5 Advice

Programme sind wie Kinder:
Sie machen nie das, was sie sollen.
(Oli B.)

Wir sind in den vorigen Kapiteln schon mit Advices in Berührung gekommen, um die Wirkungsweise von Pointcuts kennen zu lernen. In diesem Kapitel werden wir einen genaueren Blick auf die Advices selbst werfen und

- Pointcuts definieren,
- innerhalb eines Advices auf den Kontext des Joinpoints zugreifen,
- mit Exceptions umgehen,
- die unterschiedlichen Advice-Arten (*before*, *after*, *arround*) einsetzen,
- *proceed* kennen lernen und
- die Vorrangregeln verstehen lernen.

Advices enthalten den eigentlichen Code, der vom AspectJ-Compiler über die Pointcuts an den entsprechenden Stellen eingeflochten wird.

5.1 Ziehung der Lottozahlen

Lottotrommel in Software

Als Beispiel für dieses Kapitel soll die Ablösung der mechanischen Lottomaschine durch eine Programmlösung dienen. Der Anfang einer Java-basierten Software-Trommel könnte so aussehen:

```
public class Trommel {

    private Vector balls = null;

    Trommel() {
        balls = new Vector(49);
        for (int i = 1; i <= 49; i++) {
            balls.add(new Integer(i));
        }
    }
    ...
```

Abbildung 5.1
Sechs Richtige

Ziehung der Lottozahl

Die Kugeln der Lottomaschine werden durch einen Vector repräsentiert, der die Zahlen 1 bis 49 enthält. Diese werden über den Konstruktor in der Trommel platziert. Für die Ziehung der Lottozahlen dient die Methode »getNumber()«, die eine beliebige Kugel aus unserer virtuellen Trommel zieht:

```
public Integer getNumber() {
    Random random = new Random(System.currentTimeMillis());
    int n = random.nextInt(balls.size());
    return (Integer) balls.remove(n);
}
```

Wie man an diesem Codeausschnitt sieht, wird die Kugel tatsächlich aus der Trommel entfernt. Was gegenüber der realen Lottoziehung noch fehlt, ist der Mischvorgang und der Notar, der sich von dem ordnungsgemäßen Zustand der Maschine überzeugt hat. Über Letzteres werden wir uns später Gedanken machen, das Mischen realisieren wir dadurch, dass wir eine beliebige Kugel ziehen und wieder an das Ende des Vektors anfügen:

```
public void mixNumbers() {
    mixNumbers(100);
}

public void mixNumbers(int n) {
    for (int i = 0; i < n; i++) {
        Integer ballnumber = getNumber();
        balls.add(ballnumber);
    }
}
```

Mix it!

Standardmäßig wird die Trommel hundert Mal durchmischt, auf

Wunsch bzw. über die Verwendung der zweiten mixNumbers()-Methode kann die Durchmischung auch öfter stattfinden. Jetzt haben wir alles beisammen, um die erste Ziehung vorzunehmen:

Ziehung der Lottozahlen

```
public static void main(String[] args) {
    System.out.print("Ziehung:");
    Trommel trommel = new Trommel();
    for (int i = 0; i < 6; i++) {
        trommel.mixNumbers();
        System.out.print(" " + trommel.getNumber());
    }
}
```

Damit sich der Notar von der ordnungsgemäßen Funktionsweise der virtuellen Lottotrommel überzeugen kann, müssen wir neben den Lottozahlen auch noch Ablauf und Zustand der Trommel protokollieren. Dazu werden wir in diesem Kapitel eine Reihe unterschiedlicher Advices zu Hilfe nehmen.

5.2 Definition

Syntax

[**strictfp**] *AdviceSpec* [**throws** *Exception-Liste*] **:** *Pointcut* {
    Body
}
AdviceSpec :=
    before (*Parameter*) |
    after (*Parameter*) |
    after (*Parameter*) **returning** [(*Forma*)] |
    after (*Parameter*) **throwing** [(*Forma*)] |
    Typ **around** (*Parameter*)

Um den Aufruf sämtlicher Lotto-Methoden protokollieren zu können, definieren wir uns als Erstes einen Pointcut, den wir anschließend im Advice verwenden werden:

```
pointcut callAnyLottoMethod() :
    call(public * lotto..*(..));

before() : callAnyLottoMethod() {
    System.out.println(thisJoinPoint);
}
```

Advice $\simeq$ Methode

Schauen wir uns den Advice etwas genauer an: Eingeleitet wird er durch die Art des Advices (hier der Before-Advice), gefolgt vom Pointcut, der die Stelle adressiert, an der der Advice eingefügt werden soll. Der Rumpf des Advices enthält den eigentlichen Code und ist mit dem Rumpf einer normalen Methode vergleichbar. Ebenso wie bei Methoden können an Advices auch Parameter übergeben werden:

```
pointcut callMixNumbers(int i) :
    call(public void mixNumbers(int)) && args(i);

before(int i) : callMixNumbers(i) {
    System.out.println(thisJoinPoint + " arg = " + i);
}
```

Wie wir später beim around()-Advice sehen werden, können Advices sogar einen Rückgabewert besitzen. Innerhalb des Advices hat man Zugriff auf den Kontext des Joinpoints, andere Methoden (wie hier »System.out.println()«) können aufgerufen werden.

Was an der bisherigen Lösung noch etwas unglücklich ist, ist die Vermischung der Ausgabe der Ziehung mit der Ausgabe des Advices zur Protokollierung. Zur Verbesserung wollen wir eine Logging-Lösung wie Log4J[1] einsetzen. Um nicht alles über Aspekte und Advices abhandeln zu müssen, erstellen wir uns eine Log-Klasse, die die Vorarbeiten für Log4J übernimmt:

```
public class Log {

    private static Logger logger = Logger.getLogger(Log.class);

    static {
        init();
    }

    private static void init() {
        try {
            Appender appender = new FileAppender(new TTCCLayout(),
                                        "Lotto.log");
            BasicConfigurator.configure(appender);
        } catch (IOException e) {
            BasicConfigurator.configure();
        }
    }

    public static void info(String msg) {
        logger.info(msg);
    }

}
```

Für uns ist nur die info()-Methode interessant – der Rest dient lediglich zur Initialisierung und Konfigurierung von Log4J[2] (sie sorgt u. a. dafür, dass die Ausgaben in »Lotto.log« abgelegt werden). Mit dieser Methode können wir die System.out.println()-Anweisung ersetzen:

[1] http://jakarta.apache.org/log4j/

[2] Alternativ können Sie auch eine Datei »log4j.properties« erstellen und im Classpath unterbringen.

```
before(int i) : callMixNumbers(i) {
    Log.info(thisJoinPoint + " arg = " + i);
}
```

Beim Start unserer Lottotrommel werden jetzt die normalen Ausgaben nicht mehr mit den Log-Meldungen vermischt, sondern erscheinen in »Lotto.log«. Sollen weitere Punkte protokolliert oder ein anderes Logging-Framework verwendet werden, muss künftig nur die Aspekt-Klasse mit dem obigen Advice und die Log-Klasse abgeändert werden, ohne dass der Rest des Systems angepasst werden muss.

Sollte es Sie stören, dass das Logging auf eine Klasse (»Log.java«) und einen Aspekt (»LogAspect.aj«) aufgeteilt ist, können Sie auch alles in die Aspekt-Klasse packen. Allerdings ist die Unterstützung der IDE für reine Java-Klassen (noch) ein klein wenig besser.

5.3 Kontext

Um den Mischvorgang etwas dynamischer zu gestalten, führen wir eine neue mixNumbers()-Methode ein, die zwei Parameter akzeptiert: eine Untergrenze und eine Obergrenze für die Anzahl der Mischvorgänge:

```
public void mixNumbers(int min, int max) {
    Random random = new Random(System.currentTimeMillis());
    mixNumbers(min + random.nextInt(max - min));
}
```

args()

In Kapitel 4.4.3 auf Seite 96 haben wir den *args*-Pointcut kennen gelernt, den wir bereits weiter oben im Advice verwendet haben, um die Argumente zu protokollieren. Liegen mehrere Argumente vor, auf die zugegriffen werden soll, so wird *args* mit mehreren Parametern aufgerufen:

```
pointcut callMixNumbers2(int min, int max) :
    call(public void mixNumbers(int, int)) && args(min, max);

before(int min, int max) : callMixNumbers2(min, max) {
    Log.info(thisJoinPoint + " min = " + min + ", max = " + max);
}
```

Mit diesem Advice werden die beiden int-Parameter, die beim Aufruf an die mixNumbers()-Methode übergeben werden, mitprotokolliert.

5.3.1 Formale Parameter

Neben *args* gibt es in AspectJ weitere Möglichkeiten, über *formale Parameter* an den Kontext eines Joinpoints zu gelangen. Im Einzelnen sind dies:

- **args()** erlaubt den Zugriff auf die Argumente einer Methode.
- Über **this()** kann auf den Kontext eines Joinpoints zugegriffen werden. Bei statischen Methoden steht dieser Kontext allerdings nicht zur Verfügung.
- Der Zielkontext eines Feldzugriffs, Methoden- oder Konstruktoraufrufs steht über **target()** zur Verfügung.
- Auf den Rückgabewert kann über **returning()** zugegriffen werden, allerdings nur mittels after()-Advice.
- Die ausgelöste Exception erhält man über **throwing()**, ebenfalls nur im after()-Advice.

Einen anderen Zugang zum Kontext verschafft uns der *thisJoinPoint* mit seinen Verwandten *thisJoinPointStaticPart* und *thisEnclosingJoinPointStaticPart*. Dies werden wir uns später noch genauer anschauen. Wenden wir uns wieder dem *args()*-Schlüsselwort zu.

args und die Links-Rechts-Regel

Anzahl links = Anzahl rechts

Wichtig bei der Verwendung von Parametern ist, dass die Anzahl auf der linken Seite mit der auf der rechten Seite übereinstimmt. Vergisst man einen Parameter auf einer Seite wie in diesem Beispiel,

```
before(int min, int max) : callMixNumbers2(min) {
    Log.info(thisJoinPoint + " min = " + min);
}
```

Reihenfolge muss stimmen

so meldet dies der Compiler mit einem »*formal unbound in Pointcut*« oder einer ähnlichen Meldung. Auch die Namen und die Reihenfolge der Parameter müssen auf beiden Seiten übereinstimmen. Bei

```
before(int min, int max) : callMixNumbers2(max, min) {
    Log.info(thisJoinPoint + " min = " + min + ", max = " + max);
}
```

werden Minimum und Maximum vertauscht ausgegeben. Ein weiterer Fallstrick lauert bei der Verwendung des ODER-Operators. So schlägt der folgende Pointcut

```
pointcut callMixNumbers1or2(int min, int max) :
    call(public void mixNumbers(int, int)) && args(min, max)
    || call(public void mixNumbers(int)) && args(min);
```

fehl, da zwar im ersten Teil der Oder-Verknüpfung die Anzahl der Parameter übereinstimmt, aber im zweiten Teil nur der min-Parameter verwendet wird. Der Compiler quittiert dies mit einer »*inconsistent binding*«-Fehlermeldung.

Ähnlichkeiten zu Methoden-Parametern

Parameter-Änderung ohne Außenwirkung

Die Verwendung von formalen Parametern ähnelt denen einer Java-Methode. Parameter können abgefragt, zwischengespeichert, weiterverwendet oder an die nächste Methode weitergereicht werden. Allerdings sind die Parameter nur lokal innerhalb des Advices gültig. Das heißt, wird einem Parameter ein anderer Wert zugewiesen, hat dies außerhalb des Advices keine Auswirkung. Um dies zu überprüfen, ändern wir einen der übergebenen Parameter ab:

```
before(int min, int max) : callMixNumbers2(min, max) {
    Log.info("mixNumbers(..): min = " + min + ", max = " + max);
    min = -1;
}
```

Wird das Programm mit diesem Advice gestartet, verhält es sich genauso wie vorher. Um festzustellen, ob der min-Parameter tatsächlich unverändert in der mixNumbers()-Methode gelandet ist, wollen wir ihn mit einem after()-Advice abfragen:

```
after(int min, int max) : callMixNumbers2(min, max) {
    Log.info("after: min = " + min + ", max = " + max);
}
```

Wenn wir die Log-Datei anschauen, werden wir feststellen, dass die Änderung des min-Parameters durch den ersten Advice auf den after()-Advice keine Auswirkung hat:

```
56 [main] INFO lotto.Log - mixNumbers(..): min = 50, max = 100
79 [main] INFO lotto.Log - after: min = 50, max = 100
80 [main] INFO lotto.Log - mixNumbers(..): min = 50, max = 100
82 [main] INFO lotto.Log - after: min = 50, max = 100
...
```

Jetzt könnten Sie einwenden, dass es vielleicht daran liegt, dass der Parameter zweimal übergeben wurde. Um sicherzugehen, dass innerhalb der mixNumbers()-Methode der übergebene min-Parameter nicht verändert wurde, können wir den Debugger zu Hilfe nehmen oder eine Log-Anweisung in mixNumbers() verwenden (oder über einen Advice einbringen). Wir rufen einfach aus mixNumbers() eine weitere mixNumbers()-Methode auf und lassen die Parameter über einen weiteren Advice ausgeben. Zuerst die abgeänderte und dann die neue mixNumbers()-Methode:

```
public void mixNumbers(int min, int max) {
    mixNumbers(min, max, new Random(System.currentTimeMillis()));
}

public void mixNumbers(int min, int max, Random random) {
    mixNumbers(min + random.nextInt(max - min));
}
```

Die Änderung besteht lediglich darin, dass das Random-Objekt als zusätzlicher Parameter für die nächste mixNumbers()-Methode dient. Pointcut und Advice für die Protokollierung dieses Aufrufs lauten:

```
pointcut callMixNumbers3(int min, int max, Random random) :
    call(public void mixNumbers(int, int, Random))
        && args(min, max, random);

before(int min, int max, Random random) :
        callMixNumbers3(min, max, random) {
    Log.info("mixNumbers(..): min = " + min + ", max = " + max
            + ", " + random);
}
```

Und tatsächlich: Auch hier verrät die Log-Datei, dass der ursprüngliche min-Parameter an die nächste mixNumbers()-Methode weitergereicht wird:

```
52 [main] INFO lotto.Log - mixNumbers(..): min = 50, max = 100
72 [main] INFO lotto.Log - mixNumbers(..): min = 50, max = 100, \
                                          java.util.Random@22c95b
75 [main] INFO lotto.Log - after: min = 50, max = 100
76 [main] INFO lotto.Log - mixNumbers(..): min = 50, max = 100
76 [main] INFO lotto.Log - mixNumbers(..): min = 50, max = 100, \
                                          java.util.Random@1d1acd3
78 [main] INFO lotto.Log - after: min = 50, max = 100
...
```

call-by-value

Halten wir fest: Formale Parameter haben Gemeinsamkeiten mit Methoden-Parametern in Java. Aber eigentlich haben sie mehr Gemeinsamkeiten mit C-Funktionen: Hier werden Parameter immer als Wert (»call-by-value«) übergeben. Änderungen an Parametern haben damit nur lokale Auswirkungen.

5.3.2 Bleibende Wirkung

Mischen impossible

Ich möchte durch einen Advice den Zustand ändern. Wie kann ich das bewerkstelligen? Ganz einfach, ich rufe aus dem Advice die gewünschte Methode auf, die den inneren Zustand des Objekts entsprechend abändert. Angenommen, Sie wollen den Mischvorgang zu Ihren Gunsten etwas abändern. Das können Sie mit folgendem Advice erreichen:

```
after(Trommel trommel) : execution(public void mixNumbers(..))
                         && this(trommel) {
    trommel.sort();
}
```

Nach jeder Ausführung einer mixNumbers()-Methode wird die Lottotrommel anschließend wieder sortiert.[3] Wenn es Ihnen jetzt noch gelingt, die getNumber()-Methode zur Ziehung der Lottozahlen zu beeinflussen...

Das soll an dieser Stelle erst einmal genügen. Ich denke, Sie haben einen Eindruck davon bekommen, wie Sie dauerhaft über Advices auch den Status eines Objekts ändern können. Weitere Manipulationsmöglichkeiten überlasse ich Ihrem Spieltrieb.

5.3.3 Boxing

Boxing = automatische Umwandlung von int & Co

Als Boxing (oder auch Autoboxing) bezeichnet man die automatische Umwandlung primitiver Datentypen (wie int oder boolean) in einen Objekt-Datentyp (wie java.lang.Integer oder java.lang.Boolean). Greifen wir für dieses Thema auf die Konto-Klasse aus Kapitel 3 auf Seite 51 zurück und ergänzen die Attribute um einen Kontoinhaber:

```
public class Konto {

    public double kontostand;
    public String inhaber;

    public Konto(String inhaber) {
        this.inhaber = inhaber;
        kontostand = 0.0;
    }
    ...
```

Zur Erinnerung: Die Attribute sind »public«, um auf sie über Aspekte zugreifen zu können. Damit niemand Unsinn mit dieser Freiheit anstellt, wollen wir diese Attribute überwachen und den Zugriff protokollieren. Dazu haben wir den set-Pointcut kennen gelernt, mit dem dieser Joinpoint angesprochen werden kann:

```
pointcut setAttribute(Object newValue) :
    set(* Konto.*) && args(newValue);

before(Object newValue) : setAttribute(newValue) {
    System.out.println(thisJoinPoint + " set to " + newValue);
}
```

Wenn man sich den setAttribute()-Pointcut anschaut, fällt auf, dass hier eigentlich Objekte angesprochen werden. Wie sieht es aber mit primitiven Datentypen wie dem Kontostand aus, der in unserem Beispiel vom

[3] Nehmen wir der Einfachheit halber an, es gebe (aus welchen Gründen auch immer) eine sort()-Methode in der Trommel-Klasse, die den Vektor mit den Lottokugeln sortiert. In den Beispielen zum Buch finden Sie in der Lotto-Klasse diese sort()-Methode.

Typ »double« ist? Werden die mit diesem Ausschnitt auch protokolliert? Probieren wir es einfach aus:

```
set(String bank.Konto.inhaber) set to Max Mustermann
set(double bank.Konto.kontostand) set to 0.0
set(String bank.Konto.inhaber) set to Max Mustermann
set(double bank.Konto.kontostand) set to 0.0
set(double bank.Konto.kontostand) set to 1.23
...
```

Dies ist ein Ausschnitt aus einem Testlauf. Wie man sieht, werden auch die primitiven Datentypen wie »double« ausgegeben. Intern werden sie von AspectJ vorher in einen Objekt-Datentyp umgewandelt, in diesem Fall in ein java.lang.Double-Objekt. Überprüfen können wir dies, indem wir einfach den Typ im Before-Advice mit ausgeben:

```
before(Object newValue) : setAttribute(newValue) {
    System.out.println(thisJoinPoint + " set to " + newValue
                       + " ("+ newValue.getClass() +")");
}
```

Und tatsächlich, im Fall des Kontostands erscheint der java.lang.Double-Typ in der Ausgabe:

```
set(String bank.Konto.inhaber) set to Max Mustermann (class java.lang.String)
set(double bank.Konto.kontostand) set to 0.0 (class java.lang.Double)
set(String bank.Konto.inhaber) set to Max Mustermann (class java.lang.String)
set(double bank.Konto.kontostand) set to 0.0 (class java.lang.Double)
set(double bank.Konto.kontostand) set to 1.23 (class java.lang.Double)
...
```

Ab Java 5 ist Autoboxing Bestandteil des Sprachumfangs. Dadurch klappt auch folgender Pointcut:

```
before(Double betrag) : set(double Konto.kontostand) && args(betrag) {
    System.out.println("neuer Kontostand: " + betrag);
}
```

Achtung: Unterschied Java 5 – JDK 1.4

Obwohl `betrag` hier als »Double« deklariert ist, funktioniert es durch den Autoboxing-Mechanismus von Java 5 auch in diesem Beispiel, in dem das Kontostand-Attribut vom einfachen Datentyp »double« ist (man beachte den Unterschied von Groß- und Kleinschreibung). Wenn Sie allerdings in den Projekteinstellungen den Java-Compiler auf »1.4« zurückdrehen, werden Sie feststellen, dass dieser Pointcut ins Leere greift – hier findet kein Autoboxing seitens des Java-Compilers statt. Mischen Sie also nicht unnötig primitive und zusammengesetzte Datentypen, sondern versuchen Sie immer genau den Datentyp anzugeben, den Sie treffen wollen.

5.3.4 Zugriff über Reflexion

Zugriff über thisJoinPoint()

Um auf Argumente zuzugreifen, haben wir uns des *args*-Pointcuts bedient. Eine weitere Möglichkeit liefert uns *thisJoinPoint*:

```
before() : set(* Konto.*) {
    Object[] values = thisJoinPoint.getArgs();
    System.out.println(thisJoinPoint + " set to " + values[0]);
}
```

Die getArgs()-Methode gibt ein Objekt-Array der Argumente zurück, die diesen Joinpoint ausgelöst haben. Da ein set-Pointcut nur ein Argument haben kann, können wir direkt auf dieses erste Element über `values[0]` zugreifen.

Neben dem Zugriff auf den Kontext über spezielle AspectJ-Konstrukte wie *args*, *this* oder *target* stehen in einem Advice folgende Objekte zur Verfügung:

- **`thisJoinPoint`**
  Mit den Methoden *getArgs()*, *getThis()* und *getTarget()* kann auf die gleichen Informationen zugegriffen werden, die auch mit formalen Variablen erreichbar sind. Darüber hinaus können über die Methoden *getSignature()*, *getSourceLocation()* oder *getKind()* weitere Informationen zum Joinpoint abgefragt werden.
- **`thisJoinPointStaticPart`**
  stellt zwar weniger Informationen (Signatur, Auftreten, Art des Joinpoints) zur Verfügung, benötigt dafür aber keinen zusätzlichen Speicher, wenn er verwendet wird.
- **`thisEnclosingJoinPointStaticPart`**
  In der Verwendung ähnlich wie thisJoinPointStaticPart adressiert er den Joinpoint, der den aktuellen Joinpoint umgibt.

Der Zugriff auf den Kontext des Joinpoints findet mit Hilfe von Reflexion statt.

Wann ist es sinnvoller, diese Objekte zu verwenden, und wann der Einsatz von formalen Parametern?

Plädoyer für *thisJoinPoint* & Co

Mehr Möglichkeiten

Bei einer variablen Anzahl von Argumenten haben Sie es mit *thisJoinPoint* einfacher, die einzelnen Argumente aufzulisten. Auch stehen Ihnen über *thisJoinPoint* & Co weitere Informationen zur Verfügung, an die Sie mit den formalen Parametern nicht herankommen. So erhalten Sie über die getSourceLocation()-Methoden u. a. auch die Zeilennummer, die vor allem in der Entwicklungsphase wertvolle Zusatzinformation liefern kann.

Kostet Laufzeit

Auf der anderen Seite findet der Zugriff vorwiegend über Reflexion statt und ist damit langsamer und Ressourcen-intensiver als der Zugriff über formale Parameter. Setzen Sie thisJoinPoint und seine Verwandten nur wohldosiert ein und vermeiden Sie ihn bei breit gestreuten Pointcuts – Sie handeln sich sonst unnötigen Overhead ein.

Plädoyer für formale Parameter

Eine variable Anzahl von Argumenten bedeutet nicht, dass Sie nicht formale Parameter einsetzen können. Wenn Sie beispielsweise nur am ersten und letzten Parameter interessiert sind, können Sie ihn folgendermaßen adressieren:

```
// nur 1. und letzten Parameter protokollieren
before(Object first, Object last) :
        call(public void *(..))
        && args(first, *, last) {
    System.out.println("ARGS = " + first + "..."
            + last + " for " + thisJoinPoint);
}
```

Formale Parameter sind sicherer und lesbarer.

Wollen Sie hingegen wirklich eine variable Anzahl von Parametern abarbeiten, haben Sie mit formalen Parametern schlechte Karten. Trotzdem sollten Sie, wenn immer es möglich ist, formale Parameter vorziehen, schon aus Performance-Gründen. Durch die Verwendung von *args*, *this* oder *target* dokumentieren Sie, worauf Sie zugreifen wollen und machen damit Ihre Aspekte für Außenstehende (und für sich selbst) leichter lesbar. Der Zugriff über Reflexion hingegen verschleiert Ihre Absicht – ein Zugriff auf ein Argument muss erst »herausgelesen« werden. Auch nehmen Sie damit dem Compiler die Gelegenheit, den Zugriff überprüfen zu können, da ihm die Typ-Information fehlt:

```
after(): execution(public void Konto.abheben(double) {
    Konto konto = (Konto) thisJoinPoint.getThis();
    Double betrag = (Double) thisJoinPoint.getArgs()[0];
    double gebuehr = betrag.doubleValue() * 0.005;
    System.out.println("Betrag: " + betrag + "\tGebuehr: " + gebuehr);
    konto.kontostand -= gebuehr;
}
```

Überlegen Sie, was dieser Code macht, und vergleichen Sie ihn mit folgendem Code:

```
after(double betrag, Konto konto):
        execution(public void Konto.abheben(double))
        && args(betrag)
        && this(konto) {
    // kleinen Obolus fuer die Bank einbehalten
    double gebuehr = betrag * 0.005;
```

```
    System.out.println("Betrag: " + betrag + "\tGebuehr2: " + gebuehr);
    konto.kontostand -= gebuehr;
}
```

Bei der zweiten Lösung mit formalen Parametern ist sofort ersichtlich, dass im Advice auf ein Argument zugegriffen werden soll (sonst hätte man sich den Einsatz von *args* sparen können) und auf das Objekt selbst (erkennbar an *this*). Auch für den Compiler ist es leichter verdaulich, da er die nötigen Typ-Informationen hat, um den Einsatz des Parameters »betrag« überprüfen und notfalls einen Compiler-Fehler ausgeben zu können. Auch der unschöne Cast[4] aus der ersten Lösung entfällt. Dort kann es auch beim Zugriff über *thisJoinPoint* zu Laufzeitfehlern kommen, wenn der falsche Typ bzw. Cast verwendet wird.

Haben Sie inzwischen herausbekommen, was dieser Advice macht? Für jede Abhebung wird anhand des abgehobenen Betrags eine Gebühr von 0,5% fällig, die vom Kontostand abgebucht wird.

5.3.5 Advices und Exceptions

Auch Advices dürfen Exceptions werfen!

Bleiben wir noch ein Weilchen beim Konto-Beispiel. Sämtliche Überweisungen sollen jetzt zusätzlich in einer Datei mitprotokolliert werden. Dabei kann natürlich aus verschiedenen Gründen (Netzwerk nicht verfügbar, Service-Pack legt Platte lahm, ...) eine IOException ausgelöst werden. Genauso wie in Java die »Checked« Exceptions[5] im Kopf einer Methode deklariert werden müssen, ist dies auch für Advices nötig, damit sie eine Exception auslösen dürfen:[6]

```
after(double betrag, Konto empfaenger) throws IOException :
        execution(public void Konto.ueberweisen(double, Konto))
        && args(betrag, empfaenger) {
    FileWriter writer = new FileWriter("ueberweisung.log", true);
    writer.write("Ueberweisung " + betrag + " auf " + empfaenger + "\n");
    writer.close();
}
```

Ohne diese throws-Anweisung meldet der Compiler eine *»unhandled exception type IOException«* und mit ihr eine *»can't throw checked exception java.io.IOException at this point«*. Warum? Weil die ueberweisen()-Methode, in der dieser Advice eingewebt wird, keine solche Exception werfen darf. Was können Sie mit dieser Exception an dieser Stelle machen?

[4] Es gibt keine schönen Casts!

[5] Das sind die Exceptions, die nicht von »RuntimeException« abgeleitet sind.

[6] Ein »`import java.io.*`« am Anfang des Aspekts ist noch nötig für diesen Advice.

- Sie handeln die Exception im Advice selbst ab.
- Sie verpacken sie in eine »erlaubte« Exception (z. B. eine RuntimeException)
- Sie wenden Exception-Softening (s. Kapitel 6.10.3 auf Seite 195) an.

```
after(double betrag, Konto empfaenger) :
        execution(public void Konto.ueberweisen(double, Konto))
        && args(betrag, empfaenger) {
    String msg = "Ueberweisung " + betrag + " auf " + empfaenger;
    try {
        FileWriter writer = new FileWriter("ueberweisung.log", true);
        writer.write(msg + "\n");
        writer.close();
    } catch (IOException e) {
        System.out.println("msg");
    }
}
```

Hier wurde die Exception im Advice selber abgehandelt, indem die Ausgabe auf dem Bildschirm landet.

Fassen wir zusammen:

- Exceptions (außer den RuntimeExceptions), die in einem Advice geworfen werden können, müssen deklariert werden (analog wie in Java).
- Es dürfen nur die Exceptions geworfen werden, die für den entsprechenden Joinpoint zulässig sind.

Was bedeutet dies für die Exceptions in den einzelnen Advices?

- *call-* und *execution-*Joinpoints dürfen nur solche Exceptions werfen, die beim Joinpoint deklariert sind.[7]
- *(pre)initialization-*Joinpoints dürfen nur die Exceptions werfen, die in *allen* Konstruktoren deklariert wurden.
- *staticinitialization-*Joinpoints können keine »checked« Exceptions werfen.
- *set-* und *get-*Joinpoints dürfen ebenfalls keine checked Exceptions werfen.
- Handler-Joinpoints können nur die Exceptions werfen, die auch der Exception-Handler werfen kann.
  Beispiel: Angenommen, im angesprochenen Joinpoint darf eine IOException geworfen werden. Dann ist der folgende Codeausschnitt legal:

[7] und natürlich die RuntimeExceptions – die dürfen immer geworfen werden

```
    catch (Exception e) {
        if (isNetworkNotAvailable()) {
            throw new IOException("Network unavailable");
        }
    }
```

5.4 Die verschiedenen Advice-Typen

Bis jetzt haben wir die verschiedenen Arten von Advices nur oberflächlich kennen gelernt:

- den *before()*-Advice, der *vor* dem Joinpoint zur Ausführung kommt,
- den *after()*-Advice für die Ausführung *danach*,
- den *around()*-Advice, der anstatt des Joinpoints zum Tragen kommt.

Wann man welchen Advice verwendet, hängt von der Absicht ab, die man damit verfolgt. Möchte man bei einer Überweisung prüfen, ob das Konto gedeckt ist, ist es *nach* der Ausführung zu spät. Möchte man die erfolgreiche Überweisung protokollieren, ist vermutlich das Ende dieser Transaktion der geeignete Augenblick.

Damit wollen wir das Konto-Beispiel wieder verlassen und uns dem Lotto-Beispiel vom Anfang des Kapitels zuwenden. Wir werden die Trommel-Klasse um eine Klasse »Ziehung« ergänzen, die die eigentliche Ziehung und spätere Auswertung softwaretechnisch nachbilden soll:

```
public class Ziehung {

    int[] zahl = { 1, 2, 3, 4, 5, 6 };

    /**
     * Die Ziehung der Lottozahlen beginnt...
     */
    public void start() {
        start(new Trommel());
    }

    public void start(Trommel trommel) {
        for (int i = 0; i < 6; i++) {
            zahl[i] = zieheZahl(trommel);
        }
    }
```

```
    private int zieheZahl(Trommel trommel) {
        return trommel.getNumber().intValue();
    }

}
```

Die Ziehung-Klasse verwendet dabei die bereits vorgestellte Trommel-Klasse (s. Abschnitt 5.1 auf Seite 117), um sechs Richtige zu ermitteln.

5.4.1 Before-Advice

Syntax

before(*formale Parameter*) [**throws** *Exception-Liste*] **:** *Pointcut* {
Body
}

Ausführung vor *dem Joinpoint*

Der Before-Advice ist der einfachste von allen, weswegen er auch in den meisten Beispielen bis jetzt neben dem after()-Advice zum Einsatz kam. Er wird dann verwendet, wenn noch irgendetwas vor dem Joinpoint passieren soll oder Voraussetzungen überprüft werden müssen:

```
pointcut executeZieheZahl(Trommel t) :
    execution(private int Ziehung.zieheZahl(Trommel))&& args(t);

before(Trommel t) : executeZieheZahl(t) {
    t.mixNumbers();
}
```

Bevor eine Kugel gezogen werden soll, wird noch schnell die Trommel mit Hilfe des Before-Advices durchmischt. Natürlich hätte man diese Anweisung ganz konventionell direkt in die zieheZahl()-Methode einfügen können, aber die Auslagerung in einen Aspekt hat den Vorteil, dass man so die Mischvorgänge zentral zusammenfassen kann, um sie später evtl. durch eine andere Mischmethode einfacher austauschen zu können.[8]

Preconditions

Design by Contract

Der Before-Advice eignet sich hervorragend zum Einstreuen von Preconditions (Vorbedingungen), wie sie von Design by Contract (kurz: DbC) vorgeschlagen werden. Dort gehen die aufrufende und aufgerufene Methode einen »Vertrag« ein: Die aufrufende Methode garantiert die Einhaltung der Preconditions, die aufgerufene Methode sichert im

[8] Ob diese Trennung hier gerechtfertigt ist, ist fraglich – wahrscheinlich eher nicht. Es ist eben nur ein Beispiel.

Gegenzug die vereinbarten Postconditions (Nachbedingungen) zu.[9] Beschäftigen wir uns zuerst mit der Precondition für den Start der Ziehung:

```
pointcut executeZiehungStart(Trommel t) :
    execution(public void Ziehung.start(Trommel)) && args(t);

/**
 * Precondition fuer den Start der Ziehung
 */
before(Trommel t) : executeZiehungStart(t) {
    if (t.getAnzahlKugelnInside() != 49) {
        throw new IllegalArgumentException("keine 49 Kugeln in Trommel");
    }
}
```

Wenn die Precondition verletzt ist, d. h., wenn keine 49 Kugeln in der übergebenen Lostrommel sind, wird eine IllegalArgumentException geworfen. Leider kann hier nur eine RuntimeException oder davon abgeleitete Exception (wie die IllegalArgumentException) geworfen werden, da die start()-Methode keine Exception in ihrer Signatur trägt.

Vielleicht werden Sie sich bei diesem Ausschnitt fragen, wo denn die Methode getAnzahlKugelnInside() herkommt. Hier ist sie:

```
public int getAnzahlKugelnInside() {
    return balls.size();
}
```

Damit hat der beaufsichtigende Notar die Chance, diesen Teil des »Vertrages« zu überprüfen. Eine andere Möglichkeit, mit Vor- und Nachbedingungen umzugehen, liefert ab JDK 1.4 das assert-Schlüsselwort:

Ab JDK 1.4: assert

```
before(Trommel t) : execution(public void Ziehung.start(Trommel))
                    && args(t) {
    assert(t.getAnzahlKugelnInside() == 49);
}
```

Allerdings sind diese Assertions (Annahmen) nicht automatisch aktiviert. Die Überwachung der assert-Anweisung muss beim Aufruf des Programms über die Option »*-enableassertions*« (oder kurz: »*-ea*«) eingeschaltet werden:[10]

```
java -enableassertions ...
```

[9] Näheres zu DbC siehe z. B. `http://www.eiffel.com/doc/manuals/technology/contract/ariane/page.html`.

[10] Wenn Sie mit Eclipse arbeiten, müssen Sie bei den Java-Compiler-Einstellungen unter »Compliance and Classfiles« die Source- und »generated class files«-Kompatibilität mindestens auf »1.4« setzen; s. Anhang C auf Seite 389).

Sollte dann die Vorbedingung während eines Testlaufs tatsächlich verletzt sein, erhält man an dieser Stelle einen *java.lang.AssertionError* mit der Ausgabe des angegebenen Textes. Natürlich kann man diese Asserts auch fest im Code ohne Aspekte verankern, aber manchmal will man die Vorbedingungen außerhalb des eigentlichen Codes dokumentieren.

return

Kann ein Before-Advice überhaupt einen return-Wert zurückgeben, wie es die Überschrift suggeriert? Die Antwort ist *nein*. Es macht auch keinen Sinn, da dieser Advice ja am Anfang einer Methode bzw. eines Joinpoints zum Tragen kommt. Trotzdem darf ein *return* in einem Before-Advice auftauchen, wie aus dem folgenden Beispiel ersichtlich ist:

```
before(Trommel t) : executeZieheZahl(t) {
    if (t.isMixed()) {
        return;     // Mischen nicht nötig
    }
    System.out.println("Mix it, Baby...");
    t.mixNumbers();
}
```

return = Advice beenden

Dies ist das Beispiel vom Anfang dieses Abschnitts, leicht abgewandelt. Um die virtuelle Trommelmechanik zu schonen, soll das Mischen nur dann erfolgen, wenn die Kugeln noch sortiert sind. Hat eine Durchmischung bereits stattgefunden, steigt dieser Advice mit der *return*-Anweisung aus. *return* hat nur die Aufgabe, den Kontrollfluss (und damit auch den Advice) an dieser Stelle zu verlassen. Wenn Sie wollen, können Sie sich den Before-Advice auch als void-Methode vorstellen.

Das Gleiche gilt für den After-Advice, der im nächsten Abschnitt vorgestellt wird. Auch der After-Advice verhält sich wie eine void-Methode – mit *return* können Sie ihn verlassen.

5.4.2 After-Advice

Syntax

after(*formale Parameter*) [**throws** *Exception-Liste*] **:** *Pointcut* {
Body
}

oder

after(*formale Parameter*) **returning** [(*formaler Parameter*)] [**throws** *Exception-Liste*] **:** *Pointcut* {
Body
}

oder

after(*formale Parameter*) **throwing** [(*formaler Parameter*)] [**throws** *Exception-Liste*] **:** *Pointcut* {
Body
}

Der After-Advice ist schon etwas umfangreicher als der Before-Advice. Zur Einstimmung beginnen wir mit dem einfachsten aller After-Advices, dem unqualifizierten Advice.

Unqualifizierter Advice

Syntax

after(*formale Parameter*) [**throws** *Exception-Liste*] **:** *Pointcut* {
Body
}

Um später eine eventuelle Manipulation aufdecken zu können, soll die Ziehung der Lottozahlen in einer Log-Datei mitgeschrieben werden. Dazu werden nach dem Start der Ziehung die ausgelosten Zahlen über einen after()-Advice protokolliert:

```
pointcut executeZiehungStart(Ziehung z) :
    execution(public void lotto.Ziehung.start())
    && this(z);

after(Ziehung z) : executeZiehungStart(z) {
    Log.info("gezogene Zahlen: " + z);
}
```

Zur Protokollierung wird neben einer einfachen Log-Klasse[11] die *to-String()*-Methode verwendet. Da die Default-Implementierung von to-String() nicht sonderlich informativ ist, ändern wir dies kurzerhand:[12]

```
public String toString() {
    return "" + zahl[0] + " " + zahl[1] + " " + zahl[2] + " "
        + zahl[3] + " " + zahl[4] + " " + zahl[5];
}
```

Als Nächstes möchten wir eine Datei, in der sämtliche Lottoscheine abgespeichert sind, auswerten und die Gewinner ermitteln. Dazu dient die Methode »getWinners(String)«, die die Liste der Gewinner ausspuckt:

```
public List getWinners(String filename) throws IOException {
    Reader reader = new FileReader(filename);
    List winners = getWinners(reader);
    reader.close();
    return winners;
}
```

[11] Die Implementierung finden Sie auf Seite 120 in Kapitel 5.2.

[12] Es empfiehlt sich generell, die toString()-Methode zu überschreiben – es erleichtert das Debuggen und die Ausgabe von Log-Meldungen.

Die genaue Implementierung der getWinners(Reader)-Methode soll hier nicht verraten werden.[13] Wir wollen uns lieber darauf konzentrieren, wie wir den Namen der Datei protokollieren können, um später notfalls rekonstruieren zu können, aus welcher Datei die Gewinner ermittelt wurden:[14]

```
pointcut executeGetWinners(String s) :
    execution(public List Ziehung.getWinners(String))
    && args(s);

after(String s) : executeGetWinners(s) {
    Log.info("Gewinner wurden aus <" + s + "> ermittelt.");
}
```

Wenn man sich die getWinners()-Methode nochmals genauer anschaut, wird man feststellen, dass hier Exceptions auftreten können. Einmal durch den FileReader-Konstruktor, aber auch durch die getWinners(Reader)-Methode. Die spannende Frage ist nun: Was wird aus unserem After-Advice, wenn eine Exception auftaucht? Wird er oder wird er nicht ausgeführt? Und wo wird er eingefügt?

Dieser Advice wird immer ausgeführt.

Die Antwort: Der unqualifizierte After-Advice wird auf jeden Fall ausgeführt. Stellen Sie sich dazu einfach vor, der After-Advice wird in einen finally-Abschnitt eingebettet:

```
try {
    // Ausführung des Joinpoints
} finally {
    // Ausführung des After-Advices
}
```

Damit ist der After-Advice prädestiniert für Aufräumarbeiten, die auf jeden Fall ausgeführt werden sollen. Dies kann das Schließen von Datenbank-Verbindungen beinhalten, die Freigabe von Ressourcen oder das Löschen nicht mehr benötigter Objekte bedeuten.[15]

```
pointcut executeGetWinners2(Reader r) :
    execution(public List Ziehung.getWinners(Reader)
        throws IOException)
    && args(r);
```

[13] Für Neugierige: Sie finden sie unter den Beispielen zum Buch.

[14] Wenn bei Ihnen die Warnung »*does not match because declaring type is java.lang.Object*« kommt, haben Sie vermutlich »`import java.util.*` vergessen – damit kennt der Compiler den *List*-Typ nicht.

[15] Auch wenn in Java der Garbage Collector für die Beseitigung des (Objekt-)Mülls zuständig ist, können Sie ihm helfen, indem Sie nicht benötigte Objekte auf *null* setzen. Solange nämlich noch eine Referenz auf ein Objekt existiert, wird die VM dieses Objekt weiterhin im Speicher halten.

```
after(Reader r) : executeGetWinners2(r) {
    try {
        r.close();
    } catch (IOException ignored) {}
}
```

Dieser Aspekt schließt die übergebene Datei, wenn getWinners(Reader) beendet wird.[16] Auch wenn der close()-Aufruf nicht unbedingt notwendig wäre, da spätestens am Ende der Anwendung die offene Datei geschlossen wird, zeugt es doch von gutem Stil. Der try-catch-Block ist hier notwendig, da die close()-Methode selbst wieder eine IOException werfen kann.

Postconditions

Wenn der Before-Advice für Preconditions geeignet ist, drängt sich die Vermutung auf, dass der After-Advice eigentlich prädestiniert für Postconditions sein müsste. Und das ist er auch. So wie vorhin in Kapitel 5.4.1 auf Seite 132 vor dem Ziehungsstart abgesichert wurde, dass auch wirklich 49 Kugeln in der Trommel sind, dürfen nach der Ziehung von sechs Zahlen (ohne Zusatzzahl) nur noch 43 Kugeln in der Trommel liegen:

```
/**
 * Postcondition
 */
after(Trommel t) : executeZiehungStart(t) {
    assert(t.getAnzahlKugelnInside() == 43);
}
```

After Returning

Syntax

after(*formale Parameter* **) returning** [**throws** *Exception-Liste*] **:** *Pointcut* {
Body
}

Dieser Advice wird bei Exception nicht ausgeführt.

In vielen Fällen ist beim unqualifizierten Advice dessen Ausführung nach dem Auftreten einer Exception hinderlich. Dies berücksichtigt der After-Returning-Advice:

```
after(String s) returning: executeGetWinners(s) {
    Log.info("Gewinner wurden aus <" + s + "> erfolgreich ermittelt.");
}
```

[16] Anmerkung: Die Throws-Anweisung im Execution-Pointcut ist nicht unbedingt nötig, gibt aber für den Leser einen wertvollen Hinweis – nämlich, dass bei dieser Methode eine IOException auftreten kann.

Das entspricht dem Beispiel von oben (s. Seite 136) mit dem Unterschied, dass diese Log-Meldung nur dann ausgegeben wird, wenn *keine* Exception auftritt.

Auswertung des Rückgabewerts

Syntax

after(*formale Parameter*) **returning** (*formaler Parameter*) [**throws** *Exception-Liste*] : *Pointcut* {
  Body
}

Zugriff auf den Rückgabewert

In der nächsten Ausbaustufe sollen aus der zurückgegebenen Liste die Namen der Gewinner im Protokoll erscheinen. Für den Zugriff auf den Rückgabewert einer Methode steht das Schlüsselwort *returning* zur Verfügung:[17]

```
after(String s) returning(List l): executeGetWinners(s) {
    Log.info("Liste der Gewinner: ");
    for (Iterator i = l.iterator(); i.hasNext();) {
        Schein schein = (Schein) i.next();
        Log.info("xxxxxx  " + schein.getSpieler());
    }
}
```

Der Typ des Rückgabewerts (hier: *List*) muss natürlich mit dem Rückgabewert der angesprochenen Methode übereinstimmen bzw. darf auch einer der Oberklassen sein. Ist dies nicht der Fall, lässt sich zwar der Advice übersetzen, aber er kommt nicht zur Ausführung.

Hat man einen Pointcut, der mehrere Joinpoints abdeckt, muss der angegebene Typ in *returning* die gemeinsame Oberklasse sein. Im Zweifelsfall setzt man hier *Object* ein. Dies schließt auch die Basis-Typen wie *int* oder *char* ein, da diese von AspectJ automatisch in ihre Klassen-Verwandten *Integer* bzw. *Character* umgewandelt werden (»Autoboxing«). So protokolliert der folgende Advice

```
after() returning(Object o) : execution(public * Trommel.*(..)) {
    Log.info(thisJoinPoint +  " returns " + o);
}
```

sämtliche Rückgabewerte aller öffentlichen Trommel-Methoden, auch die, die einen *int* oder anderen Basistyp zurückgeben. Bei Void-Methoden wie der mixNumbers()-Methode erscheint als Rückgabewert »*null*«.

[17] Wenn Sie an der Schein-Klasse interessiert sind, werfen Sie einen Blick auf die Beispiel-Sourcen zum Buch.

After Throwing

Syntax

after(*formale Parameter*) **throwing** [(*formaler Parameter*)] [**throws** *Exception-Liste*] **:** *Pointcut* {
Body
}

After Throwing wird nur bei Exceptions ausgeführt.

Als nächstes Beispiel wollen wir die Exceptions, die in der getWinners()-Methode auftreten können, automatisch protokollieren lassen. Dazu bedienen wir uns des After-Throwing-Advices:

```
after(String s) throwing : executeGetWinners(s) {
    Log.error("Fehler in " + thisJoinPoint);
    Log.error(s + " konnte nicht gelesen werden!");
}
```

Tritt jetzt eine Exception wie das Fehlen der einzulesenden Datei auf, erscheint in der Log-Datei tatsächlich ein entsprechender Hinweis:

```
...
Fehler in execution(List lotto.Ziehung.getWinners(String))
Lotto.schein konnte nicht gelesen werden!
```

After Throwing wird immer bei Exceptions ausgeführt.

Dabei ist es unerheblich, ob die überwachte Methode getWinners() die Exception in einem try-catch-Block selber behandelt oder nur weiterreicht: Der After-Throwing-Advice wird auf jeden Fall ausgeführt. Auf diese Weise lassen sich sehr einfach sämtliche Exceptions protokollieren. Auch leere try-catch-Blöcke à la

```
try {
    Reader reader = new FileReader(filename);
    ...
} catch (FileNotFoundException e) { /* leerer catch-Block */ }
```

verlieren damit ihre Brisanz, da dieser Block durch den After-Throwing-Advice mit Code gefüllt wird.

Einen Haken hat unsere Lösung allerdings noch. Es wird zwar protokolliert, wenn eine Exception auftritt, aber es wurde vergessen, die Exception selbst mit auszugeben. Dies wollen wir nachholen und ergänzen den After-Throwing-Advice um die erwartete Exception:

```
after(String s) throwing (IOException e): executeGetWinners(s) {
    Log.error("Fehler in " + thisJoinPoint, e);
    Log.error(s + " konnte nicht gelesen werden!");
}
```

Auch hier gilt: Ist der Joinpoint allgemeiner gehalten, so dass alle möglichen Arten von Exceptions auftauchen können, nimmt man die gemeinsame Oberklasse aller Exceptions. Passen Sie aber auf, dass Sie die richtige Klasse erwischen, da AspectJ keinen Fehler bei der Übersetzung ausgibt.

```
after(String s) throwing (ParseException e):
        executeGetWinners(s) {
    ...
}
```

Auf richtige Exception achten!

Dieser Advice wird vom AspectJ-Compiler anstandslos übersetzt. Nur leider kommt er nicht zum Einsatz, da »ParseException« die falsche Oberklasse ist. Meist wird man bei »Exception« oder »Throwable« als kleinste gemeinsame Oberklasse landen, vor allem dann, wenn der Pointcut etwas allgemeiner gehalten ist.

5.4.3 Around-Advice

Syntax

> **around**(*formale Parameter*) [**throws** *Exception-Liste*] **:** *Pointcut* {
> Body
> }

Totale Kontrolle mit around()

Der Around-Advice ist der mächtigste aller Advices, zugleich aber auch der komplexeste. Während Sie mit dem Before- und After-Advice an einem Joinpoint Code hinzufügen können, haben Sie mit dem Around-Advice die totale Kontrolle über diesen Joinpoint. Sie können nicht nur Code hinzufügen, sondern ihn komplett ersetzen.

Fangen wir mit einem einfachen Beispiel an, um die Möglichkeiten des Around-Advices kennen zu lernen. In der Klasse »Ziehung« soll zu Beginn eine Willkommensmeldung ausgegeben werden. Um sie flexibel zu halten, kann sie als System-Property »welcome.message« (z. B. über die Option »–D« des Java-Interpreters) gesetzt werden. Der Inhalt dieser Property wird in der main()-Methode ausgegeben:

```
public static void main(String[] args) {
    System.out.println(System.getProperty("welcome.message"));
    ...
}
```

Was passiert nun, wenn die Property »welcome.message« nicht gesetzt ist? Dann ist der Rückgabewert von System.getProperty() *null* und es erscheint

```
null
```

Wir basteln uns einen Property-Wrapper.

an dieser Stelle. Wäre es nicht schöner, wenn in diesem Fall ein leerer String von System.getProperty() zurückgeliefert würde? Kein Problem – wir ersetzen den Aufruf mit Hilfe des Around-Advices:

```
pointcut getSystemProperty(String key) :
    call (String System.getProperty(String)) && args(key);

String around(String key) : getSystemProperty(key)
```

```
        && !within(PropertyAspect) {
    String value = System.getProperty(key);
    if (value == null) {
        return "";
    } else {
        return value;
    }
}
```

Jeder Aufruf von System.getProperty() wird durch diesen Advice ersetzt. Wenn der Rückgabewert dieser Methode *null* ist, wird er nicht weitergereicht, sondern durch einen leeren String ("") ersetzt. Die Ausfilterung durch *!within(PropertyAspect)* (»PropertyAspect« ist der Name des Aspektes, in dem dieser Advice platziert ist) ist nötig, weil sonst der Advice auf sich selbst auch zutreffen würde. Fehlt diese Angabe, landet das Programm an dieser Stelle in einer Endlosschleife, aus der es sich mit einem *StackOverflowError* befreit.

Allerdings kann so ein Advice manchmal auch irritierend (und frustrierend) sein. Betrachten Sie dazu folgendes Beispiel:

```
String msg = System.getProperty("welcome.message");
if (msg == null) {
    System.out.println("Willkommen zur Lotto-Ziehung");
} else {
    System.out.println(msg);
}
```

Selber ausgetrickst!

Wenn man den Code ohne Kenntnis des vorigen Advices liest, erwartet man, dass »Willkommen zur Lotto-Ziehung« ausgegeben wird, da keine Property »welcome.message« gesetzt ist. Stattdessen erscheint aber nichts (bzw. genauer: ein Leerstring) auf dem Bildschirm. Mit dem AspectJ-Browser oder dem AspectJ-Plugin, die wir in Kapitel 9 auf Seite 261 kennen lernen werden, existieren einige rudimentäre Tools, die bei der Suche nach dem unbekannten Advice helfen. Als Vorgeschmack sehen Sie in Abbildung 5.2 auf der nächsten Seite, wie über den AspectVisualizer die einzelnen Aspekte sichtbar gemacht werden können.

Proceed()

proceed() = Aufruf des Joinpoints

Was ist aber mit dem eigentlichen Joinpoint? Kommt er wie beim Before- oder After-Advice auch zum Zug? Im vorigen Beispiel ist er ja nur ersetzt worden. Wie Sie aus der Überschrift richtig vermuten, ist es die *proceed()*-Anweisung, mit der der eigentliche Joinpoint aufgerufen wird. Damit lässt sich der Around-Advice um den System.getProperty()-Aufruf auch folgendermaßen schreiben:

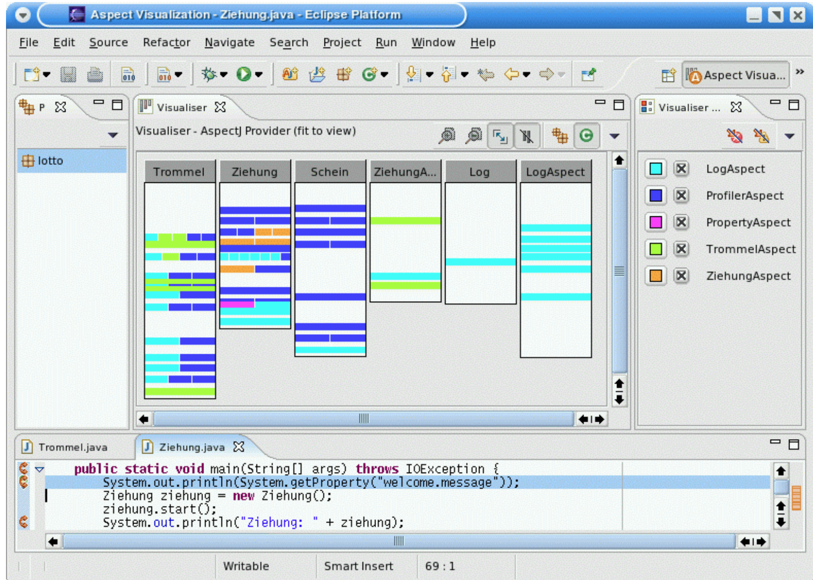

Abbildung 5.2
AspectVisualizer in Eclipse

```
String around() :  call (String System.getProperty(String)) {
    String value = proceed();
    if (value == null) {
        return "";
    } else {
        return value;
    }
}
```

proceed() beinhaltet bereits die Parameter.

Was gegenüber der vorigen Lösung auffällt, ist das Fehlen der Parameter. Tauchte dort noch »*String key*« in der Argument-Liste auf, ist dies jetzt nicht mehr nötig: »*proceed()*« steht für den kompletten Joinpoint mitsamt seinen Parametern. Auch die Absicherung durch die within-Anweisung kann entfallen, da der Advice die proceed()-Ausführung selber nicht instrumentiert. Somit ist auch keine Rekursion von diesem Around-Advice zu befürchten.

proceed() und return

Typ des Around-Advices = Typ des Rückgabewerts

Im Gegensatz zum Before- und After-Advice hat der Around-Advice einen Typ, der den Rückgabewert kennzeichnet. Es ist der Rückgabewert der proceed()-Anweisung und muss dementsprechend auch vom Around-Advice zurückgegeben werden. Fehlt die return-Anweisung, wird sich AspectJ weigern, diesen Advice zu übersetzen. Im obigen Beispiel ist der Typ des Rückgabewerts klar: System.getProperty() liefert einen String zurück, also muss auch der Around-Advice einen String

zurückgeben. Was aber, wenn man seinen Pointcut allgemeiner gehalten hat:

```
/**
 * ermittle Zeitverhalten
 */
Object around() : execution(public * lotto..*(..)) {
    long t = System.currentTimeMillis();
    Object toreturn = proceed();
    t = System.currentTimeMillis() - t;
    Log.profile("time: " + t + " ms for " + thisJoinPoint);
    return toreturn;
}
```

Dieser Advice stoppt für jede öffentliche Methode aus dem »lotto«-Paket die Zeit vor und nach der Ausführung und protokolliert sie über die Log.profile()-Methode. Danach wird der zwischengespeicherte Rückgabewert zurückgegeben. Da der Pointcut auf eine Reihe von Methoden zutrifft, bleibt als gemeinsame Oberklasse nur »Object« übrig. Über den Autoboxing-Mechanismus werden damit auch die primitiven Datentypen wie *int* oder *char* erfasst und entsprechend in ihre Gegenstücke *Integer* bzw. *Character* von AspectJ automatisch konvertiert.

Dem erfahrenen Java-Guru wird vielleicht aufgefallen sein, dass sich für die Zeit-Ausgabe auch der finally-Block anbietet, der garantiert am Ende einer Methode aufgerufen wird, unabhängig davon, ob eine Exception auftritt oder nicht. Damit wird die Zeit auf jeden Fall protokolliert. Auch gewinnt der Advice dadurch etwas an Übersichtlichkeit:

```
Object around() : execution(public * lotto..*(..)) {
    long t = System.currentTimeMillis();
    try {
        return proceed();
    } finally {
        t = System.currentTimeMillis() - t;
        Log.profile("time: " + t + " ms for " + thisJoinPoint);
    }
}
```

Noch ein Wort zum Rückgabewert: AspectJ konvertiert zwar den Rückgabewert automatisch in den passenden Datentyp, überprüft beim Kompilieren aber nicht, ob der richtige Typ zurückgegeben wird. So wird es beim folgenden Pointcut

```
Object around() :  call (String System.getProperty(String)) {
    Object value = proceed();
    if (value == null) {
        // ACHTUNG: ClassCastException im Anmarsch...
        return new Character(' ');
```

```
    } else {
        return value;
    }
}
```

bei der ersten Return-Anweisung eine ClassCastException geben, wenn das Character-Objekt in einen String umgewandelt wird. Wenn Sie das Beispiel mit dem obigen Beispiel vom Anfang dieses Unterkapitels 5.4.3 vergleichen, werden Sie feststellen, dass der Typ des Advices nicht mehr »String«, sondern »Object« heißt. Wäre er »String«, käme es nicht zu dieser Situation, da dann der AspectJ-Compiler den Typ-Konflikt erkennen und beim Kompilieren eine entsprechende Fehlermeldung ausgeben könnte. Versuchen Sie daher, den Typ des Around-Advices so speziell wie möglich zu halten.

proceed() und void

Vielleicht haben Sie sich beim vorigen Around-Advice zum Stoppen der Zeit gefragt, was denn mit den void-Methoden ist. Der Pointcut erfasst auch diese. Aber die Anweisung

```
Object toreturn = proceed();
```

void mutiert zu »null«.

macht für void-Methoden keinen Sinn. Wenn Sie als Programmierer versuchen, das Ergebnis einer solchen Methode einer »Object«-Variablen zuzuweisen, beschwert sich darüber der Compiler. Wie aber macht es AspectJ? Auch für void-Methoden führt der Compiler ein Autoboxing durch, nur dass er in diesem Fall das »null«-Objekt zuweist. Dieses Verhalten lässt sich einfach dadurch überprüfen, dass wir den Rückgabewert zusätzlich ausgeben:

```
Object around() : execution(public * lotto..*(..)) {
    long t = System.currentTimeMillis();
    Object toreturn = proceed();
    t = System.currentTimeMillis() - t;
    Log.profile(thisJoinPoint + ": t = " + t
        + " ms, return = " + toreturn);
    return toreturn;
}
```

Im Falle einer void-Methode müsste jetzt »null« in der Ausgabe erscheinen. Dies belegt der folgende Ausschnitt:

```
...
execution(void lotto.Trommel.mixNumbers()): t = 87 ms, return = null
execution(Integer lotto.Trommel.getNumber()): t = 1 ms, return = 34
...
```

Spricht man über den Pointcut nur void-Methoden an, definiert man am besten auch den Around-Advice vom Typ »void«. In diesem Fall kann auf die Return-Anweisung verzichtet werden:

```
void around() : execution(public void lotto..*(..)) {
    long t = System.currentTimeMillis();
    proceed();
    t = System.currentTimeMillis() - t;
    Log.profile(thisJoinPoint + ": t = " + t + " ms");
}
```

Mit diesem Advice werden jetzt natürlich nur die Zeiten der void-Methoden gestoppt und mitprotokolliert. Ansonsten entspricht er dem vorigen Advice.

Spiel's noch einmal, Sam

2 x proceed() ist erlaubt.

Ein proceed() darf auch mehrfach aufgerufen werden. So wollen wir zur besseren Durchmischung der Lotto-Kugeln jeden Aufruf der mixNumber()-Methoden verdoppeln. Das schaffen wir über den folgenden Around-Advice:

```
pointcut callMixNumbers() :
    call(public void Trommel.mixNumbers(..));

void around() : callMixNumbers() {
    proceed();
    proceed();
}
```

Manipulation des Kontexts

Um die virtuelle Mechanik der Lotto-Trommel zu schonen, soll der Mischvorgang jetzt nur dann stattfinden, wenn die Kugeln noch nicht gemischt sind. Der erste Ansatz führt zu folgendem Versuch:

```
pointcut executeMixNumbers(Trommel trommel) :
    execution(public void Trommel.mixNumbers())
    && this(trommel);

void around(Trommel trommel) : executeMixNumbers(trommel) {
    do {
        proceed(); // geht nicht, AspectJ erwartet hier einen Parameter
    } while (!trommel.isMixed());
}
```

Wenn man diesen Code übersetzt, beschwert sich der Compiler über »*too few arguments to proceed, expected 1*«. Sobald ein Around-Advice ein Argument (hier: »Trommel trommel«) erwartet, wird dieses Argument auch im proceed()-Aufruf erwartet:

```
void around(Trommel trommel) : executeMixNumbers(trommel) {
    do {
        proceed(trommel);   // jetzt mit Parameter!
    } while (trommel.isSorted());
}
```

Jetzt ist der Compiler zufrieden. Beachten Sie, dass der trommel-Parameter des proceed()-Aufrufs nichts mit den Parametern der angesprochenen mixNumbers()-Methode zu tun hat,[18] sondern sich auf die Argumente des Around-Advices bezieht. Wie bereits auf Seite 142 erwähnt, beinhaltet proceed() bereits den kompletten Joinpoint mitsamt den Parametern.

Was ist, wenn weitere formale Parameter auftauchen, müssen die auch in der Parameter-Liste von proceed() erscheinen? Probieren wir es doch einfach aus:

```
pointcut executeMixNumbers(Trommel trommel, int n) :
    execution(public void Trommel.mixNumbers(int))
    && this(trommel)
    && args(n);

void around(Trommel trommel, int n) :
        executeMixNumbers(trommel, n) {
    do {
        proceed(trommel, n);          // 2 Parameter!
    } while (!trommel.isMixed());
}
```

Wie man an diesem Beispiel sieht, sind tatsächlich zwei Parameter nötig, wenn man nicht mit der Fehlermeldung *»too few arguments to proceed, expected 2«* konfrontiert werden will. Die Reihenfolge im proceed()-Aufruf muss dabei mit der Reihenfolge der Parameter-Liste des Around-Advices übereinstimmen.

Während Sie beim Before- und After-Advice den Kontext nur indirekt über den Aufruf weiterer Methoden beeinflussen konnten, haben Sie beim Around-Advice mehr Einfluss auf den Kontext. So können Sie z. B. den Return-Wert von proceed() nach Belieben manipulieren. Dies wollen wir ausnutzen, um den Rückgabewert der Random.nextInt()-Methode, die u. a. von Trommel.getNumber() verwendet wird, vorhersagbarer zu machen:

```
pointcut callRandomNextInt() :
    call(public int java.util.Random.nextInt(int));

int around() : callRandomNextInt() {
    int x = proceed();
```

[18] Geht auch nicht, da in diesem Beispiel die parameterlose mixNumber()-Methode adressiert wird.

```
    return 1;
}
```

Damit liefert jeder Random.nextInt(int)-Aufruf eine »1« zurück, was die Vorhersage der Lottozahlen erheblich vereinfacht. Dieser Aspekt muss nicht unbedingt der kriminellen Energie eines Mafia-Programmierers entstammen, sondern kann durchaus auch für die Testphase interessant sein, in der reproduzierbare Ergebnisse erwünscht sind.

Eine andere Manipulationsmöglichkeit bietet uns das this-Objekt. Dahinter verbirgt sich das Objekt, das zur Ausführung der proceed()-Anweisung herangezogen wird. Es lässt sich einfach austauschen und durch ein anderes ersetzen. Dies wollen wir an der Ziehung.getWinners()-Methode ausprobieren:

```
pointcut executeGetWinners(Ziehung z) :
    execution(public List Ziehung.getWinners())
    && this(z);

List around(Ziehung z) : executeGetWinners(z) {
    return proceed(z);
}
```

Dieser Around-Advice ruft den Original-Joinpoint auf. Jetzt tauschen wir das Original-Objekt »z« durch ein anderes Objekt der Klasse »Ziehung« aus:

```
List around(Ziehung z) : executeGetWinners(z) {
    Ziehung getuerkt = new Ziehung();
    return proceed(getuerkt);
}
```

Hier wurde ein frisches Ziehungs-Objekt, das mit den Zahlen 1, 2, . . . , 6 initialisiert wurde, erzeugt, und mit diesem Objekt wurden die Gewinner ermittelt. Ein anderes Szenario für den Austausch des Original-Objekts kann z. B. ein Standby-Objekt sein, das dann einspringt, wenn das Original-Objekt gerade unpässlich ist. Oder man möchte in sicherheitskritischen Bereichen den Rückgabewert durch zwei weitere redundante Objekte absichern lassen.

Anloges gilt auch für das target-Objekt. Wenn Sie den call- statt des execution-Joinpoints verwenden wollen, tauschen Sie im obigen Beispiel *this* durch *target* aus, und Sie erhalten das gleiche Resultat.

Argumenten-Tausch

Nachdem wir das darunter liegende Objekt ausgetauscht haben, fahren wir mit einer einfacheren Übung fort: dem Austauschen der Parameter einer aufgerufenen Methode. Dazu knöpfen wir uns die Methode

»getWinners(String)« der Ziehung-Klasse vor und definieren uns dafür einen Pointcut:

```
pointcut callGetWinners(String filename) :
    call(public List Ziehung.getWinners(String))
    && args(filename);
```

Wenn diese Methode mit dem Dateinamen als Parameter aufgerufen wird, wollen wir diesen Dateinamen gegen einen anderen austauschen:

```
List around(String filename) : callGetWinners(filename) {
    return proceed("Mafia.schein");
}
```

Hier erwartet AspectJ für die proceed()-Anweisung den Parameter, der an die Methode getWinners() übergeben wird. Wir übergeben aber nicht das Original (filename), sondern unseren eigenen String (»Mafia.schein«).

Ein weiteres Beispiel für die Manipulation von Argumenten ist die Erweiterung bestehender Log-Meldungen um zusätzliche Informationen:

```
/*
 * Log.info-Calls um weitere Infos erweitern...
 */

public pointcut callLogInfo(String msg) :
    call(public void Log.info(String)) &&
    args(msg);

void around(String msg) : callLogInfo(msg) {
    String extended = thisJoinPointStaticPart.getSourceLocation()
        + ": " + msg;
    proceed(extended);
}
```

Hier wird jeder Aufruf der Log.info()-Methode um die Codestelle, an der diese Methode aufgerufen wird, ergänzt. Weitere Möglichkeiten könnten Uhrzeit, Rechnername oder Informationen der Java-VM selbst sein.

Nachdem das mit den Log-Ausgaben so gut geklappt hat, wollen wir im nächsten Schritt die normalen System.out.println()-Ausgaben um einen kleinen Werbe-Banner anreichern. Dazu definieren wir uns folgenden Pointcut:

```
public pointcut callSystemOutPrintln(Object msg) :
    call(public void java.lang.System.out.println(*)) &&
    args(msg);
```

Leider funktioniert diese Wildcard nicht wie gewünscht: »out« ist keine Klasse, sondern ein Attribut der System-Klasse. Damit greift dieser Pointcut ins Leere. Da »System.out« aber ein Objekt der PrintStream-Klasse ist, verwenden wir stattdessen diese Klasse für den Pointcut:

```
public pointcut callSystemOutPrintln(Object msg) :
    call(public void java.io.PrintStream.println(*)) &&
    args(msg);
```

Weil wir alle Arten von Argumenten der println()-Methode erfassen wollen, nehmen wir die gemeinsame Oberklasse, die Object-Klasse, als Argument-Typ. Diesen Pointcut verwenden wir in einem Around-Advice, der im ersten Schritt nur den Joinpoint ohne weitere Veränderung aufruft.

```
void around(Object msg) : callSystemOutPrintln(msg) {
    proceed(msg);
}
```

Mit diesem Advice verändert sich noch nichts an unserer Ausgabe. Im nächsten Schritt wollen wir das Argument ab und zu durch eine kleine Werbe-Botschaft auflockern:

```
void around(Object msg) : callSystemOutPrintln(msg) {
    if (new Random().nextInt(10) == 0) {
         proceed ("*** " + msg + " ***\n"
            + "Diese Ausgabe wurde ihnen präsentiert von: "
            + "JavaTux.de");
    } else {
        proceed(msg);
    }
}
```

Dieser Advice bringt im Schnitt nach jedem zehnten Aufruf eine zusätzliche Ausgabe auf dem Bildschirm. Wenn man das Programm startet, wird vermutlich alles wie erwartet funktionieren. Nur ab und zu steigt es mit einer *ClassCastException* (*»java.lang.String can not be converted to char…«* o. Ä.) aus. Schuld daran sind Anweisungen wie

```
System.out.println('x');
```

Da der Advice einen Parameter vom Typ »Object« entgegennimmt, an dieser Stelle aber einen *char*-Basistyp bekommt, wird er durch den Autoboxing-Mechanismus in ein Charakter-Objekt konvertiert. Damit werden `"*** "` und die anderen Strings innerhalb der proceed()-Anweisung verknüpft. Anschließend versucht AspectJ, dieses temporäre Objekt (vom Typ »String«) auf den urprünglichen Parameter-Typ des Joinpoints, »char«, zu konvertieren. Dies schlägt fehl und führt zu dieser ClassCastException.

Hätten Sie statt »Object« den Typ »String« bei der Definition des Pointcuts und Advices verwendet, wäre dieses Problem nicht aufgetreten. Dann wäre auch die obige System.out.println('x')-Anweisung nicht verändert worden. Versuchen Sie also, nicht immer »Object« als Datentyp zu verwenden, sondern nur die Klasse einzusetzen, die die kleinste gemeinsame Oberklasse der betrachteten Argumente ist.

Ansonsten erkennt AspectJ die meisten Fehlgriffe beim Einsatz der proceed()-Anweisung wie

```
proceed(0);                      // falscher Datentyp
```

oder

```
proceed(msg, "don't worry");    // falsche Anzahl
```

Die Regel, die AspectJ hier anwendet, lautet:

> Die Argument-Liste von proceed() muss typmäßig mit der Liste des Around-Advices übereinstimmen.

Damit werden die meisten Probleme bereits zur Compile-Zeit abgefangen. Leider nicht alle, wie wir am obigen Beispiel gesehen haben.

5.5 Rangeleien

Syntax

declare precedence *Typ-Muster1*, *Typ-Muster2*, . . . ;

Wie ist die Reihenfolge bei Mehrfachtreffern?

Solange für einen Joinpoint nur ein Advice ausgeführt wird, braucht man sich über Vorrangregeln keine Gedanken zu machen. Was aber, wenn mehr als ein Advice zutrifft? Dann kann das Verhalten des Programms sehr wohl von der Reihenfolge abhängen, in der die einzelnen Advices abgearbeitet werden. Sehen wir uns dazu folgendes Beispiel an:

```
public aspect MafiaAspect {
    after(Trommel trommel) : execution(public void mixNumbers())
                             && this(trommel) {
        trommel.sort();
        System.out.println("Ätsch, Trommel wurde sortiert!");
    }
}
```

Dieser Aspekt sortiert nach jedem Mischvorgang die Trommel neu. Keine Ahnung, wie er Eingang in das Lotto-Projekt gefunden hat, aber um solche suspekten Aspekte zu neutralisieren, existiert ein CopAspect mit folgendem Inhalt:

```
public aspect CopAspect {
    void around(Trommel trommel) :
            execution(public void Trommel.mixNumbers(..))
            && this(trommel) {
        proceed(trommel);
        if (!trommel.isMixed()) { {
            proceed(trommel);   // misch noch einmal, Sam
        }
        System.out.println("Mischen abgeschlossen.");
    }
}
```

Sollte nach dem Mischvorgang die Trommel immer noch sortiert sein, wird die Mischung einfach nochmal ausgeführt. Die spannende Frage hierbei aber ist: Welcher Aspekt gewinnt bzw. welcher Aspekt wird als letzter ausgeführt?

5.5.1 Bestimmung der Vorrangregeln

Sobald ein Advice auf mehr als einen Joinpoint zutrifft, bestimmt AspectJ die Reihenfolge der einzelnen Advices. Dabei unterscheidet AspectJ, ob er im selben oder in unterschiedlichen Aspekten untergebracht ist.

Vorrangregeln zwischen Aspekten

Bei Advices in *unterschiedlichen* Aspekten kommen folgende Regeln zum Einsatz:

1. **Precedence-Deklaration:**
   Mit Hilfe von »**declare precedence : A, B;**« kann man die Reihenfolge explizit definieren: Advice *A* hat Vorrang vor Advice *B*, d. h., *A* wird nach *B* ausgeführt. Für unseren CopAspect bedeutet dies, dass wir ihn als Ersten in der Precedence-Liste aufführen sollten:[19]

   ```
   public aspect CopAspect {
       declare precedence : CopAspect, MafiaAspect;
       ...
   ```

   Damit wird der Advice aus CopAspect *nach* dem MafiaAspect-Advice ausgeführt. Soll der CopAspect generell den höchsten Vorrang haben, kann man auch Wildcards einsetzen:

   ```
   declare precedence : CopAspect, *;
   ```

[19]Die Precedence-Deklaration muss innerhalb eines Aspektes stehen.

2. **Unter sticht Ober:**
   Wenn ein Aspekt einen anderen Aspekt erweitert (s. Kapitel 7.3 auf Seite 207), bekommt der Sub-Aspekt einen höheren Vorrang als der Super-Aspekt, damit er dessen Verhalten überschreiben kann.

AspectJ 5

3. **Direkte Übereinstimmung schlägt Autoboxing:**
   Ein Muster, das direkt auf eine Methode zutrifft (z. B. »Vector.new(int)«, bekommt Vorrang vor einem Muster, das nur über Autoboxing zutrifft (»Vector.new(Integer)«).
4. **Rest unbestimmt:**
   Trifft keiner der Fälle zu, ist das Verhalten unbestimmt. Im obigen Beispiel bleibt damit das Verhalten, ob der Mafia- oder der Cop-Aspect gewinnt, dem Zufall überlassen.

Vorrangregeln innerhalb von Aspekten

Wenn zwei konkurrierende Advices innerhalb desselben Aspekts definiert sind, hängt die interne Hackordnung von Art und Typ des Advices ab. Zwei Situationen gibt es dabei zu unterscheiden:

1. Einer der Advices ist ein After-Advice:
   Der Advice, der *später* definiert wird, erhält den *höheren* Vorrang
2. Es ist kein After-Advice darunter:
   Der Advice, der *zuerst* definiert wird, erhält den *höheren* Vorrang.

Um diese Regeln besser zu verstehen, wollen wir sie kurz durchexerzieren. Nehmen wir dazu an, wir hätten einen Aspekt mit folgenden Advices:

```
public PrecedenceDemo {
    // Before1
    // Before2
    // After1
    // After2
}
```

Nach Regel 2 gilt: Before1 > Before2.[20] Sobald ein After-Advice ins Spiel kommt, gilt Regel 1. Damit ergibt sich: After1 > Before1, After1 > Before2, dasselbe für After2, After2 > After1. Insgesamt ergeben sich damit folgende Vorrangregeln:

After2 > After1 > Before1 > Before2

[20] Das »>«-Zeichen soll hier die Vorrangregeln abbilden, wobei links der Advice mit dem höheren Vorrang steht. Das heißt, im Beispiel hat Before1 Vorrang vor Before2.

Aspect 1.0 hatte ein etwas einfacheres Regelwerk, das aber zusammen mit dem Ausführungszeitpunkt eines Advices zu Verwirrungen geführt hat. Um dieses zu verstehen, schauen wir es uns zunächst einmal an.

Ausführungszeitpunkt eines Advices

Vorrangregel und Advice-Art bestimmen das »Wann«.

Wann wird ein Advice ausgeführt? Diese Frage wird vor allem in Verbindung mit den Vorrangregeln interessant, da bei Advices, die sich gegenseitig beeinflussen oder gar eine Aktion rückgängig machen können (wie im Eingangsbeispiel), der zuletzt ausgeführte Advice gewinnt. Wann ein Advice ausgeführt wird, hängt neben den Vorrangregeln von seinem Typ ab:

- Der **After**()-Advice wird nicht sofort ausgeführt, sondern gibt die Kontrolle an den Advice mit dem *nächsthöheren Vorrang* weiter. Erst danach kommt er zur Ausführung, wenn er für den Joinpoint zutrifft. (So wird z. B. der *after() throwing*-Advice nur dann aufgerufen, wenn die angegebene Exception aufgetreten ist.)
- Der **Before**()-Advice wird sofort ausgeführt. Danach wird die Kontrolle an die Advices mit höherem Vorrang weitergegeben. Tritt eine Exception auf, kommen die Advices mit niedrigerer Priorität nicht zum Zuge.
- Der **Around**()-Advice wird ebenfalls sofort ausgeführt. Durch seine proceed()-Anweisung kann der nächste Advice zum Einsatz kommen. Wenn er allerdings vorher eine Exception wirft oder vor der proceed()-Anweisung terminiert (z. B. mit einer return-Anweisung), werden Advices mit niedrigerer Priorität (sowie der Joinpoint) nicht ausgeführt.

Betrachten wir dazu eine Welcome-Klasse, die einen kleinen Willkommensgruß auf den Bildschirm zaubert:

```
package demo;

public class Welcome {
    public void greet() {
        System.out.println("Welcome to " + this.getClass());
    }
    public static void main(String[] args) {
        new Welcome().greet();
    }
}
```

Die Ausgabe dieser Klasse wollen wir um ein paar zusätzliche Grüße verschiedener Advices anreichern. Dazu dient folgender Aspekt:

```
package demo;

public aspect PrecedenceDemo {

    // Before1
    before() : call(public void demo.Welcome.greet()) {
        System.out.println("Welcome to Before1");
    }

    // Before2
    before() : call(public void demo.Welcome.greet()) {
        System.out.println("Welcome to Before2");
    }

    // After1
    after() : call(public void demo.Welcome.greet()) {
        System.out.println("Welcome to After1");
    }

    // After2
    after() : call(public void demo.Welcome.greet()) {
        System.out.println("Welcome to After2");
    }

}
```

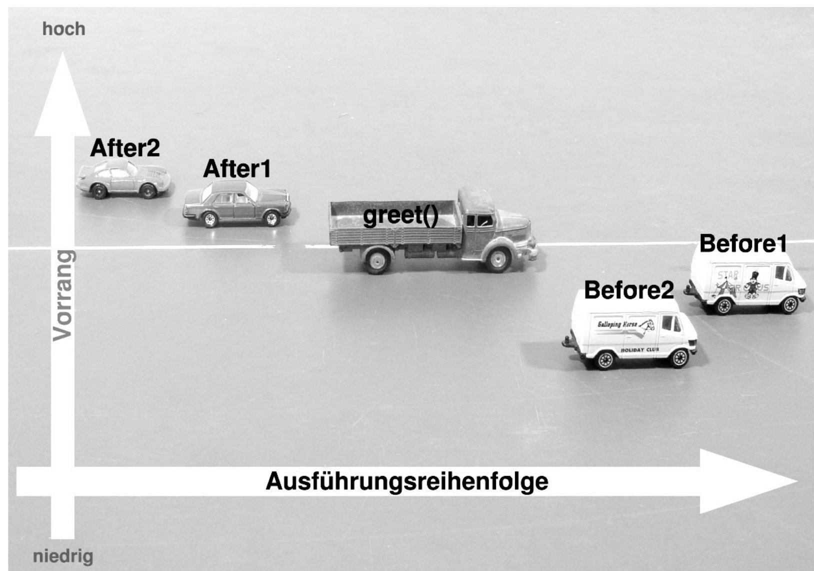

Abbildung 5.3 *Vorrangregeln und Reihenfolge der Ausführung*

Wie bereits im vorigen Abschnitt dargelegt, sind die Vorrangregeln »After2 > After1 > Before1 > Before2«. Zusammen mit dem Ausfüh-

rungszeitpunkt der einzelnen Advices ergibt sich folgende Ausgabe (vgl. Abb. 5.3 auf der vorherigen Seite):

```
Welcome to Before1
Welcome to Before2
Welcome to class demo.Welcome
Welcome to After1
Welcome to After2
```

After2 hat zwar die höchste Priorität, wird aber erst am Ende ausgeführt. Damit erscheint auch seine Ausgabe als Letzte auf dem Bildschirm. Ähnliches gilt für den After1-Advice. Als Nächstes in der Prioritätsschlange wartet Before1 auf seinen Einsatz. Er wird sofort ausgeführt, weshalb seine Ausgabe als Erste erscheint. Auch Before2 kommt noch vor der eigentlichen greet()-Methode (aber nach Before1) zur Ausführung.

5.5.2 Zirkuläre Abhängigkeiten

Dadurch dass jeder Aspekt seinen eigenen Satz von Precedence-Regeln deklarieren kann, kann es zu zirkulären Abhängigkeiten kommen:

```
declare precedence: A, B;
declare precedence: B, C:
declare precedence: C, A;
```

Aus den ersten beiden Zeilen ergibt sich, dass `A` eine höhere Priorität als `C` hat. Dies steht im Widerspruch zur letzten Zeile. Da aber die Vorrangregeln nur dann zum Zuge kommen, wenn mehrere Advices auf denselben Joinpoint zutreffen, müssen diese Regeln nicht zwangsläufig in Konflikt stehen. Wenn z. B. `C` keine gemeinsamen Joinpoints mit `B` hat, wird AspectJ nur die dritte Regel zur Bestimmung der Priorität heranziehen. Treffen dagegen Advices aus allen drei Aspekten für einen Joinpoint zu, kann AspectJ diesen Widerspruch nicht auflösen und gibt eine Fehlermeldung aus. Dies kann manchmal etwas irritierend sein, wenn später ein Pointcut mit Advice hinzugefügt wird und plötzlich ein Fehler wegen zirkulärer Abhängigkeiten gemeldet wird.

5.6 Zusammenfassung

- Während der Pointcut die Joinpoints definiert, die in Frage kommen, enthält der Advice den Code, der an diesen Joinpoints zum Tragen kommen soll.
- Für den Zugriff auf den Kontext eines Joinpoints gibt es eine Reihe von Möglichkeiten:

- *this()* erlaubt den Zugriff auf den Kontext des Joinpoints.
- *target()* erlaubt den Zugriff auf den Kontext des Ziel-Joinpoints (beim call-Joinpoint ist dies die aufgerufene Methode).
- *args()* erlaubt den Zugriff auf die Argumente.
- *returning()* erlaubt den Zugriff auf den Rückgabewert (After-Advice).
- *throwing()* erlaubt den Zugriff auf die ausgelöste Exception (After-Advice).

- Auf den Kontext kann auch per Reflexion mit Hilfe folgender Objekte zugegriffen werden:
  - *thisJoinPoint* besitzt eine Reihe von Methoden wie *getArgs()*, *getThis()* oder *getTarget()*, die den Zugriff auf den Kontext erlauben.
  - *thisJoinPointStaticPart* stellt weniger Methoden für den Zugriff zur Verfügung, kommt aber ohne zusätzlichen Speicher aus.
  - *thisEnclosingJoinPointStaticPart* erlaubt den Zugriff auf den umgebenden Joinpoint.
- Advices dürfen nur die gleichen Exceptions wie der umgebende Joinpoint werfen.
- Der Before-Advice wird vor dem eigentlichen Joinpoint ausgeführt.
- Der unqualifizierte After-Advice (*ohne* returning und throwing) wird nach dem Joinpoint ausgeführt, auch dann, wenn eine Exception auftritt.
- *After Returning* erlaubt den Zugriff auf den Rückgabewert. Tritt eine Exception auf, wird er nicht ausgeführt.
- Der Exception-Handler einer Methode lässt sich über *After Throwing* erweitern.
- Der Around-Advice wird anstatt eines Joinpoints ausgeführt. Mit *proceed()* kann dabei der eigentliche Joinpoint aufgerufen werden.
- Die Argumente von *proceed()* müssen vom Typ her mit dem Original-Joinpoint übereinstimmen, sonst erhält man zur Laufzeit eine *ClassCastException*.
- Über »*declare precedence*« lässt sich die Rangfolge von Aspekten festlegen. Dabei sollte man sich vor zirkulären Abhängigkeiten hüten, da diese u. U. erst später bei der Erweiterung eines Aspekts vom Compiler beanstandet werden.

- ❏ Bei Vererbung erhält ein abgeleiteter Aspekt eine höhere Rangordnung als der Vater-Aspekt.
- ❏ Ansonsten gilt: Zwischen After-Advices erhält der später definierte Advice den höheren Vorrang, ansonst der zuerst definierte Advice.
- ❏ Ein Before-Advice mit höherem Vorrang wird wie der Around-Advice sofort ausgeführt, während beim After-Advice der mit dem höchsten Vorrang zuletzt dran ist.

5.7 Übungen

1. Legen Sie einen KontoAspect an. Falls das Konto überzogen wird, soll automatisch eine Exception ausgelöst werden.
2. Protokollieren Sie alle aufgetretenen RuntimeExceptions (inkl. davon abgeleiteter Exceptions) mit. Ermitteln Sie auch die Stellen, an denen sie aufgetreten sind.

   Kleiner Tipp: Falls eine RuntimeException nicht abgefangen wird, bekommen Sie diese Exception nicht über den handler()-Pointcut mit (er wird ja nicht aufgerufen!). Probieren Sie es über den Konstruktor der RuntimeException-Klasse.

3. Jedesmal, wenn der Kontostand geändert wird, soll er automatisch mit Hilfe der store()-Methode abgespeichert werden.
4. Der Croupier darf jetzt zusätzlich neben dem Konto des Spielers die Hausbank verwalten, deren Grundkapital ebenfalls 5000 Chips beträgt. Lassen Sie den Computer die Rolle des Spielers übernehmen und 100 Runden Roulette spielen. Dabei soll der simulierte Spieler in jeder Runde 200 Chips auf »Pair« (gerade Zahl) setzen (bitte beachten: bei »0« gewinnt die Bank). Vergleichen Sie am Ende den Kontostand des Spielers mit dem der Hausbank. Was vermuten Sie, wer am Ende mehr in der Tasche hat?
5. Wiederholen Sie den Simulationslauf aus der vorigen Aufgabe mehrere Male. Erhalten Sie jedesmal das gleiche Ergebnis?
6. Wie oft müsste bei 100 Würfen »Pair« erscheinen? Schreiben Sie eine Testmethode dafür und überprüfen Sie Ihre Vermutung.
7. Machen Sie den Zufall vorhersagbar: Tauschen Sie die Implementierung der Random-Klasse mit Hilfe des Around-Advices aus, so dass werfeKugel() nacheinander die Zahlen 0, 1, 2, ..., 36 zurückliefert.
8. Starten Sie wieder Ihren Simulationslauf. Erhöhen Sie die Anzahl der Simulationen so weit, dass eine Überziehung des Kontos zu

erwarten ist. Wird tatsächlich eine RuntimeException ausgelöst? Wie reagiert Ihre Anwendung auf diese Exception? Wird sie richtig behandelt?

9. Verwenden Sie einen eigenen Sourcepfad »test« für die Testfälle und den TestAspect. Nehmen Sie »test« von der Kompilierung aus (der Sourcepfad kann in Eclipse über »Project ▷ Properties« eingestellt werden), und starten Sie die Simulation erneut.
10. Binden Sie den vorigen Test-Sourcepfad wieder mit in die Kompilierung ein. Steuern Sie dieses Mal den Algorithmus für die Zufallszahlen über einen if-Pointcut. Führen Sie dazu eine Testing-Klasse mit der Methode isEnabled() ein, die *true* zurückliefert, wenn ein »testing«-Property gesetzt ist und den Wert »true« enthält.