07.04

Deutschland € 5,50 Österreich € 6,50 Schweiz SFr 10,70

S&S

Javamagazin

Internet & Enterprise Technology

XML extra included

Java Web Services

Alles über Apache Axis: Architektur, Einsatz, Erweiterung

Oracle 10g Database

Was gibt es Neues für (Java-)Entwickler?

www.javamagazin.de

Macromedia Flex
Serverseitige Rich Internet-Anwendungen

Tomcat Cluster
Clustering auf Basis von Tomcat 5

Jakarta Commons
Verborgene Apache-Schätze

FOP
Workshop PDF-Generierung

TomC@ – die Kolumne
Embedded Tomcat als JBoss Servlet Engine

mit CD!

D 45867

JBoss-Features und -Tools, Teil 5: JBoss und CMP 2.0

von Dirk Weil

Aus dem Leben eines Zauberers

Im vorigen Teil unserer Serie über den Open Source Application Server JBoss haben wir über die Integration in die Entwicklungsumgebung JBuilder berichtet und gezeigt, dass damit in recht einfacher Weise u.a. EJBs entwickelt werden können [1]. Der vorliegende Artikel betrachtet nun einen speziellen Typ der EJBs, nämlich die Entity EJBs mit CMP 2.0, und erläutert die verschiedenen Konfigurations- und Optimierungsmöglichkeiten, die JBoss uns in diesem Zusammenhang anbietet.

Persistenz in J2EE-Anwendungen

Nahezu jede Anwendung arbeitet mit Daten, die dauerhaft gespeichert werden müssen. Der überwiegende Teil davon nutzt relationale Datenbanken wie Oracle, DB2 oder MySQL. Daher stellt sich immer wieder die Aufgabe, die objektorientierte Welt der Java-Anwendung mit der tabellenorientierten, relationalen Sicht der Datenbank zu verbinden. Wurden früher dazu häufig so genannte O/R-Mapper verwendet, so ist deren Technologie heute zu einem großen Teil in EJB-Containern aufgegangen. So hat auch JBoss eine CMP Engine, die das Mapping von Java-Daten auf relationale Datenbanken inklusive Relationen unterstützt.

„Container Managed Persistence nutzen wir nicht!“

Dies ist die Aussage vieler Projektverantwortlicher, die sich schon relativ früh mit dem Thema EJBs beschäftigt haben. Gründe für diese ablehnende Haltung gibt es viele, u.a. die beiden folgenden: In der Version 1.1 der EJB-Spezifikation berücksichtigte CMP keinerlei Beziehungen zwischen verschiedenen Entitäten. Konnte man also recht einfach z.B. Kundendatensätze per CMP in der Datenbank ablegen, so war man bei der Ermittelung aller Aufträge eines Kunden – eine 1 : n-Beziehung – zur expliziten Programmierung dieser Relation außerhalb von CMP gezwungen. Zudem gab es früher nahezu keine Optimierungen. So hat etwa das Lesen von N Datensätzen regelmäßig zur Ausführung von N+1 Datenbankzugriffen geführt, unveränderte Werte wurden am Transaktionsende unnötigerweise wieder in die Datenbank zurückgeschrieben etc.

Mit Einführung von EJB 2.0 hat sich das Bild allerdings grundlegend gewandelt. Zum einen können Relationen zwischen EJBs nun mit CMP abgebildet werden. Zum anderen zeigt sich der Erfolg diverser Optimierungsmaßnahmen. Man kann mittlerweile davon ausgehen, dass die Performanz von Anwendungen mit CMP 2.0 besser ist als bei manueller Programmierung der Datenbankzugriffe.

Die Entwicklung einer Entity EJB mit CMP 2.0 ist relativ einfach, wie der Kasten „CMP – ein Einstiegsbeispiel“ weiter hinten zeigt. Es gibt allerdings noch viele weitere Programmier- und Konfigurationsmöglichkeiten, die im Rahmen dieses Artikels nicht erschöpfend behandelt werden können. Es sei dazu auf die weiterführende Literatur verwiesen (z.B. [2], [3], [4]).

Container-Konfiguration

EJBs werden zur Laufzeit nicht direkt aufgerufen, sondern per Delegation aus Hüllobjekten, die der Container generiert. Dieses Delegationskonzept erlaubt die Ausführung von Code vor und nach dem eigentlichen Methodenaufruf (Abb. 1).

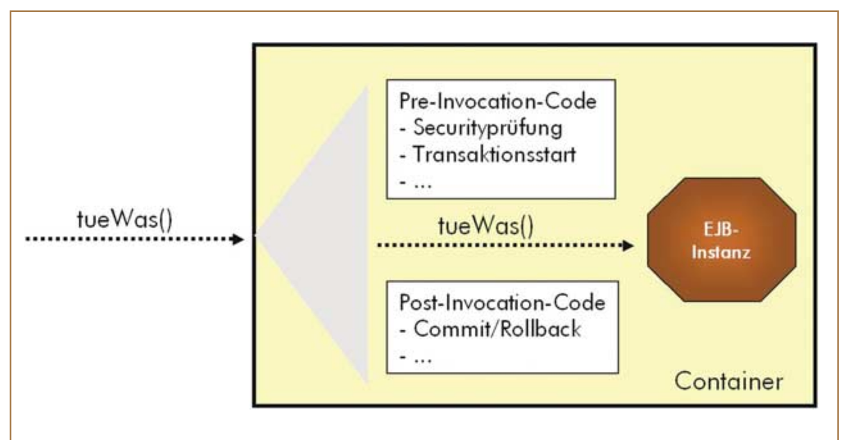

Abb. 1: Delegationskonzept

Die Konfiguration dieser Hüllobjekte geschieht im Deployment Descriptor *jboss.xml*. Hierin können beliebige Container-Konfigurationen hinterlegt und später einer EJB zugewiesen werden.

```
<jboss>
 <container-configuration>
  <container-name>Customized CMP 2.x EntityBean</
                                       container-name>
  ...
 </container-configuration>
 ...
 <enterprise-beans>
  <entity>
   <ejb-name>car</ejb-name>
   <jndi-name>demo/car</jndi-name>
   <configuration-name>Customized CMP 2.x
                       EntityBean</configuration-name>
  </entity>
 </enterprise-beans>


</jboss>
```

In der Datei *standardjboss.xml* im Verzeichnis *conf* des Servers befinden sich einige vordefinierte Container-Konfigurationen. Eine Anpassung an eigene Wünsche kann nun direkt in dieser Datei geschehen. Günstiger ist es jedoch, neue Konfigurationen als Abwandlung der Standardeinträge zu definieren, entweder als Ergänzung der *standardjboss.xml* oder im Deployment Descriptor *jboss.xml*:

```
<jboss>
 <container-configuration extends="Standard CMP 2.x
                                        EntityBean">
  <container-name>Customized CMP 2.x
                       EntityBean</container-name>
  <commit-option>A</commit-option>
 </container-configuration>
 ...
</jboss>
```

Commit-Optionen

Das oben Gesagte gilt für jeden EJB-Typ. Ich möchte mich im Folgenden allerdings auf nur einen Konfigurationsparameter beschränken, der für Entity EJBs – und damit auch für CMP – interessant ist, und zwar auf die im Beispiel bereits angeführte *commit-option*. Mit ihrer Hilfe kann man bestimmen, wie lange die einmal eingelesenen Felder einer Entity EJB gültig bleiben:

- A Hier wird angenommen, dass der Container exklusiven Zugriff zur Datenbanktabelle hat, es also keine andere EJB oder Applikation gibt, die in die gleiche Tabelle schreibt. Der Zustand der EJB wird beim ersten Aufruf einer Business-Methode gelesen und verbleibt so lange im Speicher, bis die Bean passiviert wird. Erst danach wird ggf. neu gelesen.
- B Der Zustand der Bean wird zwar im Cache gehalten, jedoch stets zu Beginn einer Transaktion neu gelesen und am Ende der Transaktion gespeichert, falls nötig. Nur Methoden, die nicht in einem Transaktionskontext ausgeführt werden, können die Cache-Daten verwenden; aber Achtung: Die sind ggf. nicht mehr aktuell!
- C Wie B, jedoch wird der Bean-Zustand außerhalb von Transaktionen nicht im Cache behalten.
- D Wie A, jedoch wird der Zustand regelmäßig nachgelesen. Dies geschieht standardmäßig alle 30 Sekunden. Mithilfe des Konfigurationselements *optiond-refresh-rate* kann die Zeit aber auch angegeben werden.

Voreingestellt ist im Standardfall die Option B. In vielen Fällen kann jedoch garantiert werden, dass Schreibzugriffe auf eine Datenbanktabelle ausschließlich durch die zugehörige EJB geschehen. Dann ist es günstiger, auf Option A zu wechseln.

Das N + 1-Problem

Viele Anwendungen nutzen Finder dazu, Listen oder Tabellen mit Daten zu füllen. Eine typische Programmsequenz ist:

```
CarHome home = (CarHome) jndi.lookup("CAR");
Collection cars = home.findAll();
Iterator iter = cars.iterator();
while ( iter.hasNext() ) {
 Car car = (Car) iter.next();
 String model = car.getModel();
 String type = car.getType();
 // ...
}
```

Bei der Ausführung dieser Sequenz zeigt sich ein recht unangenehmer Effekt des objektorientierten Zugriffs auf die Datenbank. Einerseits werden bei Benutzung einer EJB-Instanz grundsätzlich alle Attributwerte aus der Datenbanktabelle gelesen,

Listing 1

```
<jbosscmp-jdbc>
 <enterprise-beans>
  <entity>
   <ejb-name>car</ejb-name>
   ...
   <cmp-field>...</cmp-field>

   <load-groups>
    <load-group>
     <load-group-name>Liste</load-group-name>
     <field-name>type</field-name>
     <field-name>model</field-name>
    </load-group>

    <load-group>
     <load-group-name>Detail</load-group-name>
     <field-name>type</field-name>
     <field-name>model</field-name>
     <field-name>color</field-name>
     <field-name>age</field-name>
     <field-name>accidentCount</field-name>
    </load-group>
   </load-groups>

   <eager-load-group>Liste</eager-load-group>

   <lazy-load-groups>
    <load-group-name>Detail</load-group-name>
   </lazy-load-groups>

  </entity>
 </enterprise-beans>
</jbosscmp-jdbc>
```

Listing 2

```
<jbosscmp-jdbc>
 <enterprise-beans>
  <entity>
   <ejb-name>car</ejb-name>
   ...
   <lazy-load-groups>...</lazy-load-groups>

   <query>
    <query-method>
     <method-name>findAll</method-name>
     <method-params/>
    </query-method>
    <read-ahead>
     <strategy>on-load</strategy>
     <page-size>5</page-size>
     <eager-load-group>Liste</eager-load-group>
                                        <!--optional -->
   </read-ahead>
   </query>

  </entity>
 </enterprise-beans>
</jbosscmp-jdbc>
```

CMP – ein Einstiegsbeispiel

Nehmen wir einmal an, wir wollten Gebrauchtwagendaten in einer Datenbank ablegen, und zwar in einer Tabelle namens *CAR* mit den folgenden Spalten:

CAR_ID	*INTEGER NOT NULL* (Primärschlüssel)
CAR_TYPE	*VARCHAR(25) NOT NULL*
CAR_MODEL	*VARCHAR(25) NOT NULL*
CAR_COLOR	*VARCHAR(25) NOT NULL*
CAR_AGE	*INTEGER NOT NULL*
CAR_ACCIDENT_COUNT	*INTEGER NOT NULL*

Die Implementierungsklasse der passenden Entity EJB könnte so aussehen:

```
public abstract class CarBean implements EntityBean {

 public abstract Integer getId();
 public abstract void   setId(Integer id);

 public abstract String  getType();
 public abstract void   setType(String type);

 public abstract String  getModel();
 public abstract void   setModel(String model);

 public abstract String  getColor();
 public abstract void   setColor(String color);

 public abstract int    getAge();
 public abstract void   setAge(int age);

 public abstract int    getAccidentCount();
 public abstract void   setAccidentCount(int accidentCount);

 public Integer ejbCreate(Integer id, String type, String model,
             String color, int age, int accidentCount) {
  setId(id);
  setType(type);
  setModel(model);
  setColor(color);
  setAge(age);
  setAccidentCount(accidentCount);

  return null;
 }

 public void ejbPostCreate(Integer id, String type, String model,
             String color, int age, int accidentCount) {
 }

 public void setEntityContext(EntityContext BeanContext) {
 }

 public void unsetEntityContext() {
 }

 public void ejbActivate() {
 }

 public void ejbPassivate() {
 }

 public void ejbRemove() {
 }

 public void ejbLoad() {
 }

 public void ejbStore() {
 }
}
```

Es fällt auf, dass die persistenten Felder durch abstrakte Zugriffsmethodenpaare repräsentiert sind. Die Implementierung wird vom EJB-Container in einer abgeleiteten Klasse hinzugestellt.
Wir gehen davon aus, dass Entity EJBs nicht remote aufgerufen werden, definieren daher nur Local Interfaces:

```
public interface Car extends EJBLocalObject {
 public Integer getId();

 public String  getType();
 public void   setType(String type);

 public String  getModel();
 public void   setModel(String model);

 public String  getColor();
 public void   setColor(String color);

 public int    getAge();
 public void   setAge(int age);

 public int    getAccidentCount();
 public void   setAccidentCount(int accidentCount);
}

public interface CarHome extends EJBLocalHome {
 public Car create(Integer id, String type, String model, String color,
          int age, int accidentCount) throws CreateException;

 public Car findByPrimaryKey(Integer id) throws FinderException;

 public Collection findAll() throws FinderException;
}
```

Im Deployment Descriptor *ejb-jar.xml* wird die EJB sodann mit ihren persistenten Feldern deklariert:

```
<ejb-jar>
 <enterprise-beans>
  <entity>
   <ejb-name>car</ejb-name>
   <local-home>de.gedoplan.seminar.carejb.common.CarHome</local-home>
   <local>de.gedoplan.seminar.carejb.common.Car</local>
   <ejb-class>de.gedoplan.seminar.carejb.server.CarBean</ejb-class>
   <persistence-type>Container</persistence-type>
   <prim-key-class>java.lang.Integer</prim-key-class>
   <reentrant>False</reentrant>
   <abstract-schema-name>car</abstract-schema-name>
   <cmp-field><field-name>id</field-name></cmp-field>
   <cmp-field><field-name>type</field-name></cmp-field>
   <cmp-field><field-name>model</field-name></cmp-field>
   <cmp-field><field-name>color</field-name></cmp-field>
   <cmp-field><field-name>age</field-name></cmp-field>
   <cmp-field><field-name>accidentCount</field-name></cmp-field>
   <primkey-field>id</primkey-field>
   <query>
    <query-method>
```

CMP – ein Einstiegsbeispiel, Fortsetzung

```
      <method-name>findAll</method-name>
      <method-params/>
    </query-method>
    <ejb-ql>SELECT OBJECT(s) FROM car AS s</ejb-ql>
   </query>
  </entity>
 </enterprise-beans>
</ejb-jar>
```

Darin wird auch der Finder *findAll* mit einem Suchausdruck in EJB-QL verknüpft. In diesem einfachen Beispiel liefert der Finder alle Einträge der Tabelle.
Der JBoss-abhängige Descriptor *jboss.xml* enthält in vielen Fällen lediglich den JNDI-Namen der EJB:

```
<jboss>
 <enterprise-beans>
  <entity>
   <ejb-name>car</ejb-name>
   <jndi-name>demo/car</jndi-name>
  </entity>
 </enterprise-beans>
</jboss>
```

Der Descriptor kann sogar komplett entfallen, wenn *jndi-name* und *ejb-name* identisch sind. Das Mapping zur Datenbank wird im JBoss-abhängigen Descriptor *jbosscmp-jdbc.xml* bestimmt. Hierin werden die zu benutzende Datasource und Tabelle angegeben und den Feldern der EJB Spaltennamen zugewiesen:

```
<jbosscmp-jdbc>
 <enterprise-beans>
  <entity>
   <ejb-name>car</ejb-name>
   <datasource>java:/DefaultDS</datasource>
   <table-name>CAR</table-name>
   <cmp-field>
     <field-name>id</field-name>
     <column-name>CAR_ID</column-name>
   </cmp-field>
   <cmp-field>
     <field-name>type</field-name>
     <column-name>CAR_TYPE</column-name>
   </cmp-field>
   <cmp-field>
     <field-name>model</field-name>
     <column-name>CAR_MODEL</column-name>
   </cmp-field>
   <cmp-field>
     <field-name>color</field-name>
     <column-name>CAR_COLOR</column-name>
   </cmp-field>
   <cmp-field>
     <field-name>age</field-name>
     <column-name>CAR_AGE</column-name>
   </cmp-field>
   <cmp-field>
     <field-name>accidentCount</field-name>
     <column-name>CAR_ACC_COUNT</column-name>
   </cmp-field>
  </entity>
 </enterprise-beans>
</jbosscmp-jdbc>
```

selbst wenn sie in der aktuellen Transaktion gar nicht benutzt werden.

Zudem werden bei der Ausführung eines Finders wie *findAll()* zunächst nur die Primärschlüssel der gefundenen Tabelleneinträge gelesen. Bei der anschließenden Iteration über die Ergebnismenge wird dann jeder einzelne Eintrag mittels eines weiteren SQL Statement eingelesen. Sind *N* Einträge in der Datenbanktabelle vorhanden, werden also insgesamt *N+1* SQL Statements ausgeführt; dabei hätte auch ein einzelner Zugriff direkt alle Daten liefern können. Kommt dies in Programmen häufiger vor, werden aus akzeptablen Antwortzeiten indiskutable Lieferzeiten. JBoss bietet uns nun einige Optionen im Deployment Descriptor, mit denen das Ladeverhalten der CMP Engine beeinflusst werden kann.

Die Entdeckung der Faulheit

Geschäftsobjekte haben häufig eine große Menge von Feldern, von denen nur ein bestimmter, teilweise recht kleiner Anteil oft benötigt wird, während ein anderer Teil der Attribute nur selten zur Verarbeitung herangezogen wird. So werden von einem Kundenobjekt beim Aufbau einer Auswahlliste vielleicht nur die Attribute Kundennummer und Name verwendet. Die restlichen Attribute wie Adresse, Telefon etc. werden nur bei einer Detailansicht benötigt. In diesem Fall kann das Lazy Loading Pattern Anwendung finden, bei dem zunächst nur einige Attribute des zu verarbeitenden Objektes aus der Datenbank geladen werden. Findet in der Folge ein Zugriff auf eines der restlichen Attribute statt, wird es in einem zusätzlichen Datenbankzugriff nachgeladen.

JBoss unterstützt dieses Pattern durch die Berücksichtigung so genannter Load Groups, die im Deployment Descriptor *jbosscmp-jdbc.xml* deklariert werden. Eine dieser Load Groups kann dann als Eager Load Group ausgezeichnet werden, beliebig viele weitere als Lazy Load Groups (Listing 1).

Die Idee ist nun, dass beim ersten Zugriff auf eine EJB-Instanz nur die Felder der Eager Load Group mit Werten aus der zugehörigen Datenbanktabelle gefüllt werden. Wird anschließend auf eines der weiteren Felder zugegriffen, muss dessen Wert durch erneuten Zugriff auf die Datenbank nachgelesen werden. Befindet sich das Feld in einer oder mehreren Lazy Load Groups, werden bei diesem Zugriff alle Felder gefüllt, die sich ebenfalls in einer dieser Lazy Load Groups befinden.

Durch den obigen Beispielcode würden für die Erstellung einer Überblicksliste für jeden Eintrag also nur die Felder *type* und *model* gelesen. Diese Optimierung hinsichtlich des übertragenen Volumens beim Aufbau der Liste wird dadurch erkauft, dass für eine Detailanzeige nun zwei Datenbankzugriffe nötig sind, einer für *type* und *model*, einer für *color*, *age* und *accidentCount*.

Es muss allerdings noch dazu gesagt werden, dass sich die Definition von Load Groups nur dann positiv auswirkt, wenn die EJB die Commit-Option A oder D verwendet oder wenn die Zugriffe auf die ver-

schiedenen Felder der EJB innerhalb einer Transaktion liegen. Anderenfalls wird ja ohnehin jedes Mal auf die Datenbank zugegriffen, da der Zustand der EJB zwischenzeitlich ungültig wird.

Suchen mit Strategie

Das oben beschriebene N+1-Problem lässt sich auf JBoss ebenfalls weitestgehend entschärfen. Dazu können Finder mit einer Read-ahead-Strategie verknüpft werden, was dazu führt, das bei der Ausführung des Finders oder bei der anschließenden Iteration über die Ergebnismenge mehr Daten eingelesen werden, als für den Moment nötig wären, um ein späteres Nachladen von weiteren Werten ohne Datenbankzugriff zu erlauben. Es stehen zwei Strategien zur Verfügung: *on-load* und *on-find*.

Ein Finder mit der Read-ahead-Strategie *on-load* lädt bei seiner Ausführung unverändert nur die Primärschlüssel der gefundenen Einträge. Die darauf folgende Verarbeitung des Suchergebnisses geschieht dann aber portionsweise: Der Zugriff auf das erste Element der Liste führt zum Lesen von *page-size*-Datenbankeinträgen. Die nächsten *page-size*-1-Elemente können ohne weiteren Datenbankzugriff verarbeitet werden. Erst danach wird wieder ein SQL-Befehl für weitere *page-size*-Einträge ausgeführt. Die Deklaration des Read-ahead geschieht wiederum in *jbosscmp-jdbc.xml* (Listing 2).

Standardmäßig werden alle Felder der portionsweise eingelesenen Einträge geladen; die ggf. deklarierte Eager Load Group der EJB hat bei der Verarbeitung des Finder-Ergebnisses keine Bedeutung. Es kann allerdings für den Finder eine nur hier gültige Eager Load Group deklariert werden.

Die zweite Read-ahead-Strategie ist *on-find*. Hierbei werden schon bei der Ausführung des Finders mehr Felder als nur die Primärschlüssel eingelesen. Für die Verarbeitung der Ergebnismenge innerhalb der gleichen Transaktion ist dann ggf. gar kein weiterer Datenbankzugriff mehr nötig. Wie schon bei *on-load* ist die Angabe einer Eager Load Group optional möglich; der Finder liest dann nur die darin angegebenen Felder. Standardmäßig werden alle Felder gelesen, unabhängig von einer evtl. vorhandenen Eager Load Group auf Ebene der EJB. Die Deklaration in *jbosscmp-jdbc.xml* (Listing 3).

Achtung: Findet der Finder eine große Menge von Einträgen, so wird durch die Strategie *on-find* nicht unerheblich viel Speicher belegt! Man hat also mit den gezeigten Elementen einige Stellräder in der Hand, mit denen man das Verhalten von EJBs ge-genüber der Datenbank optimieren kann. Ich denke, es ist klar geworden, dass damit erhebliche Leistungszuwächse möglich sind. Eine unbedachte Konfiguration kann jedoch auch das Gegenteil bewirken.

Listing 3

```
<jbosscmp-jdbc>
 <enterprise-beans>
  <entity>
   <ejb-name>car</ejb-name>
   ...
   <lazy-load-groups>...</lazy-load-groups>

   <query>
    <query-method>
     <method-name>findAll</method-name>
     <method-params/>
    </query-method>
    <read-ahead>
     <strategy>on-find</strategy>
     <eager-load-group>Liste</eager-load-group>
                                  <!--optional -->
           </read-ahead>
   </query>

  </entity>
 </enterprise-beans>
</jbosscmp-jdbc>
```

Alternative Techniken

Natürlich sind EJBs mit CMP 2.0 nicht die einzige Möglichkeit, auf Datenbanken zuzugreifen. Die direkte Verwendung von JDBC bedeutet allerdings wesentlich mehr Kodierungsaufwand. Zudem sind die besprochenen Optimierungen dabei nur schwer zu erreichen. Eine interessante Alternative zu Entity EJBs sind sicher Java Data Objects, die ein noch leichtgewichtigeres, nahezu transparentes Persistenzschema darstellen. EJBs sind allerdings schon länger etabliert, haben (noch) eine breitere Unterstützung durch Hersteller und Entwickler. Marc Fleury hat JDO vor einiger Zeit mit dem Huhn verglichen, das die Straße überquerte, als der CMP-Laster vorbeikam [5]. Ob das so stimmt, ist selbst bei JBoss zweifelhaft (JBoss 4.0 DR2 enthält eine JDO-Implementierung, die allerdings in DR3 zugunsten von Hibernate entfallen ist) – wir werden es beobachten.

Fazit

Die Entwicklung von EJBs mit CMP 2.0 stellt sich als relativ einfache Sache heraus – zumindest, was den Java-Anteil angeht. Im Vergleich mit direktem Datenbankzugriff per JDBC – sei es aus einer normalen Klasse heraus oder aus einer Entity EJB mit BMP – ist die CMP-Programmierung weitaus kürzer und übersichtlicher – und damit weniger fehlerträchtig.

Die verschiedenen Optimierungsmöglichkeiten, die per Deployment Descriptor erreicht werden können, spielen die große Stärke des CMP-Modells auf JBoss aus. Hier zeigt sich aus meiner Sicht besonders deutlich, wie Container-Konzepte für überschaubaren Programmcode bei gleichzeitig optimaler Performanz sorgen können.

Der Erfolg stellt sich allerdings nicht ohne Mühe ein. Der Entwickler muss sich schon relativ intensiv mit Architekturen und Patterns, Use Cases und Mengengerüsten beschäftigen, um das Maximum aus seiner Anwendung herauszuholen. Es lohnt sich aber, diese Aktivitäten auch in Zeiten von Termin- und Kostendruck im Projektplan vorzusehen. ■

Dirk Weil ist Geschäftsführer der GEDOPLAN GmbH [6]. Er ist zertifizierter JBoss Consultant und seit mehr als fünf Jahren als Berater im Bereich Java tätig. Er hält Vorträge und leitet Seminare zu zahlreichen Themen der J2EE. GEDOPLAN ist autorisierter Servicepartner der JBoss Inc.

■ Links & Literatur

[1] Marcus Redeker: In zwanzig Klicks zur EJB, in *Java Magazin* 6.2004

[2] Stefan Denninger, Ingo Peters: Enterprise JavaBeans 2.0, Addison-Wesley, 2002

[3] Andreas Holubek, Rudolf Jansen et al.: Java Persistenz-Strategien, Software & Support Verlag, 2004

[4] java.sun.com/blueprints/enterprise/

[5] Marc Fleury: Why I Love EJBs, www.jboss.org/modules/html/blue.pdf

[6] www.gedoplan.de/