Modularität durch aspektorientierte Techniken

Ein ursprüngliches Ziel der objektorientierten Sprachen war die Wiederverwendung durch die Modularität der implementierten Lösungen zu erhöhen. In der Realität ist dies nur bedingt der Fall. Warum ist es so? Eine Ursache dafür ist die Vermischung von fachlichen und technischen Belangen. In diesem Artikel zeigen wir wie man durch aspektorientierte Programmiertechniken dieser Ursache vorbeugt und die gewünschte Modularität erreicht.

Andrea Vicentini, Robert Jakubowski

Das Design eines Software-Projektes wird meistens von der Fachlichkeit bestimmt. Die Infrastrukturbelange sind verstreut in den fachlichen Klassen zu finden. In single-inheritance Sprachen ist diese Vermischung nur schwer zu vermeiden.

Ein Ansatz, um dieses Problem zu lösen, ist umfangreiche Programmierrichtlinien zur Verfügung zu stellen, an die sich jeder Entwickler halten muss. Diese Richtlinien beschreiben, wie die Infrastrukturbelange in den fachlichen Code einzubauen sind. Ob diese Richtlinien korrekt umgesetzt werden, kann nur durch Codereviews sichergestellt werden.

In diesem Artikel schlagen wir eine andere Lösung vor. Wir wollen funktional abgeschlossene und wiederverwendbare Einheiten implementieren. Mit diesem Ansatz vermeiden wir Codeduplizierung und erreichen eine höhere Modularität. Um unsere Lösung zu realisieren, verwenden wir aspektorientierte Techniken. Wir setzen absichtlich keines der vorhandenen, aspektorientierten Frameworks ein, weil aus unserer Sicht folgende Ziele mit ihnen nicht erreichbar sind:

- Einfachheit: die Lösung besteht aus nur wenigen Basisklassen.
- Pure Java: die Lösung setzt nur die Standard-Sprachmittel von Java ein (ab J2SE 1.3).
- Direkte Intergrierbarkeit: zusätzliche Konfiguration oder Deploymentschritte (z.B. zusätzliche Ant-Tasks) sind nicht notwendig.
- Leicht erlernbar: es wird keine neue Syntax benötigt.

Im folgenden Kapiteln gehen wir zuerst auf die Architektur unserer Lösung ein. Anschliessend zeigen wir anhand einiger Beispiele, wie unsere Konzepte eingesetzt werden. Zuletzt gehen wir auf Probleme und offene Punkte ein.

Architektur

Starten wir mit etwas Theorie und betrachten die drei zentralen Begriffe unserer Lösung: **Entität**, **Modell** und **Aspekt**.

Der erste Begriff ist die **Entität.** Darunter verstehen wir ein Objekt, welches das Modell und die Aspekte in sich bündelt. Zur Spezifikation einer Entität führen wir eine EntitySpec ein . Die EntitySpec definiert eine API in Form eines Interfaces:

```
interface MyEntity extends MyModel, Aspect1, Aspect2, Interceptor1, Interceptor2 { }
```

Wie aus der Definition ersichtlich, tragen sowohl die Methoden des Modells wie die Methoden der Aspekte zur API der Entität bei. Anders ausgedrückt, die Entität ist ein um Aspekte angereichertes Modell. In unserer Lösung wird das EntitySpec-Interface von einem dynamischen Proxy implementiert. Die Proxy-Instanzen werden von der Klasse *EntityFactory* erzeugt.

Die beschriebene Umsetzung der Entitäten bietet, aus unserer Sicht, klare Vorteile. Der gesamte Code der Anwendung kompiliert gegen die EntitySpec-Interfaces. Konfigurationsdateien sind nicht nötig. Die Lösung kommt allein mit Java-Mitteln aus. Zusätzlich fördert die Aufteilung in Modell und Aspekte die Modularisierung. Ein weiterer Vorteil ist, dass cast- und instanceof-Operationen nicht notwendig sind um zur Laufzeit feststellen zu können, ob die Entität einen gewissen Aspekt enthält oder nicht. Mit unserem Ansatz ist der Code lesbarer und der Kompiler stellt sicher, dass keine Typ-Verletzungen zur Laufzeit auftreten.

Der zweite Begriff das **Modell**. Hier können wir uns kurz fassen, denn das Modell ist ein auf die fachliche Domain bezogenes Objekt. Mit den Modellen bilden wir die fachlichen Anforderungen ab. Die Gesamtheit der Modelle stellt unsere Domain dar.

In unserer Lösung verfügt jedes Modell über das Interface IModel. Anhand dieses Merkmals können wir zwischen Aspekten und Modellen unterscheiden.

Der dritte Begriff ist der **Aspekt**. Es handelt sich dabei um ein abgeschlossenes Modul, das in verschiedenen Entitäten wiederverwendbar ist. Aspekte können sich auf zwei unterschiedliche Weisen auf eine Entität auswirken. Erstens, sie können die Entität um Methoden und Felder erweitern. Zweitens, sie können das Verhalten der Entität beeinflussen. Die erste Art der Aspekte bezeichnen wir als Mixins, die Zweite als Interceptoren.

Die Methoden eines Aspekts werden im zugehörigen Interface zusammengefasst. Dieses Interface wird von uns AspectSpec genannt.

Die Aspekte werden in einer Registry verwaltet. Die Registry wird von der Klasse *Aspects* implementiert. Gleichzeitig ist *Aspects* die Factory für die Aspekte. Sie liefert anhand einer AspectSpec die entsprechende Aspekt-Instanz.

Aspects stellt für die Aspekte einen Namensraum zur Verfügung. Dieser Namensraum ermöglicht einerseits, dass für eine AspectSpec verschiedene Implementierungen für unterschiedliche Entitäten registriert werden können (näheres dazu im Kapitel mit den Beispielen). Andererseits können globale Aspekte definiert werden, die automatisch beim Erzeugen jeder Entität zugeordnet werden.

Das Zusammenspiel der drei Begriffe erläutern wir anhand eines Methodenaufrufs der Entität. Der Aufruf richtet sich gegen den dynamischen Proxy. Dieser Proxy repräsentiert die Entität und hält die Referenzen auf ihre Elemente (Aspekte und Modell). Anhand der Methodensignatur identifizieren wir welches Element den Aufruf entgegen nehmen muss. Bevor der Aufruf ausgeführt wird, durchläuft er die Kette der Interceptoren (s. Abbildung 1).

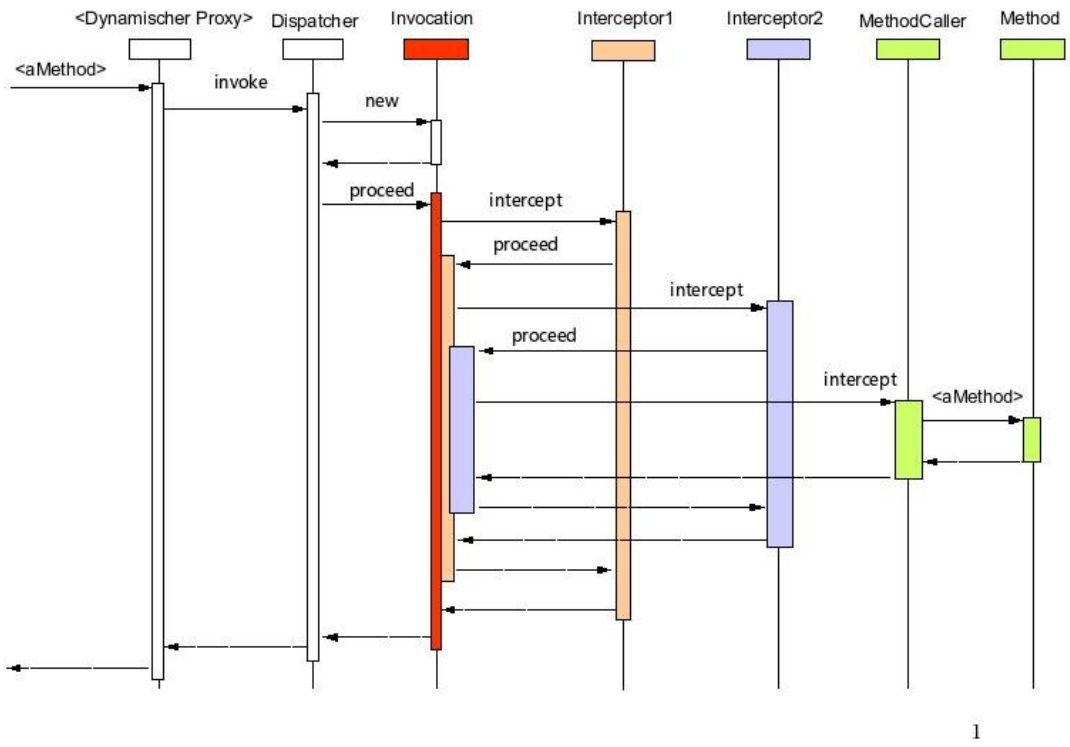

Abb. 1: Verarbeitung eines Methodenaufrufs auf der Entität. In der Abbildung besteht die Interceptor-Kette aus Interceptor1, Interceptor2 und MethodCaller. Jede Farbe entspricht einem Verarbeitungsschritt.

Jeder Interceptor verfügt über die Methode intercept. In dieser Methode wird er entsprechend seiner Funktion aktiv. Anschliessend reicht er die Verarbeitung an seinen Nachfolger weiter oder unterbricht sie. Der Zustand in dem sich der Methodenaufruf befindet, wird in einer Instanz der Klasse Invocation gehalten.

Zum Abschluss des Kapitels verweisen wir auf die Quellen auf der beiliegenden CD. Ihnen können weitere Implementierungsdetails zu den vorgestellten Konzepten und Klassen entnommen werden.

Beispiele

Nach der Einführung in die Architektur wollen wir nun die Konzepte anhand von Beispielen erläutern.

Zuerst gehen wir auf die Syntax der Aspekte und der Entitäten ein. Dazu wählen wir folgendes Beispiel. Wir betrachten einen Kundenauftrag, der aus mehreren Positionen besteht. Die erste Anforderung, die wir stellen, betrifft die Auftragspositionen. Jede Position soll uns Auskunft darüber geben können, ob sie gültig ist oder nicht. Wir realisieren diese Anforderung in vier Schritten.

Im ersten Schritt definieren wir ein Interface für das Modell der Auftragsposition und implementieren dieses Interface in einer Modell-Klasse (s. Listing 1).

Listing 1

```
public interface IOrderPositionModel extends IModel {
    int getQuantity();
    void setQuantity(int quantity);
}

public class Model implements IOrderPositionModel {
    private int quantity;

    public int getQuantity() {
        return quantity;
    }
    public void setQuantity(int quantity) {
        this.quantity = quantity;
    }
}
```

Die Validierung fügen wir im zweiten Schritt hinzu. Dazu realisieren wir eine Variante des Interceptor-Aspekts „Validation“. Um die Implementierung zu vereinfachen führen wir zuerst eine Basisklasse ein (s. Listing 2).

Listing 2

```
public abstract class ValidationAspect implements IValidationAspect {
    private boolean isValid = false;

    public abstract boolean validate(IModel model);

    public boolean isValid() {
       return isValid;
    }

    public Object intercept(Invocation ctx) throws Throwable {
        try {
            return ctx.proceed();
        } finally {
            isValid = validate(ctx.model);
        }
    }
}
```

Der Validation-Aspekt der Auftragsposition leitet die Basisklasse ab und wird somit dazu aufgefordert die Methode *validate* zu implementieren. In dieser Methode entscheiden wir wann eine Auftragsposition gültig ist. In Listing 3 ist dies der Fall, wenn die Quantity der Position grösser 0 ist.

Listing 3

```
public static class OrderPositionValidationAspect extends ValidationAspect {
    public boolean validate(IModel model) {
        if (!(model instanceof IOrderPositionModel))
             throw new IllegalArgumentException();

        return ( ((IOrderPositionModel) model).getQuantity() > 0 );
    }
}
```

Im dritten Schritt spezifizieren wir für die Auftragsposition die zugehörige Entität

```
public interface Entity extends IOrderPositionModel, IValidationAspect {}
```

Zuletzt fügen wir im vierten Schritt die einzelnen, implementierten Bausteine in einer Klasse zusammen (s. Abbildung 2).

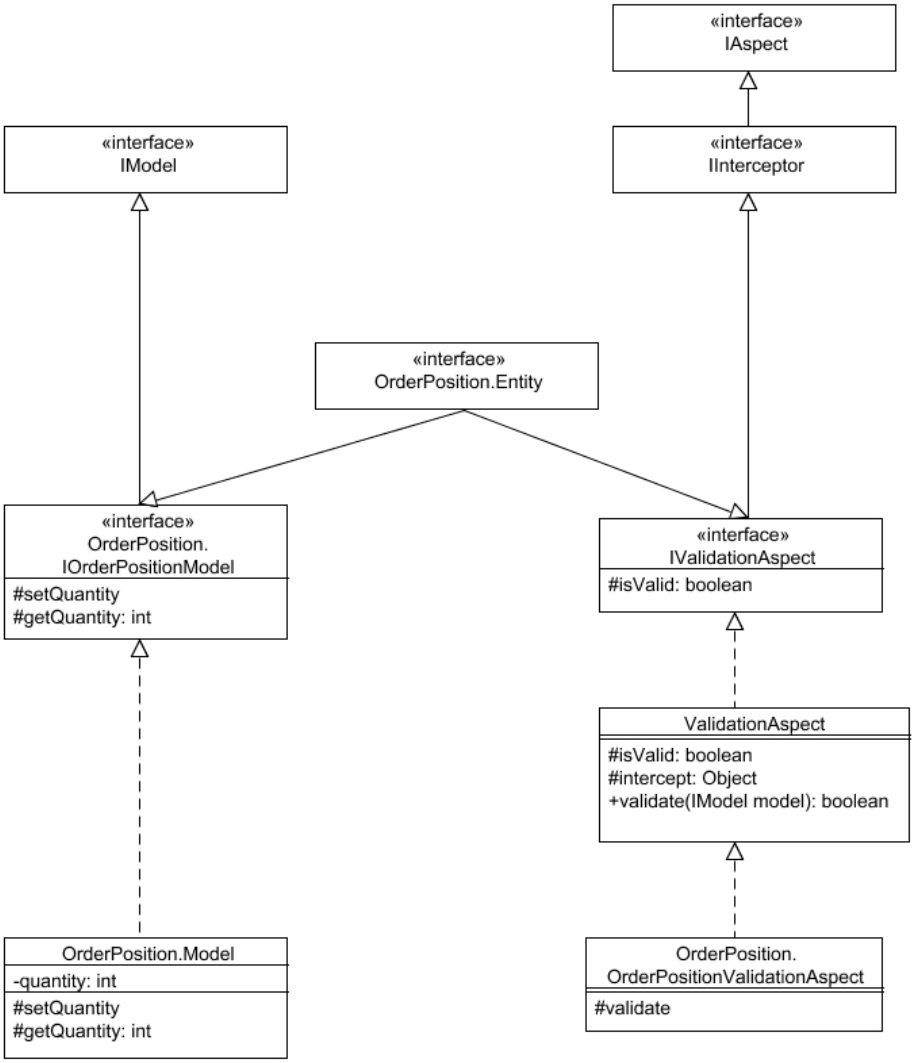

Abb. 1: Diagramm der Klasse OrderPosition aus dem ersten Beispiel.

Damit haben wir die Anforderung realisiert. Wir registrieren das Validation-Aspekt mit dem Aufruf:

```
Aspects.register(IValidationAspect.class, OrderPositionValidationAspect.class);
```

und können die Gültigkeit einer Auftragsposition mit Hilfe der zugehörigen Entität prüfen (s. Listing 4). Zusätzlich zeigt uns das Listing 4 wie auf das Modell der Entität zugegriffen wird.

Listing 4

```
Aspects.register(IValidationAspect.class, OrderPositionValidationAspect.class);

OrderPosition.Entity position = (OrderPosition.Entity) EntityFactory.create(
       OrderPosition.Entity.class, new OrderPosition.Model()
);
```

```
position.setQuantity(2);
if (position.isValid()) System.out.println(„Position is valid“);

position.setQuantity(0);
if (!position.isValid()) System.out.println(„Position is not valid“);
```

Im vorherigen Beispiel sahen wir, wie Aspekte implementiert und einer bestimmten Entität zugeordnet werden. Ausserdem lernten wir, wie eine um Aspekte angereicherte Entität zur Validierung eingesetzt wird.

Als nächstes wollen wir auf Beziehungen zwischen Entitäten eingehen und zeigen, wie sich diese Beziehungen auf die Modelle und die Aspekte auswirken. Dazu erweitern wir unser Beispiel und ordnen die Auftragspositionen einem Auftrag zu.

Analog zu der Auftragsposition definieren wir eine Entität für den Auftrag. Die Beziehung zwischen den Entitäten „Auftragsposition“ und „Auftrag“ wird in den Modellen sehr einfach abgebildet. Das Modell des Auftrags enthält eine Liste von Positionen (s. Listing 5) und bietet Zugriffsmethoden, um z.B. weitere Positionen hinzufügen zu können.

Listing 5

```
public interface IOrderModel extends IModel {
    List getPositions();
    void addPosition(OrderPosition.Entity position);
}

public static class Model implements IOrderModel {
    private List positions;

    public Model() {
       this.positions = new ArrayList();
    }
    public List getPositions() {
       return positions;
    }
    public void addPosition(OrderPosition.Entity position) {
       this.positions.add(position);
    }
}
```

Welche Folgen die Beziehung der Entitäten untereinander für die Aspekte hat, zeigt sich, wenn wir unsere bisherigen Anforderungen an die Validierung erweitern. Wir wollen nicht nur die Gültigkeit einer einzelnen Position, sondern des gesamten Auftrags überprüfen können. Dabei gilt ein Auftrag als gültig, wenn alle Positionen gültig sind.

Eine mögliche Lösung dieser Aufgabe, wäre die beiden Aspekte IOrderValidationAspect und IOrderPositionValidationAspect einzuführen und die beiden Entitäten, wie in Listing 6 dargestellt, zu definieren.

Listing 6

```
public class OrderPosition {
    public interface Entity
       extends IOrderPositionModel, IOrderPositionValidationAspect {
    }
    ...
}

public class Order {
    public interface Entity
       extends IOrderModel, IOrderValidationAspect {
    }
    ...
```

```
}
```

Die Implementierung von IOrderPositionValidationAspect würde ganz analog zur Klasse OrderPositionValidationAspect aus Listing 3 erfolgen. Der Aspekt IOrderValidationAspect würden wir in der neuen Klasse OrderValidationAspect realisieren. Diese Lösung hätte einen wesentlichen Nachteil. Wir verlieren mit ihr die Eigenschaft der Aspekte als querschnittliche Funktionalitäten.

Für die Definition unserer Entitäten ist die konkrete Ausprägung des Validation-Aspekts irrelevant. Wesentlich ist nur, dass die Entität validierbar ist und wir ihre Gültigkeit über die Methode isValid überprüfen können. Diese Methode kann allerdings zur Laufzeit nur dann ein sinnvolles Ergebnis liefern, wenn die Entität eine passende Implementierung anbietet. Listing 7 zeigt wie wir das Problem lösen.

Listing 7

```
public class OrderPosition {
    public interface Entity extends IOrderPositionModel, IValidationAspect {
        Aspects LOCAL = Aspects.forMe()
        .define(IValidationAspect.class, OrderPositionValidationAspect.class);
    }
    ...
}

public class Order {
    public interface Entity extends IOrderModel, IValidationAspect {
        Aspects LOCAL = Aspects.forMe()
        .define(IValidationAspect.class, OrderValidationAspect.class);
    }
    ...
}
```

Einerseits wird die Entität über IValidationAspect als validierbar gekennzeichnet. Andererseits wird über die Methode define pro Entität eine konkrete Implementierung des Validation-Aspekts hinterlegt. Damit können wir die Klasse OrderPositionValidationAspect weiterhin nutzen und benötigen nur eine neue Implementierung für die Validierung des Auftrags (s. Listing 8).

Listing 8

```
public static class OrderValidationAspect extends ValidationAspect {

    public boolean validate(IModel model) {
        boolean result = true;

        if (!(model instanceof IOrderModel))
               throw new IllegalArgumentException();

        for (Iterator iter = ((IOrderModel) model).getPositions().iterator();
             iter.hasNext() && result;) {

               OrderPosition.Entity pos = (OrderPosition.Entity) iter.next();
               result = pos.isValid();
        }

       return result;
    }
}
```

Fassen wir das Ergebnis zusammen. Die Beziehungen zwischen Entitäten untereinander wirken sich nicht auf ihre Definition aus. Erst in der konkreten Implementierung der zugehörigen Modelle und Aspekte werden die Beziehungen sichtbar. Damit können wir die Definition einer Entität als Komposition bestimmter Eigenschaften betrachten. Damit diese Komposition bei den Aspekten

immer auf die gleiche Art und Weise funktioniert, haben wir eine Art *Aspekt-Polymorphismus* eingeführt und ermöglichen, dass jede Entität ihre eigene Ausprägung eines bestimmten Aspektes implementiert.

Die bisherigen Aspekte arbeiteten auf der Basis von Informationen aus dem Modell. Als nächstes zeigen wir, dass Aspekte auch als Erweiterung des Modells fungieren können. Dazu stellen wir eine neue Anforderung und verlangen, dass für jeden Auftrag und für jede Auftragsposition folgende Informationen hinterlegt werden: „erzeugt am“, „erzeugt von“, „modifiziert am“, „modifiziert von“. Diese Informationen werden in dem Mixin-Aspekt IStandardAttributesAspect verwaltet. Die zugehörige Implementierung ist im Listing 9 abgebildet.

Listing 9

```
public class StandardAttributesAspect implements IStandardAttributesAspect {

    private String creationUser;
    private String modificationUser;
    private Date creationDate;
    private Date modificationDate;

    public Date getCreationDate() {
        return creationDate;
    }
    public void setCreationDate(Date creationDate) {
        this.creationDate = creationDate;
    }
    ...
}
```

Was wir nun tun müssen, damit die erforderlichen Daten für die Entitäten Order und OrderPosition angegeben werden können, ist den Aspekt mit dem Aufruf

```
Aspects.register(IStandardAttributesAspect.class, StandardAttributesAspect.class)
```

zu registrieren und die Definition der Entitäten, wie in Listing 10 dargestellt, zu erweitern. Nach diesen Anpassungen sind beiden Entitäten passend angereichert und der Entwickler kann die verlangten Informationen über die entsprechenden Setter setzen oder über Getter abfragen. Der Aspekt IStandardAttributesAspect ist somit ein Mixin der Entitäten Order und OrderPosition.

Listing 10

```
public class OrderPosition {
    public static interface Entity extends IOrderPositionModel, IValidationAspect,
         IStandardAttributesAspect {
        ...
    }
    ...
}

public class Order {
    public static interface Entity extends IOrderModel, IvalidationAspect,
        IStandardAttributesAspect {
        ...
    }
    ...
}
```

Zum Abschluss des Kapitels möchten wir noch einen Ausblick geben, welche weiteren Möglichkeiten unseres Aspekt-Konzeptes bietet.

Als global bezeichnen wir Aspekte, die nicht explizit bei der Definition einer Entität angegeben werden müssen. Das System fügt sie bei der Instanziierung der Entität automatisch hinzu. Zu den globalen Aspekten zählen wir z.B. Tracing, Profiling.

Auf Basis unserer Lösung sind natürlich solche Aspekte wie Synchronize-Aspekt (Methodenaufrufe werden synchronisiert), Oneway-Aspekt (Aufruf einer void-Methode in einem separaten Thread), oder Security-Aspekt (überwacht und schränkt den Zugriff auf das Modell ein) realisierbar.

Einschränkungen und offene Punkte

In den vorherigen Kapitel zeigten wir, welchen Mehrwert unsere Lösung besitzt. Trotzdem bleiben einige Punkte offen. Dazu gehört, dass einige Operationen in der Klasse EntityFactory und Aspects nicht im Hinblick auf die Performance implementiert sind. Dies könnte durch Einführung von geeignetem Caching verbessert werden.

Für die Entität ist die Realisierung von Methoden, bei denen die Entität als Ganzheit betrachtet wird, problematisch. Dazu gehören z.B. toString, hashCode, equals, compareTo. Um dies zu verdeutlichen, greifen wir nochmal unser Beispiel der Standard Attributes auf. Beim Aufruf von toString auf einer Entität müsste diese Methode die Informationen aus dem Modell und aus den Standard Attributen liefern. Unsere Lösung sieht momentan dafür keine API vor.

Ein weiteres Problem tritt auf, wenn sich die öffentlichen Methoden des Modells gegenseitig aufrufen. Dies findet ausserhalb der Kette der Interceptor-Aspekte statt. Ein Profiling-Aspekt misst z.B. die Anzahl der Methodenaufrufe des Modells. Bei Verschachtelung der Methoden liefert der Aspekt ein falsches Resultat. Wir stellen also fest, dass beim Design der Modelle die Aspekte mitbetrachtet werden sollten.

Die Reihenfolge der Aspekte bei der Deklaration der Entität legt ihre Reihenfolge in der Interceptoren-Kette fest. Dies bedeutet, wenn einer der Aspekte die Verarbeitung unterbricht, werden die darauf folgenden Aspekte nicht angesprochen.

Die Integration unserer Lösung in bestehenden Code erfordert zwei wesentliche Anpassungen. Zu jedem existierenden Modell muss ein Interface mit den public-Methoden extrahiert werden. Dadurch ändert sich die Instanziierung der Modelle. Die bisherigen new-Aufrufe müssen um create-Aufrufe der passenden Entität erweitert werden. Beispiel:

Code vor der Anpassung:

```
MyModel a = new MyModel(param1, param2, param3);
```

Code nach der Einführung der Interfaces:

```
MyModel a = new MyModelImpl(param1, param2, param3);
```

Code nach der Einführung der Entitäten:

```
MyModel a = (MyModel) EntityFactory.create(MyEntity.class,
                new MyModelImpl(param1, param2, param3);
```

Zusammenfassung

Zur Beginn unseres Artikels stellten wir fest, dass objektorientierte Sprachen kein Garant für Wiederverwendung oder Modularität sind. Die vorgestellte Lösung zeigt einen möglichen Weg wie man diese Ziele erreicht. Sie fördert die Modularität. Sie bietet die Basis, um technische und fachliche Belange zu trennen und legt den Fokus der Entwicklung auf das Wesentliche. Ein weiterer Vorteil liegt darin, dass nur pure Java zum Einsatz kommen. Damit ist die Lösung leichter verständlich und besser integrierbar in bestehende Projekte.

Dennoch ist die vorgestellte Implementierung noch nicht abgeschlossen und an manchen Stellen prototypisch. Sie soll als Anregung und ein Ausgangspunkt für weitere Ideen dienen.

Andrea Vicentini arbeitet seit 1998 als Software Entwickler und Architekt. Seine Schwerpunkte sind die Architektur von verteilten Systemen, Design Patterns und J2EE. Kontakt: avicentini@gmail.com

Robert Jakubowski arbeitet seit 1999 als Software Entwickler und Architekt. Seine Schwerpunkte sind Design Patterns, J2EE und Web Portale. Kontakt: robert.jakubowski@web.de

Links & Literatur

- [1] JBoss AOP User Guide 1.0, 2005
- [2] Egon Wuchner, AOP ist mehr als Modularisierung, Java Spektrum 1.2005
- [3] Nicole Wengatz, Spring und EJB, Java Spektrum 1.2005
- [4] Robert E. Filman, Tzilla Elrad, Siobhan Clarke, Mehmet Aksit, Aspect-Oriented Software Development, Addison-Wesley, 2005