Kapitel 12: Übersetzung objektorienter Konzepte

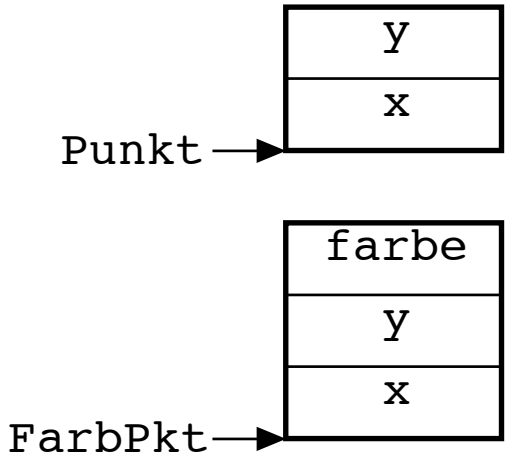
Themen

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

Klassendarstellung bei Einfachvererbung

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

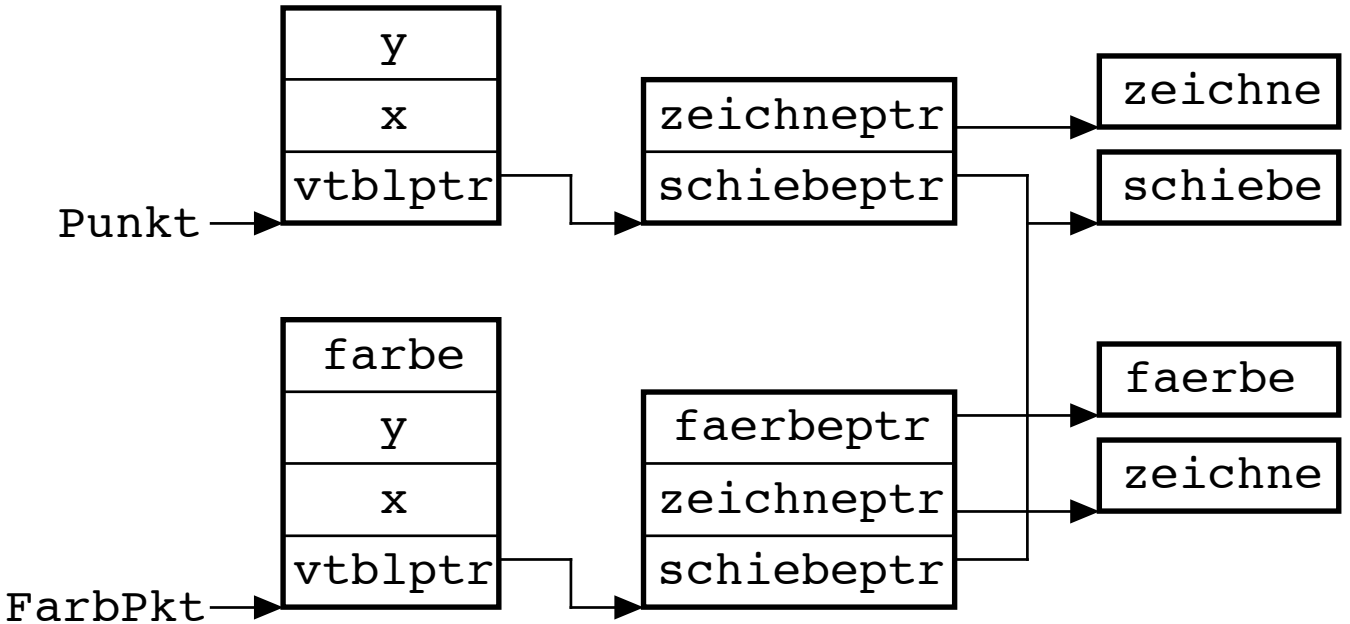
```
class FarbPkt extends Punkt {
  int farbe;
  }
```



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(inf f) {...}
  }
```



Methodenaufruf bei einfacher Vererbung

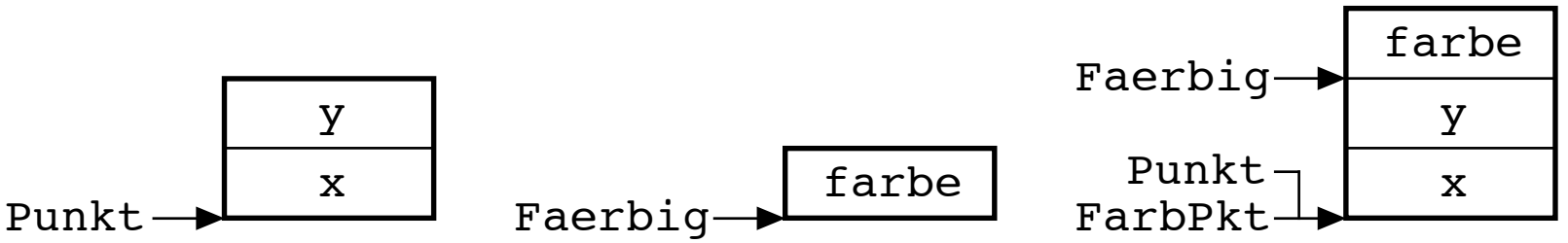
```
LDQ vtblptr,(obj)
LDQ mptr,method(vtblptr)
JSR (mptr)
```

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.

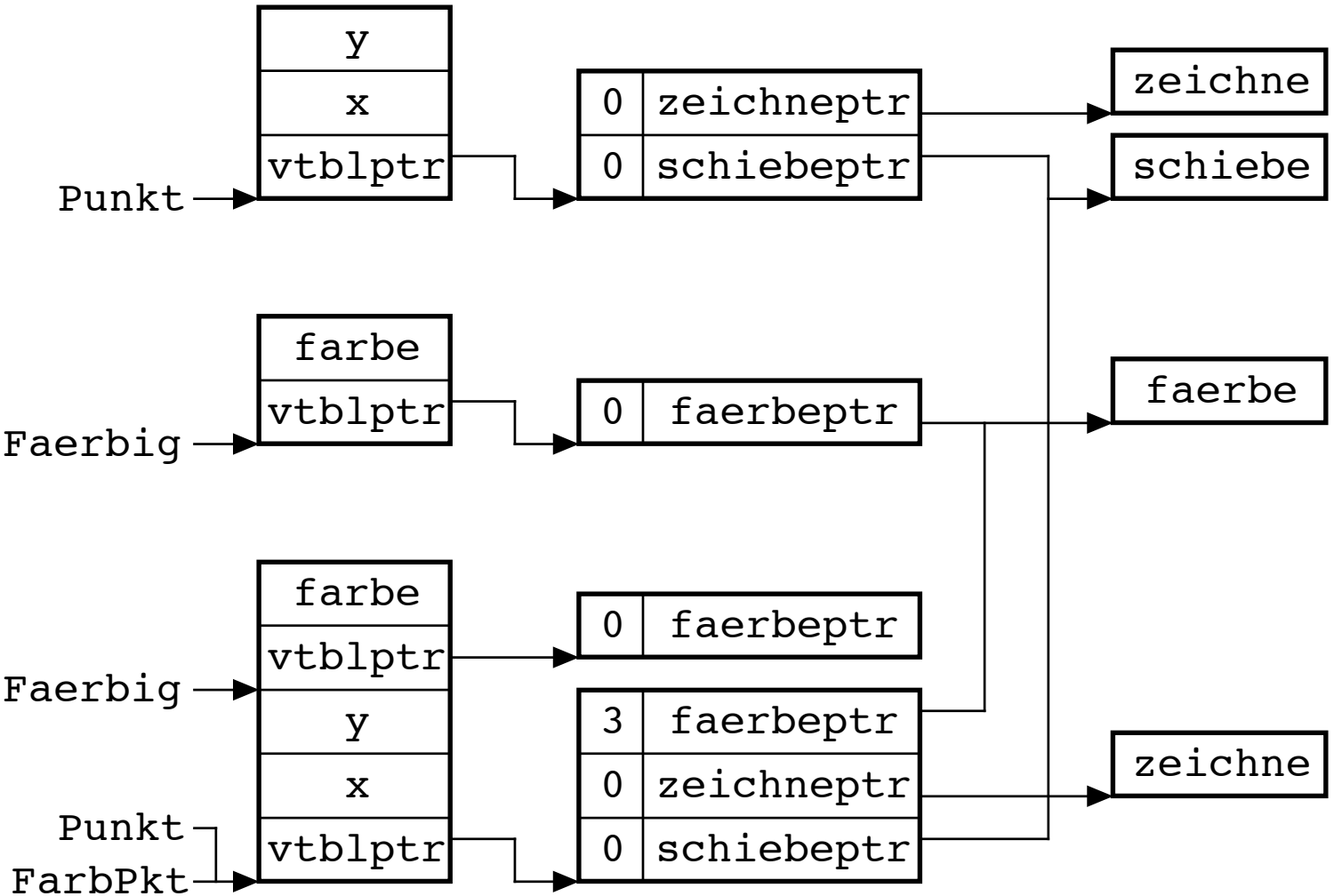
Mehrfachvererbung

```
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() {...}
  }
```

Methodenaufruf bei Mehrfachvererbung



Maschinecode für Methodenaufruf

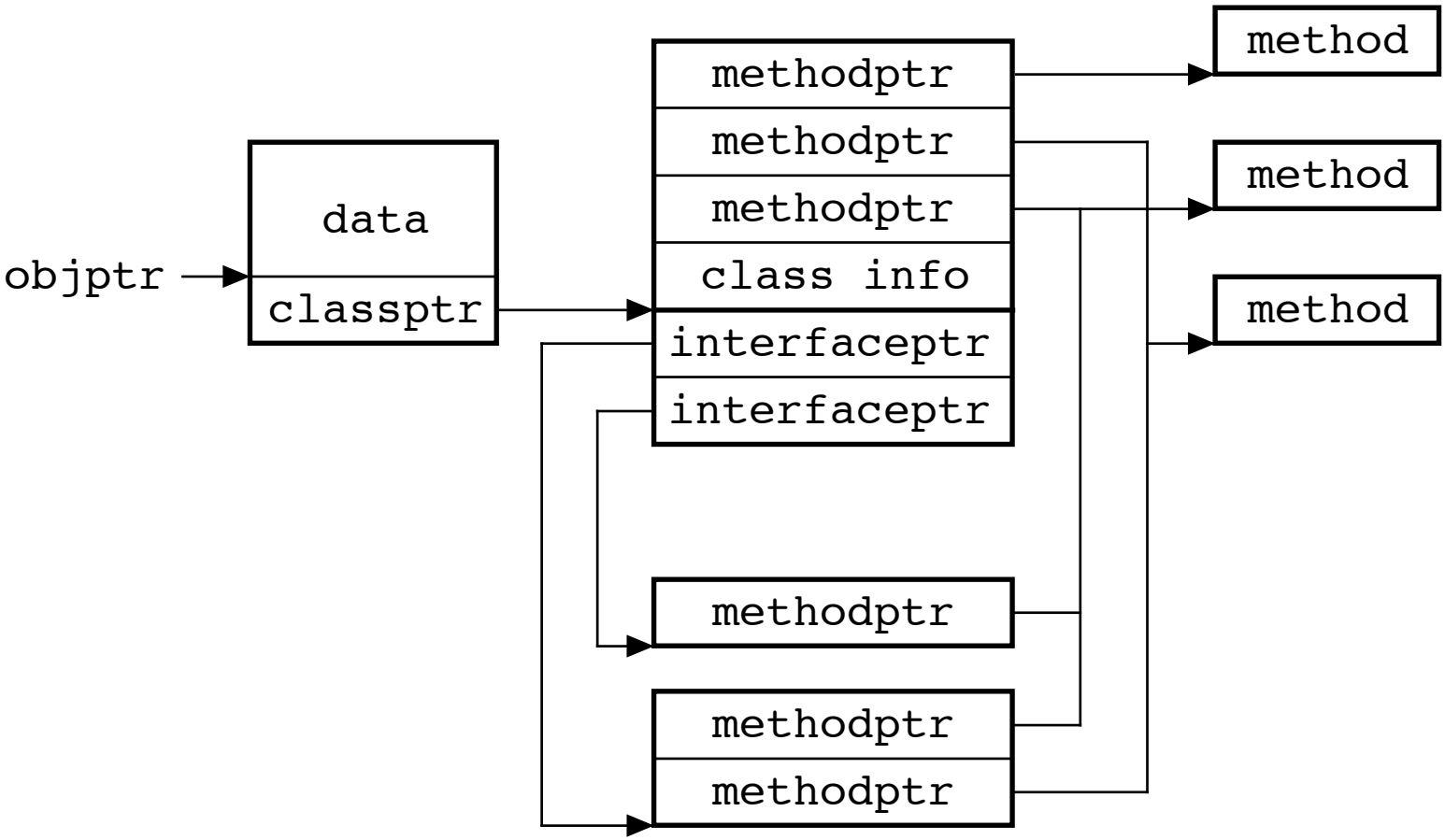
```
LD  vtblptr,(obj)                ; lade vtbl-Zeiger
LD  mptr,method_ptr(vtblptr)     ; lade Methodenzeiger
LD  delta,method_off(vtblptr)    ; lade Korrekturwert
ADD obj,delta,obj                ; korrigiere Objektzeiger
JSR (mptr)                       ; rufe Methode auf
```

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
```

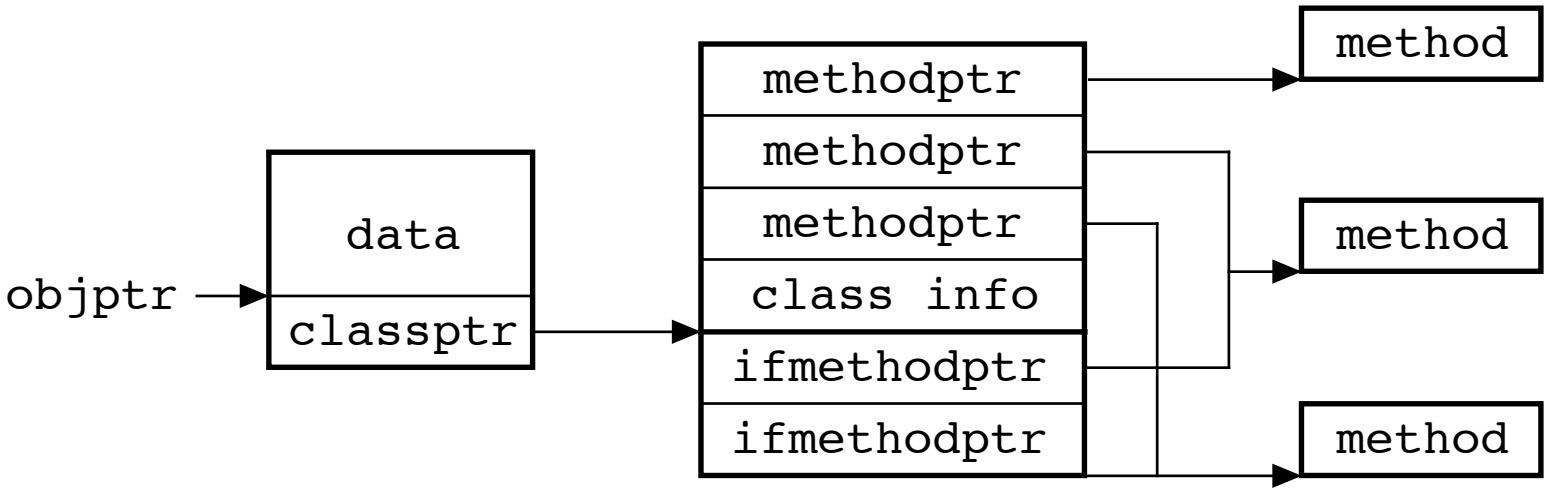
Wenn Instanzvariablen von gemeinsamen Oberklassen gemeinsam benutzt werden, muss der Offset dieser Variablen in der Methodentabelle gespeichert werden. Bei jedem Variablenzugriff muss der Offset zum Objektzeiger addiert werden.

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
```

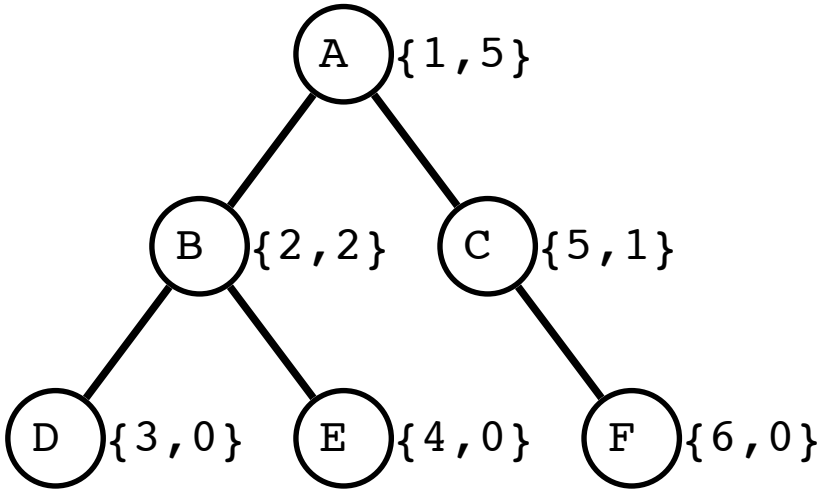
Direkter Methodenaufruf mit Typüberprüfung

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

```
if id_x <= T_B
  if id_x <= T_A then f_A(x)
  else f_B(x)
else if id_x <= T_C then f_C(x)
  else f_D(x)
```

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

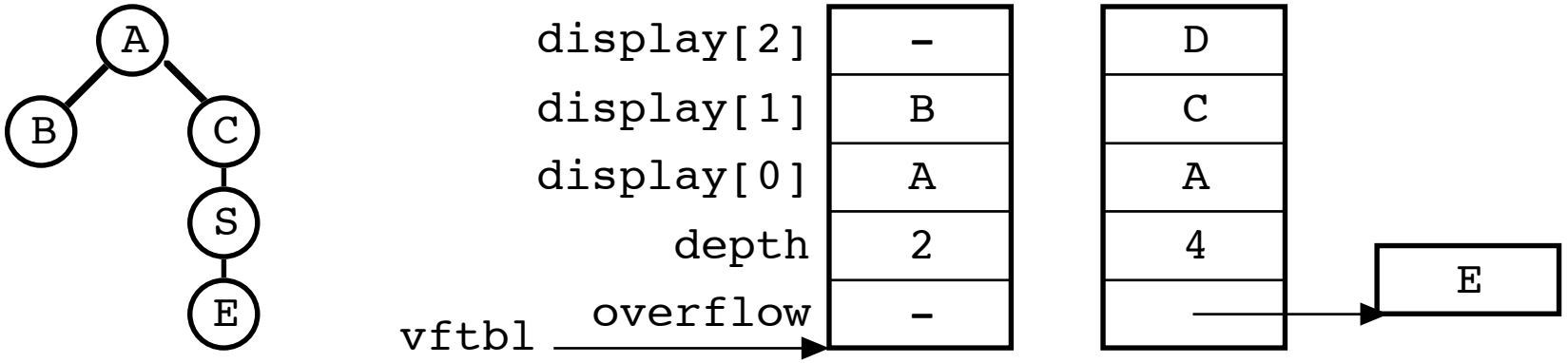
Dynamische Typüberprüfung Relative Numbering




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

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

Dynamische Typüberprüfung Display




```
return sub->vftbl->display[super->vftbl->depth] == super->vftbl;
```

```
return sub->vftbl->depth >= super->vftbl->depth &&
       sub->vftbl->overflow[super->vftbl->depth - DISPLAY_SIZE]
                           == super->vftbl;
```

Klassenhierarchieanalyse

Berechnet möglichst minimalen Aufrufgraphen und Klassenhierarchie

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

Schnelle Typanalyse (rapid type analysis)

```
callgraph := main
hierarchy := {}
for each m ∈ callgraph do
    for each new c occurring in m do
        if c ∉ hierarchy then
            add c to hierarchy
            for each m_mark ∈ marked(c) do
                add m_mark to callgraph
    for each m_stat() occurring in m do
        if m_stat ∉ callgraph then
            add m_stat to callgraph
    for each e.m_vir() occurring in m do
        for each c ∈ subtype(type(e)) do
            m_def := method(c,m_vir)
            if m_def ∉ callgraph then
                if c ∉ hierarchy then
                    mark(m_def,c)
            else
                add m_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