
Kovariante Probleme
Überladen
Multimethoden
Visitor
Annotationen
Aspekte

Einfache Typhierarchie

```
abstract class Food { ... }

class Grass extends Food { ... }

class Meat extends Food { ... }
```

Dynamische Typabfrage für kovariantes Problem

```
abstract class Animal {
    public abstract void eat(Food x);
    ...
}
class Cow extends Animal {
    public void eat(Food x) {
        if (x instanceof Grass) { ... } else fallIll();
    }
}
class Tiger extends Animal {
    public void eat(Food x) {
        if (x instanceof Meat) { ... } else showTeeth();
    }
}
```

Überladene Methode

```
class Grass extends Food { ... }

abstract class Animal {
    public abstract void eat(Food x);
}

class Cow extends Animal {
    public void eat(Grass x) { ... }
    public void eat(Food x) {
        if (x instanceof Grass) eat((Grass)x);
        else fallIll();
    }
}
```

Überladen = statisches Binden

nur deklarierte Typen für Überladen entscheidend:

```
Cow  cow   = new Cow();
Food grass = new Grass();
cow.eat(grass);          // eat(Food x) in Cow
cow.eat((Grass)grass);   // eat(Grass x) in Cow
```

Achtung: deklarierten Typ von `cow` betrachten:

```
Animal cow   = new Cow();
Food   grass = new Grass();
cow.eat(grass);          // eat(Food x) in Cow
cow.eat((Grass)grass);   // eat(Food x) in Cow !!
```

Empfehlungen für Überladen

Überladen generell fehleranfällig ⇒ eher vermeiden

Überladen in verschiedenen Klassen schwer sichtbar

⇒ Überladen von Methoden aus Oberklasse stets vermeiden

Unterscheidung zw. statischem und dynamischem Binden soll nicht entscheidend sein

⇒ für je zwei überladene Methoden soll gelten:

Unterscheidung an Hand einer Parameterposition,
wobei Parametertypen keinen gemeinsamen Untertyp haben

oder alle Parametertypen einer Methode spezieller als die der anderen,
und allgemeinere Methode verzweigt nur auf speziellere, falls möglich

Multimethoden = dynamisches Binden

Multimethoden entsprechen syntaktisch überladenen Methoden,
aber Bindung erfolgt anhand dynamischer Typen der übergebenen Argumente

dadurch oft einfachere Programme möglich:

```
class Cow extends Animal {
    public void eat(Grass x) { ... }
    public void eat(Food x) {
        fallIll();
    }
}
```

Achtung: NICHT in Java !!!
(in Java nur Überladen mit statischem Binden)

Simulation von Multimethoden (Element-Klassen)

```
abstract class Animal {
    public abstract void eat(Food food);
    ...
}
class Cow extends Animal {
    public void eat(Food food) {
        food.eatenByCow(this);
    }
}
class Tiger extends Animal {
    public void eat(Food food) {
        food.eatenByTiger(this);
    }
}
```

Simulation von Multimethoden (Visitor-Klassen)

```
abstract class Food {
    public abstract void eatenByCow(Cow cow);
    public abstract void eatenByTiger(Tiger tiger);
}
class Grass extends Food {
    public void eatenByCow(Cow cow) { ... }
    public void eatenByTiger(Tiger tiger)
        { tiger.showTeeth(); }
}
class Meat extends Food {
    public void eatenByCow (Cow cow)
        { cow.fallIll(); }
    public void eatenByTiger(Tiger tiger) { ... }
}
```

Komplexität

Nachteil simulierter Multimethoden: Anzahl der Methoden

M Tierarten, N Futterarten $\Rightarrow M \cdot N$ inhaltliche Methoden

Generell für n Bindungen: $N_1, N_2, \ldots, N_n$ Möglichkeiten

$\Rightarrow$ $N_1 \cdot N_2 \cdots N_n$ inhaltliche Methoden
insgesamt $N_1 + N_1 \cdot N_2 + \cdots + N_1 \cdot N_2 \cdots N_n$ Methoden

echte Multimethoden verwenden daher Komprimierungstechniken und Vererbung

Eindeutigkeit bei Vererbung muss garantiert werden:

```
void eatTwice(Food x, Grass y) {...}
void eatTwice(Grass x, Food y) {...}
void eatTwice(Grass x, Grass y) {...}   // notwendig
```

Annotationen

Annotation ist optionaler Parameter, der

- an (fast) beliebige Sprachkonzepte anheftbar ist,
- im Java-Code statisch gesetzt wird,
- von gesamter Werkzeugkette bis zur Laufzeit auslesbar ist.

```
@Override
public String toString() { ... }

@BugFix(who="Kaspar", date="2023-11-20", level=3,
        bug="class unnecessary and maybe harmful",
        fix="contents of class body removed")
public class Buggy { }
```

Definition von Annotationen

Syntax von Interfaces adaptiert

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE})
public @interface BugFix {
    String who() default "me";  // author of bug fix
    String date();              // when was bug fixed
    int level() default 1;      // importance level 1-5
    String bug();               // description of bug
    String fix();               // description of fix
}
```

Einschränkungen in @interface

nur parameterlose Methoden

als Ergebnistypen nur erlaubt:

- alle elementaren Typen (`int`, `double`, ...)
- Aufzählungstypen (`enum`)
- `String`
- `Class`
- andere Annotationen
- sowie eindimensionale Arrays dieser Typen (als „Mengen“)

Methodenname `value` → Name in Annotation optional

Annotationen für Annotationsdefinition

```
@Retention(RUNTIME)              // @Retention(value=RUNTIME)
@Target(value=ANNOTATION_TYPE)
public @interface Retention {
    RetentionPolicy value();
}
public enum RetentionPolicy { CLASS, RUNTIME, SOURCE }

@Retention(value=RUNTIME)
@Target(value=ANNOTATION_TYPE)
public @interface Target {
    ElementType[] value();
}
public enum ElementType { ANNOTATION_TYPE, CONSTRUCTOR,
    FIELD, LOCAL_VARIABLE, METHOD, PACKAGE, PARAMETER, TYPE
}
```

Annotationen zur Laufzeit

Falls `@Retention(RUNTIME)` wird echtes Interface erzeugt:

```
public interface BugFix extends java.lang.annotation.Annotation {
    String who();
    String date();
    int level();
    String bug();
    String fix();
}
```

Zugriff zur Laufzeit (Reflexion)

Annotationen über Reflection zur Laufzeit zugreifbar
(falls `@Retention(RUNTIME)`):

```
String s = "";
BugFix a = Buggy.class.getAnnotation(BugFix.class);
if (a != null) {  // null if no such Annotation
    s += a.who() + " fixed a level " + a.level() + " bug";
}

Annotation[] as = Buggy.class.getAnnotations();  // alle
Method[] ms = Buggy.class.getMethods();
// verschiedene analoge Methoden auch auf Method, Field
```

Aspektorientierte Programmierung – Konzeption

Ergebnisse von Berechnungen hängen ab von

- dem Programm,
- der Semantik der Programmiersprache,
- den Daten

Aspektorientierte Programmierung ändert Sprachsemantik auf kontrollierte Weise

herkömmlich: Metaprogrammierung, Precompilation – gefährlicher Graubereich

Ziel: Semantik gezielt an bestimmten Programmstellen ändern

Knackpunkt: relevante Programmstellen identifizieren (Parametrisierung)

Separation of Concerns

= im Wesentlichen Klassenzusammenhalt, aber nicht nur auf eine Klasse bezogen

Unterscheidung:

- Kernfunktionalität (core concern) — häufig
  - Faktorisierung erlaubt klare Zuordnung zu Klassen,
    hoher Klassenzusammenhalt und schwache Objektkopplung erreichbar
- Querschnittsfunktionalität (cross cutting concern) — selten
  - Eingriffe an vielen Stellen nötig (z. B. Methodenaufrufe über ganzes Programm verteilt),
    keine Faktorisierung in unabhängige Klassen möglich,
    objektorientierte Programmierung kann damit nur schwer umgehen

Aspektorientierte Programmierung zielt nur auf Querschnittsfunktionalität ab

- z. B.: Zugriffsrechte überprüfen, Debug-Information generieren, Log-Dateien schreiben

Elemente von AspectJ

Join-Point
: zur Laufzeit identifizierbare Stelle in einem Programm

Pointcut
: syntaktisches Element zur Kennzeichnung von Join-Points im Programmtext

Advice
: syntaktisches Element spezifiziert Programmcode, der an Join-Points auszuführen ist; davor (`before()`), danach (`after()`) oder stattdessen (`around()`)

Aspect
: syntaktisches Element, das alle Teile kombiniert (vergleichbar mit Klasse)

Join-Points (Beispiele)

Methodenausführung
Konstruktoraufruf
Methodenaufruf

Klasseninit

Objektinit
Feldzugriff(write)

Feldzugriff(read)

```
public class Test{
   public static void main(String[] args) {
      Point pt1 = new Point(0,0);
      pt1.incrXY(3,6);
   }
}
public class Point {
   private int x, y;
   public Point(int x, int y) {
      this.x = x;
      this.y = y;
   }
   public void incrXY(int dx, int dy) {
      x = this.x + dx;
      y =+ dy;
   }
}
```

Pointcut

```
public pointcut AddListener(EventListener el) :
   call(* javax..*.add*Listener(EventListener)) && args(el);
```

Pointcut-Typen (Beispiele):

- `execution(`*Methodensignatur*`)`
- `call(`*Methodensignatur*`)`
- `get(`*Feldsignatur*`)`
- `set(`*Feldsignatur*`)`
- `this(`*TypOderIdentifier*`)`
- `target(`*TypOderIdentifier*`)`
- `args(`*TypOderIdentifier*, ...`)`
- `cflow(`*Pointcut*`)`

Advice (Beispiele)

```
before() : call(* Account.*(..)) { checkUser(); }


pointcut connectionOperation(Connection connection) :
   call(* Connection.*(..) throws SQLException)
   && target(connection);
before(Connection connection) :
   connectionOperation(connection) {
      System.out.println("Operation auf " + connection);
   }
```

Aspect (Beispiel)

```
public aspect JoinPointTraceAspect {
   private int callDepth = -1;
   pointcut tracePoints() : !within(JoinPointTraceAspect);
   before() : tracePoints() {
      callDepth++;
      print("Before", thisJoinPoint);
   }
   after() : tracePoints() {
      callDepth--;
      print("After", thisJoinPoint);
   }
   private void print(String prefix, Object message) {
      for (int i=0, spaces = callDepth*2; i<spaces; i++)
         System.out.print(" ");
      System.out.println(prefix + ": " + message);
   }
}
```