PROGRAMMER'S CHOICE

Alfred Zeitner
Birgit Linner
Martin Maier
Thorsten Göckeler

Spring 2.5

Eine pragmatische Einführung

3 Spring AOP

Auftraggeber: *»Ich habe ein externes Auditierungssystem. Ich möchte, dass einige Ereignisse der Applikation, z.B. wenn eine Bestellung reinkommt, dort angezeigt werden. Im Laufe der Zeit könnten wir auch andere Aktionen dort sichtbar machen. Welche das sein werden, kann ich jetzt aber noch nicht sagen.«*

Wir: *»Kein Problem, wir sind da flexibel.«*

Auftraggeber: *»Zusätzlich möchte ich, dass ich über manche Aufträge, die z.B. besonders viel Umsatz versprechen, per Mail verständigt werde. Es wäre auch praktisch, dass für andere Ereignisse eine Mail verschickt wird. Welche das sein sollen, kann ich jetzt noch nicht sagen. Wenn das System mal läuft, dann weiß ich, über welche Ereignisse ich informiert werden möchte.«*

Wir: *»Sie wünschen, wir spielen.«*

3.1 Motivation

Objektorientierung

Objektorientierung ist ein erfolgreiches Programmierparadigma. Kein Wunder – mit Objekten lässt sich die Realität eben am besten beschreiben. Alles um uns herum ist ein Objekt. Obwohl alle diese Objekte irgendwie anders sind, können doch viele zu Gruppen zusammengefasst werden, die Gemeinsamkei-

ten haben. Objekte einer solchen Gruppe haben ein ähnliches Aussehen und ähnliche Eigenschaften und Funktionalität.

Funktionalität wiederverwenden

Eine Programmiersprache dient dazu, Objekte aus der Realität beschreibbar zu machen. Wir beschreiben dabei nicht nur die Eigenschaften, sondern auch das Verhalten dieser Objekte. Java ist eine exzellente Sprache, in der es einfach ist, die komplexen Zusammenhänge dieser Objekte zu erfassen und zu definieren. Gleiche Funktionalität von Objekten fassen wir in einer Klasse zusammen, sinnvolles Design vorausgesetzt. Dadurch können wir jede Funktion an einer Stelle entwickeln, testen und warten. Alle Objekte, die mit dieser Klasse erzeugt werden, besitzen dieselbe Funktionalität. Wollen wir Teile davon in anderen Klassen wiederverwenden, gibt es dafür zwei Möglichkeiten:

- Ableitung einer Klasse
- Objektkomposition

Ableitung

Durch Ableitung einer Klasse können wir Funktionalität einer Klasse überschreiben und dadurch das Verhalten einer Klasse in Form einer Tochterklasse ändern. Wir ändern dabei aber die Tochterklasse selbst, ohne die Funktionalität der Objekte zu ändern, die mit der Mutterklasse erzeugt wurden. Objekte, die mit der Tochterklasse erzeugt werden, enthalten die neue Funktionalität und die der Mutterklasse, sofern sie nicht überschrieben wurde. In Java gibt es zusätzlich zur Vererbung auch das Konzept der Interfaces. Es erlaubt uns, von Klassen, die nicht in einer Vererbungshierarchie stehen, ähnliche Funktionalität zu erwarten – oder zumindest dieselbe Schnittstelle. Wir können daher Klassen auf dieselbe Art und Weise behandeln, unabhängig davon, ob sie voneinander ableiten oder nicht. Wir können jedoch nicht davon ausgehen, dass diese Klassen dasselbe Verhalten mitbringen. Wir wissen nur, dass sie sich gleich ansprechen lassen.

Objektkomposition

Objektkomposition bezeichnet die Technik, Objekte bestehender Klassen in einer Struktur einzubinden. Dabei wird einem Objekt eine Implementierung einer Funktionalität angegeben, ohne dass das Objekt selbst weiß, was dabei ausgeführt wird. Das Objekt kennt nur das Interface, an das es die Funktionalität delegiert. Ein einfaches Beispiel dafür ist die Verwendung der Klasse `StringTokenizer`. Der Stringtokenizer weiß, wie er einen String zu teilen hat. Verwendet nun eine Klasse diese Funktionalität, muss sie nicht etwa von `StringTokenizer` ableiten, sondern sie verwendet Objektkomposition. Dabei hält die Klasse eine Referenz und delegiert Funktionalität an den Stringtokenizer. Daraus folgt, dass Funktionalität in mehreren Klassen verwendet werden kann, ohne dass die Klassen voneinander ableiten müssen.

Limitierung herkömmlicher Mittel

Mit beiden Techniken kann Funktionalität über mehrere Klassen verteilt werden, ohne dass der Code redundant wird und an mehreren Stellen steht. Vererbung verteilt Funktionalität ohne Aufwand. Mit der Ableitung von einer Klasse werden automatisch alle vererbbaren Methoden in die neue Klasse übernommen. Wird eine Methode überschrieben, wird fortan die überschriebene Version der Methode weitergegeben. Wir können daher nur davon ausgehen, dass wir die Funktionalität der direkten Mutterklasse erhalten. Möchten wir die Funktionalität der Mutter der Mutterklasse beerben, haben wir in Java keine Möglichkeit, dies zu tun, wenn in der direkten Mutterklasse überschrieben wird. Zusätzlich können wir nur Methoden überschreiben, die auch als überschreibbar vorgesehen sind. Ist eine Methode `private` oder `final` oder ist eine gesamte Klasse `final`, sind wir in unseren Möglichkeiten begrenzt. Zugegeben, meist besitzen wir die Möglichkeit, die Mutterklasse zu ändern und so unsere Klassenhierarchie zu entwickeln. Verwenden wir Code von anderen Entwicklern, ist dies nicht immer so einfach.

Redundanter Code

Sehr oft haben wir in unserem Objektmodell mehrere Klassen, die über eine ähnliche Funktionalität verfügen, aber nicht voneinander ableiten sollen oder können. Andere Sprachen wie C++ erlauben Mehrfachvererbung. Java besitzt aus vielen guten Gründen diese Möglichkeit nicht. Ähnliches kann nur mit Interfaces erreicht werden. Die Anforderung, dass eine Klasse dieselben Methodennamen besitzt, kann in Java über Interfaces gelöst werden. Dabei wird aber nur dieselbe Schnittstelle garantiert, nicht dieselbe Implementierung. Sollen dann mehrere Klassen die gleiche Funktionalität enthalten, müssen diese redundant implementiert werden. Das sollte vermieden werden. Als einziges Mittel bleibt die Objektkomposition, die aber wiederum einige Nachteile besitzt: Beispielweise muss jedes Objekt ein neues Objekt einer anderen Klasse erzeugen oder sich eine Referenz zu diesem von irgendwo besorgen. Zusätzlich sind die Aufrufe dieser Delegationsobjekte immer noch redundant, auch wenn es sich nur um ein paar Zeilen handelt. Ändert sich die Signatur der Methode, müssen alle aufrufenden Stellen geändert werden. Sehen wir uns diese Problematik an einem einfachen Beispiel in Listing 3.1 an.

```
01 public class SimpleBestellService implements BestellService
02 {
03    private SecurityChecker checker;
04
05    public void bestelleBuch(String isbn)
06    {
07       checker.checkSecurity();
08       // Bestellcode
09    }
10
```

```
11      public void bestelleZeitschrift(String isbn)
12      {
13         checker.checkSecurity();
14         // Bestellcode
15      }
16
17      public void bestelleHeft(String isbn)
18      {
19         checker.checkSecurity();
20         // Bestellcode
21      }
22
23      public void setSecurityCheck(SecurityChecker checker)
24            throws SecurityException
25      {
26         this.checker = checker;
27      }
28  }
```

Listing 3.1: Eine Klasse mit redundanten Methodenaufrufen durch Objektkomposition

Objektkomposition ist wartungsaufwändig

Listing 3.1 zeigt das Prinzip eines Delegate[1], eine Form von Objektkomposition. Dabei wird in mehreren Methoden Funktionalität eines Delegate-Objekts aufgerufen. Bei jeder öffentlichen Bestellmethode wird die Sicherheit überprüft und für den aktuellen Benutzer evaluiert, ob er zur Bestellung berechtigt ist. Um diese Überprüfung nur an einer Stelle definieren zu müssen, wird sie an ein Objekt delegiert und in jeder Methode aufgerufen. Dadurch wird der Code nur an einer Stelle in der Klasse `SecurityChecker` definiert und kann somit wiederverwendet werden. Wir sehen schon anhand dieser Klasse, dass diese Form der Verteilung der Funktionalität nicht einfach zu warten ist. Erwartet hier beispielsweise die Implementierung von `SecurityChecker` zusätzlich den Namen des Benutzers, müssen wir den Code an jeder Stelle ändern. Außerdem kann es passieren, dass wir es übersehen, den Code in einer Methode aufzurufen. So könnte es passieren, dass eine Bestellung ohne Sicherheitsüberprüfung durchgeführt wird, was fatal wäre. Die Situation wäre ähnlich, wenn wir jeden Aufruf einer Bestellmethode in eine Protokollierungsdatei schreiben würden. Wir müssten ebenso in jeder Methode zumindest eine neue Zeile Code hinzufügen, um dies zu erreichen.

1 Das Prinzip eines Delegate findet sich in vielen Design Patterns wieder, wie beispielsweise dem Design Pattern Strategy. Mehr zu Design Patterns finden Sie in Gamma E., Helm R., Johnson R., Vlissides J., Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Professional, 1994.

Die Hauptaufgabe der Klasse `SimpleBestellService` in Listing 3.1 ist es, Bücher und andere Dinge zu bestellen. Alles andere ist keine Geschäftslogik und sollte daher auch nicht in der Verantwortung dieser Klasse liegen. Im besten Fall sollte der Bestellservice gar nicht wissen, dass Protokollierung oder Sicherheitsüberprüfungen stattfinden. Erreichen könnte man dies, indem man eine Wrapper-Klasse entwickelt, die das Interface `BestellService` implementiert und Aufrufe an das eigentliche Bestellservice delegiert. Doch auch in diesem Fall entfällt nicht der Wartungsaufwand bei Änderungen des Codes oder Refaktorierungen. Zusätzlich ist die Definition, an welchen Stellen die Sicherheit überprüft wird, wenig flexibel. Ändern sich die Anforderungen an den Bestellservice und soll die Sicherheit an anderen Stellen auch überprüft werden, muss derselbe redundante Code an jede Stelle extra kopiert werden.

Belange trennen – jede Klasse sollte nur eine Aufgabe haben

Genau für Situationen wie die oben beschriebene wurde aspektorientierte Entwicklung konzipiert. Dabei wird die zu verteilende Funktionalität an einer Stelle zusammengefasst und mit deklarativen Mitteln auf Objekte und Methoden angewendet. Im Fall der Klasse `SimpleBestellService` bedeutet das, dass für die Methode `checkSecurity` deklarativ definiert werden kann, wann sie aufgerufen wird. Diese Definition umfasst etwa alle Methoden, die mit »bestell« beginnen und sich in der Klasse `SimpleBestellService` befinden. Damit muss der Aufruf nicht mehr im Code selbst passieren, sondern ein Framework übernimmt diese Arbeit.

Die Lösung liegt in aspektorientierter Programmierung

Da unser Kunde von uns erwartet, dass die Applikation sicher ist und die notwendigen Berechtigungsprüfungen durchführt, müssen wir uns um das Thema Sicherheit kümmern. Auch wir wollen, dass nur berechtigte Benutzer auf unsere Applikation zugreifen dürfen. Da wir aber davon ausgehen, dass sich die Kundenwünsche ändern, möchten wir unser Sicherheitssystem so flexibel wie möglich gestalten. Kommen neue Benutzer oder Benutzergruppen hinzu, soll sich die Applikation ohne große Änderungen auf die neue Situation einstellen können. Auch wenn wir große Erweiterungen am System machen müssen, wäre es ideal, wenn wir uns um das Thema Sicherheit nicht zu kümmern hätten und uns auf die Geschäftslogik konzentrieren könnten. Aspektorientierte Programmierung kann hier hilfreich sein und uns eine elegante und leicht zu wartende Lösung anbieten. Darum wollen wir das Spring-eigene Framework für aspektorientierte Programmierung namens Spring AOP in diesem Kapitel etwas genauer ansehen.

Flexibilität durch Deklaration

3.2 Aspektorientierte Programmierung

Querschnittsbelange

Aspektorientierte Programmierung (AOP) ist eine Programmiertechnik, die es ermöglicht, verschiedene Aspekte eines Programms getrennt voneinander zu entwerfen, zu entwickeln und zu testen. Ein Aspekt ist dabei eine Teilfunktionalität eines Programms, die auf beliebige Klassen und deren Methoden angewendet werden kann, ohne dass diese voneinander ableiten oder Referenzen zueinander halten müssen. Aspekte kapseln »querschnittliche Belange« (im Englischen oft als »*cross-cutting concerns*« bezeichnet). Klassische Beispiele für solche Querschnittsbelange sind Sicherheit, Protokollierung oder Transaktionsmanagement. Diese bezeichnen Funktionalität, die in mehreren Modulen verwendet werden soll, aber so integriert ist, dass sie selbst nicht modularisiert werden kann. Aspektorientierung ist damit kein Ersatz für Objektorientierung, sondern setzt darauf auf und bietet neue Möglichkeiten im Design und in der Programmierung.

3.2.1 Begrifflichkeiten

Begrifflichkeiten

In der Welt der Aspektorientierung gibt es einige Begriffe, die wir vorab definieren sollten. Die Terminologie in der Welt von AOP ist leider nicht unbedingt intuitiv, aber sehr weit verbreitet und in allen gängigen AOP-Frameworks konsistent.

- **Aspekt**: Ein Aspekt ist eine Teilfunktionalität eines Programms wie z.B. Sicherheit. Eine Aspektklasse implementiert Aspektfunktionalität.
- **Advice**: Ein Advice definiert, welche Funktionalität eines Aspekts wo und in welcher Form aufgerufen wird. Bei Spring AOP werden Advices über Annotationen oder XML definiert. Ein Advice definiert beispielsweise, ob ein Aspekt vor oder nach einer Methode aufgerufen wird.
- **Joinpoint**: Advices werden auf bestehende Klassen angewendet. Advices können vor oder nach den Methoden dieser Klassen aufgerufen werden. Ein Joinpoint bezeichnet dabei die Stelle, an der ein Advice aufgerufen werden kann. Spring AOP kennt nur eine Art von Joinpoint: den Aufruf einer Methode. Andere AOP Frameworks wie AspectJ[2] können auch den Zugriff auf ein Feld als Joinpoint verwenden.
- **Pointcut**: Ein Pointcut ist eine Sammlung von Stellen, an denen Advices aufgerufen werden. Ein Pointcut ist eine Liste von Joinpoints. Ein Pointcut kann also beispielsweise alle Methoden der Klasse `BestellService` umfassen, die mit »bestell« beginnen.

2 AspectJ ist eine aspektorientierte Erweiterung für Java. Dokumentation findet sich unter *http://www.eclipse.org/aspectj/*.

- **Weaving**: Weaving ist der Prozess des Anwendens der Advices auf die Klassen einer Applikation. Dabei gibt es unterschiedliche Techniken, die vom AOP-Framework abhängig sind. Das sehr weit verbreitete AOP-Framework AspectJ verwendet Compile-Time-Weaving. Spring AOP verwendet Runtime-Weaving mithilfe von Proxy-Objekten.

3.2.2 Spring AOP

Spring AOP ist eine der Schlüsselkomponenten im Spring Framework. Der »Inversion of Control«-Container von Spring ist nicht von Spring AOP abhängig, wird aber intensiv für einige Spring-Services wie z.B. Transaktionsmanagement eingesetzt. Spring AOP bietet uns als Entwickler gute und vor allem einfache Möglichkeiten, aspektorientiert zu programmieren. Spring AOP wird in Teilen des Spring Frameworks intern selbst eingesetzt, wie beispielsweise für Transaktionsmanagement.

Spring AOP verwendet Proxy-Objekte

Spring AOP ist seit den Anfängen Teil des Spring Frameworks und basiert vollständig auf Java. Andere AOP Frameworks wie z.B. AspectJ verwenden eigene Classloader und/oder eigene Compiler, um Aspekte auf bestehende Klassen anzuwenden. Spring AOP basiert auf dem Ansatz des dynamischen AOP, bei dem Proxy-Objekte gebildet werden, die Methodenaufrufe für die eigentlichen Objekte entgegennehmen. Die Klassen, die wir entwickeln, bleiben bei Spring AOP damit unverändert und funktionieren in jeder herkömmlichen Java-Umgebung.

Proxy-Mechanismus

Ein Aufruf auf ein Objekt

Um Spring AOP erfolgreich einsetzen zu können, ist es wichtig, den Proxy-Mechanismus von Spring AOP gut zu verstehen, obwohl die Details der Implementierung für den Entwickler transparent sind. In Abbildung 3.1 sehen wir ein einfaches Java-Objekt `bean`, das über die BeanFactory instanziiert wurde. Die Methode `machWas` liefert einen Rückgabewert und der aufrufende Code nimmt diesen entgegen.

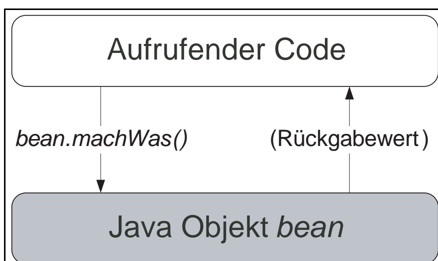

Abbildung 3.1: Ein Aufruf einer Methode auf ein Java-Objekt

Ein Aufruf auf ein Proxy-Objekt

Wird Spring AOP verwendet, werden für die Beans der BeanFactory Proxy-Objekte gebildet. Für jede Bean-Instanz gibt es ein zusätzliches Objekt im Speicher. Wird eine Bean von der BeanFactory mit `getBean` oder über Dependency Injection angefordert, gibt die BeanFactory nicht das instanziierte Objekt der konfigurierten Klasse zurück, sondern das Proxy-Objekt. Dieses sieht für den Client der BeanFactory wie das erwartete Objekt aus, indem es dieselben Methoden bereitstellt. Durch dieses Proxy-Objekt hat Spring AOP die Möglichkeit, vor oder nach dem Methodenaufruf des ursprünglichen Objekts Code auszuführen. Dieser Ablauf ist in Abbildung 3.2 dargestellt und wie folgt im Detail beschrieben.

1. Der Client der BeanFactory fordert ein Objekt mit dem Namen »bean« von der BeanFactory an, erhält ein Proxy-Objekt, castet auf den richtigen Typ und ruft die Methode `machWas` auf.
2. Der Aufruf am Proxy-Objekt ermöglicht Spring AOP die Ausführung von Aspekt-Code, den wir selbst entwickeln können und so konfiguriert haben, dass er an dieser Stelle aufgerufen wird. Die Parameter des Aufrufs sind im Aspekt-Code bekannt und könnten, falls gewollt, verändert werden.
3. Spring AOP ruft die Methode `machWas` am ursprünglichen, unveränderten Objekt auf.
4. Spring AOP nimmt den Rückgabewert des Objekts entgegen.
5. Wie in Punkt 2 können wir hier Code ausführen lassen und gegebenenfalls den Rückgabewert verändern. Wir können auch Exceptions werfen, wenn wir wollen.
6. Das Proxy-Objekt liefert den Rückgabewert an den aufrufenden Code zurück, für den der Aufruf des Aspekt-Codes transparent ist.

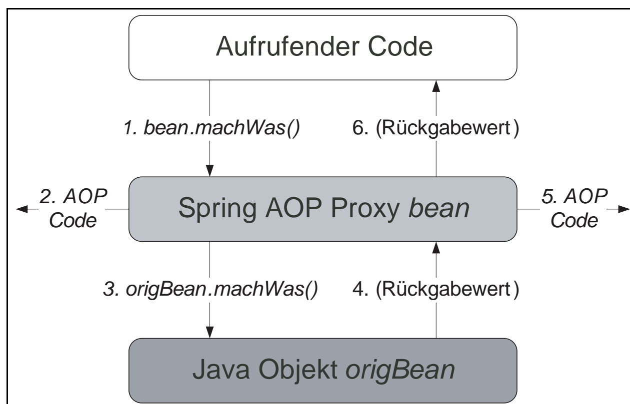

Abbildung 3.2: Proxy-Objekte werden den eigentlichen Beans vorgeschaltet.

Erzeugung von Proxy-Objekten

Zwei Arten der Bildung von Proxy-Objekten

Ein wichtiger Punkt in der Funktionalität des Proxy-Mechanismus ist, wie die Proxy-Objekte gebildet werden. Dafür gibt es bei Spring zwei Möglichkeiten, die von der Beschaffenheit der Bean-Klasse abhängen:

- Proxy-Bildung über die Klasse `java.lang.reflect.Proxy`
- Erzeugung des Proxy-Objekts mittels der Code Generation Library (cglib)

Proxy-Objekt mit `java.lang.reflect.Proxy`

Implementiert die Klasse, von der das Proxy-Objekt gebildet werden soll, ein Interface, wird ein Proxy-Objekt mit der Klasse `java.lang.reflect.Proxy` gebildet, das alle Interfaces des ursprünglichen Objekts implementiert. Somit kann das Proxy-Objekt auf jedes dieser Interfaces gecastet werden und dessen Methoden können aufgerufen werden. Der Aufruf

```
System.out.println(bean.getClass())
```

auf ein Proxy-Objekt, das die BeanFactory erzeugt hat, liefert

- `class $Proxy6`

Daran ist erkennbar, dass sich die Klasse des Proxy-Objekts von der ursprünglichen unterscheidet. Das bedeutet auch als Konsequenz, dass die Bean nicht mehr auf die Implementierungsklasse gecastet werden kann. Möchte man Methoden aufrufen, die nur in der Klasse enthalten sind, müssen diese in einem Interface definiert werden, um Zugriff darauf zu erhalten. Da Spring das Programmieren gegen Interfaces erleichtert, kann dies in den meisten Fällen ohne großen Aufwand erreicht werden. Zusätzlich wird dadurch die Trennung zwischen Implementierung und Schnittstelle gefördert. Die Verwendung von Proxy-Objekten ist nur einer der vielen Vorteile, die man dadurch erhält.

Proxy-Objekte mit cglib

Problematisch wird es bei Klassen, die keine Interfaces implementieren. In diesem Fall verwendet Spring AOP die Code Generation Library *cglib*, um von den Klassen Subklassen zur Laufzeit zu erzeugen. Der Aufruf der Methode `getClass` eines Proxy-Objekts mit *cglib* liefert beispielsweise für die Klasse `de.azeitner.BeanClass`

```
class de.azeitner.BeanClass$$EnhancerByCGLIB$$6021c59
```

Erzeugung von Proxy-Objekten zur Laufzeit

Spring AOP erzeugt mittels dieser Bibliothek zur Laufzeit eine neue Klasse, die von der ursprünglichen Klasse ableitet. Damit kann das Proxy-Objekt auf die ursprüngliche Klasse gecastet werden. Als Konsequenz bedeutet dies jedoch, dass Methoden oder Klassen, die als `final` deklariert sind, nicht verwendet werden können. Aufrufe auf unüberschreibbare Methoden werden direkt an das ursprüngliche Objekt weitergeleitet, ohne dass Aspekt-Code ausgeführt werden kann. Anders gesagt: Methoden, die als `final` deklariert sind,

bieten mit *cglib* keinen Joinpoint für aspektorientierte Programmierung. Klassen, die `final` sind, können nicht verwendet werden, da Spring bei der Erzeugung eine `AopConfigException` wirft.

> * * * TIPP
> *Aus Gründen des guten Designs, der Erweiterbarkeit und Testbarkeit ist es ratsam – zumindest für die wichtigsten Klassen, auf die AOP angewendet werden soll – Interfaces zur Verfügung zu stellen, um Proxy-Objekte leicht erstellbar zu machen. Sind Interfaces vorhanden, ist es irrelevant, ob Klassen oder Methoden* `final` *sind. Für den Fall von externen Bibliotheken, in denen Klassen ohne Interfaces enthalten sind, ist cglib jedoch eine gute Variante, um AOP auch in solchen Situationen anwenden zu können.*

3.2.3 Wieso AOP?

Aspektorientierte Programmierung ermöglicht es uns, Querschnittsbelange an einer Stelle zusammenzufassen, um redundanten Code zu verhindern. Obwohl ein objektorientiertes Modell ausgezeichnete Möglichkeiten bietet, ein System von Objekten zu erfassen und darzustellen, erweitert Aspektorientierung das herkömmliche Objektmodell. Durch den deklarativen Ansatz von Spring ist AOP besonders wartungsfreundlich und Aspekte können, ohne Code zu ändern, angewendet werden.

Aspektorientierung bietet einige Vorteile, die wir durch den Einsatz von Spring AOP erhalten und im Folgenden kurz erläutert sind.

- **Klare Verantwortung von individuellen Modulen**: AOP erlaubt es den Aspekten, nur ihre Verantwortlichkeit zu implementieren. Keine Funktionalität dieser Querschnittbelange wird in anderen Modulen verwendet.
- **Höhere Modularisierung möglich**: Aspekte werden nicht an den Code, an dem sie aufgerufen werden, gekoppelt. Dadurch kann besser modularisiert werden und das System enthält weniger redundanten Code.
- **Leichte Weiterentwicklung und Flexibilität**: Durch den deklarativen Ansatz in der XML-Datei oder mit Annotationen können Aspekte leicht angewendet und entfernt werden. Kommen neue Module im System hinzu, können die Aspekte ohne großen Aufwand auf diese übertragen werden.
- **Wiederverwendbarkeit**: Aspekte können leicht wiederverwendet werden, da es sich um einfache POJOs handelt.
- **Spätes Treffen von Designentscheidungen**: Wird Funktionalität in Aspekte ausgelagert, können einige Entscheidungen, wie z.B. Sicherheit oder Protokollierung, erst später getroffen werden, da sie die Geschäftslo-

gik nicht beeinflussen. Änderungen im Design können leicht von Aspekten übernommen werden.

- **Delegation der Querschnittsfunktionalität**: Da Aspekte eigene Einheiten bilden, können sie leicht innerhalb eines Teams delegiert werden. Da deklarativ angewendet wird, müssen sich die Entwickler auf keine gemeinsame Schnittstelle einigen.

Natürlich löst AOP keine Probleme, die auch schon zuvor unlösbar waren. Es bietet lediglich für manche Anforderungen einen alternativen Weg der Implementierung mit weniger Aufwand und bessere Wartbarkeit. Vor allem mit Spring AOP ist der Einsatz von AOP sehr einfach, weil alles innerhalb der BeanFactory passiert, ohne dass externe Konfigurationsdateien oder Bibliotheken eingebunden werden müssen.

AOP kann unübersichtlich werden

Kritik an AOP gilt vor allem der Übersichtlichkeit. In einem objektorientierten Modell ist es für das geschulte Auge leicht erkennbar, welche Codezeilen einander im Ablauf folgen. Vor allem durch die Unterstützung der Entwicklungsumgebung ist es einfach zu erkennen, welche Methoden überschrieben werden und welche Klassen voneinander ableiten. Mit AOP erhöht sich in dieser Hinsicht die Komplexität, da nicht leicht zu erkennen ist, welche Funktionalität wann ausgeführt wird. Betrachtet man die Methode einer Klasse und den aufrufenden Code, kann man sich ohne AOP sicher sein, dass zwischen Aufruf und Ablauf einer Methode »nichts« passiert. Durch die deklarative Natur von AOP können mehrere Advices dazwischen aufgerufen werden, die Exceptions werfen, Parameter verändern oder das System auf eine Weise modifizieren, die wir nicht erwarten. Darum ist es umso wichtiger, klare Richtlinien und Dokumentation beim Einsatz von AOP zu verfassen und es nur für tatsächliche Querschnittsbelange einzusetzen, die die Geschäftslogik nicht behindern oder ändern. Die Querschnittsthemen Sicherheit und Protokollierung sind gute Beispiele, die die Funktionalität der Anwendung nicht beeinflussen. Die Geschäftslogik funktioniert auch ohne sie. Dadurch sind sie zur Implementierung mit AOP optimal geeignet.

EJB 3 verwendet AOP

EJB 3 ermöglicht per Spezifikation die Verwendung von AOP mittels Interceptor-Klassen. Interceptoren können für jede Methode einer Enterprise JavaBean definiert werden, die die Aufrufe zu einer EJB abfangen und Aspekt-Funktionalität ausführen. Möchte man jedoch einfach Java-Objekte, die nicht in einem EJB Container leben oder keine EJBs sind, aspektorientiert programmieren, ist Spring AOP eine ausgezeichnete Lösung. Verwendet man Spring bereits für andere Teile der Applikation, ist die Einführung von AOP mit nur wenig Aufwand verbunden.

3.3 Umsetzung

Unser Kunde fordert die Anbindung unserer Geschäftslogik an ein externes Auditierungssystem. Wir gehen davon aus, dass der Zugriff auf dieses System über eine proprietäre Bibliothek erfolgt, der wir in diesem Kapitel keine Aufmerksamkeit schenken wollen. In unserer Geschäftslogik, die über mehrere Klassen verteilt sein kann, soll diese Funktionalität aufgerufen werden. Damit bietet sich die Anbindung an das Auditierungssystem zur Implementierung als Aspekt an. Aspekt oder nicht, der Code bleibt bei Spring AOP gleich, da Aspekte als POJOs implementiert werden. Dies bietet u.a. den Vorteil, dass wir den Aspekt separat von der Bestelllogik testen können.

»Zutaten« von AOP

Im folgenden Kapitel 3.3.2 sehen wir einen AOP-Prototypen, der

- einen Aspekt implementiert,
- ein Joinpoint definiert und
- einen Advice definiert.

Diese drei »Zutaten« sind notwendig, um AOP einsetzen zu können. Im weiteren Verlauf dieses Kapitels werden wir mehr über die weiteren Möglichkeiten von Spring AOP erfahren und diese Implementierung verfeinern.

3.3.1 Voraussetzung

Für die im Folgenden beschriebene Umsetzung benötigen wir folgende JARs, die im Klassenpfad liegen müssen. Die Datei `aspectjweaver.jar` enthält die AspectJ-Annotationen, die von Spring AOP verwendet werden. Zur Laufzeit wird jedoch kein Code von AspectJ ausgeführt.

Projektname	Name der Bibliothek	Speicherort
Spring AOP	spring-aop.jar (oder spring.jar)	/modules
AspectJ	aspectjweaver.jar	/spring_dependencies/aspectj
Code Generation Library	cglib-nodep-2.1.3.jar	/spring_dependencies/cglib

Tabelle 3.1: Bibliotheken für Spring AOP

3.3.2 Implementierung der Auditierung

Wir implementieren die Anbindung an das Auditierungssystem in Form einer einfachen Klasse `AuditAspekt` in Listing 3.2, die als Aspekt dienen soll. Darin ist eine Methode ohne Parameter enthalten, die aufgerufen werden sollen, wenn eine Meldung an das Auditierungssystem erfolgen soll. Wir

kümmern uns nicht darum, wie die Nachricht weitergegeben wird, sondern verwenden einfach ein Delegate-Objekt.

Ein erster Aspekt

```
01 public class AuditAspekt
02 {
03    ExtAuditSystemDelegate auditSystem;
04
05    public void logAction() {
06       auditSystem.melde("Eine Bestellung ist eingegangen.");
07    }
08 }
```

Listing 3.2: Eine Klasse mit Funktionalität des Aspekts

Aspekte auf Geschäftslogik anwenden

Wir wollen, dass die Methode in Listing 3.2 nach jeder erfolgreichen Bestellung aufgerufen wird. Die Geschäftslogik für Bestellungen wird vom Interface in Listing 3.3 definiert. Wir wollen für die Klasse, die dieses Interface implementiert, den Aspekt anwenden. Jedoch nur auf jene Methoden, die mit »bestelle« beginnen. Die Methode `istBestellungLieferbar` soll kein Auditierungsereignis auslösen.

```
01 public interface BestellService
02 {
03    public boolean istBestellungLieferbar(List<BestellPosition> best);
04    public void bestelleBuch(String isbn);
05    public void bestelleZeitschrift(String name);
06    public void bestelle(String name);
07 }
```

Listing 3.3: Ein Interface zur Beschreibung eines Bestellservice

Aspekte definieren

Die beiden Klassen werden als Beans in der Spring-Konfiguration definiert, wie es Listing 3.4 zeigt. Die beiden Beans sind durch diese Definition einfach nur gewöhnliche Spring Beans ohne aspektorientierte Eigenschaften. Sie werden erst durch die AOP-Konfiguration in der Spring-Konfiguration mit dem Aspekt verbunden. Im Tag `<aop-config>` können beliebig viele Aspekte definiert werden. Im Beispiel enthält diese Konfiguration einen einzelnen Aspekt, der auf die Bean `auditAspekt` verweist. Damit wird Spring mitgeteilt, welche Bean als Aspekt verwendet werden soll.

Pointcuts definieren

Mit dem Tag `<aop:pointcut>` wird ein Pointcut definiert. Im Beispiel wird der Pointcut, der aus einer Anzahl von Methoden besteht, mit dem Namen `monitorBestellung` versehen. Die Zeichenfolge, die beim Attribut `execution` angegeben ist, verwendet die AspectJ-Language, mit der sich Methoden mit Wildcards leicht beschreiben lassen. In diesem Fall werden damit alle Aufrufe derjenigen Methoden beschrieben, die mit »bestell« beginnen, eine beliebige

Anzahl von Parametern besitzen und im Interface `BestellService` definiert sind. Der Rückgabewert der Methoden ist ebenfalls irrelevant.

Advices definieren

Damit haben wir definiert, welche Klasse als Aspekt verwendet werden soll und bei welchen Methoden sie aufgerufen werden soll. Eines fehlt noch: die Definition des Advice. Es gibt unterschiedliche Typen von Advices. In unserem Beispiel wird der Before-Advice verwendet, der mit dem Tag `<aop:before>` definiert wird. Damit wird festgelegt, dass der Aspekt vor den Methoden des Pointcut aufgerufen wird. Der Advice legt ebenso die Methode aus der Aspektklasse fest, die aufgerufen wird. In Listing 3.4 werden durch diesen Advice der Pointcut `monitorBestellung` und der Aspekt `auditAspekt` zusammengeführt. An dieser Stelle in Listing 3.4 können beliebig viele Advices definiert werden. Somit ist es auch möglich, am selben Joinpoint unterschiedliche Methoden eines Aspekts aufrufen zu können.

```
01 <?xml version="1.0" encoding="UTF-8"?>
02 <beans xmlns="http://www.springframework.org/schema/beans"
03   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
04   xmlns:aop="http://www.springframework.org/schema/aop"
05   xsi:schemaLocation="http://www.springframework.org/schema/beans
06         http://www.springframework.org/schema/beans/
07           spring-beans-2.5.xsd
08         http://www.springframework.org/schema/aop
09         http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">
10
11   <aop:config>
12     <aop:aspect id="myAspect" ref="auditAspekt">
13
14       <aop:pointcut id="monitorBestellung"
15         expression="execution(* de.azeitner.spring.
16           aop.BestellService.bestelle*(..))" />
17
18       <aop:before pointcut-ref="monitorBestellung" method="logAction"/>
19
20     </aop:aspect>
21   </aop:config>
22
23   <bean id="bestellService"
24     class="de.azeitner.spring.aop.BestellServiceImpl" />
25
26   <bean id="auditAspekt"
27     class="de.azeitner.spring.aop.AuditAspekt" />
28
29 </beans>
```

Listing 3.4: AOP in der XML-Bean-Definition

3.3.3 Pointcut-Ausdrücke

Pointcuts können bei Spring mit AspectJ-Language definiert werden. Diese Ausdrucksform wurde vom Projekt AspectJ[3] eingeführt und bietet gute Möglichkeiten, Mengen von Methoden in einem Klassemodell bilden zu können. Spring AOP unterstützt nur Teile dieser Sprache, was aber für die meisten Anforderungen ausreichend ist. Spring kennt als Joinpoint nur die Ausführung einer Methode. Daher beginnen Pointcut-Definitionen in Spring meist mit dem Ausdruck `execution`, der die Ausführung einer Methode kennzeichnet. Dieser Ausdruck besitzt folgende Syntax:

Definition von Pointcuts

```
execution(<Modifier> <Rückgabewert> <Typdeklaration> <Name> (<Parameter>) <Exceptions>)
```

Die hervorgehobenen Parameter der Syntax sind verpflichtend und müssen im Ausdruck enthalten sein. Die anderen Parameter sind optional, können also auch weggelassen werden. Wird ein optionaler Parameter nicht angegeben, wird keine Einschränkung beim Methodenvergleich angenommen. Anders gesagt: Für alle Parameter, die nicht angegeben werden, führt Spring keinen Vergleich durch und geht davon aus, dass das Kriterium in jedem Fall erfüllt wird. Sehen wir uns die einzelnen Parameter dieser Syntax im Detail an.

- **Modifier (optional)**: Bezeichnet die Sichtbarkeit der Methode. Sie kann `private`, `protected` oder `public` als Wert besitzen. Achtung: Spring kann nur `public`-Methoden als Joinpoint verwenden!
- **Rückgabewert**: Bezeichnet die Klasse des Rückgabewerts. Der Ausdruck kann ein Klassenname, ein primitiver Typ oder `void` sein. Klassennamen werden vollqualifiziert (inklusive Package und Klasse) angegeben und können * als Wildcard enthalten.
- **Typ (optional)**: Vergleicht die generischen Typen dieser Methode. Kann * als Wildcard enthalten.
- **Name**: Bezeichnet den vollqualifizierten Namen der Methode. Kann * als Wildcard enthalten.
- **Parameter**: Vergleicht mit den Parametern der Methoden. `..` bezeichnet beliebig viele Parameter, wohingegen * einen einzigen Parameter mit einem beliebigen Typ bezeichnet. `()` bezeichnet Methoden ohne Parameter.
- **Exceptions (optional)**: Vergleicht die Zeichenfolge mit den Checked-Exceptions der Methode.

3 siehe *http://www.eclipse.org/aspectj/.*

Durch den execute-Bezeichner können also die unterschiedlichsten Methodengruppen beschrieben werden. Um diese Art der Pointcut-Ausdrücke noch etwas klarer werden zu lassen, sehen wir uns die Beispiele in Tabelle 3.2 an, die die Möglichkeiten dieser Ausdrücke darstellen.

Einige Beispiele von Pointcut-Definitionen

Beispiele von Pointcut-Definitionen mit execute
`Execution(* de.azeitner.spring.aop.BestellService.*(..))`
Alle Methoden der Klasse/des Interface Bestellservice mit einer beliebigen Anzahl an Parametern (inkludiert auch überhaupt keinen Parameter)
`execution(public * *(..))`
Alle öffentlichen Methoden
`execution(* set*(..))`
Alle Setter-Methoden (private, protected und public)
`execution(* de.azeitner.spring.*.*(..))`
Alle Methoden in allen Klassen/Interfaces im Package de.azeitner.spring
`execution(public * de.azeitner.spring..*.*(String)`
Alle öffentlichen Methoden in allen Klassen/Interfaces im Package de.azeitner.spring oder Subpackages, die einen Parameter vom Typ String erwarten
`execution(* de.azeitner.spring.aop.MyCl.*(String, *))`
Alle Methoden der Klasse MyCl, die zwei Parameter erwarten, von denen der erste vom Typ String ist
`execution(* de.azeitner.spring.aop.MyCl.*(.., int))`
Alle Methoden der Klasse MyCl, die eine beliebige Anzahl an Parametern erwarten, von denen der letzte vom Typ int ist
`execution(* de.azeitner.spring.aop.MyCl.set*(*))`
Alle Methoden der Klasse MyCl, die mit »set« beginnen und genau einen Parameter eines beliebigen Typs besitzen
`execution(protected void de.azeitner.spring.aop.MyCl.*())`
Alle geschützten Methoden der Klasse MyCl, die keinen Parameter besitzen. Achtung: Spring kann nur öffentliche Methoden als Joinpoint verwenden!
`execution(* de..*(..) throws IllegalStateException)`
Alle Methoden aller Klassen, die im Package de oder in Subpackages liegen und eine IllegalStateException werfen. Das Package muss hier nicht angegeben werden, da sich die Klasse IllegalStateException im Package java.lang befindet.
`execution(* de.azeitner..*(..) throws *Exception)`
Alle Methoden aller Klassen, die im Package de.azeitner oder in Subpackages liegen und eine Exception werfen, deren Name auf »Exception« endet und sich im Package java.lang befindet

Tabelle 3.2: Beispiele von Pointcut-Definitionen mit execute

Beispiele von Pointcut-Definitionen mit `execute`
`execution(* de..*(..) throws java.rmi.RemoteException)`
Alle Methoden aller Klassen, die im Package `de.azeitner` oder in Subpackages liegen und eine `RemoteException` werfen

Tabelle 3.2: Beispiele von Pointcut-Definitionen mit `execute (Fortsetzung)`

Beans als Pointcuts setzen

Seit Spring 2.5 gibt es neben `execution` noch den Bezeichner `bean`, der den Ausdruck auf Beans in der XML-Definition einschränken kann. Dabei vergleicht Spring AOP den verwendeten Wert mit dem Bean-Namen in der Spring-Konfiguration. Dadurch können ebenso *-Wildcards verwendet werden. Der Pointcut wird dabei nicht auf die Klassen der Beans angewendet, sondern auf die Beans selbst. Verwendet ein Ausdruck den Wert `bean(bestellService)`, schließt der Ausdruck nur die Bean `bestellService` ein. Andere Beans bleiben davon unberührt, auch wenn sie von derselben Klasse wie die Bean `bestellService` instanziiert wurden. Den Bezeichner `bean` gibt es nur in Spring. Dieser ist eine Spring-eigene Erweiterung der AspectJ-Language.

Ausdrücke mit `and` und `or` kombinieren

Zusätzlich gibt es die Bezeichner `and` und `or`, mit denen man den Pointcut beliebig erweitern oder einschränken kann. Folgender Ausdruck umfasst alle Setter- und Getter-Methoden einer Klasse:

```
execute(* set*(..)) or execute(* get*(..))
```

! ! ! ACHTUNG

Spring AOP vergleicht jeden Pointcut-Ausdruck mit der Methode im Proxy. Möchte man beispielsweise Setter- und Getter-Methoden im Pointcut definieren, benötigt man den Bezeichner `or`*, da eine Methode nur entweder eine Setter- oder eine Getter-Methode sein kann. Es handelt sich dabei um logische Oder- und Und-Operationen, die man nicht mit dem umgangssprachlichen »und« im Sinne der Vereinigungsmenge verwechseln darf. Der Ausdruck* `and` *kann beispielsweise dafür verwendet werden, Klassen zusammenzufassen, die zwei oder mehrere Interfaces eines speziellen Typs implementieren. Tabelle 3.3 zeigt einige einfache Beispiele mit den Bezeichnern* `bean`, `and` *und* `or`.

Beispiele von Pointcut-Definitionen mit `bean`, `and` und `or`
`bean(*)`
Alle Methoden aller Beans in der Bean-Definition
`bean(bestellService)`
Alle Methoden der Bean `bestellService`

Tabelle 3.3: Beispiele von Pointcut-Definitionen mit `bean`, `and` und `or`

Beispiele von Pointcut-Definitionen mit `bean`**,** `and` **und** `or`
`bean(*Service)`
Alle Methoden aller Beans in der Bean-Definition, deren Name mit »Service« endet
`execution(* *.bestelle*(..)) and bean(bestellService)`
Alle Methoden der Bean `bestellService`, die mit »bestelle« beginnen
`bean(bestellService) or bean(lagerService)`
Alle Methoden der Bean `bestellService` und alle Methoden der Bean `lagerService`

Tabelle 3.3: Beispiele von Pointcut-Definitionen mit `bean`, `and` und `or (Fortsetzung)`

Haben wir unsere Pointcuts definiert, die alle Methoden enthalten, an denen wir AOP anwenden wollen, können wir die Advices definieren.

3.3.4 Advices definieren

Viele Arten von Advices

Advices legen fest, wie Aspekte und Joinpoint zusammenspielen. Aspekte können bei Spring AOP vor oder nach einer Methode aufgerufen werden, die mittels des Pointcut-Ausdrucks definiert wurde. Wann dies genau passiert, definiert der Advice. Es gibt mehrere Arten von Advices, die wir genauer untersuchen wollen. Ein Advice zeigt immer auf einen Aspekt, der am Pointcut angewendet wird.

Before-Advice

Before-Advices definieren

Mit `<aop:before>` können wir Advices definieren, die zur Folge haben, dass die Aspektmethode vor den Methoden eines Joinpoint aufgerufen werden. Before-Advices sind besonders nützlich, wenn wir Eingangsüberprüfungen der Bestellung durchführen oder die Sicherheit überprüfen wollen. Wir können im Advice auch Runtime-Exceptions werfen und die Ausführung des Joinpoint damit verhindern. Eine vorangehende Sicherheitsüberprüfung kann beispielsweise eine `SecurityException` werfen, falls der Benutzer nicht zur Bestellung berechtigt ist. Wir könnten auch die eingehende Bestellung auf ihre Gültigkeit prüfen, bevor wir die Geschäftslogik ausführen, damit diese sich auf eine gültige und vollständige Bestellung verlassen kann.

After-Advice

After-Returning-Advices definieren

Unser Kunde möchte, dass wir Bestellungen an das externe Auditierungssystem melden. Wir wollen dies jedoch nicht schon vor der technischen Abarbeitung im Voraus melden. Ist die Bestellung aus irgendeinem Grund nicht gültig oder tritt bei der Bestellung am Joinpoint ein Fehler auf, soll die Bestellung

nicht an das Auditierungssystem gemeldet werden. Nur erfolgreich abgeschlossene Bestellungen sollen dort erscheinen. Dafür gibt es eine andere Form der Advice-Deklaration: `<aop:after-returning>`. Der Aspekt wir dadurch erst dann ausgeführt, wenn die Methode am Joinpoint erfolgreich beendet wurde. Wirft die Methode eine Exception, wird der Advice nicht ausgeführt.

```
01 <aop:aspect id="myAspect" ref="auditAspekt">
02
03    <aop:pointcut id="monitorBestellung"
04       expression="execution(* de.azeitner.spring.aop.
05          BestellService.bestelle*(..))" />
06
07    <aop:before pointcut-ref="monitorBestellung"
08       method="logEingang" />
09
10    <aop:after-returning pointcut-ref="monitorBestellung"
11       method="logAusgang" />
12
13 </aop:aspect>
```

Listing 3.5: Definition eines After-Returning-Advice

After-Finally-Advices definieren

In Listing 3.5 wird die Aspektmethode `logAusgang` nur aufgerufen, wenn die Bestellung erfolgreich durchgeführt wurde. Mit `<aop:after>` gibt es noch eine andere Möglichkeit eines After-Advice. Dabei wird der Advice in jedem Fall ausgeführt, unabhängig davon, was in der Methode des Joinpoint passierte. Man kann diesen Advice deswegen auch als After-Finally-Advice bezeichnen, weil er in jedem Fall ausgeführt wird. Ein Beispiel eines solchen Advice ist in Listing 3.6 zu sehen.

```
01    <aop:after pointcut-ref="monitorBestellung"
02       method="logAusgang" />
```

Listing 3.6: Definition eines After-Advice

After-Throwing-Advices definieren

Für unseren Kunden ist es interessant zu wissen, wenn eine Bestellung aufgrund eines Systemfehlers nicht abgeschlossen werden konnte. Vielleicht möchte der Kunde, dass solche Fehler an einer anderen Stelle im Auditierungssystem erscheinen, oder er wünscht eine E-Mail-Benachrichtigung, um nachvollziehen zu können, warum eine Bestellung nicht durchgeführt wurde. Für unseren Kunden bedeutet immerhin jede Bestellung wichtigen Umsatz, darum darf keine Bestellung verloren gehen. Wir wollen daher, falls in der Geschäftslogik eine Exception auftritt, eine Aspektmethode aufrufen lassen. Für diese Anforderung gibt es eine eigene Art von Advice: den After-Throwing-Advice. Dafür muss unser Code zunächst etwas angepasst werden.

```
01 public class AuditAspekt
02 {
03   ...
04
05   public void logBestellFehler(RuntimeException geworfeneException) {
06     auditSystem.meldeFehler("Ein Bestellfehler ist aufgetreten: " +
07       geworfeneException);
08   }
09
10   ...
11 }
```

Listing 3.7: Auditierungsmethode für den Fall eines Bestellfehlers

Listing 3.7 zeigt den erweiterten Code der Klasse `AuditAspekt`, die jetzt eine Methode `logBestellFehler` besitzt. Diese Methode übernimmt eine Exception oder einen Subtyp davon, den es abzufangen gilt. Die Methode soll am Joinpoint aufgerufen werden, falls die Exception eintritt, die als Parameter der Methode definiert ist. Wollen wir beispielsweise nur ungültige Zustände in der Bestellung behandeln, könnten wir eine Methode verwenden, die `IllegalStateException` als Typ verwendet. In diesem Fall wollen wir jede `RuntimeException` abfangen, die während einer Bestellung auftreten kann. Wir gehen davon aus, dass Checked-Exceptions der Bestellserviceklasse bereits vom aufrufenden Code behandelt werden.

Geworfenen Typ definieren

Um einen After-Throwing-Advice zu definieren, verwenden wir das Tag `<aop:after-throwing>`, wie es Listing 3.8 zeigt. Im Unterschied zu den anderen Advice-Definitionen erwartet das Tag das Attribut `throwing`, das den Namen des Exception-Parameters aus der Aspektmethode `logBestellFehler` enthalten muss. Der Wert `geworfeneException` muss mit dem Parameternamen aus Listing 3.7 zusammenpassen, damit Spring weiß, auf welchen Typ von Exception der Advice angewendet werden soll.

```
01 <aop:aspect id="myAspect" ref="auditAspekt">
02   ...
03   <aop:after-throwing pointcut-ref="monitorBestellung"
04     method="logBestellError" throwing="geworfeneException" />
05
06 </aop:aspect>
```

Listing 3.8: Definition eines After-Throwing-Advice

After-Throwing-Advices machen kein Exceptionhandling

Mit dieser Definition wird bei jeder geworfenen `RuntimeException` oder einer Tochterklasse die Methode `logBestellFehler` ausgeführt. Ein After-Throwing-Advice beeinflusst das Exceptionhandling der Applikation nicht. Die Exception wird technisch gesehen nicht gefangen. Für den aufrufenden Code ist also ein

After-Throwing-Advice vollständig transparent und er kann jede Exception des Bestellservice auf die gleiche Art und Weise behandeln. Die Aspektmethode hat aber Zugriff auf die geworfene Exception, die dadurch gelesen oder sogar verändert werden könnte, bevor sie an die aufrufende Klasse weitergegeben wird.

Around-Advice

Around-Advices definieren

Die letzte Advice-Art, die wir in diesem Kapitel behandeln wollen, ist der Around-Advice, der auch in EJB 3 zum Einsatz kommt. Der Around-Advice ist der flexibelste von allen, weil er alle bisher vorgestellten Arten in einem Advice vereint. Listing 3.9 zeigt, wie dieser Advice mit dem Tag `<aop:around>` definiert werden kann. Die Definition in der XML-Datei funktioniert analog zu den anderen Advices. Der große Unterschied wird erst in der erwarteten Aspektmethode sichtbar.

```
01 <aop:aspect id="myAspect" ref="auditAspekt">
02
03    <aop:pointcut id="monitorBestellung"
04       expression="execution(* de.azeitner.spring.aop.
05          BestellService.bestelle*(..))" />
06    ...
07    <aop:around pointcut-ref="monitorBestellung"
08       method="logBestellung" />
09
10 </aop:aspect>
```

Listing 3.9: Definition eines Around-Advice

Joinpoint-Abarbeitung steuern

Der Around-Advice zeichnet sich dadurch aus, dass der Advice selbst den Aufruf des Joinpoint steuern kann. Wenn wir uns die Methode in Listing 3.10 ansehen, erkennen wir, dass sich die Methodensignatur geändert hat. Die Methode liefert ein Objekt zurück und übernimmt einen Parameter vom Typ `org.aspectj.lang.ProceedingJoinPoint`. Dieser Parameter kapselt den Joinpoint, der aufgerufen werden soll. Der Around-Advice kann selbst bestimmen, wann dies passieren soll. Mit der Methode `proceed` wird der Joinpoint aufgerufen und der Rückgabewert der Joinpoint-Methode zurückliefert. Ist die Joinpoint-Methode eine `void`-Methode, liefert `proceed` `null`. Auch Exceptions werden transparent von `proceed` weitergeworfen, so dass der Advice auf Exceptions aus der Geschäftslogik reagieren kann. Der große Vorteil dieses Advice ist, dass man als Entwickler sehr flexibel ist, wann der Joinpoint aufgerufen wird und was davor und danach passieren soll.

```
01 public Object logBestellung(ProceedingJoinPoint pjp) throws Throwable {
02   // aop:before
03   Object returnValue;
04   try
05   {
06     returnValue = pjp.proceed();
07     // aop:after-returning
08     return returnValue;
09   } catch (RuntimeException e)
10   {
11     // aop:after-throwing
12     throw e;
13   } finally
14   {
15     // aop:after
16   }
17 }
```

Listing 3.10: Methode eines Around-Advice

* * * TIPP

Hat der Aspekt stark zusammenhängende Logik vor und nach Aufruf des Joinpoints, ist dieser Advice eine gute Wahl. Man kann damit lokale Referenzen innerhalb einer Methode wiederverwenden und erhält somit einfacheren Code im Gegensatz zu zwei getrennten After- und Before-Advices. Allerdings ist bei der Entwicklung von Around-Advices Vorsicht geboten, da sie die Ausführung des Joinpoint auch komplett unterbinden können, ohne dass eine Exception geworfen wird. Vergisst man als Entwickler, die `proceed`*-Methode aufzurufen, kann das ungewollte Effekte im System haben. Nur in seltenen Fällen würde man bewusst die* `proceed`*-Methode ignorieren und in den Ablauf des Programms so stark eingreifen. Grundsätzlich gilt: Soll Aspektlogik davor oder danach passieren, sind die Before- und After-Advices eine gute Wahl. Für stark zusammenhängende Logik, die den Joinpoint umschließt, ist man mit einem Around-Advice gut beraten.*

3.3.5 Advices parametrisieren

Aspekte benötigen mehr Informationen

Wir haben nun den Zugriff auf das externe Auditierungssystem mit den richtigen Advices konfiguriert. Das Problem, das wir jedoch noch haben, ist, dass wir in unserer Aspektmethode nicht wissen, welche Pointcut-Methode gerade aufgerufen wird. Wir wissen auch nicht, welche Parameter übergeben worden sind. Eine sinnvolle Nachricht an das Auditierungssystem zu schicken, ist also aus einer Aspektmethode zum derzeitigen Stand nicht möglich. Aus diesem Grund gibt es bei Spring AOP die Möglichkeit, den Aspektmethoden Informationen zum Aufruf mitzugeben.

Methodenparameter im Advice verwenden

Parameter der Bestellung sollen bekanntgemacht werden

Erinnern wir uns an das Interface `Bestellservice`, das die Bestellmethoden definiert. In Listing 3.11 erkennen wir drei Bestellmethoden, die dieselben Parameter haben. Wir können daher unseren Pointcut dahingehend erweitern, dass wir definieren, dass die Methoden zumindest die beiden Parameter `isbn` und `anzahl` enthalten.

```
01 public interface BestellService
02 {
03    public boolean istBestellungLieferbar(List<BestellPosition> best);
04    public void bestelleBuch(String isbn, int anzahl);
05    public void bestelleZeitschrift(String isbn, int anzahl);
06    public void bestelle(String isbn, int anzahl);
07 }
```

Listing 3.11: Methoden des Bestellservice mit mehreren Parametern

Parameter in den Pointcut aufnehmen

Listing 3.12 zeigt eine um `and args(isbn, anzahl)` erweiterte Form des Pointcut. Der neue Ausdruck definiert, dass alle Methoden des Pointcut die beiden Parameter `isbn` und `anzahl` vom jeweils selben Typ besitzen. Die Liste kann eine beliebige Anzahl an Parameternamen enthalten, vorausgesetzt, sie heißen in allen Methoden des Pointcut gleich. Passt auch nur eine Methode des Pointcut nicht mit dem Parameter zusammen, wirft Spring eine Exception. Zusätzliche Parameter, die nicht im Pointcut definiert sind, dürfen vorhanden sein, können aber nicht im Aspekt verwendet werden.

```
01 <aop:aspect id="myAspectParams" ref="auditAspekt">
02
03    <aop:pointcut id="monitorBestellungParams"
04       expression="execution(* de.azeitner.spring.aop.
05          BestellService.bestelle*(..)) and args(isbn, anzahl)" />
06
07    <aop:after-returning pointcut-ref="monitorBestellungParams"
08       method="logBestellungParams" arg-names="isbn, anzahl" />
09
10 </aop:aspect>
```

Listing 3.12: Pointcut mit Parametern

Aspekte können Parameter auswerten

Sind die Parameter im Advice definiert, können sie einem Aspekt mitgegeben werden. Listing 3.12 zeigt einen After-Returning-Advice, der die beiden Parameter `isbn` und `anzahl` als Parameter bekommt. Die Aspektmethode muss dementsprechend die beiden Parameter übernehmen, wie es Listing 3.13 zeigt. Die beiden Parameter werden von Spring AOP an den Aspekt übergeben, der auf diese zugreifen oder sie verändern kann.

```
01 public class AuditAspekt
02 {
03    ...
04    public void logBestellungParams(String isbn, int anzahl) {
05       auditSystem.melde("Die Bestellung beträgt " + anzahl +
06          " Stück von ISBN " + isbn);
07    }
08 }
```

Listing 3.13: Parametrisierte Aspektmethode

Durch diese Implementierung können sinnvolle Werte wie z.B. die ISBN des bestellten Buchs an das Auditierungssystem weitergegeben werden. Werden Objekte an den Aspekt weitergegeben, können diese vor oder nach Aufruf des Joinpoint verändert werden. So könnte beispielsweise ein Before-Advice versuchen, einen ungültigen Zustand einer Bestellungsanfrage abzuändern und einen gültigen an den Bestellservice weiterzugeben.

Rückgabewerte im Aspekt verwenden

Rückgabewerte im Aspekt auswerten

Nach ähnlichem Prinzip kann auch der Rückgabewert einer Methode an den Advice übergeben werden. Über das Attribut `returning` kann der Name des Parameters in der Aspektmethode angegeben werden. Spring AOP führt die Pointcut-Methode aus und übergibt den Rückgabewert an die Aspektmethode. In dieser kann darauf zugegriffen und der Wert auch verändert werden.

```
01 <aop:aspect id="myAspectReturn" ref="auditAspekt">
02
03   <aop:pointcut id="monitorBestellung"
04     expression="execution(* ..*Service.istBestellungLieferbar*(..))" />
05
06   <aop:after-returning pointcut-ref="monitorBestellung"
07     method="logLagerAbfrage" returning="jaOderNein" />
08
09 </aop:aspect>
```

Listing 3.14: Definition eines After-Returning-Advice mit Rückgabewert

Namen der Rückgabewerte müssen übereinstimmen

Listing 3.14 zeigt die Definition eines After-Returning-Advice. Dabei wird die Lagerstandsabfrage des Bestellservice mit dem Methodennamen `istBestellungLieferbar` mit einem Advice belegt. Ist die Abfrage erfolgreich und liefert die Methode einen Wert zurück, kann der Aspekt diesen auditieren oder nachbearbeiten. Die Aspektmethode `logLagerAbfrage` in Listing 3.15 übernimmt den Rückgabewert des Bestellservice. Für den Typ des Rückgabewerts gelten dieselben Regeln wie für Methodenparameter: Der Typ muss ein Supertyp aller zurückgelieferten Objekte der Methoden im Pointcut sein. Liefert die

Methode `istBestellungLieferbar` den Typ `boolean` zurück, kann der Parameter `jaOderNein` in der Aspektmethode auf `boolean` gecastet werden. Noch besser wäre es, die Aspektmethodensignatur so zu ändern, dass der Parameter `jaOderNein` bereits den Typ `boolean` erwartet, um den Cast nicht machen zu müssen.

```
01 public class AuditAspekt
02 {
03    public void logLagerAbfrage(Object jaOderNein) {
04       // Advicecode
05    }
06 }
```

Listing 3.15: After-Returning-Advice mit Parameter für den Rückgabewert

Rückgabewerte nur bei After-Returning-Advices

Rückgabewerte können nur mit After-Returning-Advices verwendet werden, da kein anderer Advice einen Rückgabewert garantiert. Around-Advices rufen den Pointcut direkt auf und können dadurch auf den Rückgabewert direkt in der Aspektmethode selbst zugreifen.

Target im Advice verwenden

Target ist das ursprüngliche Objekt

Das Target bezeichnet das ursprüngliche Objekt, für das das Proxy-Objekt erstellt wurde. Da wir AOP verwenden, wollen wir üblicherweise auf das Proxy-Objekt zugreifen. Manchmal ist es aber sinnvoll, direkt auf das Target zugreifen zu können. Wollen wir beispielsweise nur jene Lagerbestandsabfragen protokollieren, die über das Proxy-Objekt aufgerufen werden, ist die Verwendung des Target in einem Aspekt sinnvoll. Methodenaufrufe am Target wissen nichts von AOP. Es werden daher keine Aspektmethoden ausgeführt, weil es sich nicht um das Proxy-Objekt handelt. Führen wir also in einer Aspektmethode eine Lagerbestandsabfrage am Target durch, wird diese nicht protokolliert.

In unserem Fall ist das Target das Bestellserviceobjekt. Wollen wir im Aspekt auf das Target zugreifen, können wir es ähnlich wie Methodenparameter an den Aspekt übergeben lassen. Für unsere Implementierung ist dies sehr sinnvoll, wenn wir Informationen über den verwendeten Bestellservice an das Auditierungssystem weitergeben wollen. So könnten wir unterschiedliche Implementierungen des Interface `BestellService` verwenden, wovon jede einer speziellen Bestellart gewidmet ist. Würden wir nur die Methodenparameter verwenden, können wir nicht unterscheiden, über welchen Bestellservice die Bestellung durchgeführt wird.

Target im Pointcut-Ausdruck

Um die Instanz der Klasse Bestellservice dem Aspekt mitgeben zu können, muss der Pointcut-Ausdruck wie in Listing 3.16 geändert werden. Durch die Zeichenfolge `target(bestellService)` wird das AOP-Target für die Verwen-

dung vorbereitet. Im Advice kann das Target der Aspektmethode über das Attribut `arg-names` mitgegeben werden.

```
01 <aop:aspect id="myAspectTarget" ref="auditAspekt">
02
03    <aop:pointcut id="monitorBestellungTarget"
04       expression="execution(* de.azeitner.spring.aop.
05          BestellService.bestelle*(..)) and target(bestellService)" />
06
07    <aop:after-returning pointcut-ref="monitorBestellungTarget"
08       method="logBestellService" arg-names="bestellService" />
09
10 </aop:aspect>
```

Listing 3.16: AOP-Target im Advice verfügbar machen

Klasse vom Target-Objekt

Die Aspektmethode muss einen Parameter mit dem konfigurierten Namen übernehmen, der von Spring AOP mit dem jeweiligen Target befüllt wird. Nachdem das Target-Objekt von einer beliebigen Klasse sein kann, muss eine gemeinsame Mutterklasse aller Target-Objekte verwendet werden, da Spring AOP andernfalls eine `ClassCastExceptions` wirft. Im allgemeinen Fall bleibt nur `java.lang.Object`. Im Fall des Bestellservice wissen wir jedoch, dass alle Targets das Interface `BestellService` implementieren, und deshalb können wir so den Parameter mit dem Namen `bestellService` im Typ einschränken. Wie Listing 3.17 zeigt, bekommt die Methode `logBestellService` den aufgerufenen Bestellservice übergeben.

```
01 public class AuditAspekt
02 {
03    ...
04    public void logBestellService(BestellService bestellService) {
05       auditSystem.melde("Das verwendete Bestellservice ist " +
06          bestellService);
07    }
08 }
```

Listing 3.17: Aspektmethode, die ein Target übernimmt

! ! ! ACHTUNG

Das Target allein verwendet kein AOP. Erst durch das Proxy-Objekt, das Spring AOP für das Target generiert, werden Aspektmethoden aufgerufen. Das Target ist das ursprüngliche Objekt, das von der BeanFactory instanziiert wurde. Möchte man das dazugehörige Proxy-Objekt verwenden, kann man sich des Pointcut-Ausdrucks `this(bestellService)` *statt* `target(bestellService)` *bedienen. Mit* `this` *wird das aktuelle Proxy-Objekt übergeben, nicht das dahinterliegende Target-Objekt.*

3.3.6 AOP mit Annotationen

Neben der XML-Definition von AOP, erlaubt es Spring AOP auch, Advices, Pointcuts und Aspekte mittels Annotationen zu definieren. Die Annotationen sind mit den Annotationen von AspectJ kompatibel, um einen etwaigen Umstieg auf AspectJ zu erleichtern, falls Spring AOP nicht mehr ausreichen sollte. Ab Spring 2.0 sind die Annotationen von AspectJ 5 in Spring enthalten. Trotzdem ist Spring AOP von AspectJ unabhängig und es wird zur Laufzeit kein AspectJ-Code ausgeführt.

Spring AOP und AspectJ sind kompatibel

Welche Variante wir für AOP in Spring verwendet wollen, unterliegt unserer Entscheidung. Beide Varianten verwenden dieselben Pointcut-Ausdrücke und Advice-Arten. Auch die Funktionalität ist weitgehend dieselbe. Würden wir Java 1.4 verwenden, hätten wir keine Wahl, da noch keine Annotationen verfügbar sind. Ab Java 5 wird von vielen Entwicklern die Variante mit Annotationen bevorzugt, weil sie Advices mit Parametern und Rückgabewerten dort definiert, wo sie entstehen. Im Gegenzug erleichtert die XML-Variante das Testen von AOP, weil die XML-Datei leichter ausgetauscht werden kann. Spring schreibt in jedem Fall keine Variante vor, sondern überlässt uns Entwicklern die Entscheidung.

Annotationen vs. XML

Werden Annotationen für AOP verwendet, müssen Advices und Pointcuts nicht mehr in der XML-Datei eingetragen werden. Um Spring AOP zu veranlassen, dass es nach AOP-Annotationen Ausschau halten soll, muss eine der beiden Varianten in Listing 3.18 gewählt werden. Beide Varianten führen zum gleichen Ergebnis. Variante 1 ist in ihrer Definition etwas kürzer und verwendet den AOP-Namespace. Variante 2 besitzt hingegen den Vorteil, dass der AOP-Namespace nicht importiert werden muss. Verwendet man nur annotationsbasiertes AOP, wird der AOP-Namespace nicht benötigt.

Spring nach AOP-Annotationen suchen lassen

```
01 <!- Variante 1 mit AOP-Namespace -->
02 <aop:aspectj-autoproxy />
03
04 <!- Variante 2 ohne AOP-Namespace -->
05 <bean class="org.springframework.aop.aspectj.annotation.
06    AnnotationAwareAspectJAutoProxyCreator" />
```

Listing 3.18: Aktivieren von annotationsbasiertem AOP

Advices und Pointcuts mit Annotationen definieren

Um Annotationen für die Implementierung des Auditierungsaspekts zu verwenden, annotieren wir die Aspektklasse mit `@Aspect`, wie es Listing 3.19 zeigt. Trotzdem müssen wir den Aspekt als Bean definieren, damit Spring AOP weiß, dass es einen neuen Aspekt im System gibt. Alles andere lässt sich sehr

Aspekte tragen die Annotation `@Aspect`

komfortabel an einer Stelle im Code definieren. Wie es Listing 3.19 zeigt, erfolgt die Advice-Definition innerhalb der Aspektklasse. Mit der Annotation `@Before` wird ein Before-Advice definiert. Die Annotation übernimmt ein Attribut `value`, das den Pointcut-Ausdruck enthält.

```
01 @Aspect
02 public class AuditAspekt
03 {
04    ExtAuditSystemDelegate auditSystem;
05
06    @Before(value="execution(*
07       de.azeitner.spring.aop.*.bestelle*(..))")
08    public void logAction() {
09       auditSystem.melde("Eine Bestellung ist eingegangen.");
10    }
11 }
```

Listing 3.19: Der Auditierungsaspekt mit der `@Aspect`-Annotation

AOP mit Annotationen bietet den Vorteil, dass besser ersichtlich ist, welche Pointcuts welchen Aspektmethoden zugewiesen werden. Es ist auf einen Blick gut zu erkennen, auf welchen Pointcut die Methode `logAction` angewendet wird. Eine Methode kann auch mehrere Annotationen tragen und so können auch zwei oder mehr Pointcuts derselben Aspektmethode zugewiesen werden.

Advices verwenden, `@Before`, `@After` u.a.

Listing 3.20 zeigt die anderen Advices, die auch mit der XML-Variante definiert werden können. Alle übernehmen einen Pointcut-Ausdruck, der definiert, für welche Joinpoints sie aufgerufen werden sollen.

```
01 @After(value="execution(*
02    de.azeitner.spring.aop.*.bestelle*(..))")
03 public void logAusgang()
04 {
05    // Aspekt-Code
06 }
07
08 @AfterThrowing(value="execution(*
09    de.azeitner.spring.aop.*.bestelle*(..))")
10 public void logBestellFehler()
11 {
12    // Aspekt-Code
13 }
14 @AfterReturning (value="execution(*
15    de.azeitner.spring.aop.*.bestelle*(..))")
16 public void logBestellung()
17 {
18    // Aspekt-Code
19 }
```

```
20
21 @Around (value="execution(*
22    de.azeitner.spring.aop.*.bestelle*(..))")
23 public Object logBestellung(ProceedingJoinPoint pjp) throws Throwable {
24    // Advice Code
25    return pjp.proceed();
26 }
```

Listing 3.20: Advice-Annotationen bei Spring AOP

Parameterübergabe bei annotationsbasiertem AOP

Annotationen definieren die Parameter-namen

Analog zur XML-Variante können Parameter, Target und Proxy-Objekte an die Aspektmethoden weitergegeben werden. Für die Definition der Parameter bieten alle Advice-Annotationen das Attribut `argNames` an, das zur Definition der Übergabeparameter verwendet werden kann. Da die Annotation direkt vor der Methode steht und nicht in der XML-Datei, ist es mit Annotationen leichter, den Namen der Parameter zwischen dem Pointcut und der Aspektmethode abzustimmen. In Listing 3.21 wurden die Beispiele, die früher in diesem Kapitel mit der XML-Variante definiert wurden, in annotationsbasiertes AOP geändert. Die Pointcut-Ausdrücke sind gleich geblieben. Geändert hat sich nur die Stelle, an der sie stehen.

Annotationen definieren den Typ des Rückgabewerts

Das Attribut `returning` der Annotation `@AfterReturning` dient zur Übergabe des Rückgabewerts. Mit dem Attribute `throwing` der Annotation `@AfterThrowing` kann die Exception behandelt werden, die im Pointcut eines After-Throwing-Advice geworfen wird. Die Funktionalität ist auch in diesem Fall identisch mit der XML-Variante.

```
01 @Before(value="execution(* de..*.bestelle*(..)) and args(isbn, anzahl)",
02    argNames="isbn, anzahl")
03 public void logIsbn(String isbn, int anzahl) {
04    // Aspekt-Code
05 }
06
07 @Before(value="execution(* de..*.bestelle*(..))
08    and target(bestService)")
09 public void logBestellService(BestellService bestService) {
10    // Aspekt-Code
11 }
12
13 @Before(value="execution(* de..*.bestelle*(..)) and this(myProxy)")
14 public void logBestellServiceThis(Object myProxy) {
15    // Aspekt-Code
16 }
17
18 @AfterThrowing(value="execution(*
```

```
19    de..*.bestelle*(..))",
20   throwing="eineException")
21 public void logBestellFehler(RuntimeException eineException)
22 {
23    // Aspekt-Code
24 }
25
26 @AfterReturning(value="execution(* ..*Service.isBestellungLieferbar*(..))",
27   returning="jaOderNein")
28 public void logLagerAbfrage(Object jaOderNein)
29 {
30    // Aspekt-Code
31 }
```

Listing 3.21: Parameterübergabe an annotationsbasierte Aspektmethoden

Pointcut-Ausdrücke wiederverwenden

Pointcut-Ausdrücke müssen nicht redundant sein

In der XML-Definition werden die Pointcut-Ausdrücke mit einem <aop-pointcut> Tag definiert, welches mehrfach referenziert werden kann. Werden viele Advices am selben Pointcut angewendet, steht der Ausdruck dadurch nur an einer Stelle und kann dort leicht gewartet werden. Mit Annotationen werden die Pointcuts direkt an den Aspektmethoden definiert. Wenn es mehrere Aspektmethoden gibt, muss die Definition des Pointcut daher mehrfach angegeben werden und wird dadurch redundant. Darum gibt es bei Spring AOP eine Möglichkeit, Pointcut-Referenzen an einer zentralen Stelle zu verwalten.

Wiederverwendung von Pointcuts im gesamten System

Die Implementierung für unseren Kunden besteht aus mehreren Schichten, die unterschiedliche Aufgaben haben. Unter anderem gibt es eine Serviceschicht, die die Geschäftslogik enthält, und eine Webschicht, die sich um Webdarstellung und -logik kümmert. Wir wollen in jeder Schicht mehrere Advices verwenden. Die Services in der Geschäftslogik sollen mit Auditierungs-, Sicherheits- und Transaktionsaspekten belegt werden. Für alle wollen wir dieselben Pointcuts verwenden. Darum definieren wir die Pointcuts einmalig in einer zentralen Klasse, die alle Pointcuts enthält, wie es in Listing 3.22 zu sehen ist. Darin sind leere Methoden enthalten, die `void` zurückliefern. Diese können mit der Annotation `@Pointcut` versehen werden, die aus der Methode eine Pointcut-Definitionsmethode erstellt. Im Listing sehen wir die beiden Methoden `webLayer` und `businessLayer`, die `@Pointcut`-Annotationen enthalten.

```
01 public class SystemPointcuts
02 {
03    @Pointcut(value="execution(* de.azeitner.spring.web.*.*(..))")
04    public void webLayer() {
05    }
06
```

```
07    @Pointcut(value="execution(*
08        de.azeitner.spring.service.*Service.*(..))")
09    public void businessLayer() {
10    }
11 }
```

Listing 3.22: Zentrale Definition von Pointcuts

Um diese Pointcuts nutzen zu können, müssen wir die Advices auf die Pointcuts zeigen lassen. Dafür wird in den Advice-Annotationen das Attribut `value` verwendet, das wir vorher schon zur Definition der Pointcut-Ausdrücke genutzt haben. Diesmal geben wir jedoch direkt die Methode an, die mit `@Pointcut` annotiert ist. Listing 3.23 zeigt, wie dies im Aspekt angegeben wird. Achtung: Hier wird nicht der `execute`-Bezeichner verwendet, sondern direkt die vollqualifizierte Methode angegeben.

Advices auf die Pointcut-Definitionsmethode zeigen lassen

```
01 @Aspect
02 public class BestellAspekt
03 {
04    @Before(value="de.azeitner.spring.SystemPointcuts.businessLayer()")
06    public void pruefeBestellung() {
07       // Bestellung überprüfen
08    }
09    @AfterReturning(value="de.azeitner.spring.
11       SystemPointcuts.businessLayer()")
12    public void logAction() {
13       // Bestellung an externes System melden
14    }
15 }
```

Listing 3.23: Pointcut im Advice wiederverwenden

Für den Fall, dass sich unsere Businessschicht erweitert und neue Klassen entstehen, kann der Pointcut-Ausdruck zentral geändert werden, ohne dass die Advices angepasst werden müssen. Es ist also durchaus sinnvoll, Pointcuts, die mehrfach im System verwendet werden, in einer Klasse mit `@Pointcut` zu definieren. Um Schichtenverletzungen zu verhindern, könnte jede Schicht eine eigene Klasse mit den `@Pointcut`-Methoden besitzen, die nur innerhalb der Schicht verwendet wird. Aus technischen Gründen darf die Pointcut-Klasse kein Interface sein.

`@Pointcut` zentral im System oder Schicht

3.4 Zusammenfassung

Aspektorientierte Programmierung

In diesem Kapitel haben wir gelernt, aspektorientiert zu programmieren. Wir kennen die Motivation von AOP und wissen, dass AOP das objektorientierte Programmiermodell um Aspekte erweitert. Diese enthalten Querschnittsfunktionalität, die in vielen Modulen der Applikation verwendet wird, aber nicht einem zugeordnet werden kann, wie beispielweise Protokollierung oder Sicherheit. In diesem Kapitel haben wir etwas über die Vorteile erfahren, die uns AOP bietet und wann man AOP einsetzen sollte.

Spring AOP

Spring AOP ist ein AOP-Framework, das sich optimal in die BeanFactory integriert. Spring AOP verwendet Proxy-Objekte, um AOP-Funktionalität anzuwenden, und kann über die XML-Datei oder Annotationen konfiguriert werden. Wir haben in diesem Kapitel gelernt, wie wir Aspekte implementieren und sie mittels Advices auf Joinpoints anwenden. Wir haben gesehen, wie wir für die unterschiedlichen Arten von Advices Parameter und Rückgabewerte verändern können und welcher Advice für welche Situationen passend ist. Die AspectJ-Language spielt dabei eine tragende Rolle, mit der es möglich ist, Pointcuts aus Methoden mittels Wildcards zu definieren.