# Die 5 Leben des AspectJ

*Markus Völter*, voelter@acm.org, www.voelter.de
*Martin Lippert*, lippert@acm.org, www.martinlippert.com

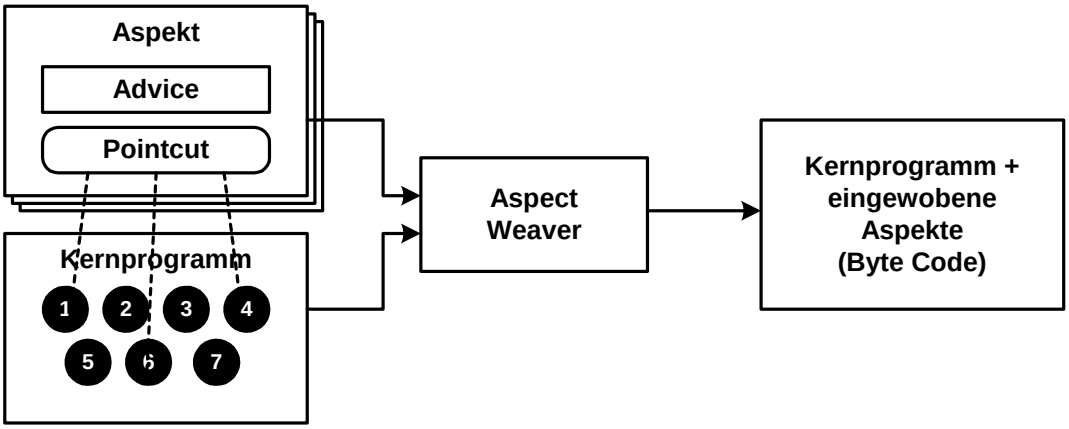
## Einleitung

Aspekte dienen dazu, querschnittliche Belange, sogenannte *Cross-Cutting Concerns*, innerhalb eines Systems zu lokalisieren und damit zu modularisieren. Im Kontext objektorientierter Programmierung sind querschnittliche Belange sind all die Dinge, die sich mit mitteln objektorientierter Programmierung nicht lokalisieren lassen, sondern an verschiedenen Stellen im Code auftauchen obwohl sie eigentlich logisch zusammengehören. Ziel ist es also, das Handling von solch querschnittlichen Belangen genauso einfach zu gestalten wie die Dinge, die sich mit klassischen Mitteln bereits gut modularisieren lassen.

- Aspekte lassen sich mit verschiedenen Mitteln, auf verschiedenen Ebenen eines Systems und mit verschiedenem „Komfort" behandeln. Alle diese Ansätze haben einige Dinge gemeinsam:
- Joinpoints definieren die Menge der Stellen während der Ausführung eines Programmes, an denen ein Aspekt eingreifen kann.
- Advices definieren, welches Verhalten der Aspekt der Kernsoftware hinzufügen soll.
- Zu guter Letzt stellt ein Pointcut eine Menge von Joinpoints dar, an denen ein bestimmter Advice nun tatsächlich greifen soll. Eine Joinpointsprache dient dazu, Pointcuts zu definieren, indem aus allen möglichen Joinpoints eines Systems eine Untermenge selektiert wird.

Ein bekanntes und inzwischen weit verbreitetes Tool ist AspectJ. AspectJ behandelt Aspekte auf Sprachebene, in diesem Fall Java. Es definiert dazu eine Reihe von Spracherweiterungen, die die oben genannten Konzepte umsetzen. Rein technisch funktioniert die Umsetzung mittels Quell- oder Bytecodeweaving, was bedeutet, dass der Aspektcode statisch mit dem Kernsystem verwoben wird[1]. Damit erlaubt es AspectJ, reinen Java-Bytecode zu erzeugen und auf jeder Java Virtual Machine auszuführen. Die folgende Skizze erläutert dies.

[1] Neben AspectJ gibt es eine ganze Reihe weiterer Spracherweiterungen und AOP-Frameworks, die die beschriebenen Techniken umsetzen. Teilweise erlauben diese Werkzeuge auch, Aspekte dynamisch mit dem Kernsystem zu verbinden.




Neben der Modularisierung von „echten“ Cross-Cutting Concerns kann AspectJ auch zu einer ganzen Reihe weiterer Aufgaben herangezogen werden. Natürlich funktionieren alle nach demselben Prinzip: Aspekte werden definiert, implementiert und anschließend mit dem Kernprogramm verwoben. Allerdings legt man bei den verschiedenen Ansätzen den Schwerpunkt auf verschiedene Dinge und löst eigentlich auch unterschiedliche Probleme. Wir haben die folgenden fünf Anwendungsgebiete für AspectJ identifiziert:

- Das Behandeln wirklicher Cross-Cutting Concerns
- Verschiedene Aspekte zu einem „Dingens“ hinzufügen
- Das Modifizieren von Software im Nachhinein (ohne Quellcode)
- Die Verwendung als Diagnose- und Entwicklungstool
- Generative Programmierung mit AspectJ

Wir werden auf diese 5 Anwendungsbereiche im Folgenden eingehen. Dabei sei noch erwähnt, dass einige dieser Anwendungsbereiche allgemein für Aspektorienterte Systeme gelten, andere sind spezifisch für AspectJ, weil sie ausnützen, wie AspectJ die AOP-Idee technisch umsetzt.

## Wirkliche Cross-Cutting Concerns

Wirkliche Cross-Cutting Concerns sind solche Dinge, die querschnittlich zu einem System liegen. In aller Regel sind sie optional in dem Sinne, dass das System auch ohne des Aspekt Sinn macht, aber eben bestimmte (querschnittliche) Features nicht besitzt. Ein gutes Beispiel ist die Frage, ob ein System threadsicher sein soll. Threadsicherheit bedeutet, dass man sich beim Zugriff auf gemeinsam genutzte Ressourcen um Locks kümmern muss. Der folgende (Pseudo-)Code ist nötig:

```
check for and acquire Lock
  use shared resource
release lock
```

In Java gibt es verschiedene Möglichkeiten, Locks zu implementieren, die einfachste unter Zuhilfenahme des spracheigenen Monitor-Konzeptes (*synchronized*-Blöcke). Soll nun ein ganzes System threadsicher werden, so muss man sich an verschiedensten Stellen um Locking kümmern. Dieser querschnittliche Belang kann mit AspectJ modularisiert und dem System separat hinzugefügt werden. Wichtig ist dabei festzuhalten, dass das System als solches auch ohne den Aspekt Sinn macht – es kann dann nur nicht in Multithreading-Umgebungen eingesetzt werden.

Neben oft technisch motivierten Querschnittsthemen lassen sich mitunter auch fachlich motivierte Belange identifizieren, die quer zur „normalen" objektorientierten Systemdekomposition liegen. Betrachten wir beispielsweise ein Bonus-Programm, welches bei bestimmten Aktionen in verschiedenen Systemteilen Bonus-Punkte gutschreiben muss, ohne die eigentliche Funktionalität des Systems zu verändern. Ein entsprechender Aspekt kann diese Aufgabe lösen. Die Funktionalität des Bonus-Programms muss nicht über die verschiedenen Systemteile hinweg verstreut werden.

Nachdem AOP anfänglich vor allem auf technische Aspekte fokussiert war, werden nun immer mehr auch fachliche Belange in Aspekten implementiert. Die Sprache CAESAR [CAESAR], die auf AspectJ beruht, hat genau diesen Fokus.

## Implementierung von Kollaborationen

Bei den „wirklichen" querschnittlichen Belangen wie oben beschrieben war die Kernaussage, dass sich ein Belang quer zu einem Softwaresystem legt. Viele Teile (Klassen) des Systems sind davon betroffen. Im Gegensatz dazu kann man AspectJ auch dazu verwenden, Verantwortlichkeiten zu strukturieren, die bisher in einem Systemteil oder auch nur in einer einzelnen Klasse realisiert wurden.

Beispielsweise kann man mittels AspectJ eine Klasse zum Subjekt eines Observer Patterns machen. Dazu ist es nötig, dass die Klasse an bestimmten Stellen im Ablauf des Programmes (Joinpoint!) eine Nachricht an eine Reihe von Observern schickt. Zu realisieren ist also sowohl das Verwalten der Observer, die an einer Veränderung des Subjektes interessiert sind, als auch das tatsächliche Benachrichtigen. Man erreicht damit, dass bestimmte Klassen des Systems zusätzlich zu ihren Kernaufgaben noch zusätzliche Rollen spielen – hier die des Subjektes.

Bezüglich des Gesamtsystems wird hier natürlich auch etwas querschnittliches implementiert (das Observer-Pattern eben). Allerdings fokussiert der Observer-Aspekt in diesem Fall auf eine oder mehrere konkrete Klassen. Insbesondere macht das System als Ganzes ohne den Observer-Aspekt ja auch keinen Sinn. Es ist also nicht optional, ob der Aspekt dazugewoben werden soll – er ist ein integraler Bestandteil des Systems. AspectJ wird lediglich verwendet, um das System besser zu strukturieren.

## Das Modifizieren von Software im Nachhinein

AspectJ kann seit der Version 1.1 auch auf Bytecodeebene arbeiten. Es ist in der Lage, Aspekte auf bereits vorhandenen Bytecode anzuwenden. Es wird damit möglich, 3rd-Party Code für den man keinen Quellcode besitzt, mit Aspekten zu beeinflussen. Man kann beispielsweise mittels *around*-Advices die Implementierung vorhandener Methoden überschreiben oder, mittels *around*, *before* und *after*, passend ergänzen. Exceptions können abgefangen werden oder mittels eines passenden Typs gewrappt werden. Es ist also möglich, Programme im Nachhinein zu modifizieren, *ohne* den betreffenden Quellcode zu besitzen.

Das klingt zunächst wie das „schmutzige" Werkzeug eines Hackers. Das Gegenteil ist der Fall. Mit der Möglichkeit, Aspekte in Bytecode zu weben, ohne den entsprechenden Quellcode zu besitzen, ergeben sich eine ganze Reihe interessanter neuer Möglichkeiten.

Aus Sicht eines Application-Servers wird es beispielsweise möglich, durch den Application-Server vordefinierte Aspekte in deployte Anwendungen zu weben. Der Sourcecode der Anwendungskomponenten muss dazu nicht vorliegen. Auch wenn dazu noch nicht unbedingt AspectJ verwendet wird, nutzt JBoss in der Version 4 AOP-Techniken in dieser Art und Weise (siehe [JBOSS]).

Als ein anderes Beispiel können unterschiedliche Classloader in Java dienen. In Eclipse wird beispielsweise der Code eines jeden Plugins von einem eigenen Classloader geladen um die Abhängigkeiten zwischen verschiedenen Plugins besser managen zu können. Aus diesem Grund darf innerhalb eines Plugins nicht direkt der *SystemClassLoader* benutzt werden. Stattdessen müssen Plugins den Classloader des Plugins verwenden, um Resourcen oder Klassen des Plugins zu laden. Daran müssen sich auch 3rd-Party-Libraries halten, die von einem Plugin verwendet werden. Benutzt aber eine 3rd-Party-Library direkt den SystemClassLoader (eine frühere Version des Xerces-XML-Parsers war beispielsweise so implementiert), hat man in seinem Plugin das Nachsehen. Mit AspectJ lässt sich allerdings ein Aspekt definieren, der den Aufruf auf den SystemClassLoader durch einen Aufruf auf den Plugin-eigenen ClassLoader „umbiegt".[2]

Die Beispiele zeigen, dass AspectJ für solche Fälle ein mächtiges Werkzeug sein kann.

## AspectJ als Unterstützung während der Entwicklung

AspectJ kann auch bei der Entwicklung von Software helfen; im auszuliefernden System ist dann von AspectJ nichts mehr zu sehen. Dabei ist zu unterscheiden zwischen der Unterstützung zur Laufzeit, und der zur Compilezeit. Betrachten wir zunächst Letzteres.

---

[2] Wir möchten darauf hinweisen, dass das Modifizieren einer 3rd-Party-Library möglicherweise rechtliche Fragen nach sich ziehen kann. Diskutieren werden wir diesen Punkt allerdings nicht im Rahmen dieses Artikels.

Ein wenig bekanntes Feature von AspectJ besteht darin, das bestimmte Joinpoints (die sogenannten „statischen" Joinpoints) bereits zur Compilezeit ausgewertet werden. Besonders nützlich ist, dass Compilerfehler und -Warnungen provoziert werden können. Damit wird es beispielsweise möglich, bestimmte Coding Conventions zu überprüfen. Beispielsweise kann eine Warnung ausgegeben werden, wenn ein (nicht statisches, nicht finales) *public* Attribut definiert wird. Eine andere Möglichkeit ist die Implementierung von feingranularer Zugriffskontrolle. Man kann den Zugriff auf bestimmte Klassen feiner regulieren als dies mit pure Java möglich ist. Man kann zum Beispiel im Rahmen der Entwicklung eines Frameworks nur den Zugriff von anderen Framework-Klassen erlauben. Darüber hinaus wäre es auch möglich, in einer Mehrschichten-Architektur Zugriffsverletzungen zwischen den Schichten zu entdecken und durch entsprechende Compiler-Warnungen zu dokumentieren (siehe [Bodkin03]).

Auch die Unterstützung der Entwicklung zur Laufzeit ist ein wichtiges Anwendungsfeld. Das Standardbeispiel für AspectJ, Logging, fällt in diese Kategorie. Es wird damit einfach möglich, die Reihenfolge des Aufrufs der Methoden im System aufzuzeichnen, inkl. der Aufruferklassen! Damit wird ein vollständiger Trace des Systems (oder Teilen davon) möglich, ohne eine Zeile Code manuell in das System zu bauen. Lediglich die Definition eines Aspektes ist nötig. Vor allem zum Debugging von Race-Conditions in Multithreading-Umgebungen ist dieser Ansatz extrem nützlich. Ein normaler Debugger verändert das Zeitverhalten des Systems oft so, dass es zu den Fehlern, die man sucht, oft gar nicht mehr kommt.

Auch das Auffinden bestimmter (potentieller) Fehlerbedingungen zur Laufzeit (z.B. Parameter oder Rückgabewert ist *null*) kann durch einen einfachen Aspekt unterstützt werden, indem er im Fall der Fälle einfach eine Fehlermeldung ausgibt. Letztendlich läuft das auf die Realisierung von Design-by-Contract mittels AspectJ hinaus (siehe [LL00]). Pre- und Postconditions sowie Invarianten werden als Aspekt modelliert und mit dem Kernsystem verwoben. Als Beispiel kann eine Invariante dienen, die sicherstellt, dass der Fahrer eines Autos immer älter als 18 Jahre alt ist. Bei der Implementierung von Invarianten gibt es zwei Aspekte (nicht im Sinne von AOP ☺) zu beachten:

- Zum einen muss der konkrete Ausdruck definiert werden, der die eigentliche Überprüfung durchführt, also beispielsweise die folgende Operation:

```
class Auto {
  // irgendwelche Operationen
  public void ensureLegalDriver() {
    if ( driver().getAge() < 18 ) throw ConstraintViolation();
  }
}
```

- Zum anderen muss entschieden werden, wann die Überprüfung ausgeführt werden soll (theoretisch, als Invariante, immer; in der Praxis geht das aber oft nicht).

Mittels AspectJ lässt sich beides recht elegant lösen. Die Operation oben lässt sich als *Introduction* zur Klasse hinzufügen. Das Anstoßen der Überprüfung kann man durch entsprechende Joinpoints realisieren, die zum Beispiel *nach* jeder das Auto oder der Fahrer verändernden Methode greifen.

Schön an dieser Lösung ist auch, dass man die Constraint-Überprüfung leicht ein- und ausschalten kann indem man den Aspekt zu dem System hinzufügt oder eben nicht.

## Softwaresystemfamilien und Generative Programmierung

Generative Programmierung hat das Ziel, im Rahmen einer Softwaresystemfamilie basierend auf einer Spezifikation optimierte, und genau auf den Anwendungsfall passende Produkte möglichst automatisch zu erstellen (was für ein Satz!). Oft definiert man die Features, die ein bestimmtes Produkt haben soll, mittels sogenannter Featuremodelle.

Mittels AspectJ lassen sich die einzelnen (querschnittlichen oder auch lokalisierten) Features implementieren. Die Spezifikation eines Produktes (der Familie) ist dann die Definition der zu verwebenden Aspekte. Je nach Art, Menge und Reihenfolge der verwobenen Aspekte wird ein anderes Produkt erstellt.

Ein oft verwendetes Beispiel für GP ist eine Produktfamilie von Stacks. Ein Stack hat immer eine gewisse Basisfunktionalität, insbesondere die Operationen *push* und *pop*. Eine skelettartige Implementierung für Integer-Stacks könnte folgendermaßen aussehen:

```
public class IntStack {
  private List elements = new ArrayList();
  public void push( int e ) { …}
  public int pop() {…}
}
```

Nun kann ein Stack – entsprechend dem Beispiel – eine ganze Reihe weiterer Features besitzen; dazu zählen unter anderem:

- Bounds checking (ok, ist in Java kein Problem…)
- Thread-Sicherheit
- Eine Operation, die die aktuelle Größe zurückliefert (*size*)
- Eine *peek* Operation

Einige dieser Features lassen sich lokalisieren (wie z.B. die size oder peek Operationen), andere sind querschnittlich. Mittels AspectJ kann man nun für jedes dieser Features einen Aspekt definieren – die lokalisierten Dinge realisiert man über Introductions. Durch die Auswahl, welche Aspekte man mit obigem Skelett verwebt erreicht man die im Zusammenhang mit GP erforderliche „Konfigurationsmöglichkeiten“ eines Produktes im Rahmen einer Softwaresystemfamilie.

Dieser Artikel zeigt, dass AspectJ ein Tool ist, welches in verschiedensten Anwendungsszenarien zu ganz verschiedenen Zwecken eingesetzt werden kann. Die Unterscheidung dieser verschiedenen Szenarien ist wichtig um bewusst entscheiden zu können, wann und wie man AspectJ verwendet. Ist es beispielsweise aufgrund einer „restriktiven" IT-Strategieabteilung nicht „erlaubt", AspectJ in Produktion einzusetzen, so kann man es trotzdem während der Entwicklungszeit einsetzen, um Code Conventions zu checken oder Constraints zu überprüfen.

## Referenzen

[CASESAR] Mezini, Ostermann: *CAESAR Project*, http://www.st.informatik.tu-darmstadt.de/static/pages/projects/caesar/CAESAR.jsp

[LL00] Martin Lippert, Cristina Videira Lopes: *A Study on Exception Detection and Handling Using Aspect-Oriented Programming*, in: Proceedings of the 22nd International Conference on Software Engineering, Ireland 2000, ACM press, pp. 418-427, 2000.

[JBOSS] http://www.jboss.org/developers/projects/jboss/aop

[Bodkin03] Ron Bodkin and Adrian Colyer: *Enterprise Aspect-Oriented Programming with AspectJ*, Präsentationsmaterial verfügbar unter: http://www.newaspects.com/