

Teil I

Grundlagen

2 Leichtgewichtige, POJO-basierte Enterprise-Applikationen

»Der Wert eines Dialogs hängt vor allem von der Vielfalt der konkurrierenden Meinungen ab.«

Karl R. Popper

2.1 Kurz gefasst

Dieses Kapitel beschreibt die architektonischen Grundlagen von EJB 3.x sowie die verwendeten Konzepte – darunter Inversion of Control, Dependency Injection, Configuration by Exception und Annotationen. Diese Konzepte sind die Zutaten für die Entwicklung von leichtgewichtigen, POJO-basierten Enterprise-Applikationen auf Basis der EJB-3.x-Technologie.

Wir beschreiben die technischen Aspekte, die hinter diesen Konzepten stehen, motivieren deren Einsatz und erläutern die Auswirkungen auf die EJB-Softwareentwicklung und das Laufzeitverhalten von EJB-Komponenten (bzw. dessen Steuerung). Klingt komplizierter, als es tatsächlich ist.

2.2 Der Blick zurück

Geht man der Frage nach, warum die oben aufgeführten Konzepte erst mit der Version 3.0 in der EJB-Technologie verankert wurden, so lassen sich im Wesentlichen zwei Gründe anführen: Einige der Konzepte gab es zum Zeitpunkt des Erscheinens der EJB-2.x-Versionen noch gar nicht. Andere wurden bei der Definition der EJB-2.x-Spezifikationen schlichtweg nicht berücksichtigt.

Neben dem einen oder anderen Ansatz, der damals noch nicht State of the Art (zumindest für die EJB-Technologie) gewesen ist, haben sich über die Zeit neue Ansätze und Erkenntnisse entwickelt.

Ausgehend von der praktischen Erfahrung mit der EJB-2.x-Technologie, wurden im Laufe der Zeit eine Reihe von Anforderungen,

Wünschen und Ansprüchen an die EJB-Technologie gerichtet: Einfachere Nutzung, Reduzierung unnötiger Aufwände bei der Programmierung sowie eine mächtigere und zugleich einfacher bedienbare Persistenzabbildung waren die »Top 3« dieser Wunschliste. In der Tat: Die Mikroarchitektur von EJB-2.x-Komponenten war zu komplex. Es existierten zu viele Fragmente, die in Korrelation gebracht werden mussten. Deployment-Deskriptoren waren sehr komplex. Entity Beans (persistente EJB-Komponenten) besaßen eine komplizierte Persistenzabbildung, der zudem noch Eigenschaften wie Vererbung oder Multi-Table-Mapping¹ fehlten. Es gab also ein enormes Verbesserungspotenzial.

2.3 Einleitung

Objektorientierung

Die Objektorientierung an sich ist etwas Feines. Man bildet die Umwelt (zumindest einen fachlichen Ausschnitt) auf objektorientierte Modelle (Klassen, Schnittstellen, Beziehungen) ab. Um Beziehungen und Eigenschaften zu modellieren, nutzt man Vererbung, Polymorphie, Delegation und weitere schöne Konzepte, die einem die Objektorientierung zur Verfügung stellt. Die Programmiersprache Java bietet die technischen Möglichkeiten, um solche objektorientierten Modelle relativ leicht umzusetzen, ohne dabei so ein tief gehendes technisches Verständnis wie bei der Programmierung in C++ zu benötigen. Das ist auch einer der Gründe dafür, dass Java eine solch große Verbreitung erfahren hat.

Irgendwann stellt man fest, dass die Klassen, die man entworfen und implementiert hat, aus den verschiedensten Gründen Aspekte wie Sicherheit, Transaktionalität, Nebenläufigkeit oder dynamische Lastverteilung aufweisen sollen. Hier treten Komponentenmodelle wie Enterprise JavaBeans auf den Plan, die eben solche Aspekte mit sich bringen und dem Entwickler einen großen Teil der Entwicklungsarbeit für diese technischen Aspekte abnehmen.

Die EJB-Idee

Die Idee bei der EJB-Technologie war und ist, im Rahmen der Java-EE-Technologie Komponenten zur Verfügung zu stellen, die dem Entwickler die Pflicht und den Aufwand zur Implementierung von infrastrukturellem Code abnehmen und die den Anforderungen an Enterprise-Applikationen (siehe folgenden Abschnitt) gerecht werden. Der Einsatz der EJB-Technologie soll zu kürzeren Entwicklungszeiten für neue Applikationen führen und gleichzeitig die Qualität der Applikationen steigern, da ein signifikanter Anteil des Programmcodes nicht mehr implementiert werden muss, sondern von der EJB-Technologie

1. Kennen Sie Multi-Table-Mapping? Wir beschreiben es in Kapitel 9.

beigesteuert wird. Diese Vision ist Realität geworden. Die EJB-Container stellen eine Menge dieser Dienste zur Verfügung, und die EJB-Komponenten, die innerhalb der EJB-Container ausgeführt werden, sind entsprechend mächtig in ihrem Leistungsumfang.

So weit, so gut. Allerdings stellten EJB-2.x-Komponenten zum einen elementare Merkmale der Objektorientierung, wie beispielsweise Vererbung und Polymorphie, aus technologischen bzw. architekturbedingten Gründen nicht oder nur eingeschränkt zur Verfügung.

EJB 2.x ist zu einfach und zu schwergewichtig.

Zum anderen wird wohl jeder, der sich bereits in der Anwendung der EJB-2.x-Technologie versucht hat, eine negative Bewertung abgegeben, wenn es um den Aufwand, die Anwendbarkeit und Komplexität sowie die Fehleranfälligkeit bei der Entwicklung von EJB-Komponenten geht.

Strukturell betrachtet waren EJB-Komponenten der Vorgängerversionen (vor EJB 3.0) weit davon entfernt, das Attribut »einfach« verliehen zu bekommen. Für eine EJB-Komponente waren in der Regel mindestens fünf Softwareartefakte notwendig. Damit einhergingen ein erheblicher Entwicklungsaufwand, eine hohe inhärente Komplexität der Komponenten, höhere Fehleranfälligkeit bei der Implementierung und eine Menge rein infrastrukturellen Codes. Genau diese Eigenschaften der EJB-Komponenten waren es, die der EJB-Technologie das Attribut »schwergewichtig« einbrachten.

Die zuvor genannten Nachteile führten in der Entwicklergemeinde zu einer latenten Unzufriedenheit gegenüber der EJB-Technologie. Die Vorteile dieser Komponenten und die Dienste, die der EJB-Container anbot (beispielsweise Concurrency Handling, Instance Pooling, Security Handling und Transaktionssteuerung) erkaufte man sich mit den oben erwähnten Nachteilen. Da die kommerzielle Softwareentwicklung sehr stark vom Faktor Zeit getrieben ist, war dies zunehmend nicht mehr akzeptabel.

Unzufriedenheit und neue Ansätze

Mit der Zeit entstanden parallel zur EJB-Technologie die sogenannten *leichtgewichtigen Ansätze*. Der Unterschied lag zum einen darin, dass hierbei einfache Java-Klassen anstelle von komplexen Komponenten verwendet wurden. Zum anderen waren die Container, in denen diese leichtgewichtigen Objekte lebten, ebenfalls leichtgewichtig in dem Sinne, dass sie nur mit den nötigsten Funktionen ausgestattet waren. Nicht alle Applikationen benötigen den Funktionsumfang der großen Applikationsserver. Die meisten Applikationen kommen mit kleinen, leichten und nicht so umfangreich ausgestatteten Containern aus.

Leichtgewichtige Ansätze

Dieser Gegensatz von schwergewichtig und leichtgewichtig übte letztlich einen zunehmenden Druck auf die EJB-Technologie (siehe

auch [Ihns05]) und auf die gesamte J2EE-Plattform aus. Dieser Druck führte zur Entwicklung der J2EE-Nachfolgeversion *Java EE 5*. Ein Bestandteil dieser Plattform ist EJB 3.0. Als Ziel für die gesamte Java-EE-Plattform und erst recht für das EJB-Komponentenmodell wurde der Wechsel von einem schwergewichtigen zu einem leichtgewichtigen Modell vorgegeben – unter der Maßgabe, nicht im Leistungsumfang abzuspecken, sondern das Leistungsvermögen dabei noch zu steigern.

EJB 3.x ist leichtgewichtig.

Seit EJB 3.0 sind EJB-Komponenten nun endlich leichtgewichtig. Auf der Ebene des Java-Quellcodes können sie sich jetzt mit den konkurrierenden leichtgewichtigen Ansätzen wie [Spring] oder [Hibernate] messen. Die Mikroarchitektur ist sehr einfach gehalten: EJB-3.x-Komponenten bestehen nur noch aus einer einfachen Java-Klasse (*Plain Old Java Object, POJO*) und einem einfachen Java-Interface (*Plain Old Java Interface, POJI*). Damit entfallen die Hauptkritikpunkte, EJB-Komponenten seien zu komplex und ihre Mikroarchitektur zu kompliziert. Die Merkmale leichtgewichtiger Ansätze lassen sich jetzt im Wesentlichen auch auf EJB-Komponenten übertragen.

EJB 3.x hat Charme.

Gleichzeitig – und das hat einen gewissen Charme – offerieren EJB-3.x-Komponenten Funktionen wie beispielsweise Transaktionsverwaltung, Concurrency Handling und Security Handling, die man bis dato nur von den schwergewichtigen Komponenten kannte. Man hat es also tatsächlich geschafft, leichtgewichtige Objekte bzw. Komponenten mit dem Funktionsumfang von schwergewichtigen Komponenten zu verbinden. Das ist der große Vorteil von EJB 3.x im Vergleich zu den sehr leichtgewichtigen Ansätzen. Dort ist der Entwickler nämlich gezwungen, die genannten Funktionen bei Bedarf selbst zu implementieren oder aus verschiedenen Produkten und Frameworks zusammensetzen. Der Programmieraufwand steigt und mit ihm die Gefahr eines instabilen (weil zusammengesetzten) Systems. Ob diese Lösung dann einem Best-of-Breed-Ansatz entspricht, ist stark abhängig vom Wissen und der Erfahrung derjenigen, die diese Aspekte selbst implementiert oder ausgewählt haben.

*Objektorientierung,
wir kommen!*

In der Reihe der EJB-Versionen stehen ab EJB 3.0 erstmals alle elementaren Merkmale der Objektorientierung zur Verfügung. Vorher hatte die EJB-Technologie Probleme mit Vererbung und Polymorphie, was den Freiheitsgrad bei der Objektmodellierung stark einschränkte. Da mit EJB 3.x die Komponenten elementar aus POJOs und POJIs bestehen, sind EJBs nun in der Lage, mit den eigentlich üblichen objektorientierten Aspekten wie Vererbung, Polymorphie und Delegation umzugehen. Dies führt dazu, dass sauber modellierte Klassenmodelle ohne technische Einschränkungen in EJB-Komponentenmodelle überführt werden können.

Enterprise-Applikationen

EJBs sind nicht dafür gedacht, um mal eben eine kleine dynamische Website zu basteln oder die klassische Adressverwaltung zu implementieren. Wer diese Technologie zu einem solchen Zweck verwendet, der darf sich nicht wundern, dass die Entwicklung solcher Applikationen viel zu lange dauert und nicht so leichtgewichtig erscheint.

EJBs sind für die Entwicklung und den Betrieb von *Enterprise-Applikationen* ausgelegt. Darunter verstehen wir Applikationen mit einer großen Anzahl an Benutzern und hohem Lastaufkommen in heterogenen Systemlandschaften, die QoS²-Aspekte wie hohe Transaktionsrate, Sicherheit, Abbildung komplexer Datenbankmodelle, Ausfallsicherheit, Lastverteilung und Skalierbarkeit aufweisen.

2.4 Hauptziele für EJB 3.x

Sie haben gesehen, dass die EJB-Version 2.1 und ihre Vorgängerversionen einige Nachteile hatten. Diese wollte man mit der Version 3.0 endgültig ausräumen. Zunächst aber sollten die Motive und die Ziele für eine neue EJB-Version konkret formuliert werden, bevor sich die Expert Group an die Spezifizierung machte. In diesem Abschnitt wollen wir Ihnen die Hauptziele von EJB 3.x ausführlich vorstellen. Anschließend können Sie losgehen und Ihre Kollegen von den Vorteilen der EJB-3.x-Technologie überzeugen. Übrigens haben sich die Hauptziele mit dem Versionswechsel von 3.0 auf 3.1 nicht verändert. Es gibt zwar ein paar erwähnenswerte Neuerungen in EJB 3.1, die wir in diesem Buch selbstverständlich vorstellen. Die grundsätzliche Architektur und Ausrichtung bleibt jedoch unverändert.

2.4.1 Motive

Einige der hauptsächlichen Motive für die Entwicklung von EJB 3.0 und Java EE 5 haben wir weiter oben bereits genannt. Es ging darum, die Entwicklung von Java-EE- bzw. EJB-basierten Applikationen deutlich zu vereinfachen, zu beschleunigen und die Komplexität der Technologie zu reduzieren.

Ein weiteres nicht zu vernachlässigendes Motiv ist die zunehmende Präsenz der .NET-Entwicklungsplattform von Microsoft, die zugegebenermaßen eine erheblich schnellere und einfachere Entwicklung von Applikationen aller Größenordnungen erlaubt. An dieser Stelle werden wir keinen dogmatischen Streit entfachen, welche Technologie-

Microsoft .NET

2. QoS = Quality of Service (Dienstgüte)

plattform die bessere aus Sicht der Softwarearchitektur und des Entwicklungsprozesses ist. Schauen wir uns lieber die Ziele an, die mit EJB 3.x verfolgt und umgesetzt worden sind.

2.4.2 »Einfach machen!«

Das Hauptziel für EJB 3.x lässt sich ganz einfach formulieren:

- Vereinfachung,
- Vereinfachung
- und nochmals Vereinfachung.

In der Vorgabe für die Spezifikation des JSR220 (EJB 3.0) sind die Ziele »Ease of Use« und »Ease of Development« formuliert. Man möchte also die Verwendung der EJB-Technologie vereinfachen, aber ebenso auch die Entwicklung von EJB-Applikationen. Der Übergang von den komplexen Komponenten hin zu einfachen POJOs und POJIs stand dabei im Fokus der neuen EJB-Mikroarchitektur.

Abb. 2-1

*POJO im Fokus der
Entwicklung der neuen
EJB-Mikroarchitektur*

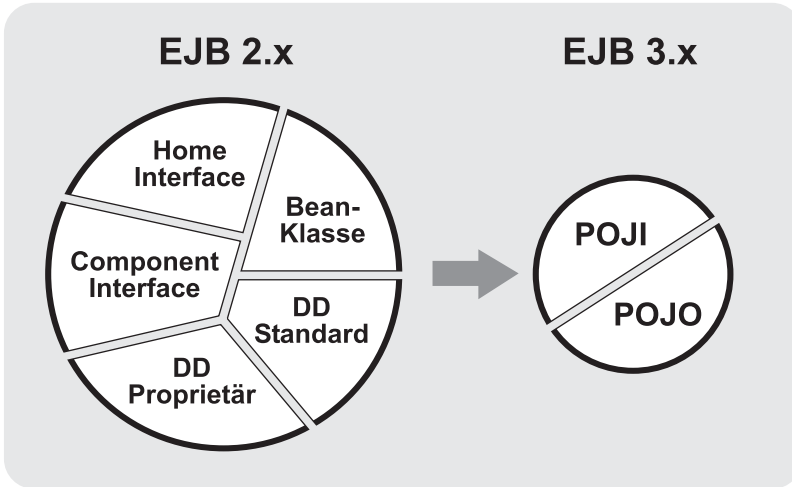


2.4.3 Vereinfachung der Mikroarchitektur von EJB-Komponenten

*Schlecht:
komplexer Aufbau*

Für gewöhnlich waren bei den EJB-2.x-Komponenten mindestens fünf Softwareartefakte notwendig, damit diese in einem EJB-Container ausgeführt werden konnten. Die negativen Folgen dieses Programmiermodells sind

- ein erheblicher Entwicklungsaufwand für jede EJB,
- eine hohe inhärente Komplexität der Komponenten,
- eine höhere Fehleranfälligkeit bei der Implementierung aufgrund fehlender Automatisierung,
- eine Menge rein infrastrukturellen Codes, den eigentlich niemand sehen, geschweige denn implementieren möchte,
- Softwareartefakte, deren Existenz nicht notwendig ist.

**Abb. 2-2**

Vereinfachung der Mikroarchitektur von EJBs (DD = Deployment-Deskription)

Dies waren genug gute Gründe, um über die bestehende Mikroarchitektur der EJB-2.x-Versionen von Grund auf neu nachzudenken. Die Mikroarchitektur der EJB-Komponenten wurde deshalb mit EJB 3.x massiv vereinfacht. Die von den EJB-Vorgängerversionen bekannten Home Interfaces gibt es nun nicht mehr. Aus den Component Interfaces werden Business Interfaces ohne Abhängigkeit zum EJB-Framework (in Abb. 2-2 als POJI bezeichnet) und EJBs lassen sich auch ohne Deployment-Deskriptoren (in Abb. 2-2 als DD bezeichnet) erstellen. Die Callback-Interfaces der verschiedenen EJB-Typen sind ebenfalls entfernt worden. Des Weiteren wurde für die persistenten Entitäten die Verpflichtung zur Implementierung eines Business Interface aufgehoben.

Reduzierung der Anzahl der Fragmente

Außerdem wurden alle Methoden aus den verschiedenen APIs und den Vorgaben der Spezifikation auf den Prüfstand gestellt. Im Falle der Stateless Session Beans beispielsweise sind die create()-Methode und die remove()-Methode überflüssig. Entwickler waren gezwungen, diese Methoden zu implementieren, da die Framework-Schnittstellen dies verlangten – obwohl ausschließlich der EJB-Container bei den Stateless Session Beans entschieden hat, wann Instanzen erzeugt bzw. gelöscht werden.

Wie bereits erwähnt, sind mit EJB 3.x aus den schwergewichtigen Komponenten leichtgewichtige, einfache Java-Klassen und Interfaces geworden. EJB 3.x setzt nun also auch auf POJO und POJI.

POJO und POJI

2.4.4 Vereinfachung des Entwicklungsprozesses

Neben der massiven Vereinfachung der Struktur bzw. Mikroarchitektur der EJB-Komponenten galt es, den Prozess für die Entwicklung von EJB-Komponenten zu vereinfachen.

Bedingt durch die Komplexität der EJB-2.x-Mikroarchitektur stellte sich die Entwicklung der EJB-Komponenten aufwendig und fehleranfällig dar. Aufgrund der Komplexität und der Vielzahl der involvierten Artefakte traten viele Fehler zum Teil erst während des Deployments oder gar erst zur Laufzeit auf. Die größte Fehlerquelle stellten die sehr umfangreichen und komplexen Deployment-Deskriptoren dar.

Diese Unzulänglichkeiten resultierten in langen Entwicklungszyklen, da der Build- und Deployment-Prozess beim Auftreten solcher Fehler jedes Mal neu angestoßen werden musste.

*EJB 3.x – einfache
Entwicklung*

Die Grundlage für eine stark vereinfachte Entwicklung von EJB-3.x-Komponenten ist die neue Mikroarchitektur auf der Basis von POJOs und POJIs. Der reduzierte Implementierungsaufwand bedeutet gleichzeitig eine Reduzierung der möglichen Fehlerquellen.

Annotationen

Die Einführung von *Annotationen* (siehe Abschnitt 2.7.3) als Ersatz für (bzw. Ergänzung zu) Deployment-Deskriptoren führt dazu, dass es im einfachsten Fall nur eine einzige Codebasis gibt: Java. Außerdem können die Annotationen vom Compiler überprüft werden, was die Fehlererkennung auf den Zeitpunkt der Übersetzung vorverlagert. Die Fehler treten dann nicht erst zum Deployment-Zeitpunkt oder zur Laufzeit auf.

*Configuration by
Exception*

Eine weitere Maßnahme, um den Entwicklungsprozess zu vereinfachen, ist die Einführung des *Configuration-by-Exception*-Ansatzes (siehe Abschnitt 2.8). Bei diesem Ansatz geht es grob gesagt darum, dass der Entwickler für den Großteil der normalen Verwendungsfälle von EJBs keine expliziten deklarativen Angaben hinterlegen muss, weil die Komponenten ein Standardverhalten besitzen. Dies spart Entwicklungszeit und trägt zur Übersichtlichkeit des Quellcodes bei – allerdings nur dann, wenn man weiß, welches Standardverhalten die Komponenten besitzen.

Die EJBs sind von Haus aus für die »Normalfälle« der Verwendung voreingestellt. Dazu hat man aus jahrelangen Erfahrungen in der Verwendung von EJB-Komponenten eine Art »Verwendungsmuster« ermittelt, das bei der Festlegung des Standardverhaltens Pate gestanden hat.

*Testgetriebene
Entwicklung*

Einen wesentlichen Beitrag zur Qualitätssicherung leistet die bessere Unterstützung der *testgetriebenen Entwicklung* (*test-driven development*). Grundlage ist wiederum die leichtgewichtige Mikroarchitektur, denn einfache Java-Klassen lassen sich auch einfach testen. Zum anderen unterstützt die Einführung der Inversion of Control bzw. der Dependency Injection die testgetriebene Entwicklung auch außerhalb des EJB-Containers. Dieses Thema wird ausführlich in Kapitel 17 behandelt.