

# Tagesprogramm

Aspektororientierte Programmierung

Annotationen

# Aspektororientierte Programmierung

- Programmierparadigma
- AspectJ
- *separation of concerns*
- Modularisierung
- Aspekte kapseln Verhalten das mehrere Klassen betrifft

## *cross cutting concern*

- *core concerns* (Kernfunktionalitäten)
- *cross cutting concern* (Querschnittsfunktionalitäten)
- Beispiele *logging*, Authentisierung, *debugging*
- Aspekte eines Programms, die nicht Kernfunktionalität sind, die aber für das Programm notwendig sind

## Elemente von AspectJ

- *join point* Ausführungspunkt in einem Programm
- *pointcut* wählt Ausführungspunkt mit Kontext
- *advice* Programmcode der am *join point* ausgeführt werden soll
- *aspect* Kombination aus *join point*, *pointcut* und *advice*

## *join point*

```
01          public class Test{
02 Methodenausführung public static void main(String[] args) {
03 Konstruktoraufruf     Point pt1 = new Point(0,0);
04 Methodenaufruf       pt1.incrXY(3,6);
05                       }
06                       }
07 Klasseninit          public class Point {
08 Objektinit           private int x;
09                       private int y;
00                       public Point(int x, int y) {
11 Feldzugriff(write)   this.x = x;
12                       this.y = y;
13                       }
14                       public void incrXY(int dx, int dy){
15 Feldzugriff(read)    x = this.x + dx;
16                       y += dy;
17                       }
18                       }
```

## *pointcut*

```
public pointcut accountOperation() : call(* Account.*(..))
```

```
    Schlüsselwort      PCName          PCTyp      Signatur
```

```
execution(Signature)
```

```
call(Signature)
```

```
get(Signature)
```

```
set(Signature)
```

```
handler(Signature)
```

```
initialization(Signature)
```

```
cflow(Pointcut)
```

```
within(Signature)
```

## *advice*

```
before() : Pointcut {Programmcode}
```

```
around()
```

```
after()
```

```
before() : call(* Account.*(..)) {Benutzer überprüfen}
```

```
pointcut connectionOperation(Connection connection) :
```

```
    call(* Connection.*(..) throws SQLException);
```

```
before(Connection connection) :
```

```
    connectionOperation(connection) {
```

```
        System.out.println("Operation auf" + connection);
```

```
    }
```

## *aspect*

```
01 public aspect JoinPointTraceAspect {
02     private int _callDepth = -1;
03     pointcut tracePoints() : !within(JoinPointTraceAspect);
04     before() : tracePoints() {
05         _callDepth++;
06         print("Before", thisJoinPoint);
07     }
08     after() : tracePoints() {
09         _callDepth--;
10         print("After", thisJoinPoint);
11     }
12     private void print(String prefix, Object message) {
13         for (int i=0, spaces = _callDepth * 2; i < spaces; i++) {
14             System.out.print(" ");
15         }
16         System.out.println(prefix + ": " + message);
17     }
18 }
```



## Literatur und Links

- *AspectJ in Action* von Ramnivas Laddad
- <http://eclipse.org/aspectj/>

# 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="2017-12-20", 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(); // all  
Method[] ms = Buggy.class.getMethods();  
// verschiedene analoge Methoden auch auf Method, Field
```