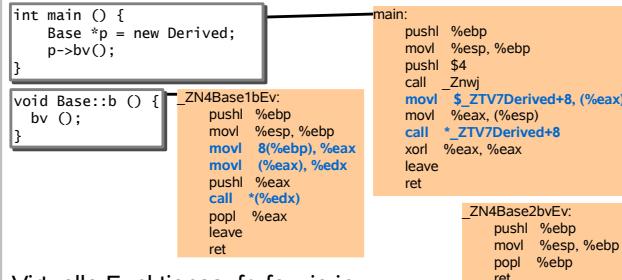
- die klassenspezifischen virtuellen Funktions-tabellen enthalten Zeiger auf den passenden Code
- der Konstruktor muss den Vtable Zeiger eintragen!
  - ggf. sogar mehrfach überschreiben!



© 2006 Olaf Spinczyk

14

## Dynamisches Binden – Ressourcen (2)



- Virtuelle Funktionsaufrufe wie in `Base::b()` und `main()` bedingen eine Indirektion
  - kein *Inlining* solcher Aufrufe möglich!
  - selbst leere virtuelle Funktionen müssen angelegt werden
- Dynamisches Binden kostet deutlich mehr als statisches!



© 2006 Olaf Spinczyk

15

## OO Techniken - Zusammenfassung

- Einfachvererbung
  - praktisch keine zusätzlichen Kosten
- Mehrfachvererbung
  - Zeigerkonvertierung Mehrfachvererbungspfad, geringer Aufwand
- Virtuelle Vererbung
  - Vtable Speicher, Objektinitialisierung, indirekter Zugriff, **Aufwand!**
- Dynamisches Binden
  - Vtable Speicher, Objektinitialisierung, indirekter Aufruf, **Aufwand!**
- Faustregel: In ressourcenbeschränkten Domänen das `virtual` Schlüsselwort nur verwenden, wenn es wirklich nötig ist



© 2006 Olaf Spinczyk

16

## Was macht Aspektorientierung aus?

„Aspect-Oriented Programming is Quantification and Obliviousness“

R.E. Filman and D.P. Friedman [4]

- Quantification
  - ein Aspekt wirkt auf viele Komponenten
- Obliviousness
  - die Komponenten müssen für die Einwirkung durch Aspekte nicht vorbereitet werden (umstritten!)
- Mit welchen Mitteln wirken Aspekte auf Komponenten?
  - Einfügungen (*Introductions*)
  - Advice für Laufzeitereignisse, z.B. Funktionsaufrufe, Konstruktion, ...
- Was kosten Aspekte im Hinblick auf Ressourcen?



© 2006 Olaf Spinczyk

17

## Einfügungen

AspectC++ Code

```
class C {  
};  
  
aspect A {  
    advice "C" : slice class {  
        int c;  
        public:  
            int get_c () {  
                return c;  
            }  
    };  
};
```

C++ Code (ag++ --keep\_acc --no\_line)

```
class C {  
    friend class ::A;  
  
private:  
    int c;  
private:  
  
public:  
    int get_c () {  
        return c;  
    }  
};
```

- in AspectC++ beschreibt ein *Pointcut*-Ausdruck die Menge der Zielklassen einer Einfügung
  - Achtung: sehr leicht sind viele Klassen und Objekte betroffen
- der Ressourcenverbrauch entspricht dem der „Handimplementierung“



© 2006 Olaf Spinczyk

18

## Advice für Laufzeitereignisse

AspectC++ Code

```
#include <stdio.h>  
  
aspect A {  
    advice call("% f()") : before() {  
        puts ("calling f()");  
    }  
};  
void f ();  
void g () { f (); }
```

```
Z1gv:  
    pushl %ebp  
    movl %esp, %ebp  
    subl $20, %esp  
    pushl $LC0  
    call puts  
    addl $16, %esp  
    leave  
    jmp _Z1fv
```

- einfacher Advice-Code führt bei ac++ zu *Inlining*
  - Voraussetzung: Aktivierung der Optimierung beim C++ Compiler
- Advice-Code wird i.d.R. wie eine Inline-Funktion übersetzt
  - Risiko der Code-Duplikation
- der Ressourcenverbrauch ist normalerweise dicht an dem der Handimplementierung



© 2006 Olaf Spinczyk

19

## AO Techniken - Zusammenfassung

### Einfügungen

- kein *Overhead*, entspricht der Handimplementierung

### Advice für Laufzeitereignisse

- durch *Inlining* i.d.R. kein *Overhead*
- nur wenige *Pointcut*-Funktionen erfordern Laufzeitüberprüfungen: *that*, *target*, *cflow*

### Vorsicht bei „Match-Ausdrücken“

- ist geboten, um zu viele Einfügungen und zu viel Advice zu vermeiden

### Advice-Code sollte bei mehr als einem „Match“ nicht zu groß sein, um Code-Duplikation zu vermeiden

- Verlagern des Codes in *non-inline Member* Funktionen des Aspekts



© 2006 Olaf Spinczyk

20

## Bewertung

Ziel: Entwurf mit geringstem Ressourcenverbrauch finden

## Dimensionen:

- Art der Variabilität  
(hier Beschränkung auf Variabilität entsprechend der Merkmaldiagramme)
    - optional, [optional] alternativ, oder
  - Bindungszeitpunkt  
(prinzipiell domänen spezifisch, hier Beschränkung auf ...)
    - Übersetzung/Konfigurierung des Systems (*compile time*)
    - beliebig zur Laufzeit (*runtime*)

### 3 x 2 Vergleiche von OO und AO Entwurf

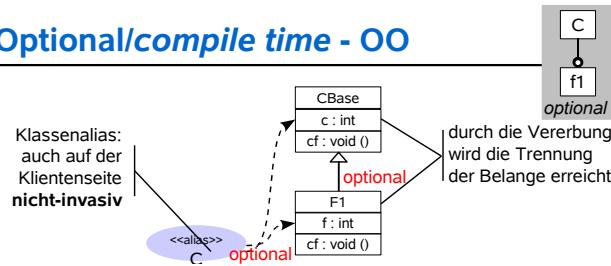
- Bewertung: (Kosten/Sonstige Eigenschaften) je +, 0, -



© 2006 Olaf Spinczyk

21

## Optional/compile time - OO



- keine besonderen Kosten durch die Einfachvererbung
  - Klassenalias für klientenseitige Transparenz nötig
  - wenn Aufrufe von cf() in CBase auch F1::cf() erreichen sollen, muss cf() virtuell sein!

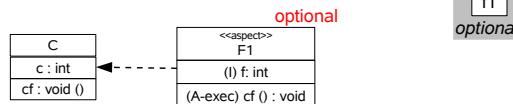


© 2006 Olaf Spinczyk

21

27

## Optional/compile time - AO



- Aspekt statt optionaler abgeleiteter Klasse
  - kein Klassenalias nötig
  - kein Overhead
  - keine virtuellen Funktionen
  - beliebig viele solcher optionalen Erweiterungen können koexistieren (siehe *oder*-Merkmale ...)

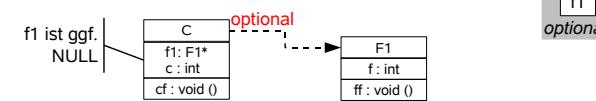


---

© 2006 CliffsNotes

( + / +

## Optional/run time - 00



- F1 und C sind getrennt, der Aufruf erfolgt aber explizit
    - ein quer schneidender Belang, Positionen hängen von F1 ab
    - wird bei vielen optionalen Merkmalen unübersichtlich
  - bei jedem Aufruf muss geprüft werden ob  $f1 \neq 0$  ist
    - Alternative: siehe optionale alternative Merkmale
  - der Klient muss die F1 Instanzen selbst verwalten
  - keine virtuellen Funktionen nötig



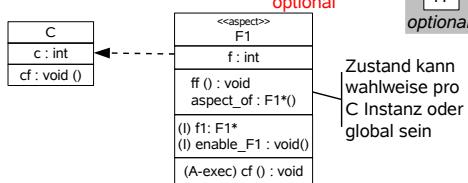
© 2006 Olaf Grunow

( + / - )

2

## Optional/run time - AO

C ist unverändert



- F1 und C sind getrennt, auch die Aktivierung von F1
  - nur möglich, wenn die Aktivierungspunkte in der Aspektsprache beschreibbar sind
- die Prüfung, ob  $f1 \neq 0$  ist, erfolgt im *Advice*
- der Klient muss keine F1 Instanzen verwalten
- Ressourcenbedarf entspricht der OO-Lösung

(+ / 0)

© 2006 Olaf Spinczyk

25

## Alternativ/compile time - OO

Klassenalias:  
auch auf der Klientenseite  
nicht-invasiv

optional altern.

- keine besonderen Kosten durch die Einfachvererbung
- Klassenalias für klientenseitige Transparenz nötig
- wenn Aufrufe von cf() in CBase auch F1::cf() erreichen sollen, muss cf() virtuell sein!

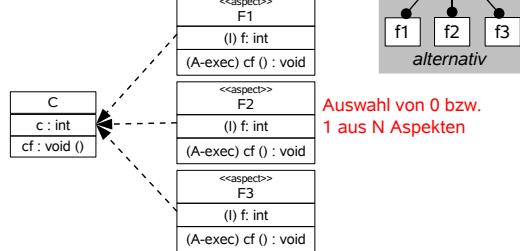
(0 / -)

© 2006 Olaf Spinczyk

26

## Alternativ/compile time - AO

Auswahl von 0 bzw.  
1 aus N Aspekten



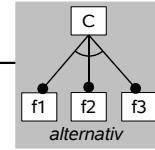
- Entspricht der Implementierung optionaler Merkmale
- die Einschränkung auf 0 bzw. 1 aus N Aspekten sollte das Konfigurationswerkzeug erledigen
- kein nennenswerter Ressourcenverbrauch

(+ / +)

© 2006 Olaf Spinczyk

27

## Alternativ/run time - OO



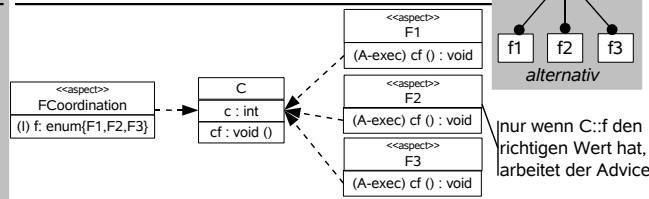
- durch die abstrakte Schnittstelle kann man zur Laufzeit die Bindung von einem C zu einem Fx Objekt umsetzen
  - Struktur entspricht dem *Strategy Design Pattern*
- die Aufrufe in C erfolgen explizit (nicht bedingt)
  - für den Zugriff auf den Zustand von C muss eine Referenz mitgegeben werden
- der Klient muss die Fx Objekte selbst verwalten
- jede Methode in FInterface ist virtuell!

(- / 0)

© 2006 Olaf Spinczyk

28

## Alternativ/run time - AO

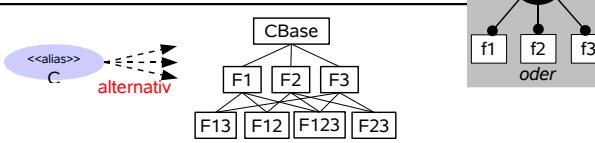


- für N Alternativen sind pro *Join Point* N Abfragen nötig
  - keine virtuelle Funktion, aber auch teuer!
- geeignet, wenn F1 bis FN keine gemeinsame Schnittstelle haben
- Alternativ kann mit einem Aspekt auch das *Strategy Pattern* gewebt werden

(- / 0)

29

## Oder/compile time – OO u. AO

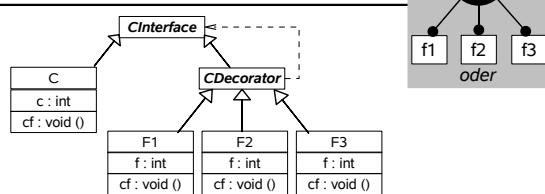


- die Vielzahl an Kombinationen führt schnell zu einer „Explosion“ der Klassenhierarchie
- ohne teure virtuelle Vererbung wären noch mehr Klassen oder *Code-Duplikation* nötig
- die AO Lösung besteht in der N-fachen und beliebig kombinierten Anwendung der Lösung für optionale Merkmale. Besser!

(+ / +)

30

## Oder/run time – OO u. AO



- durch die Anwendung des *Decorator Design Patterns* kann man flexible Objektketten erstellen
  - selbst zur Laufzeit änderbar
- alle von CInterface geerbten Methoden sind virtuell!
- mehr Flexibilität als nötig
  - kein Schutz vor dem mehrfachen Einhängen
- die AO Lösung besteht in der N-fachen Anwendung der Lösung für Optional/run time AO.

(0 / +)

31

## OO vs. AO Techniken - Vergleich

	optional	alternativ	oder
compile time	OO (0 / -)	OO (0 / -)	AO (- / -)
	AO (+ / +)	AO (+ / +)	AO (+ / +)
run time	OO (+ / -)	OO (- / 0)	AO (- / 0)
	AO (+ / 0)	AO (- / 0)	AO (0 / +)

- die AO Lösungen sind besonders bei der Konfigurierung zur Übersetzungszeit stark
- insgesamt gibt es bei den AO Lösungen viel weniger unterschiedliche Entwürfe
- im Zweifelsfall kann zur Entscheidung beitragen:
  - sind gemeinsame Schnittstellen vorhanden?
  - lassen sich die Aktivierungspunkte in der Aspektsprache beschreiben?

© 2006 Olaf Spinczyk

32

## Literatur

---

- [1] P. Wegner. *Classification in Object-Oriented Systems*,  
ACM, SIGPLAN Notices, 21(10):173-182, 1986.
- [2] B. Liskov. *Data Abstraction and Hierarchy*,  
ACM, SIGPLAN Notices, 23(5), 1988.
- [3] *C++ ABI Summary*,  
<http://www.codesourcecy.com/cxx-abi>
- [4] R.E. Filman and D.P. Friedman. *Aspect-Oriented Programming is Quantification and Obliviousness*,  
Workshop on Advanced Separation of Concerns,  
OOPSLA 2000, Minneapolis, Oct. 2000.

