Aspektorientierte Programmierung

SS 2008

Tafelanschrieb zu
Kapiteln 3-6 des Buchs von
Oliver Böhm

Berlin, 2008-06-30

07-10-26/1

Böhm 3.6.3 Signature-API (aus org.aspectj.lang)

Erhältlich mit JoinPoint.getSignature().

Methoden:

toString(), toShortString(), toLongString():
Signatur (ohne z.B. execution(..))

Class getDeclaringType():
- die den Joinpoint umgebende Klasse
- Übergang zum Reflection-API

String getName():
- Bezeichner der Methode, des Konstruktors, des Attributs usw.

Beispiel:

```
final Signature signature = thisJoinPoint.getSignature();
final Class theClass = signature.getDeclaringType();
final String identifier = signature.getName();
final Package thePackage = theClass.getPackage();

System.out.println(
  "Method " + identifier + " in " + theClass + " in " + thePackage
);
```

gibt z.B. aus z.B.:

Method report in class BasicSimulation in package telecom

Böhm 3.6.4 SourceLocation-API

erhältlich mit JoinPoint.getSourceLocation().

Methoden:

toString():
z.B. Konto.java:27

getFileName(): Nur Quelldateiname ohne Verzeichnis
z.B.: Konto.java

getLine(): Zeilennr. im Quelltext

getWithinType(): Class
- die den Joinpoint umgebende Klasse (Reflection)

Frage:
Wie Java-Stack-Trace-Format zusammenstellen?
⟶ at␣package.Class.method(File.java:999)

[Class.getPackage(): Package, Package.getName(),
JoinPoint.StaticPart.getSignature().getName(),
SourceLocation.toString()]

[Hier für Übung 2 zunächst Kapitel 4.5 drannehmen!]

Böhm 07-10-2613

3.7 Zusammenfassung

Joinpoint? Punkt im Programmablauf

Pointcut? Menge von Joinpoints

Joinpoint-Arten?

- call
- execution
- get, set
- handler
- staticinitialization, preinitialization, initialization

Wildcards (Joker)

- *
- ..
- +

Logische Operatoren? ! || &&

Signatur?

„Bezeichner" für Konstruktor? new

Zugriff auf Kontexet? thisJoinPoint

23.4.08

Böhm 4. Pointcuts definieren

07-11-02/1

Wir kennen: Joinpoint-Art-bezogene Pointcuts (kinded Pointcut)
call, execution, set, get, handler
* initialization

Wir werden lernen
- filternde Pointcuts
- Pointcuts kombinieren.

4.1 Syntax einer Pointcut-Deklaration

[Access-Typ] pointcut Identifier (Parameter) : Pointcut Expr ;
(Parameter: list)

Access-Typ ::= public | protected | private

Pointcut Expr ::= primitive Pointcut
| ! Pointcut Expr
| (Pointcut Expr)
| Pointcut Expr && Pointcut Expr
| Pointcut Expr || Pointcut Expr

primitive Pointcut ::= kinded Pointcut | filter Pointcut

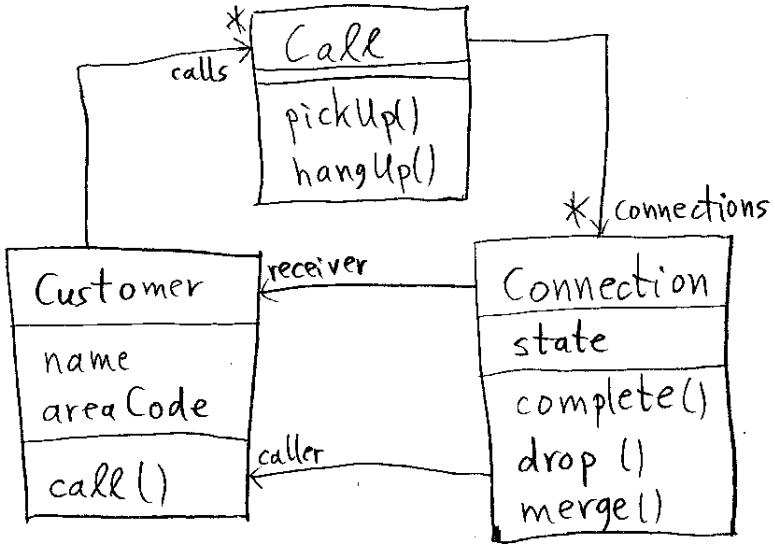
filter Pointcut ::=
adviceexecution ()
| within (TypePattern)
| withincode (MethodPattern | ConstructorPattern)
| this (Type | Id)
| target (Type | Id)
| args (Type | Id, ...)
| cflow (Pointcut)
| cflowbelow (Pointcut)
| if (Expression)

07-11-02

4.2 Anwendungsbeispiel Telecom

Einfache Simulation eines Telefonsystems.
Man kann anrufen, aufnehmen, ablegen, ...

→ AspectJ-Doku/progguide/examples-production.html
#a-simple-telecom-simulation



Bitte der den von mir ausgegebenen Quellcode verwenden!

4.3 Kinded Pointcuts

Schon behandelt.

4.4 Kontext-Pointcuts

Die Pointcuts this, target und args ergreifen Kontext eines Aufrufbaren (Methode | Konstruktor).

4.4.1 this-Pointcut

this (Type):

filtert alle Joinpoints mit aktuellem Objekt von Type: call, execution, set, get, handler, ... in Klasse Type.

Bsp.: execution(* *(..)) && this(telecom.Call)
ist äquivalent zu
execution(* telecom.Call.*(..))
ohne Klassenmethoden. [formulierbar als !static ?]

this (Id):

Mit Pointcut-Parameter Type Id
- filtert wie zuvor
- macht das aktuelle Objekt zugreifbar als Parameter Id.

Bsp.:

```
pointcut executionInCall(final telecom.Call call):
        execution(* *(..)) && this(call);
```

- filtert wie zuvor
- call ist typsicherer und effizienter Zugriff auf thisJoinPoint.getThis()

30.04.08

07-11-02

4.4.2 target-Pointcut

target (Type):
alle Joinpoints mit Zielobjekt vom Typ Type.

Bsp.: call(* *(..)) && target(telecom.Call)
ist äquivalent zu
call(* telecom.Call.*(..))
[2] mit Ausnahme des Aufrufs statischer Methoden.

target (Id):
- filtert wie zuvor
- bindet das Zielobjekt an Pointcut-Parameter Id.

Benutzungs-Bsp.:

pointcut callCall(Call c): call(* *(..)) && target(c);

07-11-02

4.4.3 args-Pointcut

args (ArgSpec {, ArgSpec})

ArgSpec ::= Type | Id | * | ..

Dabei gilt:

Type	passt zu jedem Argument dieses Typs (inkl. Unterklassen)
Id	Muss als Pointcut-Parameter "Type Id" vereinbart sein, passt zu Argument dieses Typs (inkl. Unterklassen).
*	passt zu Argument jeden Typs
..	passt zu 0 bis vielen Argumenten

Bsp.:

- args(telecom.Customer)

  nimmt alle Joinpoints mit 1 Argument vom Typ Customer, auch set/handler!
  auch z.B. caller.equals(receiver)
  (Customer: von Object: Customer)

- args(.., telecom.Customer, *)

  nimmt alle Joinpoints mit vorletztem Argument vom Typ Customer.

- before(final telecom.Customer c1): args(c1, *)

  nimmt alle Joinpoints mit 2 Argumenten, wenn das 1. vom Typ Customer ist, und bindet dieses an Parameter c1.

- this(telecom.Customer) && args(telecom.Call, ..)

  nimmt alle Joinpoints in Customer, deren 1. Argument ein Call ist.

7.5.08

07-11-07

4.5 Steuerfluss-basierende Pointcuts

Motivation: Das Protokollieren des kompletten Joinpoint-Kontexts (Übung 2) bei z.B. allen Methodenaufrufen benutzt viele toString()-Methoden.

Bsp.: Customer:

```
public String toString(){
    return name + "(" + areacode + ")";
}
```

(name: new; +: append; +: append; +: append; von StringBuilder)

call(* *(..)):

Protokollierung von z.B. Customer.call(Customer)
wird vermischt mit
Protokollierung von z.B. StringBuilder.append(String)

Wie vermeidbar?

call(* *(..)) && !withincode(String toString())
→ Kap. 4.6.1 Böhm

Was aber bei Customer:

```
public String toString(){
    return name + StringUtil.parenthesized(areacode);
}
```

Ziel: Ausschluss aller Joinpoints, die von einer toString()-Methode aufgerufen werden:

call(* *(..)) && !cflow(execution(String toString()))
↑ „control flow"

Def.: cflow(Pointcut') umfasst alle Joinpoints von Pointcut' und alle von diesen zur Laufzeit aufgerufenen Joinpoints („dynamischer Pointcut")

Def.: cflowbelow(Pointcut') ≙ cflow(Pointcut') && !Pointcut'
d.h. nur die von Pointcut'-Joinpoints aufgerufenen Joinpoints.

30.4.08

4.6 Programmtext-basierende Pointcuts

07-11-08

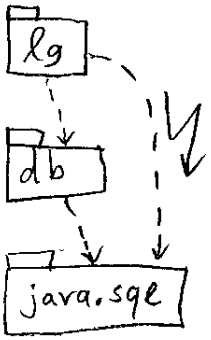
Einschränkung auf bestimmte Quelltext-Bereiche mittels

within (TypePattern) |

withincode (MethodPattern) |

withincode (ConstructorPattern)

Bsp.: Warnung beim Aufruf eines SQL-Dienstes von außerhalb der Datenbankschicht




```
[?] declare warning: call( *      java.sql..*(..))
                          && !within(db..*)
    : "Datenbank bitte nur über Paket db ansprechen!"
;
```

Bsp.: Warnung beim Aufruf einer eigenen Methode aus dem Konstruktor [unvollständig initialisiertes Objekt!] für Klasse C:

```
[?] declare warning: call(* C.*(..))
                          && withincode(C.new(..))
    : "Bitte nicht Methode für unvollständig intitialisiertes Objekt aufrufen!"
;
```

Bem.: Funktioniert leider nur für festen Klassennamen! ☹

07-11-09

4.7 Dynamische Pointcuts

Können erst zur Laufzeit ~~ausgewertet~~ werden.

if (BooleanExpression)

wobei thisJoinPoint im BooleanExpression zur Verfügung steht:

Joinpoint wird nur ausgewählt, wenn if-Bedingung wahr ist.

Bsp.: before ( ): !within (aspects.*) && if (thisJoinPoint.getArgs().length%2 == 0)
{ System.out.println(thisJoinPoint); }

[2] protokolliert nur Joinpoints mit einer geraden Anzahl von Argumenten.

7.5.08

adviceexecution() :

wählt Joinpoints bei Ausführung eines Advices.

Bsp.: Protokollierung eines anderen Aspekts

```
aspect A {
   :
   before(): ... {...}
}
aspect B {
   before(): adviceexecution() && within(A){
      System.out.println(thisJoinPoint);
   }
}
```

Bsp.: Garantierter Ausschluss von Aspekt-Selbstanwendung:

```
before(): myPointcut() && !cflow(adviceexecution()) {
   ...
}
```

07-11-08

JAMon: Java Application Monitor

+ ein freies, einfaches API
+ zur Beobachtung von Applikationen
+ effizient, für Produktionsbetrieb tauglich
+ threadsicher

Klassifizierung:

- Manuelle Codeinstrumentierung nötig (statt Profiling-Werkzeug)
+ SQL, Servlets: Aktivierbar über Konfiguration
+ Ideal zur Aktivierung durch Aspekte

Benutzung (manuell):

```
final Monitor mon = MonitorFactory.start(label);
... // zeitkonsumierende Aktivität
mon.stop();
```

Am Ende z.B.:

```
final String htmlReport
= MonitorFactory.getRootMonitor().getReport(1, "desc");
```

1: Sortierspalte; "desc": fallend ∈ {"desc" | "asc"}

Ergebnis:

Statistik aller Label mit Anzahl, Durchschnittsdauer, std-Abw. usw.

Quelle: http://jamonapi.sourceforge.net/

→ Von dort Powerpoint-Präsentation

12.11.07

07-11-14

5 Advice

Def.: Ein Advice gibt an, mit welchem Code alle Joinpoints eines Pointcuts zu instrumentieren sind.

5.1 Beispielanwendung: dito. telecom.

5.2 Definition

Syntax:

[strictfp] AdviceSpec [throws ExceptionTypeList] : Pointcut { Body }

AdviceSpec ::=

before (ParameterList) |

after (ParameterList) [(returning | throwing) [(Parameter)]] |

Type around (ParameterList)

07-11-14

5.3 Kontextzugriff

Formaler Parameter eines Advices (before, after oder around)
- kann belegt werden mit Pointcut-Parameter → Kap. 4

Gleichwertig zugreifbar im Advice-Rumpf sind:
- returning-Parameter,
- throwing-Parameter.

} → 5.4.2

AdviceSpec und throws-Klausel ähneln einem Methodenkopf.
Die AdviceSpec stellt Parameter für den Advice-Body bereit.
Die throws-Klausel gibt an, welche Ausnahmen der Advice-Body werfen darf (an der Stelle des Joinpoints).

14.5.08

Bsp.:
```
before (Customer c) throws CustomerException:
    call(* *(..)) && args(c) && !cflow(adviceexecution())
{
    if(!Customer.isValid(c)){
        throw new CustomerException(c);
    }
    ...
}
```

07-11-14

5.4 Advice-Typen

before | after | around

5.4.1 before-Advice

fügt Code vor dem Joinpoint ein.

Nutzbar für:

- Vorbedingungsprüfung → Bsp. in Kap. 5.3
- Logging
- Statistik,

07-11-14

5.4.2 after-Advice

fügt Code nach dem Joinpoint ein.

Nutzbar für:
- Invariantenprüfung
- Nachträgliche Manipulation (Sicherheit, Optimierung, Ausnahmen)
- Zeitlicher Anker für Abschlussaktionen.

a) after(): Ähnlich zu finally:

Bsp.:
```
after(): execution(public static void main(..)){
    ... // gib Statistik aus
}
```

wird immer nach der main-Methode (auch bei Abbruch durch Ausnahme) ausgeführt.

12.11.07

b) after() returning()

wird nur nach erfolgreicher Joinpoint-Beendigung ausgeführt.

Bsp.:
```
after() returning(Call call): execution(* *(..)){
    System.out.println("␣␣" + thisJoinPoint + "␣returns␣" + call);
}
```
(thisJoinPoint: StaticPart)

meldet für jede erfolgreiche Ausführung einer Methode, die ein Call-Objekt liefert, dieses:

execution(Call telecom.Customer.call(Customer)) ␣returns␣ telecom.Call@10d448

c) after() throwing()

wird nur nach Ausnahme-Abbruch des Joinpoints ausgeführt.

Bsp.:
```
after() throwing(Throwable t): execution(* *(..)){
    System.out.println("␣␣" + thisJoinPoint + "␣throws␣" + t);
}
```
(thisJoinPoint: StaticPart)

meldet für jede Ausführung einer Methode, die eine Ausnahme wirft, dieses z.B.:

execution(Call telecom.Customer.call(Customer))
throws java.lang.IllegalArgumentException: Customer Jim must not call himself!

? Propagiert die Ausnahme dadurch noch weiter?

07-11-15

5.4.3 around-Advice

ersetzt den Joinpoint mit der Möglichkeit, ihn aufzurufen.

Nutzbar für z.B.:
- Zeitmessungen
- Begleitendes Verhalten (z.B. Transaktionssicherung, Ausnahmemelden, Abrechnung)
- Ergebnismanipulation, Fehlerinjektion für Tests.

`proceed(ArgumentList);`
ruft dann den Joinpoint auf.

Bsp.:

```
Aspekt: int around(): call(int size()) {
            return 3;
        }
        ...
Anw.:   final List list = new ArrayList();
        list.add("Anton");
        System.out.println(list + ", size=" + list.size());
        // [?]. [Anton], size = 3
```

Bsp. mit Aufruf des Joinpoints:

```
Aspekt: int around(): call(int size()) {
            return 2*proceed();
        }
```

Ausgabe des Anw.-Codes:

[?] [Anton], size = 2
↑

07-11-16/1

noch 5.4.3 around-Advice

Zeitmessung von Methodenaufrufen mit JAMon z.B.:

```
Object around(): call(* *(..)) && !within(aspects.*)
{
    final Monitor mon = MonitorFactory.start(
        thisJoinPointStaticPart.getSignature().toShortString()
    );
    try {
      return proceed();
    } finally {
      mon.stop();
    }
}
```

Wirkung: [?] Messen der Ausführungsdauer von allen Methodenaufrufen, auch wenn sie durch Ausnahme abgebrochen werden. 21.5.08

Bem.: Ein Ergebnistyp des Advices (hier Object) ist nötig für alle Joinpoints, die ein Ergebnis liefern (call, execution, get). Einschränkung bis hin zu void möglich.

noch 5.4.3 around-Advice

07-11-16/2

Advice-Signatur

Regel: Ergebnistyp und Parametertypreihenfolge des around-Advices müssen zwischen proceed und around übereinstimmen.

Bsp. Fehlerinjektion:

```
Object around (Customer caller, Customer callee):
call (Call call(..)) && target(caller) && args(callee)
{
    return "Mik(650)".equals(caller.toString())
        ? proceed (callee, caller)
        : proceed (caller, callee)
    ;
}
```

Dies vertauscht Anrufer und Angerufenen, wenn Mik(650) der Anrufer ist.

26.11.07

Noch Transakt vorführen!

28.3.08

07-11-29

6 Intertype-Deklaration

Ein Advice fügte Anweisungen in andere Methoden ein.

Hier lernen wir, Deklarationen in andere Klassen einzufügen.

6.1 Erweiterung bestehender Klassen

Bsp. Ergänzen der Klasse StringBuilder um eine Methode trim.
[gibt es bisher nur in String]

```
/** Deletes leading and trailing whitespace from the content.*/
public void java.lang.StringBuilder.trim(){
    int i = ... // suche erstes nichtweißes Zeichen
    delete(0, i);
    int j = ... // suche letztes ——— " ———
    delete(j+1, length());
}
```

(length(): im Kontext von StringBuilder)

Wann welche Erweiterung einer Basisklasse B sinnvoll?

B-Quellcode ändern	• B-Quellcode wird von uns gepflegt && • Änderung ist allgemeingültig
A extends B {...}	• Änderung spezifisch && • Objekterzeugung (of A oder B) steht unter unserer Kontrolle
Intertype-Deklaration	• Objekterzeugung nicht unter unserer Kontrolle.

6.2 weggelassen

07-11-30/1

6.3 Intertype-Deklarationen-Übersicht

Möglich sind: Methoden, Konstruktoren, Attribute.

Methoden

Modifier ResultType OnType . Identifier (ParameterList)
[throws ExceptionList] ({ Body } | ;)

⇒
Die Klasse oder das Interface OnType wird um die Methode Identifier erweitert. [ohne seinen Namen zu ändern!]

Konstruktoren

Modifier OnType . new (ParameterList)
[throws ExceptionList] { Body }

⇒
Die Klasse OnType wird um den Konstruktor erweitert.

Attribute

Modifier Type OnType . Identifier [= Expression] ;

⇒
Die Klasse oder das Interface OnType wird um das Attribut „Identifier" erweitert.

Regeln dazu

- Ein bestehendes solches Element[?] kann nicht ersetzt werden! Vorsicht bei: Default-Konstruktor.
- this im „Body" oder Initialisierungs-„Expression" zeigt auf[?] das OnType-Objekt, nicht auf das Aspekt-Objekt!
- final-Attribute dürfen nur im eigenen, nicht in einem Intertype-Konstruktor initialisiert werden. [wg. Verständlichkeit].

5.12.07

noch 6.3 Intertype-Deklarationen-Übersicht

Aufgabe: In telecom zu jeder Connection die Dauer zwischen complete() und drop() mit Teilnehmern melden.

Lösung: [Ansatz?]

```
private long Connection.startMS = 0;
private long Connection.getDurationMS(){return System.currentTimeMillis()
                                        - this.startMS;}
after(final Connection conn): target(conn) && call(* complete()){
    conn.startMS = System.currentTimeMillis();
}
after(final Connection conn): target(conn) && call(* drop()){


    System.out.println(
        "Verbindung " + conn.getCaller()
        + "␣→␣" + conn.getReceiver()
        + "␣dauerte␣" + conn.getDurationMS() + "␣ms."
    );
}
```

Bsp.-Meldung:

Verbindung Jim(650) --> Mik(650) dauerte 203 ms.

4.6.08

6.4 Zugriffskontrolle

private-Attribut nur von der umgebenden Einheit (Klasse, Aspekt) zugreifbar!

Bsp.: `private Customer caller;` // nur aus Connection zugreifbar, // weil dort deklariert.

`private long Connection.startMS;` // nur aus dem Aspekt zugreifbar, // obwohl es zu Connection gehört.

privileged-Aspekte dürfen dies dennoch!

07-11-3013

6.5 Konfliktsituationen

entstehen bei Intertype-Deklaration zweier gleichnamiger Elemente in verschiedenen Aspekten.

6.5.1 Konfliktfrei bei private

Bsp.: bzgl. startMS:

```
aspect PendingLogAspect {
    private long Connection.startMS = 0;
    usw. ähnlich zuvor after... call(* complete()) { println(..)
}
aspect CompletedLogAspect {
    private long Connection.startMS = 0;
    usw. wie zuvor after ... call(* drop()) { println(...)
}
```

Regel: Jeder Aspekt verwaltet eigene private-Elemente.
⇒ Kein Konflikt.

6.5.2 Konflikt bei nicht-private

Bsp.: Compilefehler wegen printDuration bei:

```
aspect PendingLogAspect {
    private long Connection.startMS = 0;
    public void Connection.printDuration(){
        System.out.println("Connection was pending " +
            (System.currentTimeMillis() - startMS) + " ms."
        );
    }
}
aspect CompletedLogAspect {
    ...
    public void Connection.printDuration(){
        System.out.println("Connection was active " +
        + ...
    ...
}
```

Auflösbar durch Deklaration:

```
declare precedence: CompletedLogAspect, PendingLogAspect;
```

↑ dieser Aspekt hat dann Vorrang.

11.12.07

07-12-05

6.6 Eltern adoptieren

Statische Eingriffe in die Beerbung.

6.6.1 Interface hinzufügen

declare parents : TypePattern implements TypeList ;

Nützlich zum Einpassen bestehender Klassen in eine bestehende Infrastruktur.

Bsp.: Für Ablage in eine Tomcat-Session oder für Versand über RMI müssen alle Objekt(teile) Serializable (Marker-Interface) sein.

```
declare parents: fb6..* implements Serializable;
```

Bsp.: Für Nachvollziehbarkeit sollen alle Logik-Objekte einen Zeitstempel der letzen Änderung erhalten.

```
public interface Timestamped {
    long getTimestamp();
}
public aspect TimestampAspect {
    private long Timestamped.millis;
    public long Timestamped.getTimestamp(){
        return millis;
    }
    declare parents: fb6.*.lg..* implements Timestamped;
    before(Timestamped lgObject): target(lgObject)
                                  && set(* *) && !set(long millis)
    {
        lgObject.millis = System.currentTimeMillis();
    }
```

11.06.08

→ 6.6.2

07-12-05

6.6.3 Oberklasse ändern

declare parents : TypePattern extends Type ;

Regel: Type muss Unterklasse der bisherigen Oberklassen sein!

Nutzen: Zum Einziehen einer Zwischenebene,
z.B. für Zusatzfunktionalität.

Bsp.: Erweiterung von JUnit-TestCase um Assertions für Collections.

```
class DiagTestCase extends TestCase {

    public static void
    assertContains(Collection expected, Object actual){
        if(!expected.contains(actual)){
            fail("Object " + actual + " not contained in " + expected);
        }
    }
}
```

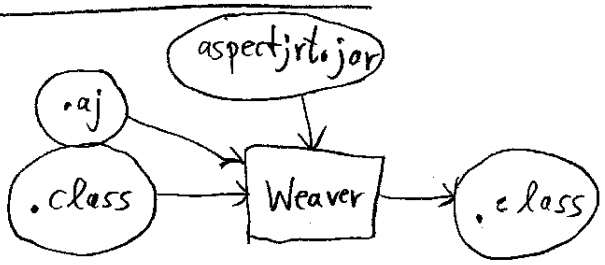
Diese Assertion kann in allen TestCases verwendet werden nach:

```
declare parents : (TestCase+ && !TestCase)
extends DiagTestCase;
```

08-06-18/1

AspectJ 5 – Load Time Weaving

PN-DFD:

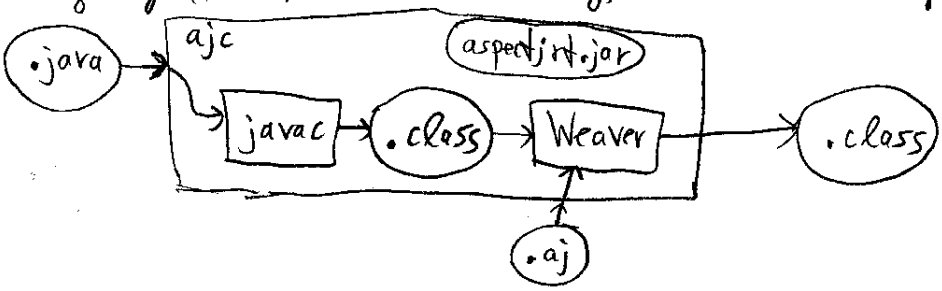


Einführung:

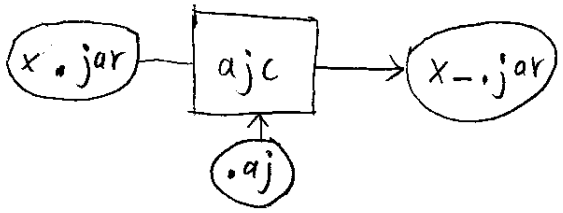
Weben („weaving"):

Webezeitpunkt

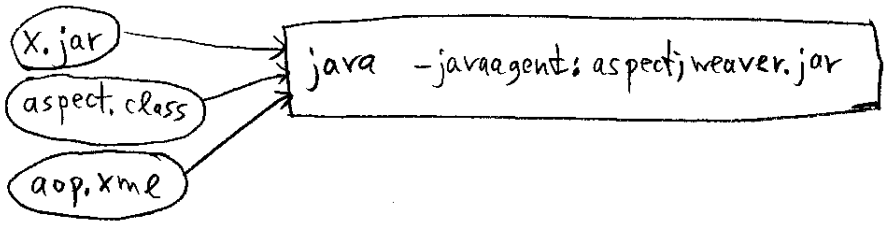
- Übersetzung („compile-time weaving"): Bisher benutzt



- Nach-Übersetzung („post-compile weaving"): Für Fremdbibliotheken mittels Optionen -inpath und -outjar:



- Laden („load-time weaving", LTW): Für Fremdbibliotheken erst beim Klassenladen



LTW

08-06-18/2

Nutzen:

+ Instrumentieren von Fremdbibliotheken
+ Keine Varianten von .jar-Dateien

Verfahren:

- Datei META-INF/aop.xml zur Konkretisierung von AspectJ-Aspekten benutzen.
- Aktivieren mittels Weaver als Java Agent oder in Eclipse im Run-Dialog > AspectJ/Load-Time Weaving Application

Zeigen:

1) Abstrakten CatchLoggingAspect im Projekt fb6

2) aop.xml

```
<concrete-aspect ... />
<weaver options="-debug">
<include within="org.castor..*" />
```

3) StartOnlyDB.bat

4) a) DbSessionTest: Run as JUnit Test ✓

b) — " — Java Application ✓ dito → Console

c) Run > AspectJ Load-Time Weaving Application > DbSessionTest LTW

- Viele Meldungen debug: Welche Klassen gewoben, welche nicht
- Stack Traces von Castor-Ausnahmen, z.B. NoSuchMethodException von hasFileName()
- Dennoch gleiches Ergebnis: OK (4 tests).

07-12-12

Abschluss

Alternative AOP-Systeme

- JBoss AOP (XML-basiert)
- Spring AOP (alternative @Annotation-Syntax)
  → Kapitel 6

08-07-08

Profiling mit Java HPROF

Ziel: Engpässe (CPU-Zeit, Speicher) herausfinden.

Texte dazu:

- HPROF Technical Article (→ Üb. 6)
- PerfAnal Technical Tip by Nathan Meyers (a. 2000, → Üb. 6)

Ab Java 5:

- HPROF basiert auf JVMTI
  (Java Virtual Machine Tooling Interface)

Lehrbeispiel Summe (→ Üb. 6)

Wo wird die meiste Zeit verbraucht bei 1000000000?

a) Methode? [_summe]

b) Zeile? [12: Schleifenkopf]

c) kumulativ Methode? [main]

Meyers-Artikel

```
java -Djava.compiler=NONE -agentlib:hprof=cpu=samples,
                                          depth=99
                                  Summe 100000000
```

(-Djava.compiler=NONE: ohne JIT)

```
java -jar PerfAnal.jar java.hprof.txt
```

HPROF-Artikel

Java-Compiler Profiling:

```
javac -J-agentlib:hprof=cpu=samples,depth=99
                                  String Multiply.java
```

25.6.08