

schwerpunkt

DI UND AOP: EINE BILANZ

„Dependency Injection“ (DI) und Aspektorientierte Programmierung (AOP) sind seit Jahren wesentliche Grundlage für Enterprise-Java-Anwendungen im Java-EE- und Spring-Umfeld. Es ist an der Zeit, Bilanz zu ziehen, den bisherigen Erfolg in der Praxis und die Vorteile dieser Ansätze darzustellen und einen Blick in die Zukunft beider Technologien zu wagen.

Normalerweise muss jedes Objekt in einem System sich die anderen Objekte, deren Methoden es aufrufen will, zusammensuchen. Bei *Dependency Injection (DI)* werden die Objekte von außen zugewiesen, also, metaphorisch gesprochen, „injiziert“. Neben der Zuweisung der abhängigen Objekte werden diese Objekte meistens auch erzeugt. Daher gehört DI zur Kategorie der Entwurfsmuster, die sich um das Erzeugen von Objekten kümmern, wie z. B. *Factory Method*, *Singleton* oder *Abstract Factory* (vgl. [Gam94]). Ähnlich wie DI beschreiben diese Patterns nicht nur, wie die Objekte erzeugt werden, sondern auch, wie auf sie zugegriffen werden kann. So gibt es beim *Singleton* meistens eine Methode, die Zugriff auf die einzige Instanz gewährt. Im Extremfall muss das Objekt noch nicht einmal erzeugt werden, sondern es kann bereits vorhanden sein.

DI hat hier aber eine andere Vorgehensweise als *Singleton* oder *Factory Method* entsprechend der *Inversion of Control*: Der eigene Code sucht sich keine Objekte zusammen und erzeugt auch keine, sondern sie werden von außen in den Code injiziert. Nehmen wir als Beispiel einen Service für die Verarbeitung von Bestellungen: Dieser bekommt abhängige Objekte, wie z. B. die Kundenverwaltung, „injiziert“. Konkret: Eine Set-Methode wird aufgerufen oder der Service wird dem Konstruktor als Parameter übergeben. Um dies zu erreichen, kann der Entwickler selbst Code schreiben – dieser wird jedoch schnell unübersichtlich, sodass sich in der Praxis DI-Container wie Spring oder Java EE durchgesetzt haben.

Vorteile von DI

DI weist eine ganze Reihe von Vorteilen auf:

Die Softwarestruktur wird verbessert. Der Bestellservice weiß nicht mehr, welche Klasse die Kundenverwaltung konkret implementiert. Natürlich muss er noch die Schnittstelle kennen, aber mehr nicht. Dadurch werden die Abhängigkeiten der einzelnen Objekte reduziert. Ohne DI können die Objekte zum Beispiel Factories nut-

zen, um Abhängigkeiten aufzulösen. Insbesondere dieses Vorgehen führt zu Problemen in der Architektur: Jedes Objekt nutzt die Factory, die wiederum jedes Objekt erzeugen kann (siehe Abbildung 1). Also hängt jedes Objekt über die Factory von jedem anderen Objekt im System ab. Möglichst wenige Abhängigkeiten sind aber Voraussetzung für die leichte Änderbarkeit von Software. Mit DI hängt jedes Objekt nur von den genutzten Schnittstellen ab. Die Implementierung der Schnittstelle ist nur dem DI-Container bekannt, zu dem die Objekte aber keine Abhängigkeit haben. (siehe Abbildung 2)

Das Testen wird vereinfacht. Dem Bestellservice kann sehr einfach eine Kundenverwaltung injiziert werden, die immer einen bestimmten Kunden zurückgibt, der zum Testen eines bestimmten Szenarios nützlich ist. Dieser Ansatz ist für Unit-Tests sinnvoll. Dabei wird eine einzelne Klasse getestet und alle abhängigen Objekte werden durch entsprechende einfachere Versionen (so genannte *Mocks* oder *Stubs*) ersetzt, die entsprechende konstante Testdaten zurückliefern. Durch DI kann man solche *Mocks* oder *Stubs* sehr einfach

der autor



Eberhard Wolff

[E-Mail: eberhard.wolff@adesso.de]

arbeitet als Architektur- und Technologie-Manager für die adesso AG in Berlin. Er wurde von Oracle in die Gruppe der Java-Champions aufgenommen, ist Autor einiger Fachbücher und spricht regelmäßig auf verschiedenen Konferenzen.

injizieren: Man erzeugt das zu testende Objekt, ohne den sonst üblichen DI-Mechanismus zu nutzen. Dann injiziert man die entsprechenden *Mocks* oder *Stubs*.

Oft wird die Konfigurierbarkeit der Komponenten als weiterer Vorteil von DI übersehen. Neben abhängigen Objekten wie der Kundenverwaltung können für den Bestellservice auch Einstellungen wie der Mindestbestellwert durch DI konfiguriert werden. Bei Spring beispielsweise können Einstellungen aus *Properties*-Dateien oder *System-Properties* genutzt werden und natürlich ist der Mechanismus auch erweiterbar, sodass Einstellungen aus praktisch jeder Quelle ausgelesen werden können. Das ist von Vorteil, weil so Fehler – wie nicht vorhandene Konfigurationseinstellungen oder Konfigurationsdateien – durch den DI-Container behandelt werden. Das reduziert den rein technischen Infrastruktur-Code in den Anwendungen.

Auf den ersten Blick ist DI eine Erfolgsgeschichte: Ursprünglich in Spring und

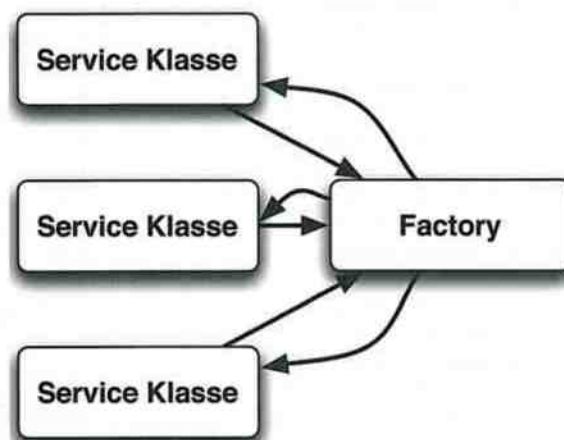


Abb. 1: Ohne DI nutzen Services häufig Factories, um Referenzen auf andere Services zu erzeugen. So entsteht eine Struktur, in der jeder Service die Factory kennt und die Factory dann jeden Service. Letztendlich hängt das gesamte System voneinander ab, was Wiederverwendbarkeit behindert und ein Zeichen für eine schlechte Architektur ist.

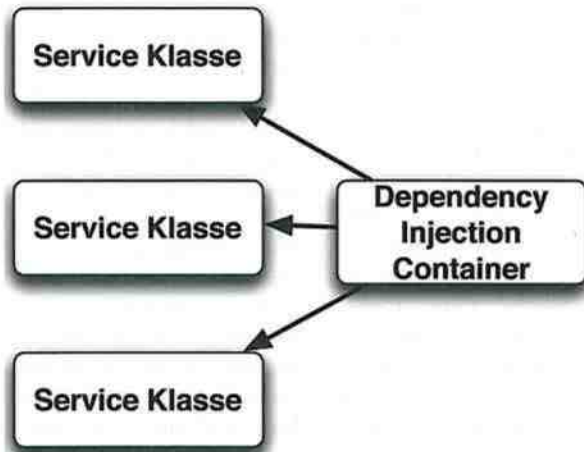


Abb. 2: Mit DI werden die Services durch den DI-Container erzeugt. Dadurch haben sie weniger Abhängigkeiten und können leicht in einem anderen Kontext eingesetzt werden oder für Tests mit anderen Abhängigkeiten versehen werden.

anderen Projekten, wie z. B. „PicoContainer“ (vgl. [Pic]), implementiert, hat es über EJB 3.0, JSR 299 (CDI) und JSR 330 Einzug in Java EE gefunden und mittlerweile kommt keine Java-EE-Anwendung mehr ohne diesen Ansatz aus. Dadurch sollte sich zumindest die Strukturverbesserung tatsächlich auch in den meisten Projekten manifestiert haben. Bei den anderen beiden Punkten ist ein solcher Optimismus nicht angebracht:

DI vereinfacht, wie bereits dargestellt, Unit-Tests. Neben diesen Tests bieten DI-Container wie Spring oder EJB 3.1 als Alternative die Flexibilität, auch außerhalb eines Applikationsservers komplexe Infrastrukturen zu starten. Dort können die ech-

ten Implementierungen des Kundenservice mit allen abhängigen Objekten ablaufen, um damit Tests durchzuführen. Diese Tests decken Fehler im Zusammenspiel der Komponenten auf, zählen also zu den Integrationstests. Vorteil der DI bei Integrationstests ist die Flexibilität, den Code in unterschiedlichen Umgebungen laufen zu lassen, also auch außerhalb eines Applikationsservers. Die zunehmende Popularität von Tools wie „Arquillian“ (vgl. [JBo-a]) zeigt aber, dass Tests in Applikationsservern zunehmend genutzt werden. Die Vorteile von DI werden also nur teilweise realisiert.

Die Konfiguration von Objekten ist leider nicht bei allen DI-Containern gegeben.

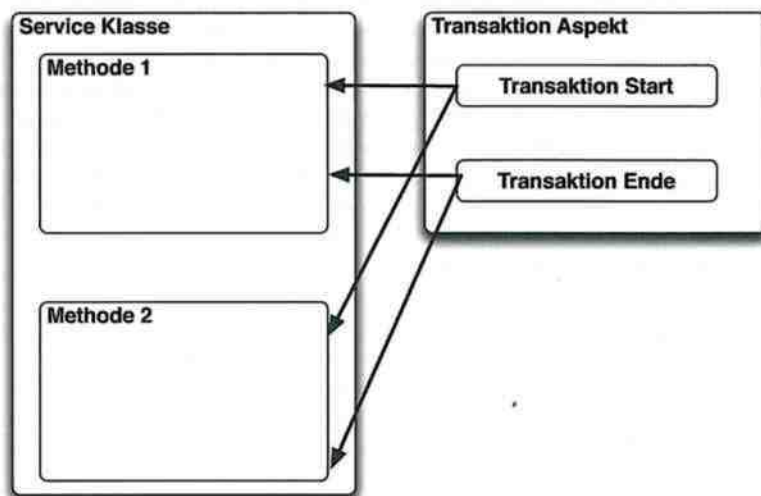


Abb. 3: Mit AOP wird die Logik für die Behandlung der Transaktionen zentral implementiert und dann an die passenden Stellen in der Geschäftslogik integriert.

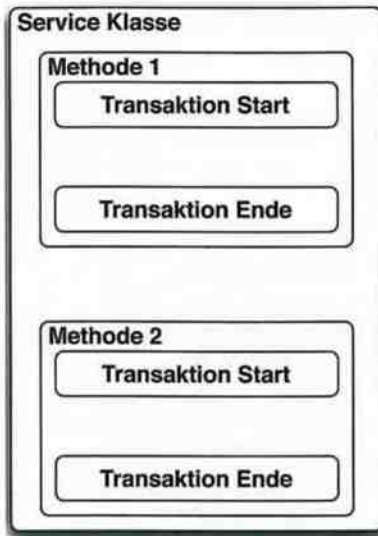


Abb. 4: Ohne AOP muss in jedem Service in jeder Methode Code – zum Beispiel für die Behandlung von Transaktionen – geschrieben werden.

Viele Projekte laden immer noch ihre Konfigurationen selbst – mit manchmal erheblichen Aufwand und nicht immer fehlerfrei. Hier werden die Möglichkeiten der DI-Lösungen immer noch zu wenig genutzt.

Lange wurde gegen DI auch ins Feld geführt, dass DI „Programmierung in XML“ sei. Durch die Möglichkeit zur Nutzung von Java-Annotationen hat sich dieses Problem mittlerweile erledigt. Aber die Annotation haben auch Nachteile: Änderungen an der Konfiguration können oft nur durch Änderungen im Programmcode erreicht werden. Das bedeutet auch ein erneutes Kompilieren und Installieren der Anwendung. Bei XML reichen eine Änderung an einer Konfigurationsdatei und ein Neustart. Dadurch sind die Systeme mit XML flexibler anpassbar.

Wie sieht die Nutzung von DI in neuen Sprachen aus? Für DI in Scala wird das *Cake Pattern* (vgl. [Ode08]) empfohlen. Dieses beschreibt die Umsetzung von DI in Scala mit bestimmten Codestrukturen. Beim Scala-Framework „Lift“ sind die Konfiguration und die benötigten Abhängigkeiten der Objekte vollständig im Code implementiert. Dieser Ansatz benötigt sehr viel (siehe oben). Trotz der Unterstützung von DI in Lift nannte der Entwickler des Frameworks Dave Pollack den größten Teil von DI einen „load of horse poop“ (vgl. [Osd]). Vielleicht baut Lift deswegen nicht auf den Erfahrungen der etablierten DI-Frameworks auf. Zusam-

menfassend kann man sagen, dass sich DI mittlerweile in den Frameworks und in der Infrastruktur etabliert hat, auch wenn nicht der volle Wert dieser Ansätze genutzt wird.

Aspektorientierte Programmierung

Ideen zur *Aspektorientierten Programmierung (AOP)* lassen sich bis in die 1990er Jahre zurückverfolgen. Der Aspekt wird in den eigentlichen Code „eingewebt“, um dadurch so genannte *Cross Cutting Concerns* zentral an einer Stelle zu implementieren (siehe Abbildung 3), statt dies ständig im Code zu tun (siehe Abbildung 4). Eine zentrale Implementierung von einigen speziellen *Cross Cutting Concern* – wie Sicherheit und Transaktionen in Enterprise-Anwendungen – wurde bereits in EJB 1.0 angeboten. EJB enthielt damals aber kein AOP-System, mit dem ein Entwickler ähnliche Erweiterungen implementieren könnte. Um eine solche Basis zu schaffen, wurde AOP plötzlich interessant und es entstanden Projekte wie das mäßig erfolgreiche JBoss AOP (vgl. [JBo-b]) oder Spring mit der AOP-Unterstützung. Spring AOP führte später eine Kompatibilität mit AspectJ ein.

Dadurch hatten Entwickler ein sehr mächtiges Werkzeug, um selbst Erweiterungen analog zu Transaktionen und Sicherheit zu implementieren. Spring bietet auch einige Aspekte an, die über Transaktionen und Sicherheit hinaus gehen. Aber AOP hatte dann irgendwann den Ruf, komplex zu sein, vielleicht, weil der Entwickler eine neue Sprache für die Definition der Stellen im Code lernen musste, an denen die Aspekte ansetzen sollen: die *Pointcut Expressions*. Ein typischer Entwickler nutzt diese Sprache nicht sehr häufig, was das Erlernen nicht erleichtert. Eine andere Befürchtung ist oft, dass der Code an irgendwelchen Stellen erweitert wird und so die Entwickler den Überblick verlieren, was wo passiert. Wenn aber technische Dienste, wie die Behandlung von *Exceptions* durch AOP, getrennt von der Hauptlogik implementiert werden, sind die Aufgaben klar getrennt – wie das bei ande-

ren technischen Services auch der Fall ist.

Durch die konsequente Nutzung von AOP können also viele Anwendungen vereinfacht und viele technische Elemente zentral implementiert werden. Diese Möglichkeiten werden bei Spring leider nur selten genutzt. In der Java-EE-Welt hat sich inzwischen ein Ansatz etabliert, in dem nur Interceptoren implementiert werden und keine vollständige Sprache für *Pointcuts* wie in der Spring/AspectJ-Welt.

Noch ein Blick auf neuere Sprache: Scala unterstützt AOP gar nicht. Es ist allerdings möglich, einzelne Methoden mit Transaktionen oder Sicherheitsabfragen zu versehen. Dafür nutzt man Funktionen, denen man die ursprüngliche Methode übergibt. Die Funktion führt dann vor und nach der Methode Logik für Transaktionen oder Sicherheit aus. Dazu nutzt man das Scala-Feature der *Higher Order Functions*. Das sind Funktionen, die selber Funktionen bzw. Methoden als Parameter akzeptieren. Mit Scala ist es aber nicht möglich, alle Methoden einer Klasse oder gar aller Klassen mit einer bestimmten Annotation mit solcher Funktionalität auszustatten. Das ist erstaunlich, denn ein solches Feature ist für Enterprise-Anwendungen sehr hilfreich. Bei dynamischen Sprachen wie Groovy sind im Gegensatz zu Scala ähnliche Features durch Meta-Programming elegant implementierbar.

Fazit

Bei AOP nutzt Java EE einfachere, aber auch weit weniger mächtige Ansätze. Die vollständige Mächtigkeit von AOP wird nur in wenigen Projekten realisiert. Bei DI ist die Situation besser, aber sicher gibt es auch dort Optimierungspotenzial. Hauptproblem ist, dass die Nutzungsmöglichkeiten von DI und AOP vielen Entwicklern nicht vollständig bekannt sind. Diese Werkzeuge sind recht mächtig, sodass sich eine intensive Auseinandersetzung auf jeden Fall lohnt. Schade ist, dass die Entwickler neuer Technologien nicht immer auf den Erfahrungen mit dem Einsatz der vorhandenen Frameworks aufbauen. ■

Literatur & Links

- [Gam94] E. Gamma, R. Helm, R.E. Johnson, Design Patterns. Elements of Reusable Object-Oriented Software, Addison-Wesley 1994
- [JBo-a] JBoss, Arquillian, siehe: <http://www.jboss.org/arquillian>
- [JBo-b] JBoss, AOP, siehe: <http://www.jboss.org/jbossaop>
- [Ode08] Martin Odersky, Lex Spoon, Bill V.: Programming in Scala, artima Press, 2008
- [Osd] Osd.com, Liftweb, siehe: <http://osdir.com/ml/liftweb/2011-07/msg00048.html>
- [Pic] PicoContainer, siehe: <http://picocontainer.org/>