

Teil I

Einleitung

1 Einführung in Aspektorientierung

Any fool can write code that a computer can understand.
Good programmers write code that humans can understand.

(Martin Fowler in »Refactoring«)

Neue Techniken und Trends bringen oft auch neue Begriffe mit sich, um sich vom Rest der (Informatik-)Welt abzuheben oder abzugrenzen. Wenn man mitreden will, reicht es aber nicht, dass man diese Begriffe kennt – ohne Kenntnis des Gesamtzusammenhangs sind sie nur leere Worthülsen. Deswegen werden wir in diesem Kapitel

- ❑ einen kurzen geschichtlichen Abriss über die Entwicklung der Programmiersprachen vorstellen,
- ❑ uns einen Überblick über die Welt der aspektorientierten Programmierung (AOP) verschaffen,
- ❑ einen Einblick in die Denkweise von AOP bekommen,
- ❑ die wichtigsten Begriffe aus dem AOP-Umfeld kennen lernen und
- ❑ in AspectJ als Vertreter der aspektorientierten Programmiersprachen hineinschnuppern.

Nach diesem Kapitel haben Sie einen groben Einblick in die Welt der Aspektorientierung und können das Potenzial, das in diesem Ansatz steckt, abschätzen. Dem handwerklichen Umgang mit AspectJ sind dann die anschließenden Kapitel gewidmet.

1.1 Von Assembler bis AspectJ

Seit der Erfindung von intelligenten Rechenmaschinen steht man vor dem Problem: Wie sag ich es dem Computer? Warum versteht er mich nicht? Während in der Anfangsphase die Rechner noch komplett über Schalter und Taster bedient und programmiert wurden, entwickelten sich bald die ersten Programmiersprachen, um dem Rechner seine Wünsche besser mitteilen zu können. Im Wesentlichen lässt sich diese Entwicklung in vier Epochen einteilen:

Die vier Epochen der Programmiersprachen

1. Maschinensprache und Assembler
2. Prozedurale und funktionale Sprachen¹
3. Objektorientierte Sprachen
4. Aspektorientierte Sprachen

Daneben gab und gibt es noch weitere Strömungen wie logische Programmiersprachen (deren bekanntester Vertreter »Prolog« ist). Aber abseits von Forschungseinrichtungen und KI (Künstliche Intelligenz) blieb diesen Sprachen der Durchbruch verwehrt. Dieses Kapitel soll fernab von akademischen Feinheiten nur einen groben Überblick über den »Mainstream« ohne Anspruch auf Vollständigkeit geben.

1.1.1 Maschinensprache und Assembler

Maschinensprache =
Ursprache

Der Prozessor im Innersten des Rechners versteht leider nur Nullen und Einsen, und auch seine Befehle setzen sich daraus zusammen. Diese »Maschinensprache« musste man in der EDV-Steinzeit beherrschen, um den Rechner programmieren zu können (stellen Sie sich einfach vor, Sie müssten direkt den Bytecode eingeben).

Assembler = lesbarere
Maschinensprache

Da der Mensch sich besser Kommandos wie »MOV« (move) oder »ADD« als irgendwelche Zahlencodes merken kann, entwickelten sich die ersten Assembler, die solche Kommandos (auch als *Mnemonics* bezeichnet) verstehen und in Maschinensprache übersetzen konnten:

```
ENTRY(strlen)
    mov    r4,r0
    and    #3,r0
    tst    r0,r0
    bt/s   1f
    mov    #0,r2
```

Was man nicht in
Assembler machen
kann, muss man löten!

Dies ist der Ausschnitt aus der `strlen`-Funktion des Linux-Kernels – immer noch nur von Eingeweihten zu lesen und zu verstehen. Dafür sind Assembler-Routinen schnell und nahe am Puls der Maschine, weswegen man sie heute hauptsächlich für zeitkritische und Hardware-nahe Aufgaben einsetzt. Allerdings hat jede Prozessorarchitektur ihren eigenen Befehlssatz, so dass Hardware-Routinen nicht portabel sind.

1.1.2 Prozedurale Sprachen

Mit dem Aufkommen von COBOL und FORTRAN löste man sich von der Abhängigkeit von der Maschine. Man war nicht mehr von

¹auch als imperative Sprachen bezeichnet

einem bestimmten Prozessor oder von einer Hardware abhängig, sondern konnte die Programme in einer maschinenunabhängigen Sprache formulieren.

Mit diesen Sprachen hielt die strukturierte Programmierung Einzug – bekannte Methoden waren hier SA/SD (Strukturierte Analyse/Strukturiertes Design) oder SADT (Structured Analysis and Design Technique). »Teile und herrsche« (engl.: »divide and conquer«) hieß das Stichwort, um größere Aufgaben angehen zu können.

*Strukturierte
Programmierung*

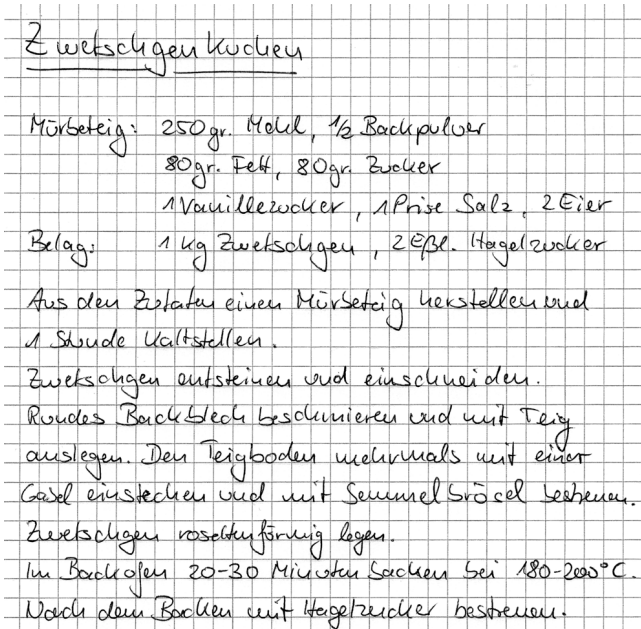


Abbildung 1.1
Die Prozedur des
Kuchenbackens

Grundelemente dieser Sprachen sind dabei Prozeduren oder Funktionen (weshalb diese Sprachen manchmal auch funktionale Sprachen genannt werden). Eine Prozedur oder Funktion wird beim Aufruf mit den Parametern und Daten versorgt, die sie zum Bearbeiten ihrer Aufgabe benötigt. Vergleichbar ist diese Vorgehensweise in etwa mit dem Kuchenbacken (vgl. Abb. 1.1)²: Hier besorgt man sich die Zutaten, um diesen »Input« weiterzuverarbeiten und als Ergebnis einen fertigen Kuchen zu bekommen. Die einzelnen Schritte dieser »Verarbeitung« entsprechen dabei dem Rumpf einer Funktion.

*Grundelemente:
Prozeduren bzw.
Funktionen*

²Quelle: Omas Backbuch

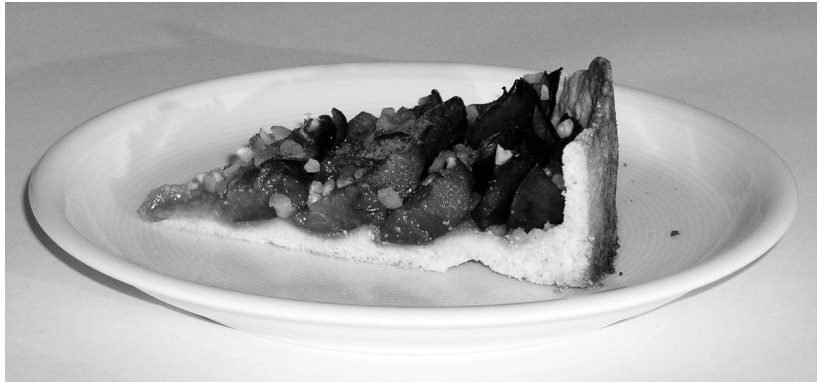
1.1.3 Objektorientierte Programmierung (OOP)

*Am Anfang steht das
Objekt...*

Objektorientierte Sprachen wie Smalltalk, C++ oder Java brachten eine weitere Entmündigung des Computers mit sich – man war nicht mehr gezwungen, wie der Computer in Prozeduren oder Funktionen zu denken, sondern durfte jetzt mit Objekten und Klassen arbeiten. Funktionen kommen zwar auch noch vor, aber sie heißen Methoden und stehen nicht mehr im Mittelpunkt der Betrachtung.

Bezogen auf das Kuchenbacken bedeutet Objektorientierung, dass man nicht mehr das Backen an sich, sondern den Kuchen als Ganzes betrachtet. Die Zutaten sind Teil des Kuchens, die Back-Methode ist nur eine unter vielen. Um zum fertigen Kuchen zu kommen, sagt man jetzt lediglich »Kuchen, back dich« (man ruft die Back-Methode auf), und der Kuchen wird schon wissen, was er zu tun hat.³

Abbildung 1.2
*Ergebnis der
Back-Methode*



Eines der ursprünglichen Versprechen der Objektorientierung, die Wiederverwendung, erwies sich in der Realität leider als frommer Wunsch – Wiederverwendung ist nicht zum Nulltarif zu haben, sondern muss (ein-)geplant werden –, aber Objektorientierung half, der wachsenden Komplexität von Programmen durch Modularisierung, Kapselung und Vererbung Herr zu werden. Und ja, trotz aller Schwierigkeiten gibt es wiederverwendbare Komponenten, wie die Vielzahl von verfügbaren Bibliotheken für die verschiedensten Zwecke belegt.

Aber man stößt immer mehr an die Grenzen. Trotz aller Modularisierung gibt es systemweite Belange wie Sicherheit oder die Protokollierung wichtiger Ereignisse (neudeutsch: Logging), die sich quer über alle Klassen und Methoden verteilen. Hier können kleine Anforderungsnuancen großen Änderungsaufwand nach sich ziehen.

³ Vermutlich wird er erst einmal eine »MissingIngredientsException« werfen, da die Zutaten noch fehlen.

1.1.4 Aspektorientierte Programmierung (AOP)



Abbildung 1.3
AOP = OOP +
Schlagsahne (Aspekte)

Die nächste Epoche der Programmiersprachen werden die aspektorientierten Sprachen wie AspectJ, AspectC++ oder AspectS (Smalltalk) einläuten. AOP wird die Objektorientierung nicht überflüssig machen, sondern darauf aufbauen. Dieses Fundament wird um Aspekte erweitert, die es erlauben, programmübergreifende Themen (im AOP-Jargon als *Crosscutting Concerns* bezeichnet) wie Logging, Transaktionen, Sicherheit ... zentral an einer Stelle verwalten zu können.

Crosscutting Concern

1.2 Klasse Konto – ein Fallbeispiel

Um ein wenig in Begriffswelt und Denkweise der AOP einzutauchen, wollen wir ein einfaches Konto-Beispiel betrachten:

```
public class Konto {  
  
    private double kontostand = 0.0;  
  
    public void einzahlen(double betrag) {  
        kontostand = kontostand + betrag;  
    }  
  
    public void abheben(double betrag) {  
        kontostand = kontostand - betrag;  
    }  
  
    public void ueberweisen(double betrag, Konto anderesKonto) {  
        abheben(betrag);  
        anderesKonto.einzahlen(betrag);  
    }  
  
}
```

Abbildung 1.4

Eine einfache
Konto-Verwaltung



1.2.1 Erweiterte Konto-Klasse

*Gleich wird's
unübersichtlich...*

Das Beispiel ist noch recht übersichtlich: Als einziges Attribut enthält es den Kontostand (der der Einfachheit halber vom Typ »double« ist); des Weiteren enthält es drei Business-Methoden zum Einzahlen, Abheben und für die Überweisung auf ein anderes Konto. Die Business-Logik ist klar verständlich und erschließt sich auch mit der Materie nicht so vertrauten Personen innerhalb weniger Minuten.

Etwas anders sieht die Sache aus, wenn noch die Fehlerbehandlung dazukommt. Es wäre zwar schön, wenn man z. B. der Steuerbehörde einen negativen Betrag überweisen könnte, aber wie wir leider wissen, spielt da die Bank nicht mit. Das Gleiche gilt für das Einzahlen bzw. Abheben: Auch hier sind nur positive Werte erlaubt.

```
public class Konto {  
  
    private double kontostand = 0.0;  
  
    public void einzahlen(double betrag) {  
        if (betrag < 0) {  
            throw new IllegalArgumentException("Betrag negativ");  
        }  
        kontostand = kontostand + betrag;  
    }  
  
    public void abheben(double betrag) {  
        if (betrag < 0) {  
            throw new IllegalArgumentException("Betrag negativ");  
        }  
    }  
}
```

```

    if (kontostand < betrag) {
        throw new RuntimeException("keine Deckung");
    }
    kontostand = kontostand - betrag;
    if (kontoUeberzogen()) {
        kontostand = kontostand + betrag;
        throw new RuntimeException("Konto überzogen");
    }
}

public void ueberweisen(double betrag, Konto anderesKonto) {
    abheben(betrag);
    anderesKonto.einzahlen(betrag);
}
}

```

Wer das Beispiel genau studiert, dem wird in der abheben()-Methode die Überprüfung aufgefallen sein, dass das Konto gedeckt ist. Das Beispiel hat viel von seiner Eleganz und Einfachheit verloren und ist nicht mehr ganz so einfach zu verstehen. Kommen jetzt noch zusätzliche gesetzliche Vorgaben wie die Protokollierung wichtiger Kontoänderungen dazu, wird es noch unübersichtlicher:

*...und
unübersichtlicher*

```

public class Konto {

    private double kontostand = 0.0;

    public void einzahlen(double betrag) {
        System.out.println(betrag + " soll eingezahlt werden");
        if (betrag < 0) {
            System.err.println("Einzahlung mit neg. Betrag verweigert");
            throw new IllegalArgumentException("Betrag negativ");
        }
        kontostand = kontostand + betrag;
        System.out.println(betrag + " auf Konto eingezahlt.");
    }

    public void abheben(double betrag) {
        System.out.println(betrag + " soll ausgezahlt werden");
        if (betrag < 0) {
            System.err.println("Abheben mit neg. Betrag verweigert");
            throw new IllegalArgumentException("Betrag negativ");
        }
        if (kontostand < betrag) {
            System.err.println(
                "Konto nicht gedeckt, Auszahlung verweigert");
            throw new RuntimeException("keine Deckung");
        }
        kontostand = kontostand - betrag;
        System.out.println(betrag + " von Konto ausgezahlt.");
    }
}

```

```

public void ueberweisen(double betrag, Konto anderesKonto) {
    System.out.println(betrag + " soll ueberwiesen werden");
    abheben(betrag);
    anderesKonto.einzahlen(betrag);
    System.out.println(betrag + " auf " + anderesKonto
        + " ueberwiesen");
}
}

```

1.2.2 Refactoring

*Es wird
übersichtlicher...*

Eine Lösung, die Martin Fowler in seinem Buch »Refactoring«^[4] vor- schlägt, heißt »*Extract Method*« – das Herausziehen von Gemeinsam- keiten in eine eigene Methode. Als Kandidat dafür bietet sich der An- fang der einzahlen()- und abheben()-Methode an, die doch einige Ge- meinsamkeiten aufweisen:

```

public class Konto {

    private double kontostand = 0.0;

    public void einzahlen(double betrag) {
        System.out.println(betrag + " soll eingezahlt werden");
        betragCheck(betrag);
        kontostand = kontostand + betrag;
        System.out.println(betrag + " auf Konto eingezahlt.");
    }

    public void abheben(double betrag) {
        System.out.println(betrag + " soll ausgezahlt werden");
        betragCheck(betrag);
        if (kontostand < betrag) {
            System.err.println(
                "Konto nicht gedeckt, Auszahlung verweigert");
            throw new RuntimeException("keine Deckung");
        }
        kontostand = kontostand - betrag;
        System.out.println(betrag + " von Konto ausgezahlt.");
    }

    public void ueberweisen(double betrag, Konto anderesKonto) {
        System.out.println(betrag + " soll ueberwiesen werden");
        abheben(betrag);
        anderesKonto.einzahlen(betrag);
        System.out.println(betrag + " auf " + anderesKonto
            + " ueberwiesen");
    }

    private void betragCheck(double betrag)
        throws IllegalArgumentException {

```

```

        if (betrag < 0) {
            System.err.println("Aktion mit neg. Betrag verweigert");
            throw new IllegalArgumentException("Betrag negativ");
        }
    }
}

```

Na ja, viel hat es nicht gebracht, aber zumindest etwas lesbarer ist es doch geworden. Schauen wir uns als Nächstes an, was die aspektorientierte Programmierung zur Behebung dieses Dilemmas bietet.

... aber nicht viel!

1.2.3 Erweiterung über Aspekte

Wie würde die Konto-Klasse unter Verwendung von Aspekten aussehen? Ganz einfach, so:

Die Business-Logik kommt wieder zum Vorschein!

```

public class Konto {

    private double kontostand = 0.0;

    public void einzahlen(double betrag) {
        kontostand = kontostand + betrag;
    }

    public void abheben(double betrag) {
        kontostand = kontostand - betrag;
    }

    public void ueberweisen(double betrag, Konto anderesKonto) {
        abheben(betrag);
        anderesKonto.einzahlen(betrag);
    }

}

```

Wie bitte? Dies ist doch genau die ursprüngliche Klasse, die wir zu Beginn hatten. Wo, bitte schön, ist die Fehlerbehandlung? Und das Logging? – Gemach, gemacht, diese Funktionalität wurde in Aspekten ausgelagert. Wie, das werden wir gleich sehen.

Vorher werden wir uns aber noch mit einigen Begriffen vertraut machen. Während wir beim Übergang von der prozeduralen zur objektorientierten Programmierung unser Vokabular um Klasse, Vererbung oder Polymorphismus erweitern mussten, beglückt uns das AOP- und AspectJ-Umfeld mit:

Joinpoint (Verbindungspunkt) ist ein Ereignis während eines Programmablaufs, das über Advices (s. u.) erweitert oder modifiziert werden kann. Solch eine Stelle kann der Ein- oder

AOP-Vokabular

Austritt aus einer Methode oder das Auftreten einer Exception sein.

Advice (Empfehlung) ist eine Erweiterung oder Modifikation, die vor, nach oder anstelle eines Joinpoints ausgeführt werden soll (z. B. Überprüfung des Kontostandes).

Pointcut (Schnittpunkt) selektiert eine Menge von Joinpoints, an denen ein bestimmter Advice greifen soll (z. B. Ausgabe einer Log-Meldung beim Eintritt in eine Methode).

Da Sie sich vermutlich noch nichts darunter vorstellen können, wollen wir wieder zum Konto-Beispiel zurückkehren, um einen ersten Eindruck von AspectJ und dessen Gedankenwelt zu bekommen. Dabei fangen wir mit der Fehlerbehandlung an und betrachten die ursprünglichen einzahlen()- und abheben()-Methoden:

```
public void einzahlen(double betrag) {
    if (betrag < 0) {
        throw new IllegalArgumentException("Betrag negativ");
    }
    kontostand = kontostand + betrag;
}
```

Der Anfang der abheben()-Methode sieht ähnlich aus. Die gemeinsamen Anweisungen stecken wir in einen *Advice*, der Ähnlichkeiten mit einer Java-Methode aufweist:

Beispiel-Advice

```
before(double betrag) : kontobewegung(betrag) {
    if (betrag < 0) {
        throw new IllegalArgumentException("negativer Betrag");
    }
}
```

Dies ist ein so genannter *Before-Advice*, der vor einem Joinpoint, der durch den Pointcut »*kontobewegung*« (Definition und Beispiel folgen gleich) definiert wird, ausgeführt wird. Oder einfacher ausgedrückt: Vor jedem Aufruf von abheben() oder einzahlen() wird diese If-Anweisung eingefügt.

Neben dem *Before-Advice* gibt es noch den *After-* und *Around-Advice*:

Advice-Typen

Before-Advice: Der Code wird *vor* dem eigentlichen Joinpoint⁴ aufgerufen.

After-Advice: Der Code wird *nach* Ausführung des Joinpoints aufgerufen.

⁴Haben Sie sich merken können, was ein Joinpoint ist? Falls nicht, setzen Sie ihn einfach in erster Näherung mit dem Aufruf einer Methode gleich.

Around-Advice: Der Code wird *anstelle* des Joinpoints ausgeführt.

Woher weiß AspectJ, an welchen Stellen er den zusätzlichen Code einfügen muss? Im obigen Beispiel über den Pointcut »*kontobewegung*«. Dieser ist folgendermaßen deklariert:

```
pointcut kontobewegung(double betrag):
    (call(public void bank.Konto.einzahlen(double))
     || call(public void bank.Konto.abheben(double)))
    && args(betrag);
```

Entscheidend an dieser Deklaration ist die zweite Zeile, die mit »call« beginnt. Das Schlüsselwort »call« deutet an, dass der Pointcut den Aufruf einer Methode beschreibt (möglich ist hier auch Zugriff auf Attribute, Auftreten einer Exception und andere). In diesem Beispiel werden genau die beiden Methoden einzahlen() und abheben() der bank.Konto-Klasse angesprochen. Über Muster können auch mehrere Klassen und Methoden damit ausgewählt werden. Der Typ des erwarteten Aufruf-Parameters ist auf »double« beschränkt, aber auch hier ist die Angabe von Mustern möglich, wie wir im 2. Teil dieses Buches sehen werden.

Der Parameter der beiden Methoden wird im Before-Advice noch für die Überprüfung benötigt und zu diesem Zweck als Pointcut-Parameter über args mitgeführt.

Damit der Advice eine Heimat hat, stecken wir ihn zusammen mit einem weiteren Advice, der den Abhebevorgang überwacht, in einen Konto-Aspekt:

Konto-Aspekt

```
public aspect KontoAspect {

    pointcut kontobewegung(double betrag):
        (call(public void bank.Konto.einzahlen(double))
         || call(public void bank.Konto.abheben(double)))
        && args(betrag);

    before(double betrag) : kontobewegung(betrag) {
        if (betrag < 0) {
            throw new IllegalArgumentException("negativer Betrag");
        }
    }

    before(double betrag, Konto konto):
        execution(public void Konto.abheben(double))
        && args(betrag)
        && this(konto) {
        if (betrag > konto.kontostand) {
            throw new RuntimeException("keine Deckung");
        }
    }
}
```

Der hinzugekommene Before-Advice wird vor der Ausführung (*execution*) der `abheben()`-Methode ausgeführt. Er vergleicht den übergebenen Betrag mit dem Kontostand⁵ und schreitet mit einer `RuntimeException` ein, wenn keine Deckung des Kontos vorhanden ist.

Was passiert nun, wenn dieser Aspekt vom Compiler übersetzt wird? Die Aspekte werden in die Konto-Klasse *eingewebt* (*weaving*), d. h., um den Aufruf der `einzahlen()`- und `abheben()`-Methoden wird der entsprechende Advice in die Class-Datei eingebunden. Stellen Sie sich einfach vor, dass der AspectJ-Compiler den Joinpoint wie ein Präprozessor (tatsächlich arbeitete der Compiler in der Anfangsphase so) durch den Advice ersetzt (vgl. Abb. 1.5 auf der nächsten Seite).

Damit hätten wir die Fehlerbehandlung abgeschlossen. Sollten noch weitere Anforderungen wie die Überprüfung *aller* öffentlichen Konto-Methoden auf Überziehung des Kontostandes hinzukommen, so kann auch dies zentral über einen Aspekt abgewickelt werden:

```
pointcut kontoMethods():
    call(public * bank.Konto.*(..));

after(): kontoMethods() {
    if (((Konto)thisJoinPoint.getTarget()).kontostand < 0) {
        throw new RuntimeException("Konto ueberzogen");
    }
}
```

Nach allen Konto-Methoden, die »public« sind (also alle Methoden, die von außen aufgerufen werden können), wird der Kontostand überprüft, ob er nicht ins Minus sinkt. Sollten Sie eine etwas liberalere Politik bei der Kontoüberziehung fahren – auch kein Problem. Sie ändern nur den entsprechenden After-Advice ab, that's all.

Nach dieser kleinen Abschweifung kehren wir wieder zurück zu den ursprünglichen Forderungen. Es sollen alle wichtigen Konto-Aktionen mitprotokolliert werden. Kein Problem, wir lassen uns einfach den Joinpoint am Anfang und Ende jeder Methode ausgeben:

Logging über
System.out

```
pointcut executePublicMethods() :
    execution(public * bank.Konto.*(..));

before() : executePublicMethods() {
    System.out.println("Start von " + thisJoinPoint);
}

after() : executePublicMethods() {
    System.out.println("Ende von " + thisJoinPoint);
}
```

⁵Damit dieses Beispiel funktioniert, wurde die Sichtbarkeit des Kontostand-Attributs auf *protected* verringert und der Aspekt ebenfalls im `bank`-Paket untergebracht.

Wir könnten statt des Joinpoints auch den Kontostand oder die übergebenen Parameter ausgeben. Aber für ein erstes Beispiel soll es an dieser Stelle genug sein. Später (so etwa ab Kapitel 3.6 auf Seite 65) werden wir dann noch weitere Möglichkeiten kennen lernen, wie man einem Joinpoint weitere Informationen entlocken kann.

Wir werden noch einige Male auf dieses Beispiel zurückkommen. In den folgenden Kapiteln wollen wir uns zunächst aber die Grundkenntnisse und die Denkweise von aspektorientierter Programmierung aneignen und ausbauen.

1.3 Weaving

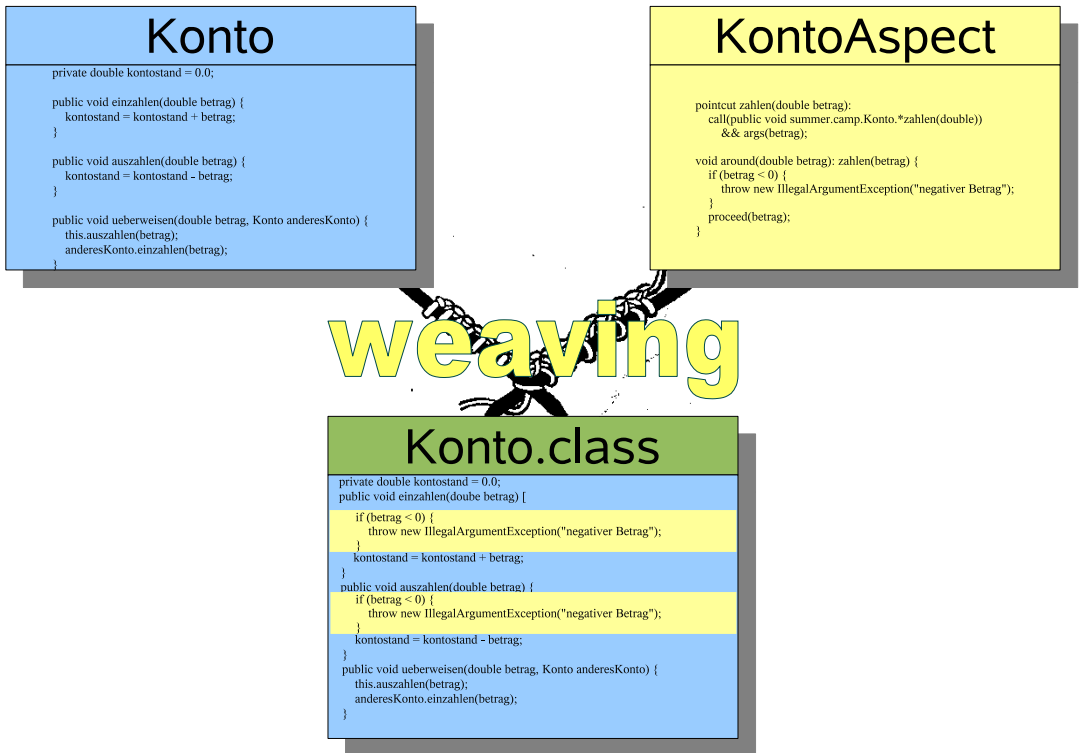


Abbildung 1.5
Der Weaving-Prozess

Aspektweber fügt Code
in Class-Datei ein

Während der Compiler am Anfang noch als Präprozessor diente, arbeitet er inzwischen als echter Compiler und fügt über den *Aspektweber* (engl. *Aspect Weaver*) die Aspekte direkt in den Java-Bytecode (mit Hilfe des Java-Compilers) ein.

Compile-Time Weaving (CTW)

Der AspectJ-Weber kennt sowohl statische als auch dynamische Joinpoints. Statische Joinpoints werden während des Kompilierungszeitpunktes ausgewertet und der Advice-Code an die entsprechenden Stellen im Byte-Code eingewebt. Dadurch ist der Laufzeitverlust während der Ausführung des Programms verschwindend gering (lediglich der Aufruf des zusätzlichen Codes).

Nicht alle Joinpoints können statisch vom Compiler ausgewertet werden. So können Joinpoints mit Bedingungen verknüpft werden, die erst zur Laufzeit ausgewertet werden können. In diesem Fall wird neben dem eigentlichen Advice-Code auch dieser Code zur Überprüfung der Bedingung in den Code mit eingewebt. Nachteile hierbei sind ein erhöhter Speicherverbrauch und ein etwas schlechteres Laufzeitverhalten, da die Auswertung des Joinpoints zur Laufzeit stattfindet.

AspectJ 5: Load-Time Weaving (LTW)

Eine weitere interessante Möglichkeit ist das *Load-Time Weaving (LTW)*, das mit AspectJ5 Einzug hält. Damit werden Aspekte über den Classloader eingewebt. Damit können z. B. fremde Bibliotheken mit zusätzlichen Log-Ausgaben angereichert werden, ohne dass sie angefasst werden müssen. Alternativ kann der AspectJ-Compiler auch direkt auf Class- oder Jar-Dateien arbeiten und Aspekte einweben (Post-Compile-Time Weaving).

1.4 Was bedeutet AOP?

OO-Modelle

In der Objektorientierung versucht man, die Wirklichkeit (genauer: den Bereich, den man gerade betrachtet, den so genannten »Problem Domain«) in Form von Klassen, die zueinander in Beziehung stehen, in einer geeigneten Software-Architektur abzubilden. Dinge der realen Welt wie »Person« oder Begriffe aus dem betrachteten Gebiet wie »Konto« finden sich dabei als entsprechende Klasse im System wieder.

Leider ist dieses Modell, das man sich anfangs von der Wirklichkeit erstellt hat, nur ein Idealbild, das im Laufe der Entwicklung durch technische Randbedingungen und geänderte Vorgaben immer mehr schwimmt. Die Anfangsarchitektur wird zusehends mit zusätzlichen Anforderungen wie Persistenz, Autorisierung, Sicherheit, Logging, Transaktionen oder Skalierbarkeit überwuchert, die querbeet fast sämtliche Klassen durchziehen.

1.4.1 Crosscutting Concerns (Querschnittsbelange)

Concern = Anliegen

Bevor ich auf *Crosscutting Concerns* näher eingehe, wollen wir uns mit dem Begriff *Concern* beschäftigen, weil er einer der zentralen Begriffe in AOP darstellt, der immer wieder auftaucht. Der Begriff »Concern« ist

in der Informatik schon länger bekannt und steht für ein Anliegen oder ein Merkmal, das man durch das geplante Softwaresystem abdecken will. So kann ein Concern ein bestimmtes Ziel, ein Konzept, ein Interessengebiet, eine Anforderung oder Ähnliches sein. In unserem Beispiel ist ein Concern die Kontoverwaltung mit allem Drum und Dran wie Einzahlung, Abhebung, Kontostandabfrage.

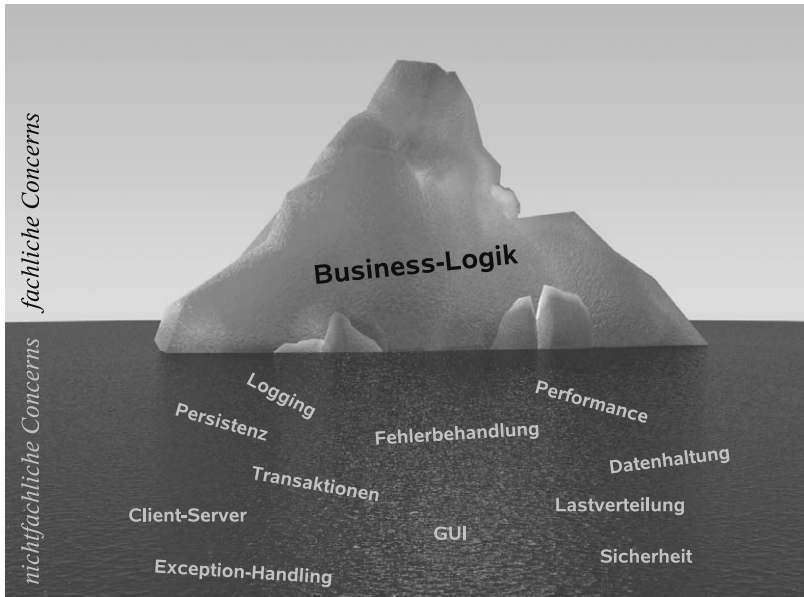


Abbildung 1.6
Die Spitze des Eisbergs

Bei der Entwicklung von Softwaresystemen spielen nicht nur *fachliche Concerns* wie Kontoverwaltung, Überweisungen oder Zinsberechnung (also das, was gemeinhin als die Geschäftslogik bezeichnet wird) eine Rolle. Oftmals sind es die *nicht fachlichen Concerns* wie Logging, Persistenz, Sicherheit oder Lastverteilung, die einen nicht unerheblichen Teil der Entwicklung verschlingen (s. Abb. 1.6).

Mit der Devise »divide et impera« (»teile und herrsche«) versucht man der steigenden Komplexität dadurch zu begegnen, dass man jeden Concern in eine eigene Klasse oder ein eigenes Modul kapselt (auch als »Separation of Concerns« bekannt). In der Praxis klappt dieser Ansatz aber leider nicht immer. Schauen wir uns dazu eine Untersuchung aus den Parc-Forschungslabors⁶ an.

Fachliche Concerns =
Geschäftslogik

»Divide et impera«
(Julius Cäsar)

⁶s. <http://www.parc.com/research/csl/projects/aspectj/default.html>

Abbildung 1.7
XML-Parsing in Tomcat

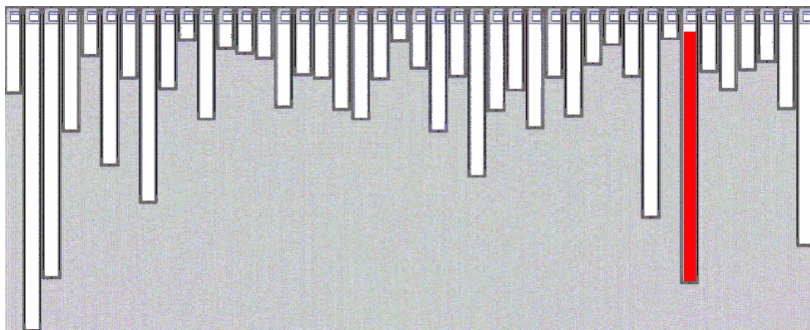
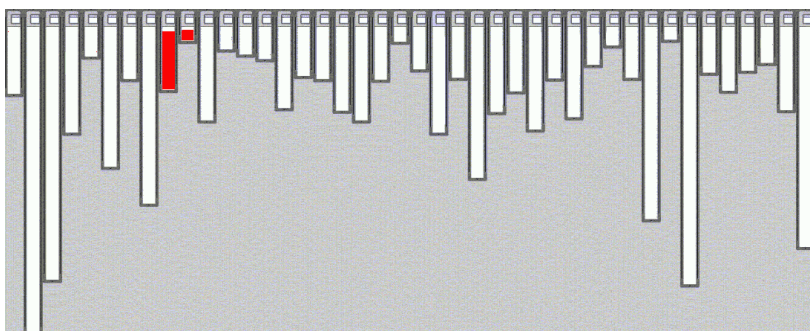


Abbildung 1.7⁷ gibt einen Überblick der einzelnen Tomcat⁸-Module. Programmteile, die für die XML-Verarbeitung zuständig sind, wurden dabei durch eine dunkle Färbung hervorgehoben. Wie man sieht, beschränkt sich dieser »Concern« auf genau ein Modul. Auch die URL-Erkennung (URL pattern matching) ist gut modularisiert und über Unterklassenbildung auf zwei Module verteilt (s. Abb. 1.8).

Abbildung 1.8
URL-Erkennung in
Tomcat



Ganz anders sieht es dagegen aus, wenn man das Logging anschaut, das über alle Module verteilt ist (s. Abb. 1.9 auf der nächsten Seite).

Ein weiteres Beispiel ist die Session-Verwaltung. Auch hier finden sich Fragmente dieses »Concerns« über viele Module verteilt. Man bezeichnet solche Concerns, die sich quer durch die Anwendung ziehen und meist erfolgreich der Modularisierung mit herkömmlichen Mitteln widersetzen, auch als »*Crosscutting Concerns*«.

⁷ entnommen aus <http://www.parc.com/research/cs1/projects/aspectj/downloads/00PSLA2002-demo.ppt>

⁸ eine freie Servlet-Implementierung von Apache.org, s. <http://jakarta.apache.org>

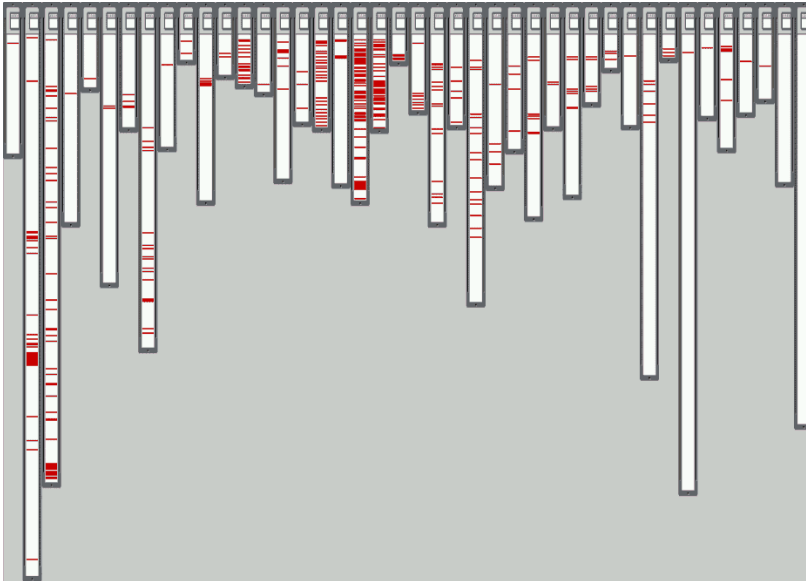


Abbildung 1.9
Logging in Tomcat

1.4.2 Probleme

Herkömmliche Herangehensweisen an diese Crosscutting Concerns haben meistens mit folgenden beiden Problemen zu kämpfen:

- ❑ Software-Module müssen mehrere Anforderungen auf einmal abdecken. Der Entwickler muss gleichzeitig auf Dinge wie Performance, Synchronisierung und Transaktionen achten, ohne die Geschäftslogik aus den Augen zu verlieren.
- ❑ Crosscutting Concerns finden sich in fast jeder Klasse wieder und sind über das ganze System verstreut.

Diese Probleme haben zwangsläufig Auswirkungen auf das Software-Design und den Entwicklungsprozess in mehrfacher Hinsicht:

- ❑ schlechte Nachvollziehbarkeit:
Wenn man mehrere Dinge gleichzeitig implementieren muss, geht damit der Zusammenhang zwischen einem Concern und seiner Realisierung verloren.
- ❑ niedrigere Produktivität:
Wenn man gleichzeitig mehrere Dinge im Auge behalten muss, kann man sich nicht auf ein Problem konzentrieren. Dadurch leidet auch die Produktivität.

- ❑ kaum Wiederverwendung:
Da ein Modul verschiedene andere Belange implementiert, sinkt die Wahrscheinlichkeit, dass dieses Modul auch in anderen Projekten verwendet werden kann, trotz ähnlicher Anforderungen.
- ❑ verminderte Code-Qualität:
Durch die Wechselwirkung der einzelnen Anforderungen kann es zu versteckten Problemen kommen. So können z. B. Performance-Anforderungen eine Caching-Lösung hervorbringen, die Auswirkung auf die Persistenz-Implementierung hat.
- ❑ erschwerte Weiterentwicklung:
Neue Anforderungen wie Client-Server-Architektur können Auswirkungen auf die gesamte Architektur und die bereits implementierten Concerns haben.

Diese Probleme sind ja nicht ganz neu, und es gibt einige Lösungsansätze dafür. So bieten Programmiersprachen wie Python die Möglichkeit von Mix-in-Klassen, die das nachträgliche Einfügen zusätzlicher Funktionalität ermöglichen. Unterklassenbildung ist eine weitere Möglichkeit, mit der fehlende Concerns nachgeschoben werden können.

Domain-spezifische Frameworks wie Enterprise JavaBeans (EJB) versuchen einige immer wiederkehrende Querschnittsbelange wie Verteilung, Skalierbarkeit, Sicherheit, Persistenz und Transaktionen zu adressieren und in Form von Konfigurationsdateien auszulagern. Allerdings sind Frameworks nur für ein bestimmtes Einsatzgebiet ausgelegt und können damit zwangsläufig nicht alle gewünschten Belange abdecken.

1.4.3 Die Quadratur der Architektur

Das »San Francisco«-Syndrom

Gute Softwarearchitekten versuchen stets, bereits die Anforderungen von morgen in das System-Design mit einzubeziehen. Und genau hierin liegt das Problem vieler Systeme: Was sind die Anforderungen von morgen? Und wie allgemein soll die Lösung gehalten werden, um auch Unvorhergesehenes abdecken zu können?⁹

Über die Anforderungen von morgen kann man nur spekulieren. Aber wie sieht die Welt morgen oder gar übermorgen aus? So mutmaßte 1946 Thomas Watson, der damalige IBM-Chef, dass es weltweit einen Bedarf von vielleicht fünf Computern geben werde, 1949 vermutete die US-Zeitschrift *Popular Mechanics*, dass die Computer der

⁹In den 90er Jahren war »San Francisco« ein sehr ambitioniertes Java-Framework von IBM, das als Basis für viele Geschäftskomponenten dienen sollte. Es litt allerdings darunter, dass es zu allgemein gehalten war und dadurch in der Anwendung umständlich wurde. Inzwischen ist Stille eingekehrt.

Zukunft vielleicht nur noch 1,5 Tonnen wiegen werde und Bill Gates bemerkte 1981, dass 640 KByte Arbeitsspeicher alles sei, was irgendeine Applikation jemals benötigen sollte.

Versucht man die künftigen Anforderungen bereits heute zu berücksichtigen, riskiert man ein System, das unnötigen Ballast mit sich schleift, da viele Vorhersagen sich genauso als falsch erweisen werden wie der Siegeszug des Netzcomputers (»Thin Client«) in den 90er Jahren. Lässt man die Zukunft außer Acht, kann man auf neue Einsatzszenarien eventuell nicht schnell genug reagieren. Wie viel Flexibilität muss man vorsehen, um künftige Erweiterungen zu berücksichtigen? Wie viel Design ist zu viel Design?

Das Dilemma

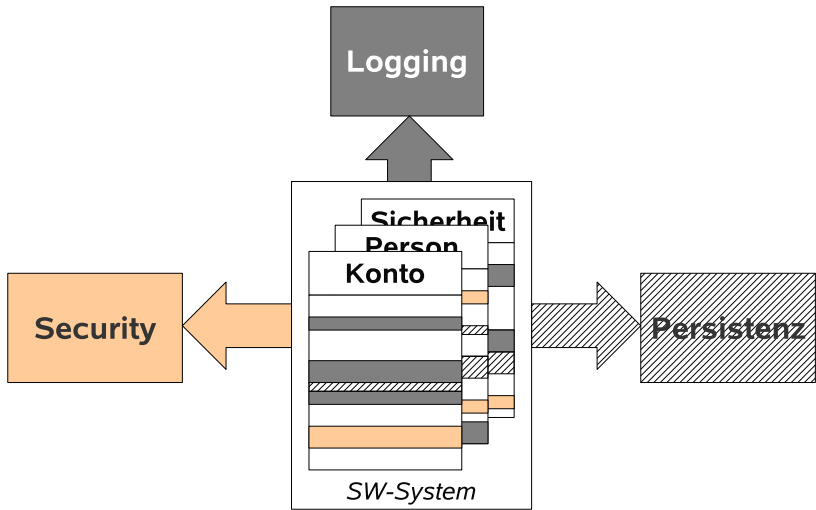
1.4.4 Die Antwort von AOP

Die Antwort von AOP auf diese Fragen besteht darin, auch Querschnittsbelange zu modularisieren. Während Objektorientierung sich vorwiegend auf die Geschäftslogik fokussiert und das Klassenkonzept als Architekturbasis bietet, konzentriert sich die Aspektorientierung auf die Aufteilung in verschiedene »Concerns«, die getrennt in »Aspekten« abgelegt und verknüpft werden können. Üblicherweise sind es drei Entwicklungsschritte, die AOP kennzeichnet:

1. **Aspekt-Zerlegung:**
Die Anforderungen werden aufgeteilt in allgemeine Belange (das, was man üblicherweise unter Geschäftslogik versteht) und Querschnittsbelange wie Logging, Security oder Persistenz (vgl. Abb. 1.10 auf der nächsten Seite).
2. **Concern-Implementierung:**
Nach der Identifizierung der einzelnen »Concerns« werden sie getrennt implementiert.
3. **Zusammenführung:**
Die einzelnen »Concerns« werden wieder zusammengeführt. Dies ist die Aufgabe des *Weavers*, dessen Funktionsweise wir bereits in Kapitel 1.3 auf Seite 15 kennen gelernt haben.

So wie bereits OOP auf die prozedurale Programmierung aufsetzt (es gibt nach wie vor Prozeduren bzw. Funktionen, auch wenn sie jetzt »Methoden« heißen), wird AOP die Objektorientierung nicht über Bord werfen. Aber die Sichtweise wird sich ändern. Bei OOP traten die Prozeduren zugunsten der Daten (bzw. des »Objekts«) in den Hintergrund, bei AOP werden es die »Concerns« sein, die stärker in den Vordergrund treten werden. Sie müssen also Ihre objektorientierte Vergangenheit nicht ablegen, sondern können darauf aufbauen.

Abbildung 1.10
SW-Belange



1.4.5 Anatomie von AOP-Sprachen

Neue Paradigmen bringen neue Programmiersprachen hervor, um die neuen Konzepte besser zu unterstützen. So wie man mit C auch objektorientiert programmieren kann (es ist zwar ein wenig mühsam, aber es geht), kann man auch mit objektorientierten Sprachen wie Java aspektorientiert programmieren (z. B. wie in AspectWerkz, wo mit XML-Dateien gearbeitet wird)¹⁰, aber die beste Unterstützung erhält man durch die Verwendung von aspektorientierten Sprachen.

Davon gibt es inzwischen einige, wie z. B. AspectJ¹¹, AspectC++¹² oder Apostle (einer Smalltalk-Erweiterung)¹³. Für die Unterstützung von AOP bieten sie:

- ❑ Implementierung von Concerns: Die einzelnen Anforderungen werden in *Advices* abgelegt, die ähnlich wie Funktionen in traditionellen Programmiersprachen wie C, C++ oder Java aufgebaut sind.
- ❑ Weaving-Regeln: bestimmen, an welchen Stellen die Concerns eingeschleust werden sollen.

Der Compiler kann dabei verschiedene Strategien anwenden, um aus diesen beiden Bestandteilen den fertigen Code zu generieren: Er kann als Präprozessor tätig werden (wie es z. B. bis AspectJ 1.0 der Fall

¹⁰<http://aspectwerkz.codehaus.org/>

¹¹<http://www.eclipse.org/aspectj>

¹²<http://www.aspectc.org/>

¹³<http://www.cs.ubc.ca/labs/spl/projects/apostle/>

war), er kann ihn direkt in den gewünschten (Byte-)Code übersetzen (ab AspectJ 1.1 wird dazu der Java-Compiler verwendet), oder aber eine entsprechende Aspect-VM übernimmt das Weaving beim Laden der Klassen. Diesen Ansatz verwendet z. B. JBoss-AOP¹⁴ über einen eigenen Classloader.

In diesem Buch werden wir AspectJ als Sprache kennen lernen, die auf Java aufsetzt und es um aspektorientierte Sprachkonstrukte erweitert. Im Januar 2005 gab es eine Ankündigung, dass AspectJ mit AspectWerkz zusammenwachsen und sich ergänzen wird¹⁵. An den Konzepten hinter AOP und AspectJ wird sich dadurch nichts ändern, dafür werden sich Aspekte künftig vielleicht auch in XML formulieren lassen.

1.4.6 Das Versprechen von AOP

AOP bietet Konzepte an, um die über die gesamten Sourcen verteilten Belange herauszuziehen und an zentraler Stelle zu implementieren. Dadurch ergeben sich folgende Vorteile:

- + Modulare Implementierung von Querschnittsbelangen:
AOP erlaubt es, jeden Concern losgelöst von anderen Concerns implementieren zu können. Man muss nicht jedesmal den gleichen Code an den verschiedenen Stellen im Programm einfügen, so dass auch unnötige Redundanz vermieden werden kann.
- + Einfachere Erweiterungen:
Dadurch, dass die einzelnen Module sich nicht mehr um verschiedene Concerns kümmern müssen, können auch neue Module schneller entwickelt werden.
- + Hinauszögern von Design-Entscheidungen:
Viele Design-Entscheidungen müssen in konventioneller Entwicklung relativ früh während der Entwicklungsphase getroffen werden, um die verschiedenen Belange wie Sicherheit, Logging oder Autorisierung unter einen Hut zu bringen. Dies ist mit AOP nicht mehr nötig, sondern kann relativ spät erfolgen und noch später ohne viel Aufwand wieder geändert werden.
- + Erhöhte Wiederverwendung:
Wenn ein Modul nur noch einen einzigen Concern implementiert und nicht mehr von verschiedenen anderen Querschnittsbelangen durchsetzt ist, wächst damit zwangsläufig auch der Grad der Wiederverwendbarkeit.

¹⁴<http://www.jboss.org/products/aop>

¹⁵<http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout-/aspectj-home/aj5announce.html>

Manche Probleme lassen sich zwar durch den Einsatz von Frameworks oder Sprachmerkmale wie dynamische Proxys entschärfen, aber zu welchem Preis? Denken Sie an EJBs – es adressiert einige allgemeine Themen wie Persistenz oder Transaktionen, ist aber nicht einfach zu beherrschen und nur für einen begrenzten Anwendungsbereich (Client-Server-Architekturen) interessant. AOP kommt hier oft mit einfacheren Mitteln zum gleichen Ziel, ohne dass es auf ein bestimmtes Gebiet eingeschränkt wäre.

1.5 Sprechen Sie Aspekt-Orientalisch?!

Eine große Hürde beim Einstieg in AspectJ sind die vielen neuen Begriffe, die durch AOP eingeführt werden. Einige haben wir bereits kennen gelernt, viele werden noch folgen. Die wichtigsten Grundbegriffe sind in diesem Abschnitt zusammengefasst, um Ihnen einen schnellen Einstieg in die neue Welt zu ermöglichen.

Noch ist Aspektorientierung relativ unbekannt, und viele Bücher, Artikel und Foren im Internet sind nur auf Englisch erhältlich. Ich werde daher auch die englischen Begriffe verwenden, um Ihnen das Stöbern und Mitreden zu erleichtern. Sofern sich ein deutscher Begriff eingebürgert hat, werde ich diesen in Klammern hinzufügen. Meist hilft die deutsche Übersetzung aber auch nicht weiter, wenn man nicht weiß, was sich in AOP hinter einem Begriff verbirgt.

1.5.1 Concern und Crosscutting Concern

Wir haben »Concern« und »Crosscutting Concern« bereits in Kapitel 1.4.1 auf Seite 16 kennen gelernt. Der Vollständigkeit halber und weil es einer der wichtigsten Begriffe in AOP ist, hier nochmals die Erklärung:

Concern ist ein Anliegen oder eine Anforderung – üblicherweise das, was man auch als *Geschäftslogik* bezeichnet, z. B. eine Konto-Verwaltung mit Einzahlung, Abhebung und Überweisung.

Crosscutting Concerns sind Querschnittsbelange, die sich quer durch das gesamte System ziehen und mit konventionellen OO-Sprachen schlecht oder gar nicht modularisiert werden können. Beispiele für Querschnittsbelange sind Logging, Persistenz oder Autorisierung.

1.5.2 Aspekt

Ähnlich wie die Klasse der Grundstein für die Objektorientierung ist, ist dies der *Aspekt* für die Aspektorientierung. Aspekte sind eine Erweiterung des Klassen-Konzepts und die Grundlage für die Implementierung der Crosscutting Concerns. Von der Bedeutung her sind sie in etwa vergleichbar mit dem Übergang von »struct« zu »class« in C++.

Anders ausgedrückt: Ein Aspekt ist der Behälter, in dem Dinge wie *Pointcuts* und *Advices* (s. Abschnitt 1.5.4 und 1.5.5 auf der nächsten Seite) definiert werden. Er kann aber auch ganz normalen Java-Code enthalten.

*Aspekt = Erweiterung
des Klassen-Konzepts*

1.5.3 Joinpoint (Verbindungspunkt)

Ein Joinpoint ist ein (Zeit-)Punkt im Programm, der über Advices (s. Kapitel 5 auf Seite 117) erweitert oder modifiziert werden kann. Joinpoints können sein:

- Aufruf einer Methode,
- Ausführen einer Methode,
- Behandeln einer Exception,
- Zugriff auf eine Variable,
- Initialisierung einer Klasse oder eines Objekts.

*Joinpoint = Ereignis im
Programmfluss*

Beispiele für Joinpoints

Schauen wir uns dazu ein einfaches Beispiel an:

```
...
Konto konto = new Konto();
konto.einzahlen(500.0);
...
```

In Abb. 1.11 auf der nächsten Seite finden Sie einige Joinpoints zu diesen beiden Zeilen aufgelistet. Joinpoints sind zwar meist statischer Natur (d. h. eine bestimmte Stelle im Quellcode), aber nicht nur. Sie können auch von der Aufrufhierarchie abhängig sein oder mit Bedingungen verknüpft werden.

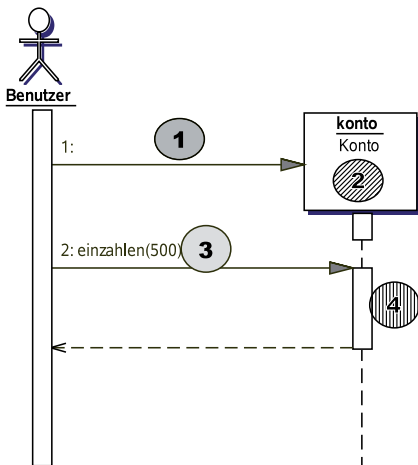
1.5.4 Pointcut (Schnittpunkt)

Ein Pointcut repräsentiert einen oder mehrere Joinpoints. Im einfachsten Fall kann dies wie in Abbildung 1.12 auf der nächsten Seite der Pointcut P2 ein einzelner Joinpoint sein, z. B. der Aufruf des Konto-Konstruktors. Es kann aber auch der Aufruf aller Konstruktoren sein oder die Ausführung aller Methoden der Konto-Klasse:

*Pointcut = Menge von
Joinpoints*

```
pointcut P3() :
    execution(public * bank.Konto.*(..));
```

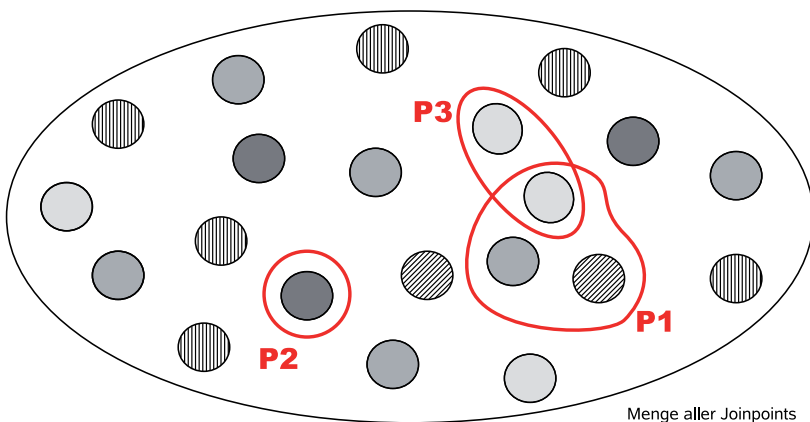
Abbildung 1.11
Einige Joinpoints



- (1) Konstruktor aufrufen
- (2) Objekt initialisieren
- (3) Methode aufrufen
- (4) Methode ausführen

Wie man an den Pointcuts P1 und P3 in Abbildung 1.12 sieht, können sich Pointcuts auch überschneiden und dieselben Joinpoints beinhalten.

Abbildung 1.12
Verschiedene
Joinpoints und
Pointcuts



Etwas komplexer wird es, wenn der Pointcut neben der Ortsangabe noch eine zeitliche Komponente beinhaltet. So gibt es Pointcuts, die erst zur Laufzeit bestimmt werden können. Wir werden noch sehen, dass Pointcuts nicht nur vom Aufruf einer Methode, sondern auch vom Zielobjekttyp des Aufrufs abhängen können, wobei die Typüberprüfung erst zur Laufzeit stattfindet.

1.5.5 Advice (Empfehlung)

Advice = Code

Ein Advice ist eine Aktion, die bei Erreichen eines Joinpoints ausgeführt werden soll. Dies kann z. B. der Aufruf einer Log-Methode sein:

```
before() : P3() {
    System.out.println("Start von " + thisJoinPoint);
}
```

Der Advice kann dabei wie hier in diesem Beispiel vor einem Joinpoint ausgeführt werden (Before-Advice), er kann aber auch nach (After-Advice) oder statt (Around-Advice) des Joinpoints ausgeführt werden. Zusammen mit dem Pointcut bildet er die Grundlage, um Crosscutting Concerns in den Griff zu bekommen.

1.5.6 Introduction

Neben den mehr dynamischen Pointcuts gibt es mit den *Introductions* auch rein statische Systemerweiterungen. Über Introduction lassen sich zusätzliche Attribute und Methoden in vorhandene Klassen einfügen, ohne dass diese Klasse im Original verändert werden muss (ideal, um bestehende Bibliotheken aufzubohren):

*Introduction =
Erweiterung von
Klassen*

```
public double bank.Konto.freistellungsbetrag;
```

Auf diese Art wird in Konto ein Attribut »freistellungsbetrag« eingeführt, ohne dass die Konto-Klasse selbst angefasst werden muss. Als *Open Classes* hat sich diese Möglichkeit in anderen Programmiersprachen (z. B. Python) schon länger bewährt.

1.6 Zusammenfassung

- ❑ Am Anfang gab es nur Assembler und Maschinensprache.
- ❑ Es folgten prozedurale und objektorientierte Sprachen, mit denen man sich zunehmend von der Arbeitsweise des Prozessors löste.
- ❑ Grenzen der Modularisierung und Objektorientierung zeigen sich bei Querschnittsbelangen (Crosscutting Concerns).
- ❑ Über Aspekte können solche Querschnittsbelange herausgezogen und zentral verwaltet werden.
- ❑ AOP baut auf OOP auf und bringt neue Begriffe und Konzepte mit. AspectJ baut auf Java auf und ist eine Programmiersprache für AOP.
- ❑ AOP-Entwicklung teilt sich auf in
 - ❑ Aspekt-Zerlegung
 - ❑ Concern-Implementierung
 - ❑ Zusammenführung (dies übernimmt der Aspekt-Weaver)

- ❑ Die wichtigsten AOP-Begriffe sind:
 - ❑ Joinpoints: stellen ein Ereignis wie z. B. das Ändern eines Attributs im Programmfluss dar.
 - ❑ Pointcuts: adressieren einen oder mehrere Joinpoints.
 - ❑ Advice: Code, der vor, nach oder anstatt eines Joinpoints ausgeführt wird.
- ❑ Weitere Begriffe sind: Aspekt (Behälter für Aspekt-Code), Introduction (Erweiterung bestehender Klassen).

1.7 Übungen

Am Ende jeden Kapitels (mit Ausnahme des letzten Kapitels) finden Sie einige Übungen, mit denen Sie den Stoff nochmals Revue passieren lassen sowie Ihr Wissen testen und vertiefen können. Damit es nicht ganz so schwer wird, finden Sie einen Vorschlag für die Lösung im Anhang D.3 auf Seite 394 und über die Webseite zum Buch¹⁶. Wenn Ihre Lösung davon abweicht, heißt das nicht, dass sie falsch sein muss – viele Wege führen nach Rom.

1. Schauen Sie unter de.wikipedia.org nach, was dort zum Thema »aspektorientierte Programmierung« zu finden ist. Gibt es dort auch einen Artikel zu AspectJ?
2. Googlen Sie nach weiteren interessanten Links zu AOP, die eine Einführung in dieses Thema bieten.
3. Schauen Sie sich Ihr aktuelles Projekt (oder ein anderes Projekt Ihrer Wahl) an. An welchen Stellen werden Exceptions abgefangen und »nur« geloggt? Wo befinden sich weitere Log-Anweisungen und wie groß ist in etwa ihr Anteil?
4. Für eine Roulette-Anwendung wird ein Bank-Modul benötigt, das den Kontostand eines Spielers verwaltet. Wie könnte so ein Bank-Modul aussehen? Implementieren Sie es in Java.
5. Schreiben Sie einige Testfälle für Ihre Anwendung, vorzugsweise mit JUnit¹⁷. Falls Sie mit JUnit noch nicht vertraut sind, schauen Sie sich das JUnit Cookbook¹⁸ von Kent Beck und Erich Gamma an oder besorgen Sie sich »Unit Tests mit Java« von Johannes Link [7]¹⁹.

¹⁶<http://www.aosd.de>

¹⁷<http://www.junit.org>

¹⁸<http://junit.sourceforge.net/doc/cookbook/cookbook.htm>

¹⁹Eines der besten Bücher zu diesem Thema – unbedingt lesen, auch wenn man sich bereits mit JUnit auskennt.