# Enterprise JavaBeans 3.0

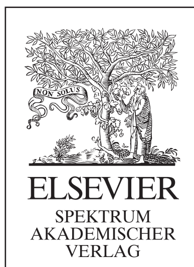
## Grundlagen - Konzepte - Praxis

## Zweite Auflage

Dies ist ein Auszug aus dem Buch.

Weitere Informationen finden Sie im Web unter

http://www.ejbbuch.de




Spektrum
AKADEMISCHER VERLAG

März 2007

© Elsevier GmbH, München
Spektrum Akademischer Verlag ist ein Imprint der Elsevier GmbH.

Das Werk einschließlich aller seiner Teile ist urheberrechtlich geschützt. Jede Verwertung außerhalb der engen Grenzen des Urheberrechtsgesetzes ist ohne Zustimmung des Verlages unzulässig und strafbar. Das gilt insbesondere für Vervielfältigungen, Übersetzungen, Mikroverfilmungen und die Einspeicherung und Verarbeitung in elektronischen Systemen.

# Vorwort

Mit Enterprise JavaBeans (EJB) existiert in der Java Enterprise Edition eine durch die Java-Community akzeptierte, serverseitige Komponententechnologie, mit der sich verteilte und mehrschichtige Geschäftsanwendungen für die Java-Plattform entwickeln lassen. EJB verspricht weitgehend automatisierte Transaktions-, Persistenz-, Verteilungs- und Sicherheitsfunktionen. Dies hilft dem Entwickler, seine Energie auf die Implementierung der Geschäftslogik zu konzentrieren.

**Entstehungsgeschichte**

In ihrer mittlerweile neunjährigen Geschichte hat die EJB-Technologie diverse umfangreiche Änderungen erfahren. Die erste EJB-Spezifikation wurde im April 1998 vorgestellt, die vor allem das Thema der transaktionalen Verarbeitung (OLTP) durch Stateless Session Beans im Fokus hatte. Bereits im Dezember 1999 folgte die in vielen Punkten überarbeitete Version 1.1. Im Juli 2001 wurde die Version 2.0 veröffentlicht. Sie bot wesentliche Erweiterungen und Verbesserungen, vor allem im Bereich Persistenz durch ein neues Entity-Beans-Modell und Messaging. Im Sommer 2002 kam dann noch Version 2.1 mit Unterstützung für Web-Services und zeitgesteuerte Ereignisse.

**EJB 3.0 vereinfacht den Entwicklungsprozess**

Nahezu vier Jahre hat es gedauert, bis die nun aktuelle EJB-Spezifikation 3.0 im Rahmen von Java Enterprise Edition 5.0 erscheinen konnte. Ein wesentliches Hauptziel von EJB 3.0 und auch Java Enterprise Edition 5.0 im Allgemeinen war es, den Entwicklungsprozess zu vereinfachen. So wurden etwa bewährte Konzepte wie leichtgewichtige persistente Java-Objekte, „Configuration by Exception" und die von Spring bekannte Dependency Injection eingeführt. Herausgekommen ist eine leichtgewichtige Komponenten- und O/R-Mapping-Technologie, die unserer Meinung nach ein optimales und stabiles Fundament für moderne Geschäftsanwendungen bildet.

Das Buch behandelt die EJB-Spezifikation 3.0 mit ihrem POJO-basierten Komponentenmodell („Simplified API") und dem neuen Entity-Konzept des Java Persistence API (JPA), das die schwergewichtigen Entity Beans der Vorgängerversionen ablöst.

### *Praxisbezug*

**Hintergrundinformationen**

**Spring und Seam BPM und SOA**

Nahezu alle Bücher auf dem Markt konzentrieren sich auf die sachliche Darstellung des EJB-Themas. Unser Buch greift einen weiteren Aspekt auf, der nach unserer Meinung nicht fehlen darf: die Einordnung und objektive Bewertung der EJB-Technologie, Hintergrundinformationen, sowie Bezüge zu verwandten Konzepten. Hier stehen die Anwendungsframeworks Spring und Seam, sowie die Themen Service-orientierte Architektur und Business Process Management im Vordergrund, die wir in den abschließenden Kapiteln detailliert und praxisnah vorstellen. Weiterhin

vermitteln wir unsere umfangreiche Praxiserfahrung in Form von Kommentaren (Vor- und Nachteile, Auswirkungen etc.) und Nutzungsempfehlungen, die sich durch das gesamte Buch ziehen.

### *Webseite zum Buch und Beispiele*

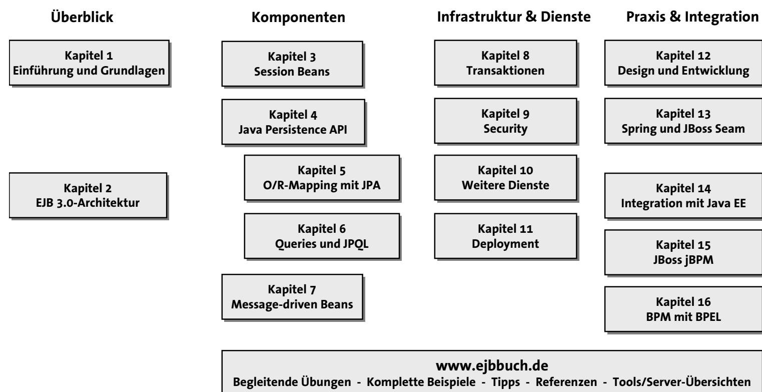
**www.ejbbuch.de**

Aktuelle Informationen zum Buch und weitere Materialien finden Sie auf der Webseite `http://www.ejbbuch.de/`.

### *Danksagungen*

An dieser Stellen wollen wir uns bei den vielen Mitarbeitern und Personen bedanken, die zum Erscheinen dieses Buches beigetragen haben. Zuerst den Ko-Autoren, deren Erfahrungen zur EJB-Technologie in die Kapitel eingeflossen sind. Auch gilt unser Dank den folgenden Personen für ihre kompetente fachliche Unterstützung und ihre entsprechend wertvollen Kommentare: Hendrik Beck, Florian Brandl, Florian Kronenberg, Torsten Winterberg, Wolfgang Dostal und Armin Wallrab. Auch möchten wir unseren Unternehmen (der mgm technology partners GmbH, der camunda GmbH sowie der Opitz Consulting GmbH) für die Ermöglichung dieses Projektes danken. Vom Spektrum-Verlag möchten wir unserer Lektorin Frau Bianca Alton für die Planung und unserem Lektor Herrn Martin Radke für die präzise Durchsicht unserer Arbeit und die gute Zusammenarbeit danken.

Die Autoren Martin Backschat und Bernd Rücker, sowie die Ko-Autoren Stefan Scheidt und Oliver-Arne Hammerstein, März 2007.



# Inhaltsverzeichnis

**1 Einführung und Grundlagen** ... 1
1.1 Software-Entwicklung im Unternehmensumfeld ... 2
1.2 Architekturen für Enterprise-Anwendungen ... 7
1.3 Middleware-Technologien ... 13
1.4 Enterprise JavaBeans ... 23

**2 Konzepte und Architektur von EJB 3.0** ... 27
2.1 Überblick ... 28
2.2 Elemente der EJB-Architektur ... 28
2.3 Enterprise Beans ... 36
2.4 Client-Sicht auf Session Beans ... 38
2.5 Bestandteile von Enterprise Beans ... 42
2.6 Die Laufzeitumgebung des EJB-Containers ... 48

**3 Session Beans** ... 65
3.1 Einführung ... 67
3.2 Client- und Container-Sicht auf Session Beans ... 71
3.3 Stateful Session Beans ... 77
3.4 Stateless Session Beans ... 85
3.5 Entwicklung von Session Beans ... 86
3.6 Ein Warenkorb als Stateful Session Bean ... 95
3.7 Stateless Session Beans als Web-Service ... 98
3.8 Web-Service-Clients ... 101

**4 Java-Persistenz mit Entities** ... 109
4.1 Die Bedeutung von Persistenz ... 110
4.2 Entities ... 114
4.3 Entity-Manager ... 123
4.4 Lebenszyklus-Ereignisse und Callbacks ... 141

**5 Objektrelationales Mapping mit JPA** ... 147
5.1 Mapping des persistenten Zustands ... 148
5.2 Vererbungshierarchien ... 171
5.3 Beziehungen ... 182
5.4 Fetching-Strategien ... 215

**6 Das Query-API und JPQL** ... 219
6.1 Überblick ... 220
6.2 Das Query-API ... 220
6.3 Select-Queries mit JPQL ... 227
6.4 Massendaten-Operationen mit JQPL ... 241
6.5 Native SQL-Queries ... 243

**7 Message-Driven Beans** ... 245
7.1 Eigenschaften von Messaging-Systemen ... 246
7.2 Java Message Service (JMS) ... 247

7.3 Einsatzszenarien von Message-Driven Beans ... 253
7.4 Der Lebenszyklus von Message-Driven Beans ... 258
7.5 Aufbau von Message-Driven Beans ... 260
7.6 Das News-Ticker-Beispiel ... 267

**8 Transaktionen** ... 273
8.1 Konzepte der Transaktionsverarbeitung ... 274
8.2 Transaktionssteuerung in EJB ... 279
8.3 Isolation von Transaktionen ... 292
8.4 Verteilte Transaktionen ... 295
8.5 Transaktionen, Bean-Typen und Rollen ... 298
8.6 Transaktionsbehandlung bei Ausnahmen ... 304
8.7 Isolation und Locking im Java Persistence API ... 309

**9 Sicherheit** ... 315
9.1 Sicherheitsmechanismen von Java EE ... 316
9.2 EJB-Sicherheitsmanagement ... 325

**10 Weitere Dienste des EJB-Containers** ... 337
10.1 Der Enterprise Naming Context (ENC) ... 338
10.2 Interception ... 346
10.3 Timer-Dienst ... 362

**11 Packaging und Deployment** ... 375
11.1 Packaging und Assembly ... 376
11.2 Deployment im Java-EE-Applikationsserver ... 382
11.3 Der EJB-Deployment-Deskriptor (DD) ... 385

**12 Design und Entwicklung** ... 397
12.1 Performance und Skalierbarkeit ... 398
12.2 Design-Tipps ... 404
12.3 Testen von Enterprise Beans und JPA-Entities ... 411

**13 Frameworks: Spring und Seam** ... 415
13.1 Leichtgewichtige Java-EE-Entwicklung ... 416
13.2 Einführung in das Spring Framework ... 416
13.3 Vergleich von Spring und EJB 3.0 ... 420
13.4 Integration von Spring und EJB 3.0 ... 441
13.5 Das Seam-Framework ... 454

**14 Integration im Umfeld von Java EE** ... 463
14.1 Überblick ... 464
14.2 Geschäftsprozesse ... 466
14.3 Lang laufende Transaktionen ... 477
14.4 Typische Integrationsarchitekturen ... 478
14.5 SOA und Java EE ... 482

**15 JBoss jBPM** ... 491
15.1 Grundlagen ... 492
15.2 jBPM und Java Enterprise ... 501

15.3 Beispielanwendung .................................... 506
15.4 Integration in JBoss Seam .................................... 517

**16 BPM mit BPEL** .................................... 521
16.1 Beispielanwendung .................................... 522
16.2 Deployment und Test .................................... 530
16.3 BPEL = Web-Services? .................................... 531
16.4 Fazit BPEL .................................... 533

**A Referenz zu EJB 3.0** .................................... 537
A.1 EJB-Ausnahmen .................................... 538
A.2 JPA-Ausnahmen .................................... 539
A.3 Annotationen .................................... 540
A.4 Der XML-Deployment-Deskriptor .................................... 547
A.5 Zugriff auf EJB-2.1-Komponenten .................................... 548
A.6 Konfigurationsdateien für das JPA .................................... 550

**B Java Enterprise Edition 5** .................................... 551
B.1 Die Java-EE-Architektur .................................... 552
B.2 Java-EE-Programmierschnittstellen .................................... 558
B.3 RMI-IIOP .................................... 570

**C Produkte** .................................... 577
C.1 Java-EE-5-Applikationsserver .................................... 578
C.2 Entwicklungsumgebungen für EJB 3.0 .................................... 585
C.3 Business Process Engines .................................... 587

**Literaturverzeichnis** .................................... 591

**Index** .................................... 595

# 2 | Konzepte und Architektur von EJB 3.0

***Martin Backschat und Bernd Rücker***

Das erste Kapitel hat EJB als serverseitige Komponententechnologie eingeführt und ihre Wurzeln, Anforderungen und Ziele dargestellt. Dieses Kapitel untersucht, durch welche Architekturmerkmale und Konzepte diese Anforderungen und Ziele umgesetzt werden. Es identifiziert dazu zunächst die wesentlichen Elemente der EJB-Architektur und erläutert ihr Zusammenwirken. Weiterhin werden die Eigenschaften und Einsatzgebiete der Komponententypen Session, Entity und Message-Driven Beans vorgestellt. Von besonderer Bedeutung in der EJB-Architektur ist der Container, der diesen Komponenten eine umfangreiche Ablaufumgebung zur Verfügung stellt.

## Übersicht

**2.1 Überblick** ... 28

**2.2 Elemente der EJB-Architektur** ... 28
2.2.1 Szenarien und Rollen ... 29
2.2.2 Der Java-EE-Server ... 31
2.2.3 Der EJB-Container ... 32

**2.3 Enterprise Beans** ... 36

**2.4 Client-Sicht auf Session Beans** ... 38
2.4.1 Merkmale der Client-Sicht ... 38
2.4.2 Lokale und Remote-Clients ... 39

**2.5 Bestandteile von Enterprise Beans** ... 42
2.5.1 Das Business-Interface ... 43
2.5.2 Die Bean-Klasse ... 44
2.5.3 Der Deployment-Deskriptor ... 45

**2.6 Die Laufzeitumgebung des EJB-Containers** ... 48
2.6.1 Ressourcen- und Instanzen-Verwaltung ... 50
2.6.2 Interceptoren ... 50
2.6.3 Laufzeit-Informationen für die Bean-Instanzen ... 51
2.6.4 Verträge des Containers ... 60
2.6.5 Programmier-Schnittstellen und -Restriktionen ... 62

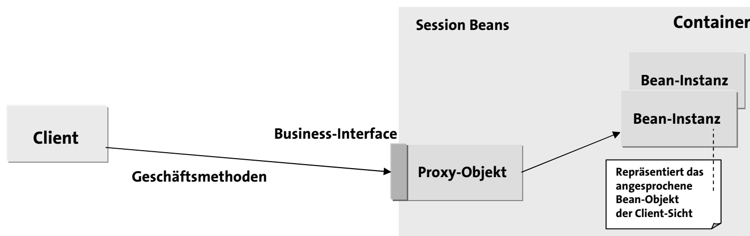



***Abbildung 2.4*** *Die Client-Sicht auf Session Beans.*

realisieren die Client-Sicht auf Session Beans. Die Abbildung 2.4 zeigt in vereinfachter Form den Zugriff eines Clients, der Methoden des Business-Interfaces aufruft, die vom Proxy-Objekte abgefangen und weitergeleitet werden. Dabei wird deutlich, dass der Client niemals direkt auf eine Bean-Instanz im Container zugreift.

**Interception durch generierte Objekte**

Der Grund für dieses Abfangen (engl. *Interception*) und Weiterleiten ist, dass es dem Container möglich wird, die Aufrufe zu kontrollieren sowie Vor- und Nachbereitungen durchzuführen. Durch die Interception kann er den Enterprise Beans die zuvor angesprochene Laufzeitumgebung bieten (Sicherheit, Transaktionen, Persistenz, etc.). Zum Beispiel kann er zunächst testen, ob der Client überhaupt das Zugriffsrecht auf die Methode hat. Weiterhin kann er die Zuordnung der „logischen" Bean-Objekte auf die Bean-Instanzen im Arbeitsspeicher dynamisch regeln. Dadurch ist es möglich, Bean-Instanzen erst bei wirklichem Bedarf zu erzeugen (*Just-in-Time-Activation*) oder vorsorglich bereitzustellen und dynamisch zuzuordnen (*Pooling*). Beide Ansätze führen zu verbesserter Performance und Ressourcen-Nutzung. Wir werden das Interception-Konzept in Kapitel 2.6.2 und vor allem in 10.2 auf Seite 346 vertiefen.

## 2.4.2 Lokale und Remote-Clients

Bezüglich der Clients von Session Beans unterscheidet man zwischen den folgenden beiden Arten:

- *Remote-Clients* liegen in einem anderen Adressraum als der Container (und damit der Bean-Instanzen). Sie laufen z. B. in einer anderen JVM auf dem Server oder liegen auf einem anderen Rechner.
- *Lokale Clients* liegen im Adressraum des Containers und werden deshalb von derselben JVM ausgeführt. Dabei kann es sich etwa um Servlets oder weitere Enterprise Beans im Container handeln.
- *Web-Service-Clients* benutzen das über WSDL exportierte Endpoint-Interface zum Aufruf. Nur Stateless Session Beans können Web-Service-Clients unterstützen.

**Remove vs. lokale Client-Sicht**

EJB stellt diesen beiden Arten von Clients unterschiedliche Sichtweisen auf das Bean zur Verfügung: die *Remote* Client-Sicht und die *lokale* Cli-

ent-Sicht. Es ist jedoch zu beachten, dass selbst Aufrufer aus demselben Adressraum wie das Bean durchaus auch die Remote-Client-Sicht benutzen können. Dies ist etwa nötig, wenn eine EJB-Komponente keine lokalen Interfaces anbietet.

**Lokaler Aufrufer kann auch Remote-Client-Sicht benutzen**

**Remote-Methodenaufrufe mit RMI und RMI-IIOP**

**Remote-Objekte**

In der Programmiersprache Java wurden Konzepte zur verteilten Objekt-Kommunikation von Anfang an berücksichtigt. Java Remote Method Invocation (RMI) erlaubt auf einfache Weise, Objekte auf einem entfernten Rechner aufzurufen (sogenannte *Remote-Objekte*).

Als Parameter und Ergebnis beim Methodenaufruf von Remote-Objekten dürfen lediglich RMI-konforme Typen eingesetzt werden, um Probleme beim Marshalling und Demarshalling zum Transport über das Netzwerk zu vermeiden. Dabei sind alle primitiven Java-Datentypen und serialisierbaren Objekte erlaubt, sowie Objekte, deren Klasse sich von `java.rmi.Remote` ableitet.

**Das Interface `Remote`**

Der Zugriff von Remote-Clients auf EJB-Komponenten basiert auf der RMI-Technologie für Remote-Objekte. Ein Remote-Objekt im Sinne von RMI implementiert ein Remote-Interface, das Methoden deklariert, die von einem Remote-Client aufrufbar sind. Hier ein einfaches Beispiel:

```
public interface DemoRemote extends java.rmi.Remote {
  public int remoteMethod(String s)
                           throws java.rmi.RemoteException;
}
```

**`RemoteException`**

Wie zu sehen ist, erweitern Remote-Interfaces immer das Interface `java.rmi.Remote`. Es deklariert selbst keine eigenen Methoden, sondern dient lediglich als Marker-Interface (vergleichbar mit `Serializable`). Jede deklarierte Methode eines Remote-Interfaces muss in ihrer `throws`-Klausel die Ausnahme `java.rmi.RemoteException` oder eine ihrer Super-Klassen angeben, etwa `java.io.IOException`. `RemoteException` wird vom RMI-Subsystem geworfen, wenn bei der Netzwerkkommunikation oder auf dem Server ein Fehler auftritt.

**Neu in EJB 3.0**

Im Gegensatz zu EJB 2.1 sind in EJB 3.0 selbst die Remote-Business-Interfaces *keine* `Remote`-Interfaces, d. h. sie leiten sich *nicht* von `Remote` ab. Ebenso müssen die deklarierten Methoden im Remote-Business-Interface keine `RemoteException` in der Throws-Klausel deklarieren. Der Grund ist, dass der EJB-Container seine eigenen `Remote`-Klassen für die Stub-Objekte auf der Remote-Client-Seite bereitstellt, die das Remote-Business-Interface implementieren. Aus diesem Grund muss man sich lediglich an die RMI-IIOP-

▼

Beschränkungen bezüglich der Parameter- und Ergebnistypen halten. Und diese Beschränkungen beziehen sich in erster Linie auf die Remote-Clients.

**Ablauf der RMI-Kommunikation**

Die Abbildung 2.5 illustriert den Ablauf bei der RMI-Kommunikation. (Die Konzepte der verteilten Objekt-Kommunikation wurden bereits im Kapitel 1.3.1 vorgestellt.) Der Remote-Client richtet seinen Methodenaufruf an das von RMI generierte Stub-Objekt, das den Aufruf an das Skeleton-Objekt auf der Serverseite weiterleitet. Das Skeleton-Objekt ist mit dem Remote Objekt verbunden und delegiert den Aufruf dahin weiter. Bei RMI-Remote-Methodenaufrufen werden Parameter und Ergebnisse als Kopien übergeben (*Call-by-Value-Semantik*). Eine Ausnahme bilden nur Remote Objekte: bei ihnen wird eine Kopie ihres Stub-Objektes weitergeleitet.

**Call-by-Value-Aufrufsemantik**

**RMI-IIOP**

Mit „RMI over IIOP“ (RMI-IIOP) existiert eine Variante, dem das IIOP von CORBA als Transportprotokoll zugrunde liegt. RMI-IIOP ist kompatibel zu CORBA und bietet somit eine theoretische Möglichkeit, um mit CORBA-Objekten zusammen zu arbeiten.

**RMI-IIOP-Stubs bei EJB**

Remote-Clients, die Methoden eines Enterprise Beans aufrufen, arbeiten mit RMI-IIOP-Stubs als *Proxy-Objekte*, die das Business-Interface clientseitig implementieren und den Aufruf an den EJB-Container weiterleiten.

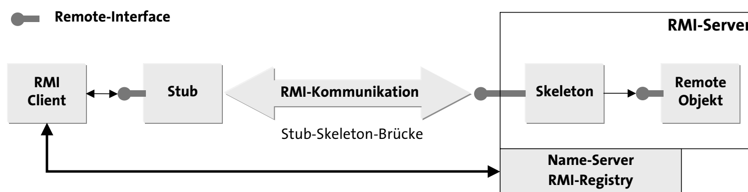



***Abbildung 2.5*** *RMI Stubs und Skeletons.*

### *Remote-Client-Sicht*

Die Remote-Client-Sicht stellt eine netzwerktransparente Schnittstelle dar, über die ein Client-Programm die exportierten Business-Interface-Methoden des Beans aufrufen kann. Für die Remote-Client-Sicht wird das Proxy-Objekt als Remote-Objekt im Sinne von RMI-IIOP realisiert (vgl. Textkasten auf der vorherigen Seite). Das Remote-Proxy-Objekt implementiert das Remote-Business-Interface.

### *Lokale Client-Sicht*

Die lokale Client-Sicht bezeichnet eine optimierte Schnittstelle, über die Clients, die in derselben JVM wie das Enterprise Bean existieren, die Bean-Methoden aufrufen. In diesem Fall sind die Proxy-Objekte keine Re-

mote Objekte. Die Methodenaufrufe geschehen deshalb *nicht* über den Umweg von RMI-IIOP und einem Netzwerk-Transportprotokoll. Insbesondere werden Parameter und Ergebnisse bei Methodenaufrufen direkt und nicht als Kopie übergeben (*Pass-by-Reference*-Semantik).

**Call-by-Reference**

Die Implementierung des *lokalen Business-Interfaces* erfolgt auf ähnliche Weise wie die ihrer Remote-Pendants. Die Objekte, die die lokalen Interfaces implementieren, sind lokale Proxy-Objekte. Darin ist dann, wie im Remote-Fall, die nötige „Glue logic" enthalten, damit der Container einen Methodenaufruf abfangen und kontrollieren kann (Interception).

## 2.5 Bestandteile von Enterprise Beans

Für den Bean-Entwickler besteht ein Enterprise Bean aus verschiedenen Interfaces und Hilfsklassen, der Bean-Klasse selbst sowie dem Deployment-Deskriptor. Viele dieser Bestandteile sind seit EJB 3.0 nun optional. Der Umfang der Bestandteile richtet sich nach dem Typ des Beans, jedoch seit EJB 3.0 *nicht* mehr danach, welche Client-Sicht unterstützt werden soll. Die folgende Übersicht zeigt alle Teile:

**Remote-Business-Interfaces (optional):** Die darin deklarierten Methoden sind die *Geschäftsmethoden* für ein Session Bean, die für die Remote-Client-Sicht zur Verfügung stehen. Zur Kennzeichnung wird die `@Remote`-Annotation benutzt.

**Lokale Business-Interfaces (optional):** Analog zu den Remote-Methoden, allerdings für die lokale Client-Sicht eines Session Beans. Die Clients laufen in derselben JVM und deshalb entfällt der Overhead der RMI-IIOP-Kommunikation. Zur Kennzeichnung wird die `@Local`-Annotation benutzt.

**Endpoint-Interface (optional):** Dieses Interface definiert die Geschäftsmethoden, die als Web-Service via JAX-WS exportiert werden. Zur Kennzeichnung wird die `@javax.jws.WebService`-Annotation benutzt.

**Message-Interface:** Message-Driven Beans implementieren dieses Interface, das Callback-Methoden enthält, damit JMS-Messaging-Systeme oder andere Messaging-Provider die Nachrichten zustellen können.

**Bean-Klasse:** Sie implementiert alle Geschäftsmethoden aus den aufgeführten Interfaces. Ein Session Bean muss mindestens ein Interface besitzen; dabei kann es sich um ein Remote-, ein lokales oder eine Endpoint-Interface handeln. Weiterhin enthält sie optional noch zusätzliche Methoden, die direkt vom Container aufgerufen und zur Verwaltung der Bean-Instanzen dienen (sogenannte Lebenszyklus-Callback-Methoden). Zur Kennzeichnung von Stateless Session Beans wird die `@Stateless`-Annotation benutzt, für Stateful Session Beans die `@Stateful`-Annotation.

Ein Message-Driven Bean implementiert in der Bean-Klasse die `onMessage`-Methoden der aufgeführten Message-Interfaces, oder das JMS-Standard-Interface `javax.jms.MessageListener`. Zur Kennzeichnung eines Message-Driven Beans wird die `@MessageDriven`-Annotation benutzt.

**Interceptor-Klassen (optional):** Seit EJB 3.0 lassen sich die Lebenszyklus-Methoden bzw. für Entities des Java Persistence API die Listener in eine eigene Klasse auslagern.

**Deployment-Deskriptor (optional):** Er enthält Metadaten der Komponente. In dieser Datei werden die Anforderungen eines Beans an seine Laufzeitumgebung beschrieben. Der Deployment-Deskriptor ist ein XML-Dokument, deren Elemente von der EJB-Spezifikation vorgegebenen sind. Seit EJB 3.0 ist der Deployment-Deskriptor optional, denn alle Angaben können nun alternativ als Annotationen direkt in der Bean-Klasse stehen.

**Client-Sichten**

Session Beans können eine *Remote-Client-Sicht* und eine *lokale Client-Sicht* bereitstellen. Es ist jedoch zu empfehlen, sich nach Abwägung des voraussichtlichen Deployment-Szenarios – z. B. wird ein Bean innerhalb derselben JVM wie die potentiellen Clients deployt –, für die Unterstützung einer bestimmten Client-Sicht zu entscheiden. Für Message-Driven Beans gilt dies nicht, denn sie bestehen nur aus ihrer Bean-Klasse und optionalen Einträgen im Deployment-Deskriptor – Message-Driven Beans besitzen keine Client-Sicht, da sie lediglich auf Nachrichten reagieren.

Die folgenden Beschreibungen der Interfaces, der Bean-Klasse und des Deployment-Deskriptors beschränken sich auf die Besonderheiten, die durch die EJB-Architektur bedingt sind. Auf die Besonderheiten der verschiedenen Bean-Typen werden wir dann im Detail in den folgenden Kapiteln eingehen.

Weiterhin beschränken wir uns auf die Remote-Client-Sicht. EJB definiert sich als verteiltes Komponentenmodell, so dass man häufiger mit dieser Sicht in Kontakt kommen wird. Im nächsten Kapitel werden die lokalen Interfaces für ein Stateless Session Bean und die beiden Client-Sichten im Detail vorgestellt.

### 2.5.1 Das Business-Interface

In diesem Interface werden alle Geschäftsmethoden deklariert, die von einem Enterprise Bean für lokale bzw. Remote-Clients angeboten werden. Seit EJB 3.0 ist es nicht mehr nötig, ein EJB-spezifisches Interface zu erweitern oder spezifische Ausnahmen anzugeben. Das Interface wird von der Bean-Klasse implementiert. Zu bemerken ist, dass eine Bean-Klasse durchaus auch mehrere Business-Interfaces implementieren kann, etwa um unterschiedliche lokale und Remote-Client-Sichten anzubieten.

Das folgende Business-Interface definiert die Methoden für einen Warenkorb als Session Bean:

```
@Remote
public interface ShoppingCart
{
  public void purchase(Product product, int quantity);
  public void emptyCart();
}
```

### 2.5.2 Die Bean-Klasse

In der Bean-Klasse von Session und Message-Driven Beans werden alle Geschäftsmethoden des Enterprise Beans aus den Business-Interfaces implementiert.

Der folgende Code zeigt die Bean-Klasse des Session Beans „ShoppingCartBean". Die Klasse implementiert das zuvor dargestellte Interface:

```
@Stateless
public class ShoppingCartBean implements ShoppingCart
{
  public void purchase(Product product, int quantity) { ... }
  public void emptyCart() { ... }
}
```

**Ausnahmen und EJB 3.0**

**System- vs. Anwendungsausnahmen**

Der EJB-Container unterscheidet bei der Ausnahmebehandlung zwischen System- und Anwendungsausnahmen. *System-Ausnahmen* werden durch server-interne Fehler ausgelöst. Aber auch die Methoden von Beans können System-Ausnahmen werfen, um transaktionale Vorgänge abzubrechen. Die Details werden in Abschnitt 8.6 auf Seite 304 ausgeführt. *Anwendungsausnahmen* werden durch Probleme oder Fehler in der Geschäftslogik ausgelöst. Eine Anwendung wird typischerweise eine eigene Exception-Hierarchie benutzen, um die anwendungsspezifischen Probleme zu klassifizieren und dem Aufrufer mitzuteilen. In der Regel muss der transaktionale Vorgang durch das Werfen einer Anwendungsausnahme nicht abgebrochen werden.

Folgende Ausnahmen werden als *System-Ausnahmen* betrachtet:

**Definition von System-Ausnahmen**

- Die gesamte `java.lang.RuntimeException`-Hierarchie (*Laufzeit-Ausnahmen)*, etwa `NullPointerException` und `IndexOutOfBoundsException`. Laufzeit-Ausnahmen müssen in Java nicht in der Throws-Klausel einer Methode dekla- ▼

# 4 | Java-Persistenz mit Entities

***Martin Backschat***

Der effiziente Umgang mit persistenten Daten ist eine wesentliche Herausforderung für jede Enterprise-Anwendung. EJB hat deshalb schon immer das Thema Persistenz berücksichtigt. Im Laufe der Zeit wurde allerdings klar, dass mit den persistenten EJB-Komponenten, den Entity Beans, in der Praxis viele Nachteile und Probleme verbunden sind. Die EJB-3.0-Spezifikation setzt daher auf die neue *Java Persistence API* (JPA), die ein auf POJOs und Annotationen basierendes Programmiermodell bietet und bewährte Konzepte von erfolgreichen Persistenz-Frameworks wie etwa Hibernate und TopLink übernimmt. JPA ist ein Persistenz-Standard, der, im Gegensatz zu den älteren Entity Beans, auf eine breite Akzeptanz in Entwicklerkreisen und bei den Produktherstellen stößt. Dieses Kapitel vermittelt eine Übersicht der wichtigsten Konzepte der Java-Persistenz. Weiterführendes zum Thema O/R-Mapping, Entity-Beziehungen und Vererbung wird im folgenden Kapitel 5 behandelt. Kapitel 6 befasst sich schließlich mit dem Query-API und der Query-Language.

## Übersicht


**4.1 Die Bedeutung von Persistenz** ... 110
4.1.1 Persistente Objekte ... 110
4.1.2 Persistenz-Frameworks ... 111
4.1.3 Das Java Persistence API im Überblick ... 112

**4.2 Entities** ... 114
4.2.1 Die Entity-Klasse ... 115
4.2.2 Konfiguration des O/R-Mappings ... 117
4.2.3 Persistenz-Archive ... 119

**4.3 Entity-Manager** ... 123
4.3.1 Der Persistenz-Kontext ... 124
4.3.2 Verwaltung und Zugriff auf Entity-Manager ... 126
4.3.3 Das `EntityManager`-Interface ... 130
4.3.4 Lebenszyklus-Verwaltung von Entity-Objekten ... 130
4.3.5 Suche nach Entity-Objekten ... 136
4.3.6 Datenbank-Synchronisation ... 138
4.3.7 Ressourcen-lokale Transaktionen ... 140

**4.4 Lebenszyklus-Ereignisse und Callbacks** ... 141
4.4.1 Callback-Methoden ... 143
4.4.2 Entity-Listeners ... 143

tionsservers einen alternativen Persistence-Provider einsetzt, etwa weil dieser performanter ist.

**Entities != Entity Beans**

Eine Konsequenz dieser Eigenschaften ist, dass Entities mit den Entity Beans aus den früheren Versionen der EJB-Spezifikation nicht kompatibel sind. Aber keine Angst, wer Entity Beans einsetzt, kann die Migration zu EJB 3.0 und dem Java Persistence API sanft vollziehen, denn ein EJB-Container muss gemäß der EJB-3.0-Spezifikation die alten Entity Beans unterstützen. Aber sie sind ein Auslaufmodell, denn es ist klar – das Java Persistence API ist der bevorzugte und viel bessere Weg. Aus diesem Grund wird auch dieses Buch nur auf das JPA eingehen. Die Entity Beans der EJB 2.x-Spezifikation werden allerdings in der ersten Auflage dieses Buches ausführlich behandelt.

**Entity Beans sind „deprecated"**

**Vorteile des JPA-Programmiermodells**

**Unabhängigkeit vom EJB-Container:** Das Java Persistence API kann sowohl inner- als auch außerhalb eines EJB-Containers verwendet werden. Dies macht das Testen des eigenen Persistenz-Codes viel einfacher, da das Deployment entfallen kann. Insgesamt erhöht dies die Produktivität während der Entwicklung. Ein weiterer Vorteil ist, dass nun eine standardisierte Persistenz-Technologie auch für Standalone-Anwendungen verfügbar ist.

**POJO-Prinzip:** Die Klassen von Entities müssen keine Interfaces implementieren und auch Home-Interfaces sind nicht mehr erforderlich, um Create- und Finder-Methoden zu deklarieren. Stattdessen können nun Entity-Objekte mit dem üblichen `new`-Operator erzeugt oder über das Query-API ermittelt werden. Weil Entities auf Sprachebene reguläre Java-Klassen sind, kann man auch *Vererbung und Polymorphie* einsetzen.

**Annotationen:** Metadaten etwa für das O/R-Mapping können nun direkt in der Java-Klasse über Annotationen spezifiziert werden. Alternativ können die Mapping-Angaben aber auch in XML-Dateien ausgelagert werden. Beide Varianten lassen sich kombinieren, wobei dann XML immer Vorrang gegenüber den Annotationen hat. Dies ist besonders nützlich, wenn man in den verschiedenen Entwicklungs-, QA- und Produktionsumgebungen unterschiedliche Konfigurationen benötigt und wechseln muss.

**Lebenszyklus-Ereignisse und Callbacks:** In der Entity-Klasse oder auch in einer separaten Listener-Klasse lassen sich Callback-Methoden zur Behandlung der Lebenszyklus-Ereignisse definieren. Dabei können auch mehrere Listener für dasselbe Ereignis registriert werden.

**Attach/Detachment:** Entity-Objekte können von ihrem Persistenz-Kontext losgelöst und in anderen Schichten der Anwendungen als reguläre Java-Objekte verwendet werden. Wenn dort Änderungen durchgeführt werden, können diese Objekte wieder mit ihrem Original im Persistenz-Kontext zusammengeführt werden, so dass die Änderungen persistent werden. Diese Fähigkeit macht die Entwicklungsmuster „Value Object" bzw. „Data Transfer Object" für diesen Zweck überflüssig und reduziert somit erheblich den Kodierungsaufwand der Anwendung (vgl. auch Textkasten auf Seite 135).

**Query-API und Verbesserungen in der Query-Sprache:** Die JPQL (eigentlich JPA-QL) unterstützt nun Operationen auf Massendaten (sogenannte Bulk-Updates und -Deletes), weiterhin Aggregatsfunktionen, Projektionen, Gruppierung, dedizierte Ergebnis-Klassen und Verbund-Abfragen (Joins).

## 4.2 Entities

Entities sind aus Entwicklersicht reguläre Java-Klassen; ein Client erzeugt und benutzt Entity-Objekte wie auch jedes andere Java-Objekt. Da sie nicht unter der Verwaltung des EJB-Containers stehen, sind Entities keine EJB-Komponenten im engeren Sinn. Und auch konzeptuell weisen JPA-Entities wesentliche Unterschiede zu Session- und Message-Driven Beans auf:

**Entities vs. Session Beans**

- Entity-Objekte besitzen mit ihrem Primärschlüssel eine *Identität*, durch die sie sich von anderen unterscheiden. Clients können sie damit eindeutig referenzieren. Die Identität kann an andere als Parameter weitergereicht werden und ermöglicht die Verarbeitung gemeinsamer Daten in verteilten Architekturen. Session Beans besitzen keine eigene Identität, aber sie können über ihr Remote-Business-Interface von entfernten Clients benutzt werden.
- Entities haben *kein Remote-Interface*. Entity-Objekte sind lokal. Aber durch den Attachment/Detachment-Mechanismus können sie – sofern sie serialisierbar sind – an entfernte Awendungsschichten weitergereicht werden.
- Entity-Objekte haben einen *persistenten Zustand*. Entsprechend ist die Lebenszeit eines Entity-Objekts völlig unabhängig von der Laufzeit der Anwendung oder des Servers. Die den Entities zugrunde liegenden Daten können auch bereits schon vor Inbetriebnahme der Anwendung in der Datenbank liegen, und sie können auch mit anderen Anwendungen außerhalb der Java-Welt geteilt werden.

### 4.2.1 Die Entity-Klasse

Für eine Entity-Klasse werden im einfachsten Fall nur zwei Metadaten benötigt: Sie muss als Entity erkenntlich sein und sie muss einen Primärschlüssel als Teil ihres persistenten Zustands besitzen. Die Metadaten für Entities können in Form von Annotationen oder einer XML-Konfigurationsdatei spezifiziert werden. Im Folgenden konzentrieren wir uns auf die erste Variante, die Alternative per XML wird jedoch in Abschnitt 4.2.2 und in den folgenden beiden Kapiteln besprochen.

**Annotationen vs. XML**

Die `@javax.persistence.Entity`-Annotation zeigt an, dass es sich um eine Entity-Klasse handelt, die Klasse also eine persistente Datenbank-Repräsentation besitzt. Die Annotation hat ein optionales Attribut `name`. Dieser Name wird verwendet, um im Entity in einer JPQL-Query zu referenzieren. Der Default ist der unqualifizierte Name der Entity-Klasse. Die Klasse muss weiterhin einen Primärschlüssel besitzen. Er wird über die Annotation `@javax.persistence.Id` festgelegt. Die Annotation kann entweder auf ein Attribut oder eine Getter-Methode angewendet werden.

`@Entity`

`@Id`

Bei JPA gilt in Bezug auf alle weiteren Metadaten, dass sie nur benötigt werden, um ein vom Default abweichendes Verhalten oder eine andere Konfiguration durchzusetzen. Wenn die Klasse keine weiteren Metadaten angibt, dann werden standardmäßig alle weiteren Attribute bzw. Getter-Methoden als Teil des persistenten Zustands aufgefasst und auf gleichnamige Spalten mit einem passenden Typ abgebildet. Einzige Ausnahme sind per Default alle transienten Attribute (mit dem Qualifier `transient`).

Die Liste der formalen Anforderungen an eine Entity-Klasse sind überraschend kurz:

**Anforderungen an Entity-Klasse**

- Sie muss eine *Top-Level-Klasse* sein, die nicht als `final` qualifiziert ist, und
- die Klasse muss einen *parameterlosen Konstruktor* mit Sichtbarkeit `public` oder `protected` besitzen. Dies ist bereits erfüllt, wenn im Code der Klasse kein Konstruktor definiert wird, denn der Java-Compiler wird dann automatisch einen Default-Konstruktor generieren.

Neben den persistenten Daten kann ein Entity beliebige weitere Methoden definieren und auch Teil einer Vererbungshierarchie sein (siehe dazu Kapitel 5.2 auf Seite 171). In der Entity-Klasse oder auch in einer seperaten Listener-Klasse lassen sich zudem Callback-Methoden zur Behandlung der Lebenszyklus-Ereignisse definieren (siehe Seite 141).

Das Listing 4.1 zeigt zum Einstieg die Klasse des Produkt-Entities, mit dem wir im Verlauf dieses Kapitels arbeiten werden. Sie hat in der relationalen Datenbank eine Repräsentation in Form einer `PRODUCT`-Tabelle.

***Listing 4.1** Beispiel für eine Entity-Klasse. Die Annotation `@Entity` weist sie als Entity-Klasse aus und `@Id` markiert den Primärschlüssel – mehr Angaben sind nicht erforderlich.*

```
package de.ejbbuch.shop.model;
import javax.persistence.*;

@Entity
public class Product {
  // Die persistenten Attribute:
  @Id
  private long productCode;
  private String description;
  private double price;

  public long getId() { return productCode; }
  public void setId(long id) { this.productCode = id; }

  public String getDescription() { return description; }
  public void setDescription(String d) { this.description = d; }

  public double getPrice() { return price; }
  public void setPrice(double p) { this.price = p; }
}
```

**Das Produkt-Entity**

Die Klasse definiert private Attribute, die den persistenten Zustand halten und die mit den Spalten der Zieltabelle korrespondieren. Ergänzend definiert die Klasse auch Methoden zum Zugriff auf den Entity-Zustand von außerhalb der Klasse.

### *Der Zugriffsstil für den persistenten Zustand*

Bei der Verwendung von Annotationen für die Mapping-Angaben stehen grundsätzlich zwei Wege zur Verfügung, um den persistenten Zustand zu definieren und zugänglich zu machen: entweder werden die Annotationen auf Instanzvariablen (*Attribute*) oder auf Getter-/Setter-Methodenpaare (*Properties*) nach JavaBeans-Manier angewendet. Der Persistence-Provider erkennt den gewünschten Zugriffsstil daran, ob die `@Id`-Annotation auf ein Attribut oder auf eine Getter-Methode angewendet wird. Für das Produkt-Entity in Listing 4.1 haben wir uns beispielsweise für den Zugriff über Attribute entschieden, deshalb steht die `@Id`-Annotation am `productCode`-Attribut.

**Zugriff über Attribute**

Wenn der Zugriff auf diese Weise angegeben wird, liest und beschreibt der Persistence-Provider zur Laufzeit für den Abgleich mit der Datenbank den Inhalt dieser Attribute direkt. Der persistente Zustand bildet sich stets aus *allen* nicht-transienten Attributen der Klasse.

Für den Entwickler hat der Zugriff über persistente Attribute zwei wichtige Konsequenzen: Die Attribute dürfen nicht `public` sein und der Zu-

griff auf diese Attribute darf nur in Methoden der Klasse geschehen. Die persistenten Attribute sind also nicht unmittelbar für den Client-Code erreichbar. Stattdessen kann ein Client den Zustand nur über Methoden abfragen oder setzen, die ihm die Entity-Klasse anbietet. Diese Beschränkung ist technisch begründet. Die Überwachung gestattet ihm zum Beispiel, Attribute erst bei Bedarf nachladen zu können – dies ist vor allem im Zusammenhang mit Beziehungen zwischen Entities relevant –, und weiterhin kann er gezielt nur den tatsächlich veränderten Teil des Entity-Zustands in die Datenbank zurückschreiben.

**Für Clients nur über Methoden erreichbar**

Der Persistence-Provider kann alternativ die Getter-/Setter-Methodenpaare des Entities für den Zugriff auf den persistenten Zustand benutzen. Dieser Zustand setzt sich in diesem Fall immer aus *allen* Getter-/Setter-Methodenpaaren (Properties) zusammen – ausgenommen sind nur Getter-Methoden mit der `@Transient`-Annotation. Die Methoden müssen dazu `public` oder `protected` sein. Hinsichtlich der Namensgebung der Getter-Methoden gibt es übrigens eine Ausnahme für `boolean`- und `Boolean`-wertige Properties: Für sie ist statt des Namens `getX` auch die Form `isX` erlaubt.

**Zugriff über Properties**

Das folgende Beispiel zeigt eine Variante des Produkt-Entities aus Listing 4.1 mit Zugriff über Properties. Die `@Id`-Annotation wird dazu auf die Getter-Methode `getProductCode`-Methode angewendet:

**Entity-Variante mit Properties**

```
@Entity
public class Product {
  private long code;
  private String desc;
  private double price;

  // Zugriff auf den persistenten Zustand durch Properties:
  @Id
  public long getProductCode() { return code; }
  public void setProductCode(long c) { this.code = c; }

  public String getDescription() { return desc; }
  public void setDescription(String d) { this.desc = d; }

  public double getPrice() { return price; }
  public void setPrice(double p) { this.price = p; }
}
```

Die Attribute werden vom Persistence-Provider bei dieser Zugriffsart nicht beachtet. Deshalb ist es auch nicht wichtig, wie sie heißen. Im obigen Listing benutzt etwa das Property „productCode“ (`getProductCode`/`setProductCode`) die Instanzvariable `code`.

### 4.2.2 Konfiguration des O/R-Mappings

Für jede Entity-Klasse gibt es eine Repräsentation im Schema der relationalen Datenbank; konkret wird ein Entity mit einer Tabelle verbunden

oder auf mehrere Tabellen verteilt. Der Zustand von Instanzen dieser Entity-Klasse ergibt sich dann aus den Zeilen dieser Tabellen.

**Standard-Mapping**

Die Java Persistence API definiert, wie diese Abbildung der Entity-Klassen auf Tabellen durchzuführen ist. Die Regeln sind in den meisten Fällen sinnvoll und müssen nicht überschrieben werden. Beispielsweise besagt eine Abbildungregel, wie die Tabellen und Spalten benannt sind: der Name der Tabelle ist der unqualifizierte Klassenname und die Spaltennamen in der Tabelle entsprechen den Namen der persistenten Attribute bzw. Properties. Eine weitere Regel bestimmt die Abbildung von Java- auf SQL-Typen.

**Metadaten nur für Ausnahmefälle**

Sollte eine Regel nicht zutreffen, so muss man durch Metadaten die abweichende Einstellung explizit formulieren. Der Tabellennamen kann etwa durch die `@javax.persistence.Table`-Annotation geändert werden.

Mit der Annotation `@javax.persistence.Column` lassen sich abweichende Angaben zur Abbildung eines Attributs bzw. Properties spezifizieren. Im Beispiel des Produkt-Entities wird diese Annotation benutzt, um für die Tabellenspalte „price“ die Gleitpunktgenauigkeit von acht Stellen zu setzen. Diese Information wird etwa benutzt, wenn die JPA-Implementierung die Tabellen in der Datenbank erzeugt.

**Schema-Generierung**

Die JPA-Implementierung generiert oder erwartet in unserem Beispiel eine Tabellendefinition, die wie folgt aussehen könnte:

**Relationale Repräsentation des Produkt-Entities**

```
CREATE TABLE Product (
   productCode LONG PRIMARY KEY NOT NULL,
   description VARCHAR(255),
   price DECIMAL(8)
);
```

Das nächste Kapitel 5 wird das Thema O/R-Mapping fortführen.

### *Unterstützte Datentypen*

Da der SQL-Standard nur bestimmte Datentypen kennt, ist es sinnvoll, dass auch die JPA-Spezifikation nur bestimmte Java-Datentypen für den persistenten Zustand abdeckt. Die unterstützten Java-Datentypen entsprechen im Wesentlichen denen von JDBC, auf das eine JPA-Implementierung normalerweise aufsetzt.

- Primitive Datentypen und Wrapper-Klassen wie etwa `Float`.
- Standard-Klassen `String`, `BigInteger`, `BigDecimal`, `Date`, `Calendar`, `Time`, `Timestamp`.
- Arrays, etwa `byte[]`, `Float[]` oder `Date[]`.

- Serialisierbare Klassen; sie sind dadurch gekennzeichnet, dass sie direkt oder indirekt das Interface `java.io.Serializable` implementieren.
- Aufzählungen (`enum`).
- Eingebette persistente Objekte.
- Referenzen auf Entities und Collections von Entities als persistente Beziehungen (siehe dazu Kapitel 5).

*Mapping-Konfiguration*

Wenn man zur Abbildung der Entities nicht Annotationen benutzen oder diese Informationen überschreiben will, ohne die Klassen zu ändern, stellt das Java Persistence API eine Alternative in Form von XML-Konfigurationsdateien bereit. Die JPA-Implementierung sucht automatisch im *META-INF*-Verzeichnis nach einer Datei *orm.xml*. Ein alternativer Dateiname oder ergänzende Dateien sind jedoch möglich. Für das Produkt-Entity mit Property-Zugriff würde das äquivalente XML-Mapping-Dokument wie folgt aussehen:

**Konfigurationsdatei *META-INF/orm.xml***

**Beispiel für *orm.xml***

```
<entity-mappings>
  <entity class="de.ejbbuch.shop.model.Product" access="PROPERTY">
    <attributes>
      <id name="productCode"/>
    </attributes>
  </entity>
</entity-mappings>
```

Die Mapping-Datei hat ein Wurzelelement `<entity-mappings>`. `<entity>` definiert die Entity-Klasse und den Zugriffsstil: `PROPERTY` (Getter-/Setter-Methoden) oder `FIELD` (Attribute). `<id>` ist ein Unterelement von `<attributes>` und definiert, welches Attribut als Primärschlüssel benutzt wird. Wie bei Mapping-Annotationen geht die JPA-Implementierung davon aus, dass alle anderen Properties der Klasse ebenfalls persistent sind, so dass man sie nicht explizit aufführen muss.

## 4.2.3 Persistenz-Archive

Entity-Klassen werden in Persistence-Units zusammengefasst und deployt. Eine *Persistence-Unit* ist mit genau einer Datenquelle assoziiert und umfasst typischerweise alle zusammengehörigen Entities, sowie ihre Mapping- und Laufzeitkonfigurationen. Die Persistence-Units der Anwendung werden gemeinsam in der Datei *persistence.xml* konfiguriert. Diese Datei ist ein Deployment-Deskriptor für den JPA Persistence-Provider; laut JPA-Spezifikation muss sie existieren (sofern man überhaupt Gebrauch von dem Java Persistence API macht). Innerhalb der Datei *persistence.xml* können eine oder mehrere Persistence-Units definiert wer-

**Persistence-Unit**

***META-INF/persistence.xml***

# 5 | Objektrelationales Mapping mit JPA

***Martin Backschat***

Das Abbilden von Entities des Java Persistence API in ein relationales Datenbank-Schema wird als objektrelationales Mapping bezeichnet. Die Herausforderung ist, die konzeptuellen Unterschiede zwischen dem Domänenobjektmodell und den relationalen Strukturen transparent und in portabler Weise zu überbrücken. Dieses Kapitel erklärt zunächst die Grundelemente des Mappings mit den spezifischen JPA-Annotationen. Es folgen die Mapping-Strategien für Vererbungshierarchien zwischen Entity-Klassen als Eckpfeiler für objektorientiertes Design. In der Praxis besitzt jedes Objektmodell auch Beziehungen zwischen den Objekten. Sie werden von JPA umfassend unterstützt und im Folgenden ausführlich behandelt.

## Übersicht

**5.1 Mapping des persistenten Zustands** ........ 148
5.1.1 Tabellen und Spalten ........ 149
5.1.2 Primärschlüssel ........ 154
5.1.3 Zusammengesetzte Primärschlüssel ........ 158
5.1.4 Mapping-Konfigurationen ........ 163
5.1.5 Sekundärtabellen ........ 167
5.1.6 Eingebettete Objekte ........ 170

**5.2 Vererbungshierarchien** ........ 171
5.2.1 Die SINGLE_TABLE-Mapping-Strategie ........ 174
5.2.2 Die TABLE_PER_CLASS-Mapping-Strategie ........ 176
5.2.3 Die JOINED-Mapping-Strategie ........ 177
5.2.4 Non-Entity-Basisklassen ........ 179

**5.3 Beziehungen** ........ 182
5.3.1 Klassifikation und Definition von Beziehungen ........ 182
5.3.2 Eins-zu-eins-Beziehungen ........ 186
5.3.3 N-zu-eins-Beziehungen ........ 193
5.3.4 Eins-zu-N-Beziehungen ........ 195
5.3.5 Bidirektionale Eins-zu-N-Beziehungen ........ 201
5.3.6 N-zu-M-Beziehungen ........ 204
5.3.7 Mapping von Collection-wertigen Beziehungsattributen ........ 208
5.3.8 Kaskadierende Operationen ........ 210

**5.4 Fetching-Strategien** ........ 215
5.4.1 Bedeutung und Einsatzgebiete der Strategien ........ 215
5.4.2 Lazy-Fetching bei losgelösten Entity-Objekten ........ 217

```
    @Transient
    public String getLastName( ) { return pk.getLastName( ); }
    @Transient
    public long getSsn( ) { return pk.getSsn( ); }
}
```

**Keine persistenten Properties**

### 5.1.5 Sekundärtabellen

Bei der Entwicklung von Geschäftsanwendungen wird man häufig mit einem bestehenden Domänenobjektmodell arbeiten und zusätzlich auf ein existierendes Datenbank-Schema aufsetzen, etwa weil die Datenbank von mehreren Anwendungen genutzt werden soll. Aber auch wenn das Schema noch nicht existiert, kann es gute Gründe geben, das Objektmodell und vor allem die Grobgranularität seiner Objekte nicht als Grundlage für das Schema-Design zu benutzen. Aus Sicht des Datenbank-Designer sind oft Kriterien wie Vermeidung von Redundanzen, die Wartbarkeit und das Sicherstellen von Datenkonsistenz entscheidend, so dass man das Schema oder Teile in normalisierte Form überführt (vgl. auch `http://de.wikipedia.org/wiki/Normalisierung_(Datenbank)`).

Das Java Persistence API bietet die Möglichkeit, ein Entity auf mehrere Tabellen abzubilden, *Multi-Table-Mapping* genannt. Dazu kann für jedes Attribut des persistenten Zustands die Zieltabelle und -spalte bestimmt werden.

Die Tabelle, die mit der `@Table`-Annotation ausgewählt wird, ist nach wie vor die Haupttabelle. Weitere Tabellen für das Entity, die *Sekundärtabellen*, werden jeweils mit der Annotation `@SecondaryTable` deklariert. Auf Ebene der persistenten Attribute bzw. Properties werden diese Tabellen dann über das `table`-Element der `@Column`-Annotation als Ziel ausgewählt. Attribute ohne Angabe einer Zieltabelle werden auf die Haupttabelle abgebildet. Der einfachste Fall sieht dann wie folgt aus:

`@SecondaryTable`

```
@Entity
@SecondaryTable(name="SECOND_TABLE")
public class Demo {
  @Id
  private int id;     // In Haupttabelle (DEMO)
  @Column(table="SECOND_TABLE")
  private int value;  // In Sekundärtabelle
}
```

Als komplexeres Anwendungsbeispiel betrachten wir nun ein Entity, das Mitarbeiter modelliert (Klasse `Employee`). Die Entity-Objekte enthalten neben dem Namen auch die Adress- und einige Statistik-Informationen des jeweiligen Mitarbeiters. Das Schema-Design hat jedoch die Adressen und Statistiken in separate Tabellen verlagert. Abbildung 5.3 auf der nächsten Seite illustriert das Prinzip und den Einsatz der `@SecondaryTable`-Annotation, um den persistenten Entity-Zustand aus den drei Tabellen zu aggregieren.

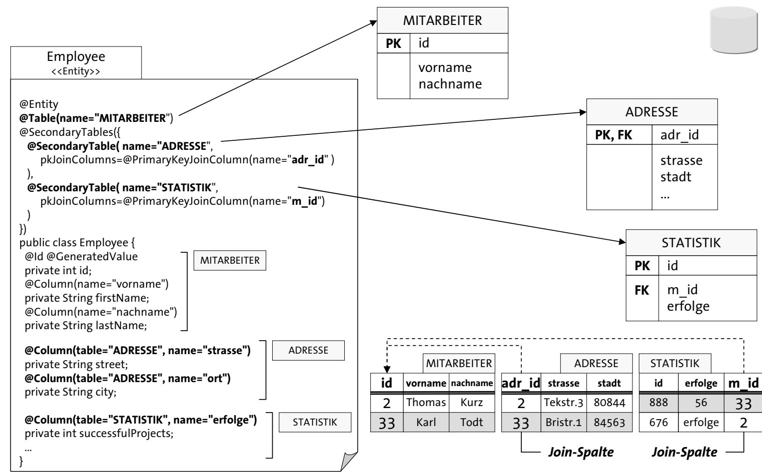



***Abbildung 5.3*** *In diesem Beispiel verteilt sich das Mitarbeiter-Entity auf eine Haupt- und zwei Sekundärtabellen.*

```
@Entity
@Table(name="MITARBEITER")
@SecondaryTables({
   @SecondaryTable(name="ADRESSE",
     pkJoinColumns=@PrimaryKeyJoinColumn(name="adr_id")),
   @SecondaryTable(name="STATISTIK",
     pkJoinColumns=@PrimaryKeyJoinColumn(name="m_id"))
})
public class Employee {
   ...
   @Column(table="ADRESSE", name="strasse")
   private String street;
   ...
   @Column(table="STATSTIK", name="erfolge")
   private int successfulProjects;
   ...
}
```

Da es mehr als eine Sekundärtabelle gibt, müssen die Deklarationen mit der `@SecondaryTables` mit Array-Notation gesammelt werden. Die `@SecondaryTable`-Annotation wirkt auf Entity-Ebene. Sie kennt alle Elemente von `@Table` (vgl. Abschnitt 5.1.1 auf Seite 150) – `name`, `catalog`, `schema` und `uniqueConstraints`) –, fügt aber das neue Element `pkJoinColumns` hinzu.

**Elemente von `@SecondaryTable`**

Unter JPA erfolgt die Verknüpfung der Haupttabelle mit den Sekundärtabellen standardmäßig über die Primärschlüssel-Spalte der Haupttabelle – oder Primärschlüssel-Spalten im Fall eines zusammengesetzten Primärschlüssels. JPA unterstellt dann, dass die Sekundärtabellen eine entsprechende Spalte mit dem gleichen Namen besitzen. Wenn diese Namenskonvention in den Sekundärtabellen nicht umgesetzt ist, hat man die Möglichkeit, die Spalten in der `@SecondaryTable`-Annotation durch das Element `pkJoinColumns` anzugeben. Das Element enthält eine oder mehrere `@PrimaryKeyJoinColumn`-Annotationen für alle Join-Spalten

**`pkJoinColumns`-Element**

der Sekundärtabelle. Zur Bedeutung von Join-Spalten siehe auch Textkasten auf dieser Seite.

`@PrimaryKey-JoinColumn`

Das `name`-Element der `@PrimaryKeyJoinColumn`-Annotation spezifiziert die jeweilge Spalte in der Sekundärtabelle. Für die Tabelle „ADRESSE“ im Beispiel ist dies `adr_id` und für die Tabelle „STATISTIK“ die Spalte `m_id`. Mit dem optionalen `referencedColumnName`-Element lässt sich der Spaltenname auf Seite der Haupttabelle angeben, der für den Join verwendet werden soll. Standardmäßig ist dies die Primärschlüssel-Spalte der Haupttabelle.

**Join-Spalten**

Als Join-Spalte bezeichnet das Java Persistence API eine Tabellenspalte, durch die man die Zeilen mit den Zeilen einer anderen Tabelle eindeutig verbinden kann. Diese Spalte besitzt dazu dieselben Werte wie eine Spalte der referenzierten Tabelle, typischerweise die Primärschlüssel-Spalte. Wenn sich der Primärschlüssel der referenzierten Tabelle aus mehreren Spalten zusammensetzt, dann werden zur eindeutigen Zuordnung entsprechend viele Join-Spalten benötigt.

JPA nutzt Join-Spalten etwa im Zusammenhang mit Sekundärtabellen, der Abbildung von Vererbungshierarchien (vgl. auf Seite 177) und bei Beziehungen (vgl. auf Seite 188).

**Join-Spalte ist Fremdschlüssel**

In der Datenbank werden Join-Spalten typischerweise explizit als Fremdschlüssel gekennzeichnet (vgl. auch `http://de.wikipedia.org/wiki/Fremdschlüssel`). Dies erlaubt dem Datenbank-System die Optimierung durch einen Index auf diesen Spalten. Zudem wird ein Constraint zur Sicherstellung der referenziellen Integrität – ein Eckpfeiler der Datenintegrität – angelegt.

Die XML-Mapping-Konfiguration steht als Alternative zu den Annotationen zur Verfügung und bietet äquivalente Elemente. Das besprochene Beispiel würde man ohne Annotationen dann wie folgt in der Mapping-Datei beschreiben:

```
<entity-mappings>
  <entity class="de.ejbbuch.shop.model.Employee" access="FIELD">
    <table name="MITARBEITER"/>
    <secondary-table name="ADRESSE">
      <primary-key-join-column name="adr_id"/>
    </secondary-table>
    <secondary-table name="STATISTIK">
      <primary-key-join-column name="m_id"/>
    </secondary-table>

    <attributes>
```

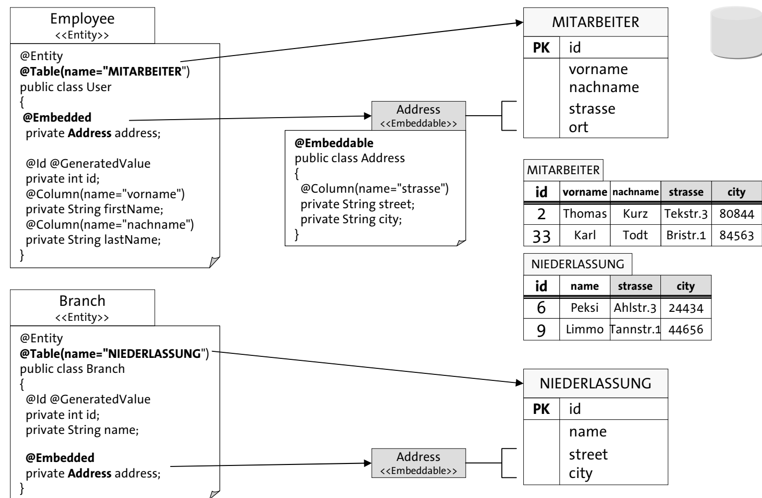



***Abbildung 5.4** Die neue Variante des Mitarbeiter-Entities gruppiert einige seiner Tabellenspalten als eigene, eingebettete Klasse (`Address`). Der Vorteil: Mehrere Entities, hier `Employee` und `Branch`, können sich die Klassendefinitionen teilen. Die Daten bleiben aber getrennt.*

```
      ...
      <basic name="street">
        <column table="ADRESSE" name="strasse"/>
      </basic>
    </attributes>
  </entity>
</entity-mappings>
```

### 5.1.6 Eingebettete Objekte

Innerhalb eines Entities können Java-Objekte eingebettet werden, die selbst nicht als Entities fungieren. Sie beziehen die Werte ihrer Attribute aus derselben Tabelle wie das Entity. Entsprechend besitzen diese Objekte keine eigene Identität, sondern dienen zur Gruppierung von persistenten Attributen des Entities, quasi als Wertobjekte. Das Konzept lehnt sich an das von eingebetteten Primärschlüssel-Klassen mit `@EmbeddedId` an, das in Abschnitt 5.1.3 auf Seite 160 vorgestellt wurde.

Zur Illustration soll uns folgende Klasse für eingebettete Objekte dienen, die dazu mittels `@Embeddable` gekennzeichnet werden muss:

```
@Embeddable
public class Address {
   @Column(name="strasse")
   private String street;
   private String city;

   public Address() {}
   // Hilfreiche Funktionen
   public Address(String s, String c) { street=s; city=c; }
   public getStreet() { return street; }
   public getCity() { return city; }
}
```

# 8 | Transaktionen

*Martin Backschat*

Das Transaktions-Management von Java EE ist ein wesentlicher Dienst für robuste serverseitige Anwendungen. Das Kapitel stellt die Konzepte und die Benutzung unter EJB und für das Java Persistence API dar.

## Übersicht

**8.1 Konzepte der Transaktionsverarbeitung** ........ 274
8.1.1 Motivation für Transaktionsverarbeitung ........ 274
8.1.2 ACID-Eigenschaften ........ 275
8.1.3 Transaktionsmodelle ........ 278

**8.2 Transaktionssteuerung in EJB** ........ 279
8.2.1 Auswahl der Transaktionssteuerung ........ 281
8.2.2 Container-gesteuerte Transaktionen ........ 282
8.2.3 Bean-gesteuerte Transaktionen ........ 287
8.2.4 Client-gesteuerte Transaktionen ........ 291

**8.3 Isolation von Transaktionen** ........ 292
8.3.1 Isolationsstufen ........ 293
8.3.2 Isolationsstufen und Enterprise JavaBeans ........ 294

**8.4 Verteilte Transaktionen** ........ 295
8.4.1 Das 2-Phasen-Commit-Protokoll ........ 296
8.4.2 Kommunikation in verteilten Transaktionen ........ 297

**8.5 Transaktionen, Bean-Typen und Rollen** ........ 298
8.5.1 Transaktionen und Session Beans ........ 298
8.5.2 Das `SessionSynchronization`-Interface ........ 299
8.5.3 Transaktionen und Message-Driven Beans ........ 301
8.5.4 Transaktionen und EJB-Rollen ........ 301

**8.6 Transaktionsbehandlung bei Ausnahmen** ........ 304
8.6.1 System-Ausnahmen ........ 304
8.6.2 Anwendungsausnahmen ........ 305
8.6.3 Ausnahmen in Callback-Methoden ........ 308

**8.7 Isolation und Locking im Java Persistence API** ........ 309
8.7.1 Optimistisches Locking von Entity-Objekten ........ 310
8.7.2 Read- und Write-Locks ........ 313

- Ausnahmen, die innerhalb einer laufenden Transaktion auftreten, bewirken, dass diese Transaktion zurückgesetzt bzw. für ein *Rollback* markiert wird.
- Die betroffene Bean-Instanz wird sofort aus dem Speicher entfernt, es können keine weiteren Methodenaufrufe stattfinden.
- Wurde die Ausnahme während der Verarbeitung einer vom Client aufgerufenen Methode ausgelöst, erhält der Client eine `EJBException`.

## 8.7 Isolation und Locking im Java Persistence API

**Tradeoff zwischen Konsistenz und Performance**

Die Wahl der Isolationsstufe für Transaktionen hat erhebliche Auswirkungen auf die Performance der Anwendung: Wenn man die höchste Stufe wählt, *Serializable*, so wird die Nebenläufigkeit der Transaktionen stark eingeschränkt und es können sogar Deadlock-Situationen entstehen. Wird dagegen eine zu niedrige Isolationsstufe, wie etwa *Read Uncommitted*, eingesetzt, so ist die Datenkonsistenz gefährdet.

Das Java Persistence API stellt zwei Mechansimen zur Verfügung, damit eine Anwendung die nötige Datenkonsistenz bereits in der niedrigen Isolationsstufe *Read Committed* sicherstellen kann:

- Optimistisches Locking unter Verwendung eines Versionsattributs.
- Explizite Lese- und Schreibsperren.

**Read Committed als Voraussetzung**

JPA geht standardmäßig davon aus, dass die Datenbank-Transaktionen mit der Isolationsstufe *Read Committed* konfiguriert sind. Diese Isolationsstufe garantiert, dass Änderungen, die innerhalb einer Transaktion stattfinden, erst beim Commit dieser Transaktion für die anderen Transaktionen sichtbar werden (vgl. auch Abschnitt 8.3 auf Seite 292).

**Optimistisches versus pessimistisches Locking**

**Kurze Geschäftsvorfälle**

Viele Anwendungen benötigen eine Flexibilisierung des Transaktionskonzeptes. Klassische transaktionale Systeme sind darauf optimiert, genau definierte, kurze Geschäftsvorfälle stark parallelisiert abzuarbeiten. So muss zum Beispiel ein Kontoführungssystem einer Bank Millionen von kurzen Einzelbuchungen pro Tag abarbeiten.

**Langlaufende Geschäftsprozesse**

Aber es gibt auch Situationen, die einen anderen Verarbeitungscharakter aufweisen. Dies sind etwa dialogorientierte Systeme oder langlaufende Geschäftsprozesse mit asynchrone Verarbeitung. In

diesen Fällen werden umfangreichere Datenmengen über einen langen Zeitraum benötigt. Schreibzugriffe finden selten statt oder die Datenbereiche der verschiedenen Instanzen überlappen nur wenig.

Diese zwei Anwendungstypen motivieren die beiden folgenden Strategien zur Gewährleistung der Datenkonsistenz:

**Optimistische Sperrstrategie:** Der Zugriff auf Daten wird während der beliebig langen Haltephasen nicht gesperrt. Erst wenn der Anwender seine Datenänderungen in die Datenbank schreibt und die Transaktion dadurch beendet, wird geprüft, ob in der Zwischenzeit andere Anwender die Daten verändert haben. In diesem Fall muss der Anwender den (statistisch seltenen) Konflikt manuell auflösen.

Die optimistische Sperrstrategie ist eigentlich falsch benannt. Denn die Strategie kommt – bis auf eine kurzen Zeitspanne am Transaktionsende – ohne Sperren in der Datenbank aus und beliebig viele Anwender können nebenläufig das gleiche Objekt bearbeiten.

**JPA unterstützt nur optimistisches Sperren**

**Pessimistische Sperrstrategie:** Die Daten werden bereits beim Lesen gegen Änderungen durch andere in der Datenbank gesperrt. Dies vermeidet bereits von Anfang an Konflikte. Die Sperren werden nur für kurze Zeit gehalten, um die Wartezeit anderer Prozesse, die auch auf diese Daten angewiesen sind, zu beschränken. Für die pessimistische Sperrstrategie gibt es seitens Java Persistence API keine Unterstützung und entsprechend keinen portable Konfiguration.

### 8.7.1 Optimistisches Locking von Entity-Objekten

Die fundamentale Annahme der optimistischen Lockking-Strategie ist, dass es nur relativ selten vorkommt, dass zwei oder mehr Transaktionen auf demselben JPA-Entity-Objekt operieren. Konsequenterweise wird man deshalb in dieser Zeitspanne keine Sperre in der Datenbank setzen wollen, sondern erst dann, wenn die gesamten Änderungen im Persistenz-Kontext in die Datenbank übertragen werden, typischerweise automatisch am Ende der Transaktion oder explizit durch einen Flush.

`OptimisticLockException`

Wenn der Entity-Manager die Datenbank-Sperren für die geänderten Entity-Objekte erfolgreich gesetzt hat, kann er überprüfen, ob die Änderungen im Persistenz-Kontext noch aktuell sind, oder ob seit Beginn der Transaktion neue Änderungen für die Entity-Objekte in die Datenbank übernommen wurden. Falls eine Änderung durch eine fremde Transaktion stattgefunden hat, dann hat die eigene Transaktion mit Daten gearbeitet, die älter als die fremden Änderungen sind. Die eigenen Änderungen im Persistenz-Kontext sind damit eigentlich veraltet und sollten nicht in die Datenbank geschrieben werden. Stattdessen muss der Entity-

Manager die Transaktion zum *Rollback* markieren und der Anwendung die Situation durch Werfen der Ausnahme `OptimisticLockException` anzeigen.

### *Beispiel für Dateninkonsistenzen ohne optimistisches Locking*

Zur Illustration betrachten wir das folgende Beispiel, in dem ein Session Bean den Prozess für die Belastung eines Bankkontos steuert.

```
@Stateless
public class AccountingServiceBean implements AccountingService {
  @PersistenceContext(unitName="Accounting")
  EntityManager em;
  public void chargeAmount(int id, double amount) {
     Account account = em.find(Account.class, id);
     int currentBalance = account.getBalance();
     // Weitere Aktionen: Überprüfe Gültigkeit des Kontos.

     // ...
     account.setBalance(currentBalance - amount);
  }
}
```

Obwohl diese Methode zunächst harmlos erscheint, kann sie zu beträchtlichen Problem führen, sofern das Konto-Entity (`Account`) nicht für das optimistische Locking vorbereitet wird. Angenommen, zwei Sachbearbeiter, S1 und S2, erhalten den Auftrag, eine Liste von Buchungsaufträgen abzuarbeiten. Die Listen überschneiden sich und beide arbeiten zur selben Zeit an einer Buchung für das Konto 42: Bearbeiter S1 will 100 Euro und S2 200 Euro abbuchen. Zuerst ruft die Anwendung von S1 die Methode `AccountingServiceBean.chargeAmount(42, 100)` auf. Sie liest zunächst die Daten des Kontos 42 aus der Datenbank und findet dabei den Kontostand 1000 Euro vor. Danach startet die Methode die Konto-Validierung und weitere Aktionen, die einige Zeit in Anspruch nehmen.

Während dieser Zeit beginnt der Bearbeiter S2 und ruft `chargeAmount(42, 200.0)` auf. Die Ausführung liest den Stand von Konto 42 ein, der immer noch 1000 Euro beträgt. Die Methode beginnt auch hier mit der Konto-Validierung und den weiteren Aktionen, die jedoch schnell erledigt sind – schneller als für S1, der immer noch wartet – und schreibt abschließend den neuen Kontostand von 800 in die Datenbank. Die Transaktion von S2 ist nun beendet. Nach einiger Zeit sind auch für S1 die Aktionen beendet und die Methode führt die Abbuchung durch – sie schreibt den Wert 900 (=1000-100) als aktuellen Kontostand in die Datenbank. Dadurch ergibt aber leider ein falscher Kontostand, zum Vorteil des Kontobesitzers: Anstatt der erwarteten 700 Euro (=1000-200-100) besitzt er noch 900. Der Grund ist, dass die letzte Änderung des Kontostands (von S1) die vorhergehende von S2 überschrieben hat. Die Methode `chargeAmount()` hätte die Transaktion von S1 nicht erfolgreich abschließen dürfen.

# 13 | Frameworks: Spring und Seam

***Stefan Scheidt, Oliver-Arne Hammerstein und Martin Backschat***

Für eine effiziente Entwicklung auf Basis der Java-EE-Plattform ist der Einsatz geeigneter Frameworks zwingend und in der Entwicklergemeinde durchgängig akzeptiert. In diesem Kapitel besprechen wir zunächst das Spring Framework, ein Dependency-Injection- und AOP-Container, der „Plain Old Java Objects" als Programmiermodell favorisiert. Spring hat sich als Ergänzung der Java-EE-Plattform und in vielen Anwendungsfällen auch als tragfähige Alternative zu „klassischen" EJB-2.x-basierten Architekturen etabliert. Wir stellen Spring zunächst im Überblick vor. Anschließend vergleichen wir Spring und EJB 3.0 und gehen dann auf Integrationsmöglichkeiten ein, insbesondere von Spring und dem Java Persistence API. Der nächste Abschnitt stellt JBoss Seam vor, ein Framework, das auf EJB 3 und JavaServer Faces basiert und eine weitgehende Integration dieser Frameworks für die Entwicklung von Web-Applikationen bereitstellt. Ein Vergleich von Spring und Seam bildet den Abschluss dieses Kapitels.

## Übersicht


**13.1 Leichtgewichtige Java-EE-Entwicklung** ..... 416

**13.2 Einführung in das Spring Framework** ..... 416

**13.3 Vergleich von Spring und EJB 3.0** ..... 420

13.3.1 Dependency Injection ..... 421
13.3.2 Spring AOP und EJB Interceptors ..... 426
13.3.3 Transaktionsmanagement ..... 428
13.3.4 Integration von Persistenz-Frameworks ..... 432
13.3.5 Stateful Beans ..... 434
13.3.6 Asynchrone Verarbeitung ..... 434
13.3.7 Sicherheit ..... 438

**13.4 Integration von Spring und EJB 3.0** ..... 441

13.4.1 Nutzung des Java Persistence API ..... 442
13.4.2 Enterprise Beans als Fassade für Spring Beans ..... 449
13.4.3 Sonstige Integrationsmöglichkeiten ..... 451
13.4.4 Zusammenfassung ..... 452

**13.5 Das Seam-Framework** ..... 454

13.5.1 Konzepte ..... 454
13.5.2 Seam im Einsatz ..... 459
13.5.3 Spring vs. Seam ..... 461

## 13.1 Leichtgewichtige Java-EE-Entwicklung

Leichtgewichtige Anwendungs-Frameworks sind in der Enterprise-Java-Community bereits seit einigen Jahren ein wichtiger Faktor für den Projekterfolg. Angefangen mit den Pionieren Hibernate und Spring und der Einführung von Technologien wie aspektorientierte Programmierung und Metadaten-Annotationen bis hin zur aktuellen EJB-3.0- und Java-EE-5.0-Spezifikation, sind leichtgewichtige Frameworks nun Mainstream geworden. Die große Nachfrage nach leichtgewichtigen Technologien resultiert größtenteils aus dem Unmut der Entwickler über das „schwergewichtige" und invasive Komponentenmodell von EJB 2.x. Durch leichtgewichtige Frameworks können Entwickler produktiver arbeiten, da Programmiermodell einerseits und Infrastruktur der Laufzeitumgebung beziehungsweise Frameworkfunktionalitäten andererseits klar voneinander getrennt werden. Zusätzlich fördern leichtgewichtige Frameworks die Realisierung leistungsfähigerer Anwendungsarchitekturen und erleichtern sowohl das Testen als auch die Wiederverwendung von Geschäftskomponenten.

Allen leichtgewichtigen Anwendungs-Frameworks im Java-Enterprise-Umfeld ist ein Kernprinzip gemeinsam: der Gebrauch von „einfachen" Java-Objekten (POJOs) für die Implementierung von Fachklassen und Service-Objekten. Es besteht also keine Notwendigkeit, aus Infrastruktur- beziehungsweise Framework-Klassen abzuleiten oder vorgegebene Interfaces zu implementieren. Leichtgewichtige Frameworks sind in diesem Sinne nicht invasiv. Domänenobjekte oder Geschäftsprozesse, die diese Domänenobjekte benutzen, werden durch reguläre Java-Klassen implementiert. Die Zusammenarbeit der Komponenten einer Applikation, z. B. Service-Komponenten und Datenzugriffskomponenten, wird gemäß einer Beschreibung durch Metadaten festgelegt. Ein wichtiges Konzept für diese Komposition ist die Dependency Injection (vgl. Abschnitt 2.6.3 auf Seite 55).

Die verschiedenen Frameworks, die dieses Paradigma umsetzen, unterscheiden sich einerseits in der Leistungsfähigkeit der Dependency-Injection-Mechanismen. Andererseits bieten sie unterschiedliche Integrationsmöglichkeiten von selbst geschriebenen Anwendungs-Komponenten und Infrastruktur-Diensten. Für das Spring Framework und EJB 3 wird in Abschnitt 13.3 ein entsprechender Vergleich durchgeführt.

## 13.2 Einführung in das Spring Framework

Gegen Ende des Jahres 2002 erschien das Buch „Expert One-on-One J2EE Design and Development" von Rod Johnson [Joh02]. In diesem Buch geht Rod Johnson sehr detailliert auf unterschiedliche Problemstellungen ein, die beim Einsatz der Java Enterprise Edition auftreten können. Insbe-

sondere kritisiert er die Komplexität und die Einschränkungen für die Anwendung guter objektorientierter Designprinzipien, die durch einen unnötigen, durch die Anforderungen nicht gerechtfertigten Einsatz von EJB in J2EE-Projekten entstehen. Als Alternative demonstriert Rod Johnson Lösungen auf Basis eines Java EE Frameworks, das die Trennung von Schnittstelle und Implementierung einer Applikationskomponente und den Einsatz von „Plain Old Java Objects" für diese Implementierung in den Mittelpunkt rückt. Für die Komposition einer Applikation aus solchen Komponenten wird konsequent das Prinzip der Dependency Injection genutzt. Infrastrukturdienste wie z. B. Transaktionsmanagement werden durch Mechanismen der aspektorientierten Programmierung in die Applikation eingewoben. Auf diese Weise lassen sich Anwendungen auf Basis der Java Enterprise Edition entwickeln, die sowohl die Vorteile eines guten objektorientierten Designs als auch die der Java-EE-Plattform für sich zunutze machen können. Dabei weisen sie keine höhere Komplexität auf, als es die umzusetzenden Anforderungen erfordern.

**Spring als Alternative zu EJB 2.x**

Die Vorschläge von Rod Johnson entsprachen dem damaligen „Zeitgeist", der eine Vereinfachung der Java Enterprise Edition forderte. Schnell wurde der Wunsch laut, den Framework-Code aus dem Buch als Open-Source-Projekt zur Verfügung zu stellen und weiter zu entwickeln. Im Februar 2003 wurde bei SourceForge.net das Spring-Framework-Projekt gegründet, siehe `http://www.springframework.org`. Das Final Release der Version 1.0 des Spring Frameworks wurde im März 2004 unter der Apache-2-Lizenz veröffentlicht. Aktuell (Oktober 2006) liegt Version 2.0 vor.

Erklärte Projektziele des Spring-Projekts sind: [JH04, S. 143 und 144]

**Ziele von Spring**

- Der Einsatz der Java Enterprise Edition soll einfacher werden.
- Die Anwendung bewährter objektorientierter Designprinzipien wie die Trennung von Schnittstelle und Implementierung und die Entkopplung von Komponenten mit verschiedenen Verantwortlichkeiten sollen leichter werden.
- Die einzelnen Bestandteile eines Systems sollen möglichst leicht in Isolation getestet werden können.
- Praxiserprobte APIs wie das Java Transaction API oder der Java Messaging Service und weit verbreitete Frameworks wie z. B. Persistenz-Frameworks sollen einfach integrierbar sein. Der wichtige Leitspruch hierbei ist: „It's all about choice!" Der Entwickler soll jederzeit die Möglichkeit haben, eine der Problemstellung angemessene Lösung in seinem Systemdesign verwenden zu können.

### *Überblick über das Spring Framework*

**Bausteine des Spring Frameworks**

Die grundlegenden Bausteine des Spring Frameworks sind ein Dependency Injection Container, der sogenannte *Spring Application Context*, das

*Spring AOP Framework* und diverse *API-Vereinfachungen*. Der Dependency Injection Container liefert die Infrastrukturgrundlage, um eine Spring-basierte Applikation aus unterschiedlichen Komponenten, wie z. B. Datenzugriffsobjekten und Business Services (siehe Abschnitt 12.2.3), aufzubauen. Mit AOP werden dann zusätzliche Aspekte, wie Transaktionsmanagement oder Sicherheitsaspekte, eingebunden. Die API-Vereinfachungen dienen der einfachen Integration und Nutzung von Java EE APIs und Frameworks wie JMS, JDBC oder diverse O/R-Mapping-Frameworks. Die eigentlichen Applikationskomponenten werden meist duch „Plain Old Java Objects" implementiert, siehe Abbildung 13.1.

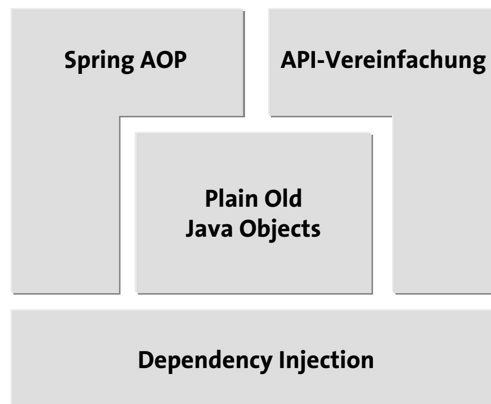



***Abbildung 13.1***
*Grundlegende Bausteine des Spring Frameworks.*

Aufbauend auf diese Basisbausteine bietet das Spring Framework weitere Funktionsblöcke an. Dabei handelt es sich z. B. um Hilfsklassen für die Entwicklung von Datenzugriffsobjekten basierend auf diversen Persistenz-Frameworks, Unterstützung bei der Bereitstellung von Remoting-Schnittstellen für Service-Implementierungen und Integrationsmöglichkeiten für verschiedene Web-Frameworks, inklusive eines eigenständigen Web-MVC-Frameworks. Einen ausführlichen Überblick über die Bestandteile von Spring findet man in der Referenzdokumentation [JH+06]. Im Umfeld des Spring Frameworks sind ferner diverse ergänzende Projekte entstanden, wie z. B. Spring Web Flow oder das Acegi Security System for Spring, welche das Spring Framework um unterschiedliche Funktionen erweitern. Siehe hierzu `http://www.springframework.org/sub-projects` und `http://www.springframework.org/associatedProjects`.

**Projekte im Spring-Umfeld**

Auf die für einen Vergleich mit Enterprise JavaBeans relevanten Eigenschaften und Funktionen des Spring Frameworks gehen wir im Abschnitt 13.3 detailliert ein. Spring bietet jedoch auch viele Funktionen, die kein direktes Pendant in der EJB-Welt habe bzw. überhaupt nicht von der EJB-Spezifikation adressiert werden. Einige Informationen dazu finden sich im nachfolgenden Kasten.

### Was leistet Spring sonst noch?

Spring versteht sich als Integrations-Framework, das die Entwicklung aller Bereiche und Schichten einer Java-EE-Applikation unterstützen möchte. Wir können hier natürlich keinen vollständigen Überblick geben. Stellvertretend seien nur einige wichtige Funktionen aufgeführt.

#### Spring Remoting-Support

Das Spring Framework bietet umfangreiche Unterstützung, um einen Business Service mit einer Remoting-Schnittstelle zu versehen. Dazu kann ein Spring Bean vergleichsweise einfach mit Proxies und sogenannten *Exportern* ausgestattet werden. Diese erlauben es, das Bean über unterschiedliche Remote-Protokolle anzusprechen. Spring unterstützt in Version 2.0 die Protokolle RMI, HTTP, Cauchos Hessian und Burlap, JAX RPC und JMS. Die Möglichkeit, ein Spring Bean mit einem Remote Session Bean als Fassade zu versehen, wird in Abschnitt 13.4.2 besprochen.

**Remote Exporter**

#### Integration mit Web-Applikationen

Das Spring Framework bietet eine einfache Integrationsmöglichkeit mit der Webschicht einer Java-EE-Applikation. Dazu bietet Spring einen Servlet Listener, der es erlaubt, die Konfiguration eines Spring Application Context einzulesen. Dadurch erhalten die Komponenten der Web-Applikation einfachen Zugriff auf die Spring Beans dieses Kontextes. Darauf aufbauend bietet das Spring Framework für diverse prominente Web-Frameworks noch weitergehenden Support. Zu den unterstützten Frameworks zählen JavaServer Faces, Struts, Tapestry und WebWork.

**Web Application Context**

#### Spring Web MVC

Neben der Unterstützung für die eben aufgezählten Web-Frameworks beinhaltet das Spring Framework auch ein eigenes, Action-basiertes Web-MVC-Framework. Vom Grundprinzip ist es vergleichbar mit dem bekannten Struts-Framework. Es zeichnet sich jedoch durch eine deutlich bessere Integration mit den übrigen Funktionalitäten des Spring Frameworks aus. Insbesondere können die Möglichkeiten der Dependency Injection besser genutzt werden. Ferner sind Komponenten der Webschicht bei Spring Web MVC einfacher zu testen. Schließlich ist die gute Unterstützung einer Vielzahl von View-Techniken wie JavaServer Pages, Velocity, FreeMarker, XSLT und JasperReports als Alleinstellungsmerkmal von Spring Web MVC zu erwähnen.

**Web-Framework**

▼

**Sonstige Funktionen**

Hilfen bei der täglichen Arbeit

Darüber hinaus bietet Spring diverse Hilfen für die tägliche Arbeit bei der Entwicklung von Java EE Applikationen. Dazu zählen Vereinfachungen beim Zugriff auf Input Streams und beim Senden von E-Mails oder das Exportieren von Spring Beans als JMX MBeans. Auch zur Zeit noch etwas „exotische" Anwendungsfälle wie das Verwenden von Skriptsprachen (Groovy, BeanShell und JRuby) finden ihren Platz im Spring Framework. Darüber hinaus stellt Spring diverse Basisklassen für die Implementierung von Integrationstests von Datenzugriffsobjekten mit JUnit bereit.

Zum Abschluss dieses Überblicks über das Spring Framework zeigt Abbildung 13.2 auf der nächsten Seite schematisch den typischen Aufbau einer Spring-basierten Applikation:

typischer Aufbau einer Spring-Applikation

- Die fachliche Domäne wird durch POJOs abgebildet.
- Datenzugriffsobjekte, getrennt nach Interface und Implementierung, stellen die notwendige Datenzugriffslogik zur Verfügung.
- Service-Objekte, ebenfalls nach Interface und Implementierung getrennt, stellen die benötigten Service-Funktionen zur Verfügung. Für Datenzugriffe nutzen sie die Datenzugriffsobjekte. Auf dieser Ebene wird auch die Transaktionsdemarkation vorgenommen.
- Lokale Clients greifen direkt auf die Service-Interfaces zu, für Remote-Clients stehen Remote-Exporter zur Verfügung.

## 13.3 Vergleich von Spring und EJB 3.0

In diesem Abschnitt wollen wir die für einen Vergleich mit Enterprise JavaBeans relevanten Funktionen von Spring zunächst genauer vorstellen, um sie dann mit den entsprechenden Funktionen von Enterprise JavaBeans zu vergleichen.

Grundsätzliches zum Vergleich

Grundsätzlich ist bei unserem Vergleich zunächst festzuhalten, dass Enterprise JavaBeans in der hier besprochenen Version 3.0 die Java Standard Edition 5 und einen vollwertigen Java-EE-Applikationsserver voraussetzen. Ferner zeichnen sie sich durch die konsequente Nutzung von Annotation-basierter Dependency Injection und „Configuration by Exception" aus. Das Spring Framework ist explizit dafür ausgelegt, es auch in Umgebungen einzusetzen, die auf Java SE 1.3 und 1.4 basieren. Abhängig von den verwendeten Funktionen ist kein vollwertiger Applikationsserver für den Betrieb einer Spring-Applikation notwendig. Viele Web-Applikationen, die etwa mit Spring, Hibernate als Persistenz-Framework und

# 14 | Integration im Umfeld von Java EE

***Bernd Rücker***

In den vorherigen Kapiteln wurden EJBs ausführlich erläutert. Dieses Kapitel geht nun auf deren Einsatz und Integration in komplexen Umgebungen und serviceorientierte Architekturen ein. Es wird gezeigt, dass Java-EE-Anwendungen ein sehr gutes Fundament dafür bilden können.

Denn Integration heterogener IT-Landschaften wird heute aus verschiedensten Gründen immer wichtiger. Es wird immer weniger mit proprietären Technologien als viel mehr mit Architekturkonzepten wie SOA gearbeitet. Die Betrachtung und direkte Umsetzung der Geschäftsprozesse wird zum Erfolgsfaktor, wobei idealerweise Prozesse die im Unternehmen betriebenen, wiederverwendbaren Services steuern. Dabei spielen Konzepte und Standards wie Business Process Engines oder der Enterprise Service Bus, eine große Rolle. Dieses Kapitel gibt daher einen Überblick über die wichtigsten Integrationsszenarien im Umfeld von Unternehmensanwendungen, sowie die beteiligten Technologien.

## Übersicht


**14.1 Überblick** ..... 464

**14.2 Geschäftsprozesse** ..... 466

14.2.1 Implementierung von Geschäftsprozessen in Java EE ..... 469

14.2.2 Bestandteile einer Business Process Engine ..... 470

14.2.3 Standards und Produkte ..... 473

**14.3 Lang laufende Transaktionen** ..... 477

**14.4 Typische Integrationsarchitekturen** ..... 478

14.4.1 Java Business Process Engine und Stateless Session Beans ..... 479

14.4.2 Prozesse mit BPEL-Server und Web-Services ..... 480

14.4.3 Enterprise Service Bus ..... 481

**14.5 SOA und Java EE** ..... 482

14.5.1 Java Business Integration (JBI) ..... 483

14.5.2 JBI am Beispiel OpenESB ..... 484

14.5.3 Service Component Architecture (SCA) ..... 487

14.5.4 Service Data Objects (SDO) ..... 489

### 14.2.1 Implementierung von Geschäftsprozessen in Java EE

Ist nun ein entsprechendes Repertoire an Geschäftsprozessen vorhanden, sollen häufig einzelne Prozesse durch eigenentwickelte Java-Software unterstützt werden. Dabei können betriebswirtschaftliche Prozesse mindestens als Anforderungsdokumente dienen.

**Defizite Java SE für BPM**

Bei genauerer Betrachtung des Problems wird schnell klar, dass Standard-Java-Konstrukte für die Implementierung eines lang laufenden Geschäftsprozess nicht ausreichen, da häufig auf externe, asynchrone Ereignisse gewartet werden muss. Nun könnte man ein Programm implementieren, dass an einer bestimmten Stelle der Ausführung auf ein externes Ereignis wartet. Bei Umsetzung mit wartenden Threads wird allerdings die Anwendung schlecht skalieren und Ausfälle nicht tolerieren. Außerdem ist die eigene Thread-Steuerung innerhalb vieler Umgebungen verboten. Eine andere Möglichkeiten der Implementierung sind persistente Zwischenzustände, wobei man nun in der Pflicht steht, entsprechend individuellen Code für den Persistenzaspekt zu schreiben.

**Defizite Java EE für BPM**

Daher liegt die Idee nahe, entsprechende Java-EE-Mechanismen des Applikationsservers zu verwenden. In der Tat propagiert Sun für diesen Anwendungsfall das Stateful Session Bean (SFSB), das seinen Zustand persistieren, so dass zu einem späteren Zeitpunkt die Ausführung des Prozesses fortgesetzt werden kann. Nun kann man lange streiten, ob SFSBs prinzipiell problematisch sind (siehe hierzu z.B. [Joh02]) oder ob diese Probleme mit EJB 3.0 behoben sind, wie es beispielsweise JBoss proklamiert. Auf jeden Fall gibt es unabhängig von dieser technischen Diskussion immer noch Anforderungen in prozessorientierten System, die von dieser Variante nicht adressiert werden:

**Durch Java EE nicht adressierte Anforderungen**

- *Versionierung:* Auf Grund lang laufender Prozesse wird eine Versionierung von Prozessdefinitionen notwendig. Denn häufig befinden sich noch laufende Prozesse in der Abarbeitung, wenn eine neue Version released werden soll. Diese Prozesse müssen oft auch noch nach dem vormals definierten Ablauf abgearbeitet werden. Und eine Migration von Prozessinstanzen auf neue Versionen ist seltenst trivial.
- *Hot-Deployment:* Ein neuer Prozess sollte nach Möglichkeit ohne Auszeit der Anwendung bereitgestellt werden. Zwar bieten die meisten Application-Server für EJBs entsprechende Hot-Redeploy-Unterstützung an, die jedoch normalerweise auch nicht ganz ohne kurze Pause auskommt. Ein reines Deployment von Prozessdefinitionen ist meist deutlich unproblematischer.
- *Prozess-Log:* Oft ist es spätestens für das Prozess-Controlling notwendig im Nachhinein Laufzeitinformationen (wie Durchlauf- oder Liegezeiten) über den Prozess zu sammeln. Dies muss in SFSBs selbst implementiert werden.

- *Grafische Prozessbeschreibung:* Idealerweise kann ein Geschäftsprozess in einer grafischen Notation dargestellt werden, die ein fachlicher Anwender versteht. Wird ein Prozess hartcodiert, z. B. in einem Stateful Session Bean, so muss die grafische Abbildung auf jeden Fall getrennt gepflegt werden und eine Single-Source Lösung ist kaum mehr möglich.

**Business Process Engines**

Um diese Probleme zu lösen, kommen nun Business Process Engines ins Spiel, welche allerdings nicht mehr Bestandteil der Java-EE-Spezifikation sind. Die Engines bieten meist neben einer Lösung für die oben genannten Aspekte mehr oder weniger weitere Features und Tools rund um die Geschäftsprozesse an. Dabei muss beachtet werden, dass es auch mit Business Process Engines und dazugehörigen Tools leider immer noch unrealistisch ist, einen Geschäftsprozess einfach „zusammenzuklicken" und dann sofort ablaufen zu lassen, auch wenn diverse Marketing-Abteilungen es immer wieder vorgeben.

### 14.2.2 Bestandteile einer Business Process Engine

Business Process Engines (häufig wird auch von BPM-Engine oder Workflow-Engine gesprochen) kann man sich als einen Container für lang laufende Geschäftsprozesse vorstellen. Dabei sind die Business Process Engines als solche oft als Java-EE-Anwendung implementiert um die technischen Dienste des EJB-Containers nutzen zu können, und integrieren sich so nahtlos in vorhandene Applikationsserver (siehe Abbildung 14.2).

Die Hauptleistungen einer Business Process Engine sind:

**Dienste der Business Process Engine**

**Interpretation von Prozessbeschreibungen:** Die meist als XML definierten Geschäftsprozesse müssen ausgeführt werden. Dazu muss die Engine die Prozessbeschreibung verstehen und zur Laufzeit Prozessinstanzen starten und verwalten können. Die Engine muss also auch die Elemente des Prozesses (wie beispielsweise Zustände, Entscheidungen und Ereignisse) kennen und interpretieren können. Dann kann sie für jede Prozess-

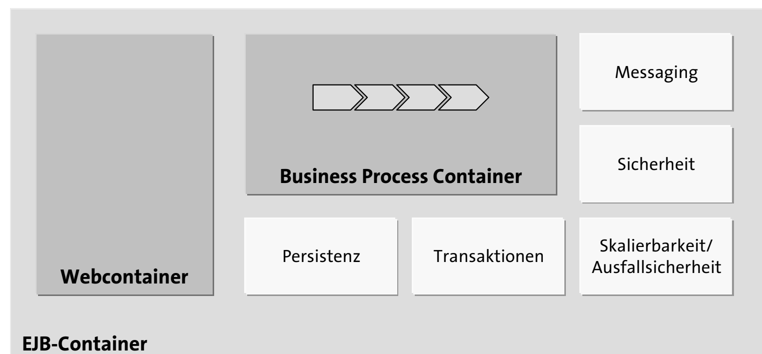



***Abbildung 14.2** Business Process Engine innerhalb des Applikationsservers.*

instanz eine entsprechende Zustandsverwaltung bereitstellen und Entscheidungen über den Ablauf selbstständig treffen.

**Persistenz:** Die Persistierung des Prozesszustandes bei Warten auf externe Ereignisse bzw das Wiederfinden der richtigen Prozessinstanz mit Auftreten externer Ereignisse ist bei lang laufenden Geschäftsprozessen eine wichtige Aufgabe. In diesem Zusammenhang ist oft auch eine Transaktionssteuerung notwendig, um die Konsistenz der Daten sicherzustellen. Hier können auf Java EE basierende Lösungen direkt auf die Dienste des Containers zurückgreifen.

**Einbindung von Services im Prozess:** Um automatisierte Prozesse umzusetzen, muss die Engine auch selbstständig andere Software aufrufen können. Je nach Engine kann dies auf unterschiedlichste Art geschehen (z. B. Aufruf eines Web-Services, Ausführung von Java-Code oder Senden einer E-Mail). Diese Leistung und die Art der Implementierung macht nun die Mächtigkeit einer Business Process Engine als zentrale Schaltstelle der prozessorientierten Software aus.

**Verwalten von Aufgaben, Timeouts und Ereignissen:** Geschäftsprozesse verlangen oft nach menschlicher Interaktion (also Aufgaben) oder nach asynchroner Kommunikation mit Fremdsystemen. In beiden Fällen muss die Abarbeitung des Prozesses unterbrochen und zu einem späteren zeitpunkt zurückkommende externe Ereignisse verarbeitet werden. Automatische Timeouts sind in diesem Zusammenhang ein wichtiger Mechanismus, um Fehler in Fremdsystemen oder auch Nichtbearbeitung durch den Menschen zu erkennen.

**Versionierung von Prozessen:** Da Geschäftsprozesse langlebig sind, ist eine wichtige Aufgabe die Versionierung der Prozessdefinitionen, da normalerweise Prozessinstanzen verschiedener Versionen laufen können. Eine Migration eines laufenden Prozesses in die neuste Version ist häufig alles andere als trivial (z.B. kann der Zustand eventuell nicht mehr vorhanden sein oder aber auch „Vorarbeiten" im Prozess fehlen). Daher sollten alte Prozesse bis zu ihrer Beendigung in der Version laufen, in der sie auch gestartet wurden. Zur Prozessdefinition gehören hierbei meist nicht nur die Prozessbeschreibungen, sondern häufig auch entsprechender Java-Code. Werden externe Services angesprochen, kann es wichtig sein, die korrekte Version des Services anzusprechen. Idealerweise können neue Prozessdefinitionen während der Laufzeit des Systems ohne jeglichen Neustart eingespielt werden.

**Prozesskontext:** Um eine prozessorientierte Anwendung zu erhalten und flexibel Fremdsoftware integrieren zu können, muss die Prozessinstanz alle für den Prozess wichtigen Daten halten. Sie muss zwar nicht die kompletten Datensätze speichern, aber zumindest entsprechende Schlüssel kennen (z.B. reicht die Kundennummer aus, wenn das entsprechende CRM-System über die Kundendaten befragt werden kann). Somit braucht

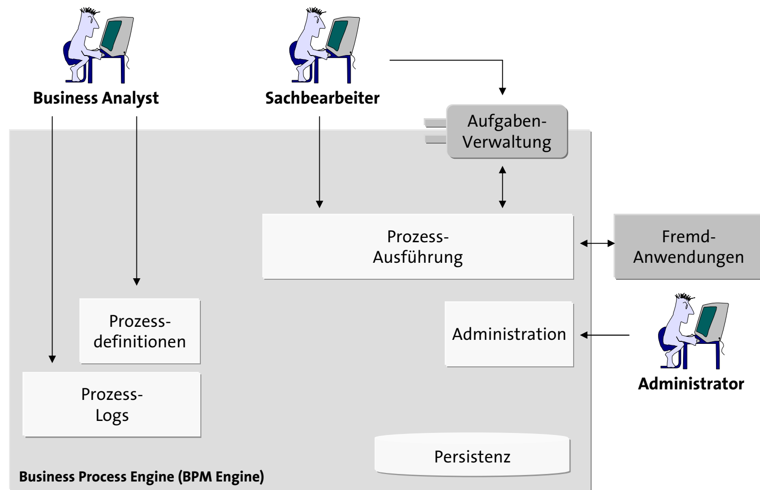



***Abbildung 14.3***
*Bestandteile einer Business Process Engine.*

jede Prozessinstanz einen entsprechenden Kontext, in dem die benötigten Daten abgelegt werden können. Der Kontext sollte dabei möglichst flexibel Daten aufnehmen, um wirklich alle für den Geschäftsprozess relevanten Daten aufnehmen zu können. Die Daten müssen dabei genau wie der Prozesszustand persistent und konsistent gehalten werden.

Zusätzlich bieten viele Business Process Engines meist weitere sinnvolle Dienste an, typische Beispiele sind:

**Zusätzliche Dienste**

**Business Activity Monitoring (BAM):** Es werden Kennzahlen für Geschäftsprozesse ermittelt (vor allem Durchlauf- und Bearbeitungszeiten) und für die Analyse aufbereitet. Somit bietet BAM detaillierte Informationen über Prozessdurchläufe und ermöglicht die Erkennung von Überlastungen, die Überwachung von Service Level Agreements (SLA) und liefert Ansatzpunkte zur Prozessverbesserung.

**Simulation:** Simulation von Prozessdurchläufen anhand statistischer Informationen kann eine große Hilfe zur Bewertung von Prozessverbesserungen oder Findung von Nadelöhren sein.

**Debugging:** Business Process Engines ermöglichen oft das Debuggen einzelner Prozessinstanzen.

**Ad-Hoc Prozesse:** Viele Prozesse in Unternehmen sind nur Teilstrukturiert. Aber auch in strukturierten Vorgängen gibt es immer wieder Ausnahmen, die nicht dem modellierten Ablauf entsprechen. In dieser Situation ermöglichen es einige Engines bestimmten Benutzern in den Prozessablauf einzugreifen und ad-hoc zu entscheiden, wie der Prozess fortgesetzt wird.

**BPMS**

Business Process Engines sind nur ein Teil der für BPM benötigten Software-Suite. Anhand des in Abbildung 14.4 auf der nächsten Seite vean-

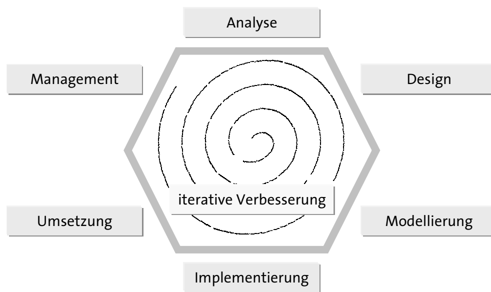



***Abbildung 14.4***
*Management von Geschäftsprozessen.*

schaulichten BPM-Kreislaufes ist ersichtlich, dass sie vor allem die Implementierung der Geschäftsprozesse unterstützen. Vollwertige Business Process Management Systems (BPMS) enthalten deshalb weitere Tools für die Modellierung und Simulation sowie das Management und die Analyse von Prozessen. Die auf den einzelnen Feldern existierenden Tools wurden dabei inzwischen zu entsprechenden Suiten integriert (durch Firmenkäufe oder Kooperationen), so dass heute die BPMS-Suiten gute Voraussetzungen schaffen, um Geschäftsprozesse IT-unterstützt zu implementieren. Dies ist vielleicht auch einer der Hauptunterschiede des heutigen BPM im Vergleich zum Workflowmanagement der 90er Jahre. Die große Herausforderung liegt nun in der Umsetzung eines Roundtrips, so dass die ständige Analyse des laufenden Prozesses zu kontinuierlichen Prozessverbesserungen führen kann. An diesem Punkt hapert es jedoch in der Praxis heute noch, so wirft z. B. die Transformation von BMPN in BPEL heute noch große Probleme auf und stellt eher eine Generierung dar, die keinen Roundtrip unterstützt.

### 14.2.3 Standards und Produkte

Es gibt heute viele verschiedene und einige wichtige Standards im Bereich der Prozessmodellierung und Ausführung, die im Folgenden kurz dargestellt werden. Abbildung 14.5 auf der nächsten Seite gibt einen groben Überblick über deren Entwicklung in den letzten Jahren.

**Standards**

**Business Process Modeling Language (BPML):** BPML ist eine formelle Sprache zur Definition von Geschäftsprozessen in XML. Sie berücksichtigt dabei wichtige Punkte wie Transaktionen, Prozessdaten, Ausnahmenbehandlung usw. und soll vor allem unabhängig von Business Process Engines oder Tools sein. Sie konnte sich allerdings gegenüber BPEL nicht behaupten und wird auch nicht weiterentwickelt.

**Business Process Modeling Notation (BPMN):** Ziel der Notation ist eine Prozessbeschreibung für den Business-Analysten, die anschließend in technische Prozessbeschreibungen (wie BPEL oder XPDL) transformiert wird. Somit kann eine Brücke zwischen Fachabteilung und Entwicklern geschlagen werden. Ihr Hauptbestandteil ist das Business Process Diagram (BPD), welches inzwischen auch von einigen bekannten Tools un-

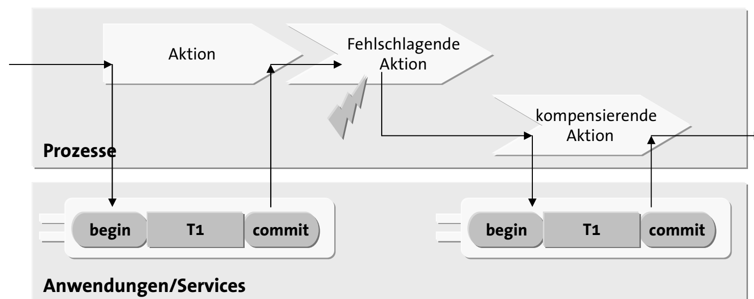



***Abbildung 14.7***
*Transaktionen und Kompensation in Geschäftsprozessen.*

„Transaktionsklammern" nachzubilden, gibt es in diesem Konzept Bereiche (sogenannte Scopes), für die jeweils eigene kompensierende Aktionen definiert werden können. So kann in unterschiedlichen Stadien des Prozesses entsprechend auf Ausnahmen reagiert werden. Ein Beispiel für Scopes und Ausnahmebehandlung mit BPEL findet sich in Kapitel 16.

Dies stellt allerdings Anforderungen an die integrierten Komponenten, da kompensierende Services entsprechend angeboten werden müssen, was heute meist nicht der Fall ist. Aber nur so kann in dem sehr asynchron geprägtem Umfeld der lang laufenden Geschäftsprozesse sinnvoll orchestriert werden.

## 14.4 Typische Integrationsarchitekturen

Im Umfeld der Java Enterprise Edition tummeln sich viele mögliche Anwendungsarchitekturen. Denkt man über Integration verschiedener Komponenten oder gar ganzer Systeme im Sinne einer serviceorientierten Architektur mit BPM nach, so sind einige Szenarien typisch:

**Prozesse mit Java Business Process Engine ohne EJB:** Es wird eine Business Process Engine innerhalb eines Webcontainers betrieben, die direkt Java-Geschäftslogik (als POJO) aufruft. In diesem Einsatzszenario muss allerdings die Transaktionssteuerung selbst durchdacht werden, was den Rahmen dieses Buches sprengen würde. Auch wird dies bei einer Lösung mit EJBs komplett vom Applikationsserver übernommen.

**Prozesse mit Business Process Engine und Stateless Session Beans:** Es kommt eine Business Process Engine innerhalb des Applikationsservers zum Einsatz, die lokale Stateless Session Beans für Geschäftslogik aufruft. Meistens existiert dann auch ein Stateless Session Bean als Facade zur Steuerung der Prozesse.

**Prozesse mit BPEL und Web-Services:** Es kommt eine BPEL Engine zum Einsatz. Geschäftslogik kann per Web-Service aufgerufen werden. Dabei ist unerheblich, ob der Service lokal oder remote angeboten wird. Auch kann Geschäftslogik, die nicht als Web-Service zur Verfügung steht, per WSIF angebunden werden (siehe Abschnitt 16.3).

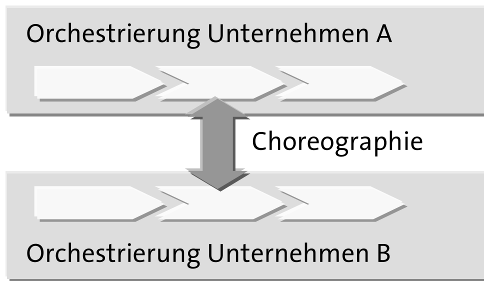



***Abbildung 14.8***
*Orchestrierung und Choreographie.*

**Enterprise Service Bus:** Mit einem ESB können Services über verschiedenste Techniken angebunden werden (EJBs, JMS, JCA, Web-Services), da ESBs typischerweise einige Standardkonnektoren mitbringen. Die Services werden dabei gegenseitig nur über den Bus angesprochen und sind somit lose gekoppelt. Ein Prozess wird meist in einer BPEL Engine modelliert, welche ebenfalls an den ESB angedockt wird.

**Plattformunabhängigkeit vs. Komplexität**

Der Unterschied dieser Lösungen liegt hauptsächlich in der Technologieneutralität und Plattformunabhängigkeit, die von oben nach unten bedeutend zunimmt. Gleichzeitig nimmt aber auch die Komplexität entsprechend zu, da nicht nur zusätzliche Middleware (BPEL Engine, ESB) zum Einsatz kommt, sondern Schnittstellen evtl. auf WSDL-Ebene gehoben werden müssen. Dies bedeutet, dass je nach Projekt durchaus kritisch betrachtet werden sollte, wie viel Plattformunabhängigkeit wirklich benötigt wird. Oft ist es dabei auch eine Option, interne Prozesse über eher Java-EE-lastige Architekturen zu orchestrieren und einzelne Services zur Choreographie dann als Web-Services zu veröffentlichen (vergleiche Abbildung 14.8).

### 14.4.1 Java Business Process Engine und Stateless Session Beans

Die naheliegendste technische Architektur ist die Integration einer Business Process Engine in den Applikationsserver. Typischerweise wird ein Stateless Session Bean als Facade davor geschaltet. Services die durch den Prozess orchestriert werden, sollten ebenfalls als Stateless Session

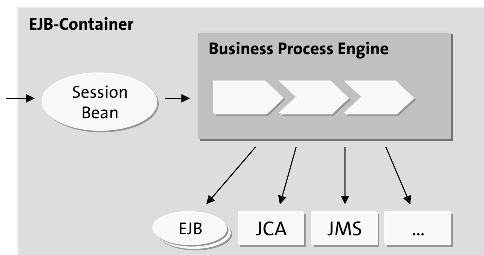



***Abbildung 14.9***
*Architektur mit Java EE Business Process Engine.*

Beans vorliegen. Des Weiteren können theoretisch alle Dienste des Applikationsservers in den Prozessen genutzt werden.

**Schnittstellen als Java-Interfaces**

Diese Lösung hat einige Vorteile. Zunächst wird keine komplizierte neue Technik außer der Business Process Engine an sich eingeführt. Dies bedeutet auch, dass Services als reine Session Beans implementiert und Schnittstellen in Java-Interfaces mit der ganzen Mächtigkeit der Programmiersprache definiert werden können. Viele mitgelieferte oder bestehende Java-EE-Komponenten können sofort verwendet werden. Auch kann die Transaktionssteuerung durch den Container übernommen werden. Mit JTA kann eine Transaktion bei der Ansteuerung der Business Process Engine über die Facade gestartet werden, die dann auch für alle Aufrufe durch die Business Process Engine Gültigkeit hat und als ganzes commited werden kann. Dies macht ein Zurückrollen bei Fehlern sehr einfach und elegant. Zu guter Letzt gelten bei einer solchen Lösung meist automatisch die Vorteile wie Clusterfähigkeit und Skalierbarkeit des Applikationsservers.

**Transaktionssteuerung durch JTA**

**Skalierbarkeit**

Ein Beispiel für eine solche Lösung ist das in Kapitel 15 ausführlich dargestellte JBoss jBPM.

### 14.4.2 Prozesse mit BPEL-Server und Web-Services

**Schnittstellen in WSDL**

Wird ein BPEL-Server eingesetzt ist man auf WSDL (Web Services Description Language) als Schnittstellenbeschreibung angewiesen. Daher müssen als Stateless Session Beans zur Verfügung stehende Dienste als Web-Services veröffentlicht werden, was jedoch seit EJB 3.0 auch nicht weiter schwierig ist. Vorsicht ist bei dieser Lösung hauptsächlich mit komplizierten Parametern angebracht, da diese nicht immer trivial in WSDL übersetzt werden können. Auch die Transaktionssteuerung ist in diesem Kontext komplizierter (siehe Kasten und Abschnitt 14.3).

**Plattformunabhängigkeit**

Diese Lösung hat den großen Vorteil der Plattformunabhängigkeit. Services können in jeder Technologie vorliegen, sofern sie über WSDL veröffentlicht sind. Auch sind in diesem Bereich inzwischen genügend

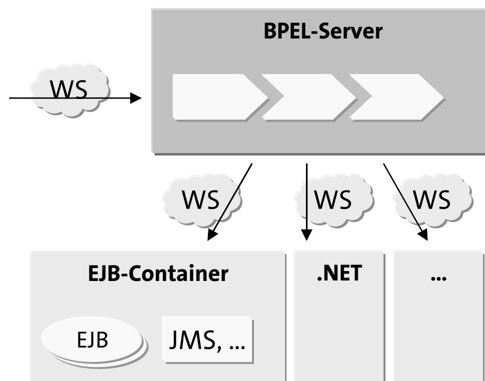



***Abbildung 14.10*** *Architektur mit BPEL-Server.*

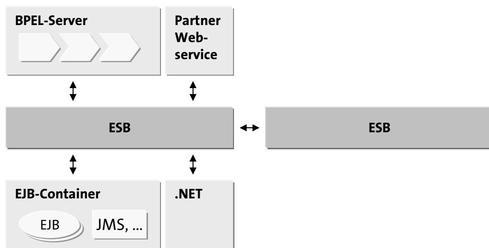



*Abbildung 14.11*
*Architektur mit Enterprise Service Bus.*

**Standards: XML, WSDL und BPEL**

Standards etabliert (wie die Sprache BPEL selbst), so dass sowohl eine Herstellerunabhängigkeit wie auch Zukunftssicherheit gegeben sind. Dies erkauft man sich durch größeren Leistungsbedarf (vor allem durch die extensive Nutzung von XML und WSDL) und eine komplexere Systemumgebung. Auch die Transaktionssteuerung bedarf komplizierter Überlegungen. Beispielhaft für diese Architektur wird der Oracle BPEL-Server in Kapitel 16 dargestellt.

**Atomare Transaktionen mit Web-Services**

Im Bereich der Webservices gibt es momentan einige Spezifikationen (z. B. `WS-AtomicTransaction` oder `WS-ACID`), um die Problematik der atomaren Transaktionen technisch auch für Web-Services abzubilden. Prinzipiell wird die gleiche Logik des 2-Phasen-Commit-Protokolls (siehe Abschnitt 8.4.1) zugrunde gelegt, allerdings mit Web-Services umgesetzt.

## 14.4.3 Enterprise Service Bus

**Zentralisierte und standardisierte Nachrichten**

Ein Enterprise Service Bus geht nun noch einen Schritt weiter, indem er die Kommunikation zwischen verschiedenen Services standardisiert und Nachrichtenverarbeitung zentralisiert. Dies hat vor allem den großen Vorteil, dass die Kopplung zwischen den Services weiter abnimmt und die Konfiguration, welche Services wo ablaufen und wie zusammenspielen, zentral im ESB gehalten wird. Möchte ein Service auf einen anderen zugreifen, kennt er im Allgemeinen nur dessen Namen (über den er dann dynamisch die Schnittstellenbeschreibung beziehen kann) oder dessen Schnittstelle. Verschiedene ESBs können dank Web-Services zusammenarbeiten, so dass diese Integration nicht an der Unternehmensgrenze Halt machen muss und es für Service-Konsumenten völlig transparent ist, wo ein Service von wem und wie implementiert wurde.

**Asynchrone Kommunikation**

Die Kommunikation in einem ESB ist asynchron geprägt, da Aufrufe über Messages und eine entsprechende Message Oriented Middleware abgewickelt werden. Dies ist für Geschäftsanwendungen meist viel natürlicher, da Geschäftsprozesse sehr stark ereignisorientiert sind (daher werden