Aspektorientierte Programmierung mit AspectJ 5

1. Auflage

von
Christoph Knabe
Technische Fachhochschule Berlin
Fachbereich VI (Informatik)
www.tfh-berlin.de/~knabe/

3 JOINPOINTS UND POINTCUTS

3.1 Joinpoints im Programm

Def.: Ein Joinpoint (Verknüpfungspunkt) ist ein Punkt in der Ausführung eines Programms, an dem in die Ausführung eingegriffen werden kann.

Bsp.: Bei

```
konto.einzahlen(1000.0);
```

ist der Aufruf von `einzahlen` ein Joinpoint.

Was für Joinpoints gibt es?

Bei der Ausführung der Zeilen

```
final Konto konto = new Konto();
konto.einzahlen(5*einzelbetrag);
konto.ueberweisenAuf(200, anderesKonto);
```

mit dem Kontobeispiel von [Böhm06], p. 7, erhältlich unter

http://www.aosd.de/buecher/AOP_AspectJ/examples/Kap03-Joinpoint/1Konto/src/bank/Konto.java

bemerkt AspectJ, wenn wir folgenden Aspekt anwenden

```
package aspects;

public aspect Logging {

    before(): !within(aspects.*) {
        System.out.println(thisJoinPointStaticPart);
    }

}
```

folgende Joinpoints:

```
call(bank.Konto())
preinitialization(bank.Konto())
initialization(bank.Konto())
execution(bank.Konto())
set(double bank.Konto.kontostand)
call(void bank.Konto.einzahlen(double))
execution(void bank.Konto.einzahlen(double))
get(double bank.Konto.kontostand)
set(double bank.Konto.kontostand)
call(void bank.Konto.ueberweisenAuf(double, Konto))
execution(void bank.Konto.ueberweisenAuf(double, Konto))
call(void bank.Konto.abheben(double))
execution(void bank.Konto.abheben(double))
get(double bank.Konto.kontostand)
set(double bank.Konto.kontostand)
call(void bank.Konto.einzahlen(double))
execution(void bank.Konto.einzahlen(double))
get(double bank.Konto.kontostand)
set(double bank.Konto.kontostand)
```

Was wird bemerkt?

2

Was wird nicht bemerkt?

3

An allen oben protokollierten Punkten kann man mit AspectJ in den Programmablauf eingreifen!

Was ist ein Pointcut

Def.: Ein Pointcut ist das Sprachkonstrukt, mit dem man Joinpoints auswählen kann.

Ein Pointcut stellt also eine Joinpointmenge dar. Eventuell verstehbar als Joinpoint-Crosscut (Eingriffspunkt-Querschnitt), d.h. Ein Querschnitt durch das Programm, der nur bestimmte Joinpoints trifft.

Bsp.: `!within(aspects.*)`

ist ein Pointcut, der alle Joinpoints innerhalb des Pakets `aspects` ausschließt (Negation per `!`).

3.2 Primitive Pointcuts in AspectJ

Für eine konkrete Programminstrumentierung muss man meist nur wenige Joinpoints aus obiger Liste ansprechen. Manchmal reichen aber diese auch nicht aus. Jetzt folgen die einzelnen Joinpoints von AspectJ.

call(*MethodPattern*) oder **call(*ConstructorPattern*)**

Aufruf einer Methode oder eines Konstruktors.

Bsp.: `call(public void bank.Konto.einzahlen(double))`

ist ein Pointcut, der alle Aufrufe der Methode `einzahlen` der Klasse `bank.Konto` auswählt.

Bsp.: `call(bank.Konto.new(..)))`

ist ein Pointcut, der alle Aufrufe von Konstruktoren der Klasse `bank.Konto` auswählt.

execution(*MethodPattern*) oder **execution(*ConstructorPattern*)**

Ausführung einer Methode oder eines Konstruktors.

Bsp.: `execution(String *..*(..))`

ist ein Pointcut, der alle Ausführungen aller String-liefernden Methoden aller Pakete auswählt.

Bsp.: `execution(bank.Konto.new()))`

ist ein Pointcut, der alle Ausführungen des parameterlosen Konstruktors der Klasse `bank.Konto` auswählt.

get(*FieldPattern*)

Lesezugriff auf ein Attribut eines Objektes oder einer Klasse

Bsp.: `get(java.io.PrintStream System.*)`

ist ein Pointcut, der alle Lesezugriffe auf die in der Klasse `java.lang.System` deklarierten Attribute auswählt. Diese sind ???`System.in`, `System.out` und `System.err`. Dies kann man gut benutzen, um ein Logging über eine etablierte Logging-Schnittstelle wie *Apache Commons Logging* zu erzwingen statt der nicht global konfigurierbaren direkten Ausgabe über `System.out`.

set(*FieldPattern*)

Schreibzugriff auf ein Attribut eines Objektes oder einer Klasse

Bsp.: `set(double bank.Konto.kontostand)`

ist ein Pointcut, der alle Schreibzugriffe auf das Attribut `kontostand` von Objekten der Klasse `bank.Konto` auswählt. Dies kann man benutzen, um Kontoveränderungen zu überwachen.

handler(*TypePattern*)

Abfangen von Ausnahmen

Bsp.: `handler(RuntimeException+)`

ist ein Pointcut, der alle catch-Zweige, in denen eine `RuntimeException` oder davon abgeleitete abgefangen wird, auswählt. Dies kann man z.B. benutzen, um Ausnahmebehandlungen zu protokollieren.

staticinitializaton(*TypePattern*)

Initialisierung einer Klasse

Bsp.: `staticinitialization(bank.Konto)`

ist ein Pointcut, der die Initialisierungen der Klasse `bank.Konto` auswählt. Zeitlich liegt dies noch vor einer eventuell in Konto enthaltenen `main`-Methode.

preinitializaton(*ConstructorPattern*)

Nach dem Aufruf eines Konstruktors, aber noch vor dem Aufruf des Konstruktors seiner Oberklasse

Bsp.: `preinitialization(bank.Konto.new())`

ist ein Pointcut, der den Zeitpunkt nach dem Aufruf von `new Konto()`, aber noch vor der Initialiserung des `Object`-Anteils von Konto auswählt.

initializaton(*ConstructorPattern*)

Ausführung eines Konstruktors nach einem (eventuell impliziten) Aufruf des Konstruktors seiner Oberklasse

Bsp.: `initialization(new(..))`

ist ein Pointcut, der die Ausführung des eigentlichen Konstruktorrumpfes aller Konstruktoren (nach eventuellem `super`(...)) auswählt.

3.3 Wildcards

Man kann eine Größe durch ihren vollständig ausgeschriebenen Namen identifizieren. Dazu kann man (auch Objektattribute) durch Angabe von *`Paket.Klasse.Attribut`* ansprechen.

Oft wird man aber mehrere Joinpoints auf einmal ansprechen wollen. Dafür können wir Muster mit folgenden Jokerzeichen verwenden:

- ***** (Stern) steht für beliebig viele Bezeichner-Zeichen (Buchstaben, Ziffern, _, `$`, aber kein Punkt)
- `..` (2 Punkte) steht für einen beliebigen Namensbestandteil, der mit Punkt beginnt und endet oder für eine beliebige Anzahl von Parametern
- **+** (Plus) steht für einen Typ und alle seine Unterklassen.

Bsp.:
`*Test` alle auf Test endenden Klassennamen
`java..*` alle Klassen aus java-Plattformpaketen (z.B. java.lang, java.sql, java.io)
`Throwable+` alle Ausnahmeklassen

3.4 Logische Operatoren

Mittels der logichen Operatoren kann man Muster kombinieren.

- **!** (Nicht) steht vor einem Annotationsmuster, Typmuster, Modifier oder Pointcut und verneint diesen.

- || (Oder) steht zwischen zwei Annotationsmustern, Typmustern oder Pointcuts und bildet die Vereinigungsmenge
- **&&** (Und) steht zwischen zwei Typmustern oder Pointcuts und bildet die Schnittmenge.

Bsp.: `call((boolean||byte||short||int||long||float||double||char) *(..))`

wählt alle Aufrufe von Methoden, die ein Ergebnis eines primitiven Typs liefern, aus.

3.5 Signaturen und Muster

Zur Auswahl eines Systemteils kann seine exakte Signatur verwendet werden. Daneben kann mit einem Signaturmuster eine Menge von Systemteilen ausgewählt werden.

Typmuster

Der Name eines Typs (Klasse oder Interface), ob mit oder ohne Paketnamen, ist seine Signatur. In einem Typmuster können die Wildcards und logischen Operatoren verwendet werden.

Bsp.: `long||Long`

wählt den primitiven Typ `long` und seine Hüllklasse `Long` aus.

Bsp.: `java..*`

wählt alle Klassen aus dem Paket `java` oder seiner Unterpakete aus.

Annotationsmuster

Java 5-Annotationen kann man durch Annotationsmuster ansprechen. Sie haben den Aufbau:

- @*Typname*, z.B. `@Foo` oder `@org.xyz.Foo`.
- @(*Typmuster*), z.B. `@(org.xyz..*)` oder `@(Foo || Boo)`
- *Annotationsmuster Annotationsmuster*, bedeutet beide müssen vorhanden sein, z.B. `@Foo @Boo`.
- !*Annotationsmuster*, bedeutet, diese Annotation darf nicht vorhanden sein. z.B. `!@Foo`

Methodenmuster

Eine Methodensignatur hat den Aufbau

[*Annotationen*] [*Modifizierer*] *Ergebnistyp Typ.Methodenname*([*Parameterliste*]) [**throws** *Ausnahmen*]

Die meisten Teile darin sind optional. Mit den Wildcards und logischen Operatoren kann man mächtige Muster zur Auswahl von Methoden schreiben.

Bsp.: `!String toString(..)`

wählt alle Methoden aus, die zwar `toString` heißen, aber keinen String liefern. Z.B. für Qualitätssicherung.

Aufgabe: Geben Sie das Muster an, um alle Methoden der üblichen `toString`-Signatur herauszufinden, bei denen die ab Java 5 empfohlene Annotation `@Override` fehlt.

4

Aufgabe: Geben Sie das Muster an, um die Aufrufe aller auf `System.out` anwendbaren `print`- oder `println`-Methoden auszuwählen, aber nicht die `printf`-Methoden.

5

Konstruktormuster

Eine Konstruktorsignatur unterscheidet sich von einer Methodensignatur nur darin, dass sie keinen Ergebnistyp hat und dass der „Methodenname“ **new** ist.

Bsp.: `java.util.Vector.new(..)`

wählt alle Konstruktoren von `Vector` aus.

Da bei `Thread`-lokaler Benutzung die mit `Vector` verbundene Synchronisation unnötig aufwändig ist, sollte man ihn eventuell durch eine `java.util.ArrayList` ersetzen. Folgende statische Deklaration in einem Aspekt verhilft uns zu einer eigenen Compiler-Warnung an allen Stellen, wo ein `Vector`-Konstruktor aufgerufen wird:

```
declare warning: call(java.util.Vector.new(..))
: "Please check, if synchronized object is necessary!";
```

Attributmuster

Eine Attributsignatur hat den Aufbau

[*Annotationen*] [*Modifizierer*] *Attributtyp* [*Typ***.**]*Attributname*

Die Teile in eckigen Klammern sind optional. Mit den Wildcards und logischen Operatoren kann man wiederum Muster zur Auswahl von Attributen (engl. *field*) schreiben.

Bsp.: `set(* (@Persistent Object+).*)`

wählt alle Stellen aus, an denen ein Attribut eines Objekts einer Klasse, die als `@Persistent` markiert ist, verändert wird. An diesen Stellen müsste man das Objekt für eine erneute Speicherung vormerken (so genanntes *dirty flag*).

Bsp.: `declare warning: set(static * *): "Please do not use static fields, but set them in a constructor or setter!";`

warnt an allen Stellen, an denen ein Klassenattribut gesetzt wird. **Bem**.: In der Komponentenarchitektur sind statische Attribute zu vermeiden. Man soll nötige Objekte immer über einen Konstruktor oder Setter injizieren (*dependency injection*).

Ausnahmebehandlermuster

Eine Ausnahmebehandlersignatur hat den einfachen Aufbau

Ausnahmetyp

Bsp.: `handler(IOException)`

wählt alle Stellen aus, an denen exakt `catch(IOException` *name*`)` steht.

Ein sinnvolleres Beispiel ist, sich auf alle Behandler von Ausnahmen einer bestimmten Ausnahmenhierarchie zu beziehen.

Bsp.:
```
declare warning: handler(java.lang.Error+)
: "Sun does not recommend to catch Error!";
```

gibt Warnungen für alle Stellen aus, an denen ein `Error` oder eine Unterklasse davon abgefangen wird. Dies ist deshalb nicht empfohlen, da bei einem `Error` normalerweise die virtuelle Maschine nicht weiterarbeiten kann und daher die Ausnahme auch nicht sinnvoll behandelt werden kann.

3.6 Zugriff auf den Kontext

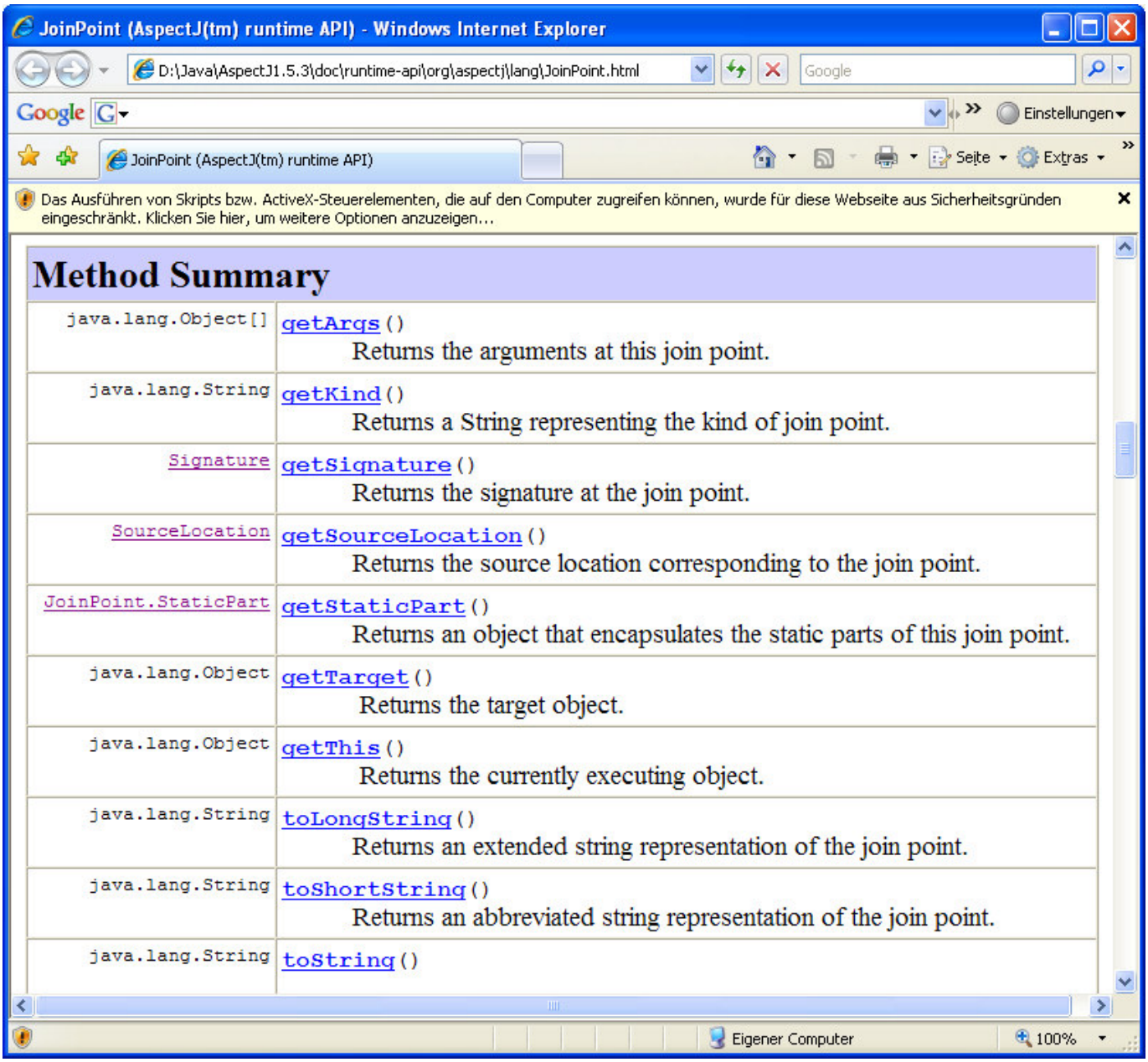
Im Advice kann auf den aktuellen Joinpoint-Kontext wie folgt zugegriffen werden:

Name	*Typ*	*Bedeutung*
`thisJoinPoint`	org.aspectj.lang .JoinPoint	Laufzeit-Kontextinfos, siehe AspectJ-API
`thisJoinPointStaticPart`	org.aspectj.lang .JoinPoint.StaticPart	Nur die statischen Anteile davon, benötigt weniger Speicher
`enclosingJoinPointStaticPart`	org.aspectj.lang .JoinPoint.StaticPart	Die statischen Anteile des umgebenden Joinpoints

Diese Zugriffe sind nützlich für Logging oder verändernde Eingriffe.

3.6.1 thisJoinPoint-Methoden

`thisJoinPoint` ist vom Typ `org.aspectj.lang.JoinPoint`. Dieses ermöglicht Zugriff auf nur zur Laufzeit verfügbare Kontextinformationen zum Joinpoint mittels Reflection.



In der API-Doku von AspectJ sind dessen Methoden beschrieben.

Benutzung z.B.:

```
final org.aspectj.lang.JoinPoint joinPoint = thisJoinPoint;
System.out.print(joinPoint.getSourceLocation());
```

Die Ausgabe erscheint im Format *Dateiname*:*Zeilennummer*, z.B.:

```
AbstractSimulation.java:32
```

Nun zu den einzelnen Methoden:

toShortString(), toString(), toLongString()

Diese Methoden liefern eine Stringdarstellung der Joinpoint-Signatur in verschiedenen Ausführlichkeitsstufen.

toShortString(): *Klassenname*`.`*Methodenname*`(..)`, z.B.

```
execution(Konto.einzahlen(..))
```

toString(): *Ergebnistyp Paketname*`.`*Klassenname*`.`*Methodenname*`(`*Parametertypen*`)`, z.B.

```
execution(void bank.Konto.einzahlen(double))
```

toLongString(): *Modifizierer Ergebnistyp Paketname*`.`*Klassenname*`.`*Methodenname*`(`*Parametertypen*`)`, z.B.

```
execution(public void bank.Konto.einzahlen(double))
```

Object getThis()

Diese Methode liefert das aktuelle Objekt des Joinpoints, wenn er sich auf eine Objektmethode bezieht. Bei einer Klassenmethode (**`static`**), die ja ohne Objektbezug ist, wird **`null`** geliefert. Das Ergebnis muss zu dem Zieltyp gecastet werden. **Bsp**.:

```
final Object thisObject = thisJoinPoint.getThis();
final bank.Konto konto = (bank.Konto)thisObject;
```

Wenn auf Grund der Pointcut-Signatur der Typ des **`this`**-Objekts zur Compilezeit fest steht, sollte man besser den speziellen typsicheren **`this`**-Pointcut benutzen [Böhm, Kap. 4.4.1].

Object getTarget()

Diese Methode liefert das Zielobjekt des Joinpoints. Nur bei einem **`call`**-Joinpoint sind **`this`** und **`target`** verschieden. Bei einer Klassenmethode (**`static`**), die ja ohne Objektbezug ist, wird **`null`** geliefert. Das Ergebnis muss zu dem Zieltyp gecastet werden. **Bsp**.:

```
final Object targetObject = thisJoinPoint.getTarget();
final bank.Konto konto = (bank.Konto)targetObject;
```

Wenn auf Grund der Pointcut-Signatur der Typ des **`target`**-Objekts zur Compilezeit fest steht, sollte man besser den speziellen typsicheren **`target`**-Pointcut benutzen [Böhm, Kap. 4.4.2].

Object[] getArgs()

Diese Methode liefert die aktuellen Argumente des Joinpoints.

Bei einem **`call`**-Joinpoint sind dies die übergebenen Werte.

Bei einem **`execution`**-Joinpoint sind dies die erhaltenen Werte.

Bei einem **`handler`**-Joinpoint enthält die Reihung das abgefangene Ausnahmeobjekt.

Bei einem **`set`**-Joinpoint enthält die Reihung den neuen Attributwert.

String getKind()

Diese Methode liefert einen die Art des Joinpoints identifizierenden String, z.B. `method-execution`.

StaticPart getStaticPart()

Diese Methode liefert den statischen Anteil eines JoinPoints. Im Allgemeinen ist es empfohlen, stattdessen das Schlüsselwort **`thisJoinPointStaticPart`** zu verwenden.

3.6.2 ThisJoinPointStaticPart-Methoden

Mit diesen Methoden kommt man an die zur Compilezeit bekannten JoinPoint-bezogenen Informationen. Dies sind die gleichen wie bei JoinPoint mit Ausnahme der Methoden:

`getArgs()`, `getThis()` und `getTarget()`.