

Kapitel 12: Übersetzung objektorientierter Konzepte

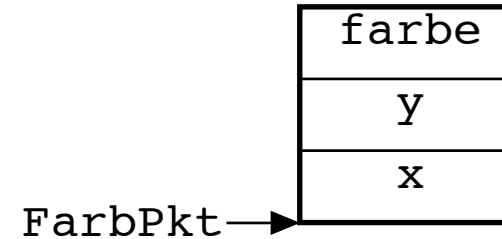
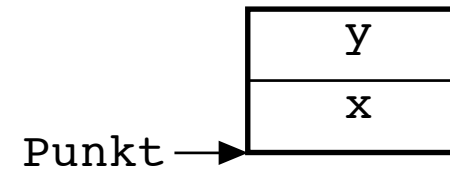
Themen

- **Klassendarstellung und Methodenaufruf**
- **Typüberprüfung**
- **Klassenhierarchieanalyse**
- **Escape Analyse**

Klassendarstellung bei Einfachvererbung

```
class Punkt {  
    int x, y;  
}
```

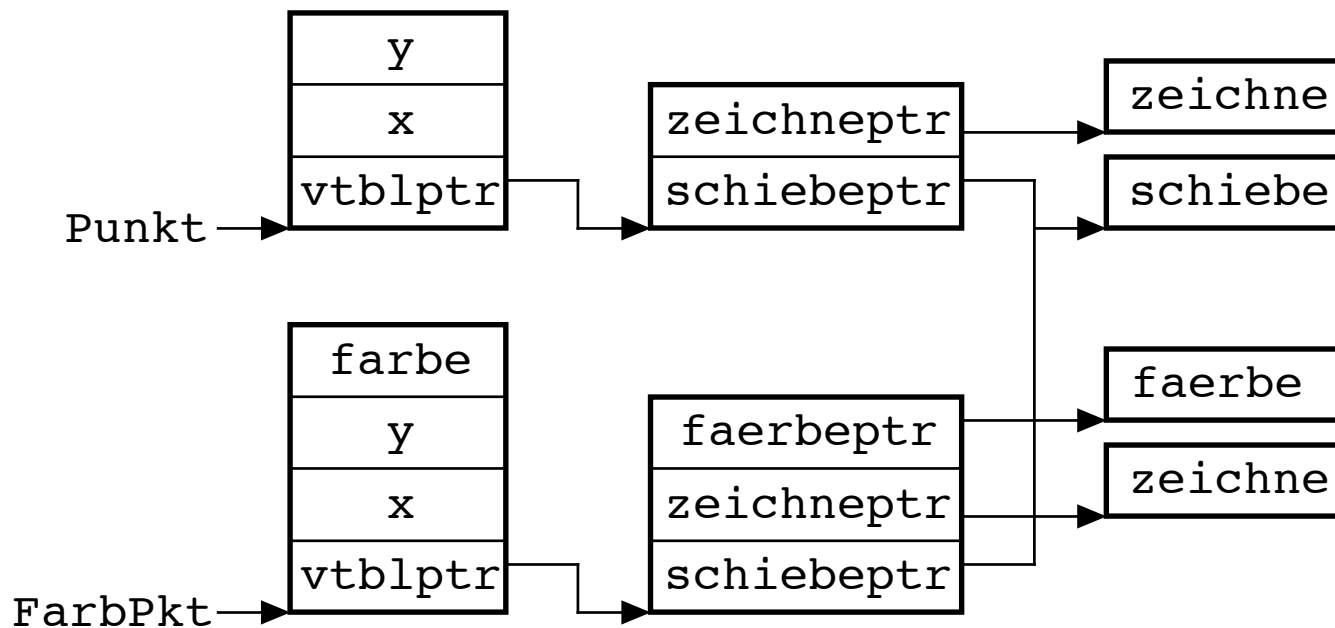
```
class FarbPkt extends Punkt {  
    int farbe;  
}
```



Einfachvererbung mit Methodentabelle

```
class Punkt {  
    int x, y;  
    void schiebe(int x, int y) {...}  
    void zeichne() {...}  
}
```

```
class FarbPkt extends Punkt {  
    int farbe;  
    void zeichne() {...}  
    void faerbe(int f) {...}  
}
```



Methodenaufruf bei einfacher Vererbung

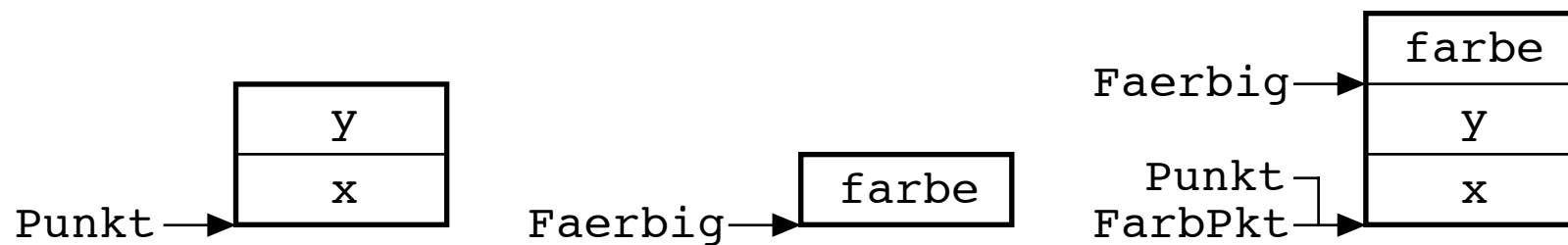
```
LDQ vtblptr, (obj)  
LDQ mptr, method(vtblptr)  
JSR (mptr)
```

Klassenlayout bei Mehrfachvererbung

```
class Punkt {  
    int x, y;  
}
```

```
class Faerbig {  
    int farbe;  
}
```

```
class FarbPkt extends Punkt, Faerbig{}
```



Zeigeranpassung

```
Faerbig f;  
FarbPkt fp;  
f = fp;      // f = fp; if (cp!=null) f = (Faerbig)((int*)fp+2)
```

Typumwandlungen außerhalb der Typhierarchie sind nicht möglich. Freispeicherverwaltung wird durch Zeiger in die Mitte von Objekten verkompliziert.

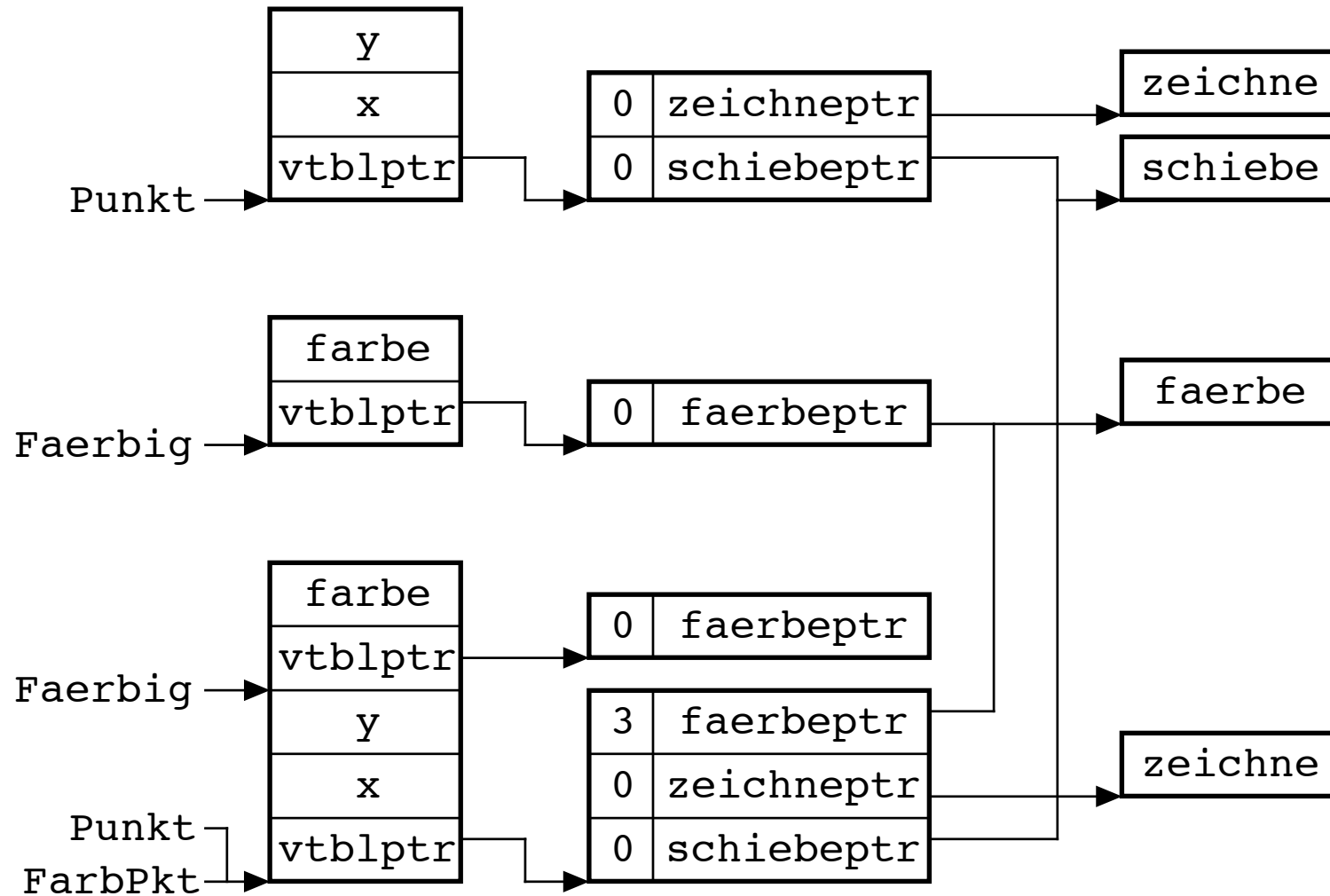
Mehrfachvererbung

```
class Punkt {  
    int x, y;  
    void schiebe(int x, y) {...}  
    void zeichne() {...}  
}
```

```
class Faerbig {  
    int farbe;  
    void faerbe(int f) {}  
}
```

```
class FarbPkt extends Punkt, Faerbig {  
    void zeichne() {...}  
}
```

Methodenaufwurf bei Mehrfachvererbung



Maschinecode für Methodenaufruf

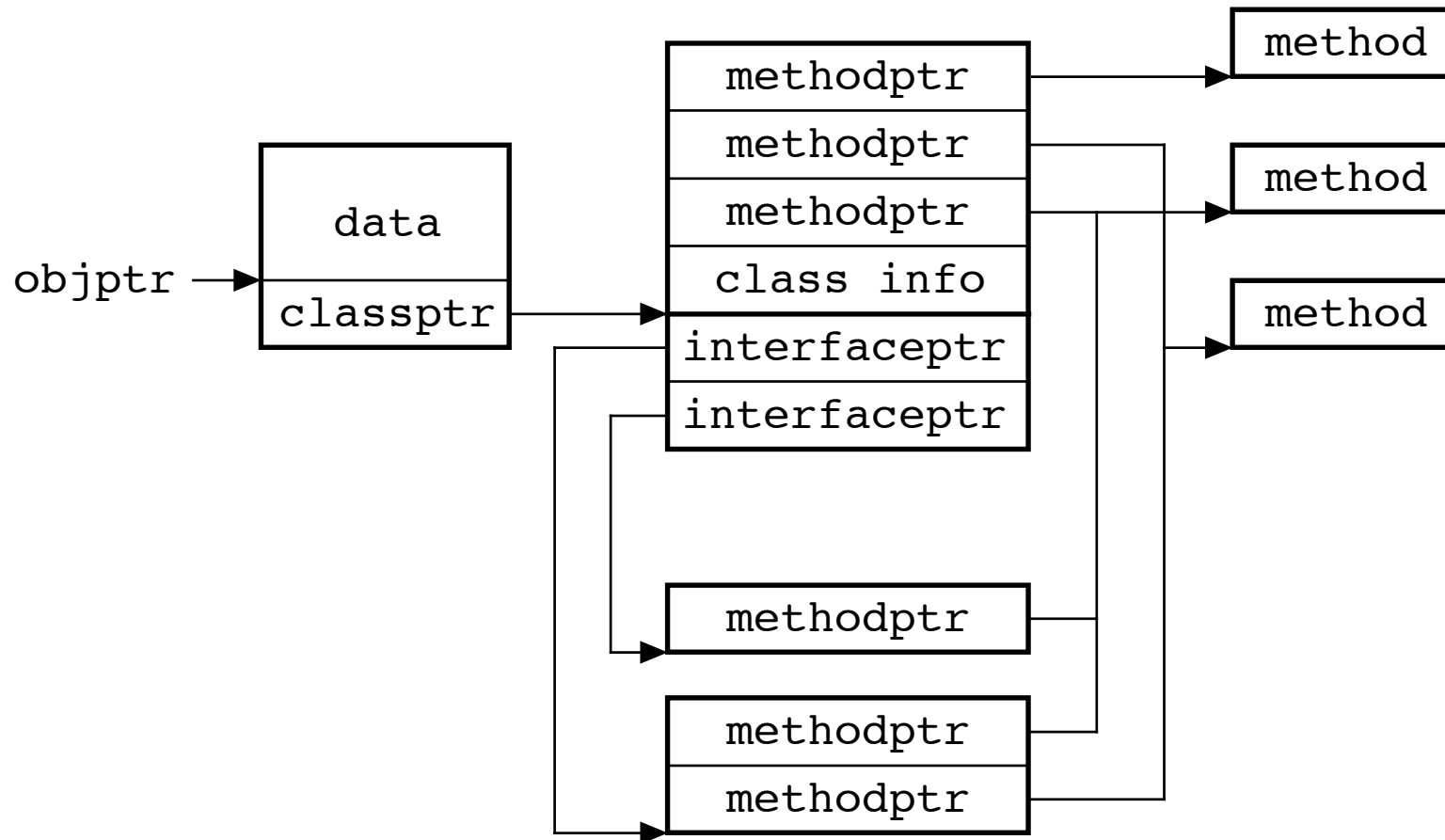
```
LD  vtblptr,(obj)           ; lade vtbl-Zeiger
LD  mptr,method_ptr(vtblptr) ; lade Methodenzeiger
LD  delta,method_off(vtblptr) ; lade Korrekturwert
ADD obj,delta,obj           ; korrigiere Objektzeiger
JSR (mptr)                  ; rufe Methode auf
```

Die Korrekturwerte sind auch dann nötig, wenn Mehrfachvererbung nicht verwendet wird. Trampolines speichern den Korrekturwert im Maschinencode:

```
ADD obj,3,obj               ; korrigiere Objektzeiger
BR  faerbe                  ; rufe Methode auf
```

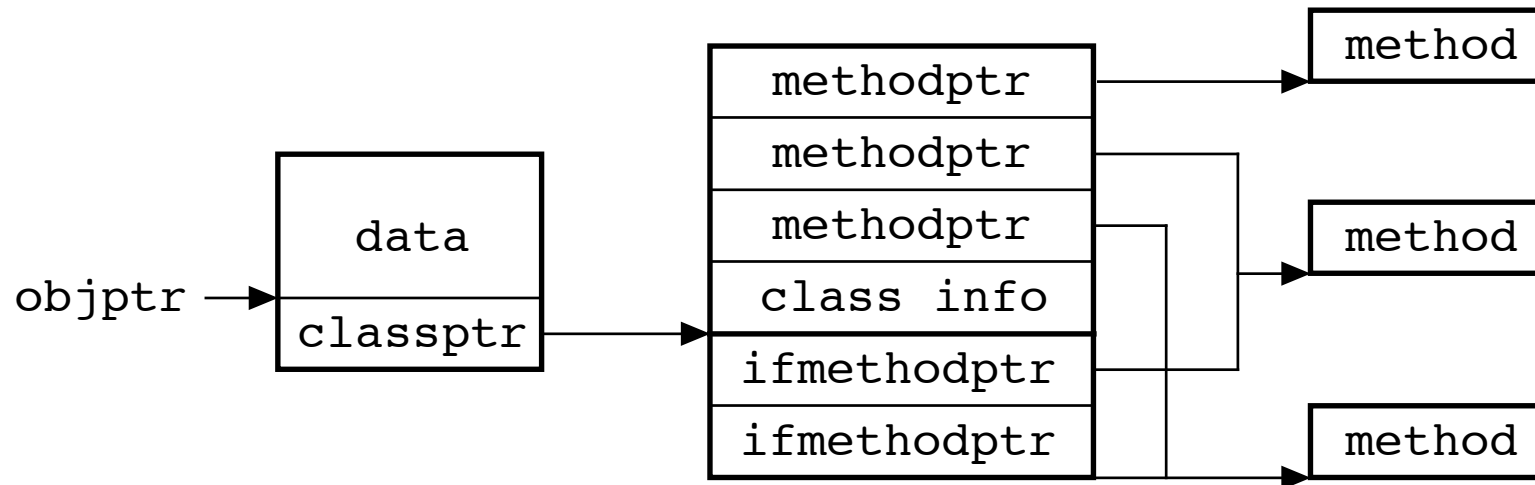
Wenn Instanzvariablen von gemeinsamen Oberklassen gemeinsam benutzt werden, muss der Offset dieser Variablen in der Methodentabelle gespeichert werden. Bei jedem Variablenzugriff muss der Offset zum Objektzeiger addiert werden.

Java Klassendarstellung (CACAO)



Der Aufruf von Interfacemethoden benötigt eine zusätzliche Indirektion.

Komprimierte Java Klassendarstellung



Die Interfacemethodentabelle kann komprimiert werden. Die JikesRVM löst Konflikte, die beim Nachladen von Klassen entstehen, über Hashing und einen Konfliktstub auf:

```
LD classptr, (obj) ; lade Klassenzeiger
LD mptr, hash(method) (classptr) ; lade Methodenzeiger
MV mindex, idreg ; lade Methodenindex
JSR (mptr) ; rufe Methode oder Konfliktstub
```

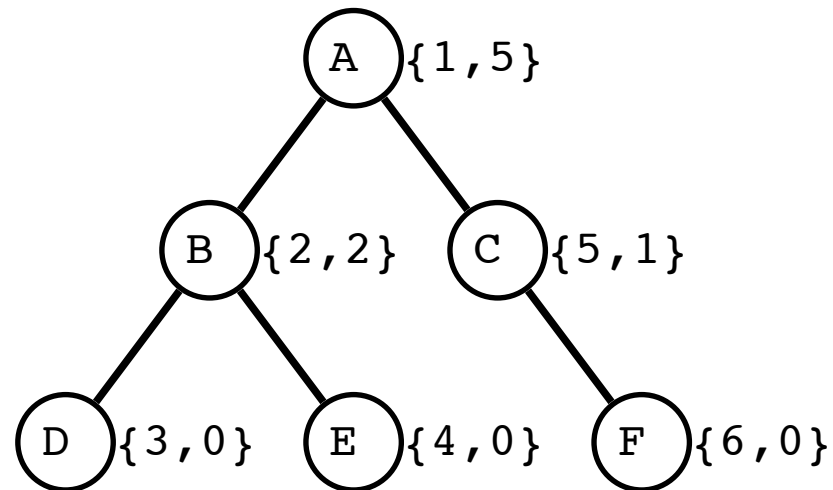
Direkter Methodenaufruf mit Typüberprüfung

In SmallEifel werden Methoden direkt aufgerufen und durch eine Typüberprüfung gesichert. Damit ist auch einfach Inlining möglich.

```
if idx <= TB
  if idx <= TA then fA(x)
  else fB(x)
else if idx <= TC then fC(x)
  else fD(x)
```

Werden Klassen zur Laufzeit nachgeladen, so muss der Maschinencode angepaßt werden.

Dynamische Typüberprüfung



```
return (unsigned) (sub->vftbl->baseval - super->vftbl->baseval) <=  
        (unsigned) (super->vftbl->diffval)
```

```
return (sub->vftbl->interfacetable[-super->index] != NULL);
```

Klassenhierarchieanalyse

Berechnet möglichst minimalen Aufrufgraphen und Klassenhierarchie

<i>main</i>	// main-Methode des Programms
<i>new c</i>	// Erzeugung eines Objects der Klasse <i>c</i>
<i>marked(c)</i>	// Menge der markierten Methoden der Klasse <i>c</i>
<i>x()</i>	// Aufruf der statischen Methode <i>x</i>
<i>type(e)</i>	// der deklarierte Typ des Ausdrucks <i>e</i>
<i>e.x()</i>	// Aufruf der virtuellen Methode <i>x</i> im Ausdruck <i>e</i>
<i>subtype(c)</i>	// <i>c</i> und alle Unterklassen der Klasse <i>c</i>
<i>method(x,c)</i>	// Methode <i>x</i> die in der Klasse <i>c</i> definiert ist
<i>mark(m,c)</i>	// markiert Methode <i>m</i> der Klasse <i>c</i>

Schnelle Typanalyse (rapid type analysis)

```
callgraph := main
hierarchy := {}
for each  $m \in \text{callgraph}$  do
    for each new  $c$  occurring in  $m$  do
        if  $c \notin \text{hierarchy}$  then
            add  $c$  to hierarchy
            for each  $m_{\text{mark}} \in \text{marked}(c)$  do
                add  $m_{\text{mark}}$  to callgraph
    for each  $m_{\text{stat}}()$  occurring in  $m$  do
        if  $m_{\text{stat}} \notin \text{callgraph}$  then
            add  $m_{\text{stat}}$  to callgraph
    for each  $e.m_{\text{vir}}()$  occurring in  $m$  do
        for each  $c \in \text{subtype}(\text{type}(e))$  do
             $m_{\text{def}} := \text{method}(c, m_{\text{vir}})$ 
            if  $m_{\text{def}} \notin \text{callgraph}$  then
                if  $c \notin \text{hierarchy}$  then
                     $\text{mark}(m_{\text{def}}, c)$ 
            else
                add  $m_{\text{def}}$  to callgraph
```

Escape Analyse

Die Escape Analyse stellt fest, ob Objekte aus einem bestimmten Bereich ausbrechen. Die Ergebnisse werden für weitere Optimierungen verwendet:

- Objekt innerhalb einer **Methode**: Objekt auf Stack anlegen
- Objekt innerhalb eines **Threads**: Entfernen von Synchronisation