Spielerwechsel

Erfahrungsbericht zur Migration von Hibernate nach OpenJPA

Uwe Hohenstein, Michael C. Jaeger

Persistenzframeworks sind komplexe Softwarelösungen für die Speicherung und das Auffinden von Objekten in relationalen Datenbanken, die in sehr vielen Softwareprodukten Einzug halten. Dementsprechend stellen diese Frameworks einen häufigen Bestandteil bei Entwurf und Realisation von Software dar. Der vorliegende Beitrag soll die Architektur eines hochverfügbaren Systems mit soft-realtime Anforderungen vorstellen und Erfahrungen erläutern, die bei der Migration eines zugrunde liegenden Persistenzframeworks gesammelt wurden. Die Migration betrifft eine Dienstplattform für Kommunikationslösungen, welche zunächst das weitverbreitete objekt/relationale Persistenzframework Hibernate eingesetzt hat. Aus verschiedenen Gründen wurde der Austausch von Hibernate durch das Alternativprodukt OpenJPA vorgenommen.

Persistenzframeworks vereinfachen den Zugriff aus einer objektorientierten Programmiersprache wie Java auf ein relationales Datenbanksystem. Sie stellen Werkzeuge für die persistente Speicherung von objektorientierten Daten in relationalen Tabellen mittels einer objekt/relationalen (O/R) Abbildung dar. Ein Persistenzframework stellt Entwicklern eine objektorientierte Programmierschnittstelle (Application Programming Interface, API) zur Verfügung, mit der diese Objekte speichern können. Die Extraktion der Daten aus dem Objekt und die Kommunikation mit der Datenbank übernimmt das Persistenzframework. Umgekehrt erlaubt das Framework die Abfrage von Objekten mit einer objektorientierten API, ohne SQL-Anfragen explizit formulieren zu müssen.

Insbesondere das Java-basierte Persistenzframework Hibernate hat in der Vergangenheit in der Java-Szene zunehmend an Beliebtheit gewonnen [Be05]. Sein Reiz liegt darin, dass es ein frei verfügbares Open-Source-Projekt ist, das eine Vielzahl von Datenbanken unterstützt wie DB2, Oracle, Sybase, SQL Server, PostgreSQL oder MySQL. Ein weiterer Grund für die Beliebtheit sind sicherlich die geringen Performanzeinbußen gegenüber einer direkten Datenbankanbindung mit JDBC und die Möglichkeit, hochgradig Einfluss auf das Verhalten und die Performanz nehmen zu können. Eine mangelnde Einflussnahme wird häufig beim konkurrierenden Standard Java Data Objects und entsprechenden Produkten bemängelt [BS05].

Mitte des Jahres 2006 wurde durch das Unternehmen Firestar eine Patentverletzung von Red Hat, dem Lieferanten von Hibernate, in den Vereinigten Staaten beklagt: Firestar hält ein Patent auf O/R-Abbildungen [HODC00]. Auch wenn dieses Patent technisch gesehen aufgrund bekannter vorangegangener Arbeiten (z. B. durch das Enterprise Objects Framework von NeXT aus dem Jahre 1994 [Ne94]) nicht tragfähig scheint, musste erwartet werden, dass auch bei klaren technischen Verhältnissen die Rechtssprechung unerwartete Entscheidungen treffen kann. Was zunächst nur als ein Problem für Red Hat erscheint, entpuppt sich schnell als größeres Problem: Jede Software, die mit Hibernate in die USA ausgeliefert wird, ist von der Patentklage ebenfalls erfasst, da die Redistribution die Rolle eines Lieferanten impliziert.

Anwendungsfall

Somit wurde die Patentklage auch zu einem Problem für ein Geschäft der Siemens AG, welches Hibernate in einer Telekommunikations-Middleware namens OpenSOA [Str07] integriert. OpenSOA implementiert eine Plattform für Dienste, mit denen eine serviceorientierte Architektur (SOA) für Anwendungen im Kommunikationsumfeld realisiert wird. Die Plattform bündelt Dienste und Funktionen, die Gegenstand jeglicher Kommunikationsanwendungen sind, und ermöglicht durch diese Form der Wiederverwendung eine effiziente Entwicklung von Anwendungen.

OpenSOAs Dienstplattform ist auf Basis einer OSGi-Laufzeitumgebung implementiert. Da die OSGi-Spezifikation nicht alle Funktionen einer serviceorientierten Architektur abdeckt, wurden Erweiterungen erstellt oder bereits erhältliche Software hinzugenommen. Besonderes Augenmerk wurde hierbei auf die Anforderungen an Performanz und Skalierbarkeit für Kommunikationsanwendungen gelegt. Hierbei dient der OGSi-Container als leichtgewichtige Basis. Hibernate wird in OpenSOA benutzt, um die Programmierung zu vereinfachen und um auf mehreren Datenbanksystemen lauffähig zu sein, im Wesentlichen solidDB, MySQL und PostgreSQL.

Um die Bewegungsfreiheit auf dem amerikanischen Markt zu erhalten bzw. auszubauen und um Produkte dem Kunden weiterhin ohne Risiken anzubieten, entschloss sich das Projektmanagement, Hibernate durch ein anderes Persistenzframework zu ersetzen.

Ersatzkandidaten für Hibernate

Die rein technische Lösung, der Patentklage zu begegnen, besteht darin, eine Alternative zu Hibernate in die OpenSOA-Software zu integrieren, schließlich ist eine Vielzahl an Persistenzframeworks verfügbar. Dabei ist aber nicht sichergestellt, ob andere Produkte nicht ebenfalls von der Patentklage betroffen sind und ob eine Klage gegen sie erhoben wird. Dennoch schien dies als der praktikabelste Weg, insbesondere weil vie-

le Produkte gängigen Standards folgen und somit einen erneuten Wechsel des Produkts mit moderatem Aufwand ermöglichen würden. Somit ist standardkonformen Produkten sicherlich der Vorzug gegenüber anderen denkbaren Alternativen wie iBATIS mit proprietären Schnittstellen zu geben.

Als Standards kommen derzeit Java Data Objects (JDO) und die Java Persistence API (JPA) als Bestandteil der EJB 3.0-Spezifikation infrage. Dabei ist JPA im Vergleich zum JDO-Standard aktueller und derzeit auch aus folgenden Gründen die bessere Wahl:

- JPA ist ein Teil des Standards für EJB3-Applikationsserver. JPA-konforme Produkte sind sowohl unter Java SE als auch unter Java EE lauffähig. Das erlaubt eine leichtere Migration von Applikationen, die für einen EJB3-Container entwickelt wurden, auch ohne Container laufen zu lassen, wie auch umgekehrt.
- JPA beinhaltet nach eigenen Aussagen das „Beste“ aus Hibernate, TopLink, JDO und EJB2.1-Container-Managed Persistence.
- Für eine Ersetzung von Hibernate ist sicherlich relevant, dass JPA eine gewisse Hibernate-Nähe aufweist, insbesondere bei der Anfragesprache JPQL.

Die Wahl fiel somit recht schnell auf den JPA-Standard. Nach kurzer Evaluierung wurde OpenJPA als Hibernate-Ersatz ausgewählt, da es Hibernate-nah und als Open Source verfügbar ist. OpenJPA ist aus dem Produkt Kodo von BEA Systems hervorgegangen. Im Jahr 2006 hat sich BEA entschlossen, den Großteil des Produkts Kodo als Open-Source-Produkt der Apache Software Foundation öffentlich weiter zu entwickeln, wohingegen sich die von BEA Systems weiterhin angebotene, kommerzielle Version durch Werkzeuge und Analysesoftware abhebt.

Auch wenn sich Red Hat und Firestar im Mai 2008 bezüglich der Patentklage geeinigt haben, wurde die Migration abgeschlossen, zum einen, da OpenJPA unter der Apache-Softwarelizenz vertrieben wird. Diese bietet Vorteile gegenüber der LGPL von Hibernate. Zum anderen ist die Standardkompatibilität bedeutend, auch wenn Hibernate einen JPA-kompatiblen Aufsatz bietet, aber ansonsten proprietäre APIs bietet. Diese potenzielle Austauschbarkeit ist nun mal für industrielle Anwendungen ein wichtiges Argument.

Die technische Migration

Ein Persistenzframework wie Hibernate oder OpenJPA benötigt Information darüber [Be05], welche Java-Klassen und welche Felder persistent sind und wie die Java-Klassen auf Datenbanktabellen, Felder auf Tabellenspalten, Assoziationen auf Fremdschlüssel usw. abzubilden sind. Das kann entweder über XML-Spezifikationen in einer Mapping-Datei oder über Java-5-Annotationen in den Persistenz-Klassen erfolgen.

In OpenSOA wurde Hibernate größtenteils mit XML-Spezifikationen verwendet. Die Mapping-Datei `Customer.hbm.xml` in Listing 1 beschreibt die Abbildung der Java-Klasse `Customer` aus Listing 2 auf eine Tabelle `C`.

Aufgrund der Mapping-Datei werden mittels der Angabe von `table="C"` Objekte der Klasse `Customer` in eine korrespondierende Tabelle `C(cid,cname)` abgelegt. Persistente Attribute werden mit `<property>` ausgezeichnet und mit `column=` auf Tabellenspalten abgebildet, beispielsweise `name` auf die Spalte `cname`. Zudem wird ein eindeutiger Identifikator (OID) mit `<id>` ausgezeichnet. Alle Objekte der Klasse sind über ihren id-Wert unterscheidbar. Die Aufträge vom Typ `Order` gelangen analog in eine Tabelle `O(oid,part,custid)`. Die 1:n-Beziehung `Customer–Order` ist in `Customer.hbm.xml` als

```
<set name="orders"> <key column="custId"/> <one-to-many class="Order"/>
```

```
<class name="Customer" table="C">
  <id name="id" column="cid" type="long">
    <generator class="native"/>
  </id>
  <property name="name" column="cname"/>
  <set name="orders" cascade="delete">
    <key column="custId"/> <one-to-many class="Order"/>
  </set>
</class>
```

Listing 1: Beispiel einer hbm-Mapping-Datei, Datei Customer.hbm.xml

```
class Customer {
  int    id;
  String name;
  Set    orders;
  <entsprechende get/set-Methoden>
}

class Order {
  int    oid;
  String part;
  <entsprechende get/set-Methoden>
}
```

Listing 2: Beispiel für eine auf Datenbanktabellen abzubildende Java-Klasse

vereinbart. Hierdurch wird die Spalte `custId` der Tabelle `O` zur Repräsentation der Beziehung genommen, die Spalte verweist als Fremdschlüssel auf den Primärschlüssel `cid` von `C`.

Hibernate ermöglicht eine umfangreiche Steuerung der Abbildung von Klassen und Beziehungen auf Tabellen, wie [Pl04] systematisch aufzeigt.

Das Quelltextbeispiel in Listing 3 zeigt, wie mit der Hibernate API programmiert wird. Nach dem Öffnen einer Session, das ist im Prinzip eine Verbindung zum Datenbanksystem, mit `openSession` wird eine Transaktion begonnen. In dieser wird zunächst ein Objekt der Klasse `Customer` angelegt, das anschließend mit einem `save`-Aufruf in der Datenbank gespeichert wird. Eine Anfrage liefert danach alle Java-Objekte der Klasse `Customer`. Alle gelesenen Objekte werden in einem Cache verwaltet [HB06]. Ein zweiter Zugriff mit dem Aufruf `get` holt direkt den Kunden mit der Nummer 4711, der sich bereits aufgrund der vorangegangenen Anfrage im Cache befindet. Die Transaktion wird zum Schluss mit `commit` abgeschlossen und die Datenbankverbindung durch das Schließen der Session freigegeben. Auf eine adäquate Behandlung von Ausnahmefällen wurde hier verzichtet.

Das Quelltextbeispiel in Listing 4 zeigt, wie mit OpenJPA programmiert wird. Die Programmierung erfolgt mit den Klassen `EntityManagerFactory`, `EntityManager` und `EntityTransaction`, die eine vergleichbare Funktionalität wie `SessionFactory`, `Session` bzw. `Transaction` bieten. So erfolgt in OpenJPA das Speichern eines Objekts mit `EntityManager.persist()` statt `Session.save()` usw. Insofern erscheint ein Wechsel von Hibernate nach OpenJPA recht einfach durchführbar.

Auch die zu persistierenden Java-Klassen können bei einer Umstellung von Hibernate nach OpenJPA unverändert bleiben. Allerdings müssen die Mapping-Dateien umgestellt werden, da die XML-Formate eine unterschiedliche Syntax aufweisen und kleinere konzeptionelle Unterschiede bestehen. Eine XSLT-Transformation zur Überführung ist allerdings recht einfach zu erstellen.

Im Grunde gibt es zwischen Hibernate und OpenJPA sehr viele Gemeinsamkeiten in grundlegenden Konzepten mit geringen syntaktischen Unterschieden. Das trifft auch für einige Besonderheiten zu wie Notifizierungen von Datenbankereignissen oder ein Konzept, das die Austauschbarkeit von Daten-

```
SessionFactory sf = new Configuration().buildSessionFactory();
Session mySession = sf.openSession();  // repräsentiert DB-Verbindung
Transaction tx = mySession.beginTransaction();
Customer c = new Customer(4711);
mySession.save(c);
Query q = mySession.createQuery("SELECT c FROM Customer c");
List result = q.list();
Customer c2 = mySession.get(Customer.class, 4711);
tx.commit();
mySession.close();
```

Listing 3: Applikationsprogrammierung mit Hibernate

banksystemen ermöglicht, aber dennoch proprietäre Konzepte eines einzelnen Datenbanksystems effizient nutzt: In Hibernate kümmert sich ein spezielles Dialect-Konzept um die bestmögliche Umsetzung der Hibernate-Features auf adäquate datenbanksystem-spezifische Mittel. In OpenJPA gibt es ein korrespondierendes Dictionary-Konzept. Der Satz an vorgefertigten Dictionary-Klassen ist geringer als bei Hibernate, umfasst aber alle bekannteren Datenbanksysteme, wobei beide solidDB, einen unserer Kandidaten, nicht unterstützen.

Um die Umstellung von Hibernate auf OpenJPA auf möglichst wenige Quellcodeänderungen zu reduzieren, wurde das Hibernate-API auf der Grundlage von OpenJPA in einem neuen Java-Package implementiert. Um Hibernate auszutauschen, war lediglich der Import auf das neue Package zu legen. Auch wenn viele Unterschiede durch einen derartigen Wrapper-Ansatz abgefangen werden konnten, traten jedoch Probleme auf, die eine weiter gehende Behandlung erfordern.

Im Folgenden richtet sich der Fokus auf diverse Probleme,

```
EntityManagerFactory emf =
  Persistence.createEntityManagerFactory("CustomerDB");
EntityManager em = emf.createEntityManager();
EntityTransaction tx = em.getTransaction();
Customer c = new Customer(4711);
em.persist(c);
Query q = em.createQuery("SELECT c FROM Customer c");
List result = q.list();
Customer c2 = mySession.find(Customer.class, 4711);
tx.commit();
em.close();
```

Listing 4: Applikationsprogrammierung mit OpenJPA

die eine Verzögerung bei der Umstellung von Hibernate auf OpenJPA bereiteten, wobei die Ergebnisse von [VS07] hier ausgelassen werden. Alle Erkenntnisse gelten im Prinzip auch für JPA als Standard.

Unterschiede bei der Konfiguration

In Listing 3 wurde bereits die Konfiguration mit Hibernate sichtbar. In Hibernate erhält man eine **`SessionFactory`** über ein **`Configuration`**-Objekt:

```
SessionFactory sf = new Configuration().buildSessionFactory();
```

Die **`SessionFactory`** repräsentiert eine Datenbank und verwaltet die zugehörige Mapping-Information. Hibernate benutzt eine Konfigurationsdatei, typischerweise mit Namen **`hibernate.cfg.xml`**, die im Klassenpfad liegen muss oder über einen expliziten Pfad referenziert werden kann. Diese Datei wird von **`buildSessionFactory()`** gelesen und verarbeitet. Listing 5a zeigt, dass diese Konfigurationsdatei typische Datenbankparameter enthält, wie die Datenbank-URL, den Treiber, Benutzer, Passwort, und mit **`<mapping>`** auf die persistenten Klassen verweist.

OpenJPA verfährt ähnlich, führt aber eine sogenannte PersistenceUnit als logischen Datenbanknamen ein. Der Name wird der **`createEntityManagerFactory`**-Methode mitgegeben. In der Konfigurationsdatei **`persistence.xml`** lassen sich zu mehreren PersistenceUnits die Verbindungsoptionen (Treiber, URL usw.) vereinbaren (s. Listing 5b). Hier steht auch ein Verweis auf die Mapping-Datei **`orm.xml`** bzw. die annotierten Klassen.

OpenJPA erwartet die Dateien **`persistence.xml`** und **`orm.xml`** in einem META-INF-Verzeichnis innerhalb des Klassenpfads. Das ist nicht immer vorteilhaft. Da wir OpenJPA in einem OSGi-Container benutzen, wird die Datenbankkonfiguration Bestandteil der Deployment-JAR-Datei. Das verletzt das projektinterne Deployment-Prinzip, bei dem das Datenbanksystem, z. B. die IP-Adresse und der Port, jederzeit nach dem Deployment im Container beim Kunden einstellbar ist. Eine solche Änderbarkeit nach dem Deployment erfordert nun eine Nachbearbeitung der JAR-Datei, die auf dem Deployment-Rechner nicht möglich ist. Um datenbanksystem-spezifische Daten außerhalb des Deployments zu verwahren, aber dennoch die OpenJPA-Initialisierung zu gewährleisten, wurden die Datenbankverbindungsdaten in eine externe Properties-Datei ausgelagert und über den Wrapper-Ansatz mitbehandelt. Die Konfiguration ist somit nicht mehr Bestandteil des Deployments.

Connection-Pool

Zu den Best Practices von Hibernate zählt es, eine Session zu beenden, sobald die Transaktion mit **`commit()`** oder **`rollback()`** beendet wird. Es mag verwundern, dass nicht mehrere Transaktionen mit einer Session abgewickelt werden, aber der Grund liegt darin, dass bei Transaktionsende der Datenbestand in der Datenbank und der im Session-Cache auseinander laufen; wenn andere Transaktionen dieselben Daten in der Zwischenzeit in der Datenbank geändert haben, werden diese Änderungen im Cache nicht sichtbar. Da eine Session eine Datenbankverbindung repräsentiert, deren Auf- und

```
<!–Hibernate: hibernate.cfg.xml -->
<property name="connection.url">
  jdbc:mysql://localhost:3306/db </property>
<property name="connection.driver_class">
  com.mysql.jdbc.Driver</property>
<property name="dialect">
  org.hibernate.dialect.MySQLDialect </property>
<property name="connection.username"> itsMe </property>
<property name="connection.password"> myPassword </property>
<!-- c3p0 Connection Pool -->
<property name="c3p0.max_size"> 50 </property>
<!-- Mapping-Dateien -->
<mapping resource="Customer.hbm.xml"/>
ence>
```

Listing 5a: Konfiguration in Hibernate und OpenJPA

```
<!–OpenJPA persistence.xml -->
<persistence-unit name="CustomerDB" transaction-type="RESOURCE_LOCAL">
  <class> Customer </class>
  <properties>
   <property name="openjpa.ConnectionDriverName"
             value="com.mysql.jdbc.Driver"/>
   <property name="openjpa.ConnectionURL"
             value="jdbc:mysql://localhost:3306/db"/>
  </properties>
 </persistence-unit>
</persistence>
```

Listing 5b: Konfiguration in Hibernate und OpenJPA

Abbau zeitaufwändig ist, ermöglicht Hibernate die Verwaltung eines Connection-Pools, der logisch freigegebene Datenbankverbindungen nicht physisch freigibt, sondern für nachfolgende Benutzungen in einen Pool stellt. Daher sind keine Performanzeinbußen durch permanentes Öffnen und Schließen zu befürchten.

Hibernate bietet eine vorgefertigte Konfiguration für den c3p0-Connection-Pool. Die Handhabung ist sehr einfach über Einträge `<property name="c3p0.min_size">10</property>` in der Konfigurationsdatei (vgl. Listing 5a) vorzunehmen. Die voreingestellte OpenJPA-Konfiguration benutzt standardmäßig keinen Connection-Pool, was zu den angesprochenen Performanzeinbußen führt. Allerdings können DBCP oder c3p0 hinzugeschaltet werden. Bei der Verwendung von DBCP mit MySQL ist die Property `openjpa.ConnectionDriverName` auf `value= "org.apache.commons.dbcp.BasicDataSource"` statt `value="com.mysql.jdbc.Driver"` zu setzen. Der eigentliche MySQL-Treibername ist dann zusammen mit der Datenbank-URL und den anderen Eigenschaften zu einer `openjpa.ConnectionProperties` zusammenzufassen:

```
<property name="openjpa.ConnectionProperties"
 value="DriverClassName=com.mysql.jdbc.Driver,
 Url=jdbc:mysql://localhost:3306/db,Username=itsMe,Password=myPassword"/>
```

Problematisch ist hierbei die unterschiedliche Wirkung der Pool-Konfigurationsparameter wie minimale und maximale Größe. Der erste Versuch, DBCP entsprechend zu konfigurieren, ergab z. B. ein unerwartetes oszillierendes Verhalten, bei dem der Pool trotz permanenter Last in kurzer zeitlicher Abfolge permanent geschrumpft und wieder angewachsen ist.

Lebenszyklus von Objekten

In Hibernate wird oft aus Performanzgründen ein Objekt als Muster zum Löschen benutzt, um einen vorangehenden Lesezugriff auf das Objekt zu vermeiden:

```
Customer c = new Customer(4711);
session.remove(c);  // Löschen des Objekts mit id=4711
```

Existiert ein Objekt mit der id=4711, so wird es gelöscht, andernfalls passiert nichts. Das funktioniert in OpenJPA so nicht, da durch den Konstruktor ein neues temporäres Objekt erzeugt wird, da ein Persistieren mit `persist()` noch nicht erfolgt ist. Somit liegt bei `remove()` kein persistentes Objekt vor, welches zu löschen wäre, und es wird keine Datenbankoperation ausgelöst. Analoges gilt für ein Zurückschreiben des Objekts. OpenJPA behandelt das Objekt ebenfalls als neu, sodass sich das Datenbanksystem mit einer Eindeutigkeitsverletzung beklagt. Hier hilft nur, in die Anwendungsprogrammierung einzugreifen, um derartige semantische Unterschiede auszugleichen.

Löschkaskadierung

Hibernate bietet eine flexible Steuerung, Datenbankoperationen wie das Löschen oder Speichern über Beziehungen mit einer `cascade`-Option zu kaskadieren. Bei der Löschkaskadierung gibt es zwei Varianten: Während mit `cascade="delete"` für die `Customer-Order`-Beziehung vereinbart wird, dass mit einem `Customer`- auch die in Beziehung stehenden `Order`-Objekte gelöscht werden, sorgt die Option `delete-orphan` dafür, dass kein `Order`-Objekt ohne Vater existieren kann. Wird also die Beziehung zwischen dem `Customer`- und dem `Order`-Objekt aufgelöst, so verliert das Sohn-Objekt seine Lebensberechtigung und wird ebenfalls gelöscht. Bei `"delete"` bleibt das `Order`-Objekt gewissermaßen ohne Vater bestehen.

Eine Kaskadierung wird von OpenJPA ebenfalls unterstützt. Allerdings stellt OpenJPA für `delete-orphan` nur einen nicht-JPA-konformen Extra-Mechanismus bereit, der eine Kaskadierung über eine Annotation `@ElementDependant` ermöglicht; ein XML-Äquivalent gibt es jedoch nicht. Um eine aufwändige, manuelle Programmierung zu vermeiden, muss folglich auf Annotationen umgestellt werden.

Unterschiede bei Anfragen

Obwohl die Hibernate Query Language HQL und OpenJPAs Pendant JPQL im Großen und Ganzen sehr ähnlich sind, treten dennoch im Praxisbetrieb Unterschiede auf, die mehr oder weniger Probleme aufwerfen können. Hierzu gehören kleinere syntaktische Unterschiede. Zum Beispiel ist statt `SELECT COUNT(*) FROM Customer` die Form `SELECT COUNT(c) FROM Customer c` zu nehmen. Im Gegensatz zu Hibernate sind in OpenJPA auch explizite Variablen in Pfadausdrücken erforderlich. Statt `SELECT c FROM Customer c WHERE name='Ms.Marple'` oder gar `FROM Customer WHERE name='Ms.Marple'` muss es in voller Länge `SELECT c FROM Customer c WHERE c.name='Ms.Marple'` heißen.

Bei der Migration trat dabei das Problem auf, dass im GUI Bedingungen der Form `Feld=Wert` zusammengestellt werden, die direkt an Hibernate weitergereicht werden konnten. Mit OpenJPA ist eine Korrelationsvariable als `x.Feld=Wert` hinzuzufügen, die passend zur Klasse zu wählen ist. Weitere syntaktische Unterschiede bestehen beim Eager Fetching. So lädt `SELECT x FROM Customer c JOIN FETCH c.orders o` in Hibernate mit dem Objekt `c` gleich die in Beziehung stehenden `Order`-Objekte mit aus der Datenbank in den Cache. In OpenJPA ist nur ein `JOIN FETCH c.orders` ohne Korrelationsvariable `o` möglich. Das Fehlen von `o` schränkt die Anfragemöglichkeiten ein. Unterschiede beim Lazy/Eager Fetching und die daraus resultierende Performanzproblematik sind ohnehin ein komplexes Thema. Performanzkritisch ist auch, dass OpenJPA aus:

```
DELETE FROM Customer c WHERE c.name= 'Ms.Marple'
```

folgende Anfrage mit Selbstbezug erzeugt, die in den meisten Datenbanksystemen verboten ist:

```
DELETE FROM tab WHERE id IN (SELECT id FROM tab WHERE c.name='Ms. Marple')
```

Dieses Problem lässt sich über eine geänderte Dictionary-Klasse lösen, indem das datenbanksystem-spezifische Dictionary so abgeändert wird, dass nunmehr eine relationale SQL-Anweisung ohne Selbstbezug generiert wird.

Schlüssel-Generierung und Objektidentität

Eine persistente Java-Klasse erfordert einen Identifikator, der in Hibernate-Mappings mit `<id>` ausgezeichnet ist (vgl. Listing 1) und Objekte der Klasse eindeutig identifiziert. Hibernate unterstützt mehrere Strategien, die über `<generator>` ausgewählt werden. Beispielsweise besitzt Hibernate eine UUID-Generierung (weltweit global eindeutiger Identifier) und eine `increment`-Strategie (inkrementiere den höchsten Schlüsselwert der Tabelle).

Obwohl der aktuelle JPA-Standard keine UUID-Schlüsselgenerierung vorsieht, bietet OpenJPA dennoch einen proprietären Mechanismus als Erweiterung der AUTO-Strategie:

```
<generated-value strategy="AUTO" generator="uuid-hex"/>
```

Leider funktioniert das nicht mit XML-Mapping-Dateien. Man kann aber auf folgende Annotation ausweichen:

```
@Id @GeneratedValue(strategy=GenerationType.AUTO, generator="uuid-hex")
```

Will man Annotationen vermeinden, so muss eine explizite Initialisierung mit einem UUID-Generator realisiert werden. Auch der `increment`-Mechanismus fehlt in OpenJPA, der allerdings durch eine `SELECT MAX`-Anfrage und nachfolgende Inkrementierung recht einfach nachgebildet werden kann.

Darüber hinaus kann Hibernate typische Mechanismen der Datenbanksysteme wie Sequenzgeneratoren (z. B. solidDB) oder Autoinkrement-Spalten (z. B. MySQL) nutzen, um Schlüsselwerte zu belegen, die dann als Strategien `sequence` bzw. `identity` wählbar sind. Da die OpenSOA-Applikationen MySQL, solidDB und PostgreSQL unterstützen müssen, ist ein abstrakter, vom Datenbanksystem unabhängiger Mechanismus nötig. Schließlich ist das Ziel eines O/R-Frameworks die Unabhängigkeit von Datenbanksystemen, was eigentlich auch für Mapping-Dateien gelten sollte. Hibernate verfügt zu diesem Zweck über eine `native`-Strategie, die, je nachdem, was das zugrundeliegende Datenbanksystem anbietet, entweder `sequence` oder `identity` auswählt.

OpenJPA bietet eine vergleichbare `AUTO`-Strategie, die ebenfalls OpenJPA entscheiden lässt, wobei es aber verfügbare Sequenzgeneratoren oder Autoinkrement-Spalten ignoriert und immer einen Hi/Lo-Mechanismus wählt, der High/Low-Werte in einer gesonderten Tabelle verwaltet. Das führt natürlich zu Wertekonflikten bei bestehenden Datenbanken im Feld, da bereits durch Sequenzen oder Autoinkrement-Spalten vergebene Werte höchstwahrscheinlich nochmals mit Hi/Lo erzeugt werden. Natürlich kann man datenbanksystem-spezifische Mapping-Dateien pflegen, die je nach gewähltem Datenbanksystem direkt `sequence` oder `identity` festlegen. Ein modellgetriebener Ansatz kann hier helfen, eine entsprechende Mapping-Datei zu erzeugen. Da die Mapping-Datei Bestandteil des Deployment-JARs sein muss, steht das wiederum im Konflikt zum datenbanksystem-unabhängigen Deployment, bei dem beim Kunden das Datenbanksystem eingestellt werden kann. Eine Änderung der Deployment- und Installationsprozedur wäre wiederum sehr aufwändig.

Das Problem wird noch dadurch verstärkt, dass einige OpenJPA-Konzepte nur als Annotation vorliegen, wie z. B. `delete-orphan`. Einerseits ist es mühselig, `delete-orphan` manuell zu implementieren, insbesondere wenn im Objektmodell Kaskaden über mehrere Stufen gehen. Nutzt man andererseits die `delete-orphan`-Option mit einer Annotation, so sind mehrere Quellcodevarianten vorzuhalten, da die Mapping-Annotationen Bestandteil des Quellcodes sind. Die fehlende Unterstützung von `native` ist für die Migration daher schwerwiegend.

Die wesentliche Idee, dennoch einen abstrakten Mechanismus in OpenJPA einzubringen, besteht nun darin, das Verhalten beider Einzelstrategien `sequence` und `identity` so abzuändern, dass OpenJPA intern zur richtigen Strategie wechselt. Das heißt, ist `identity` zwar gewählt, wird aber vom Datenbanksystem keine Autoinkrement-Spalte unterstützt, so wird intern `sequence` gewählt. Im Prinzip ist für diesen Ansatz eine Änderung des OpenJPA-Quellcodes nötig, der glücklicherweise als Open Source vorliegt. Eine Quellcode-Änderung impliziert aber, dass der build-Prozess von OpenJPA verstanden und in den OpenSOA-build-Prozess integrierbar ist. Da dieses sehr aufwändig ist, wurde AspectJ [La03] benutzt, um die Änderungen einzubringen. Einzelheiten hierzu lassen sich in [HJ09b] nachlesen.

Performanz

Das Lazy/Eager-Fetching-Prinzip ist für die Performanz von O/R-Applikationen sehr bedeutsam [HB06]. Hibernate und OpenJPA unterscheiden sich hierbei sowohl konzeptionell als auch syntaktisch sehr stark. Generell sind die Möglichkeiten des JPA-Standards wesentlich geringer, was OpenJPA durch proprietäre Konzepte kompensiert.

Aus Performanzgründen ist es auch notwendig, OpenJPA mit einem eingeschalteten Query Compilation Cache zu betreiben, da ansonsten gleiche JPQL-Anfragen wiederholt nach SQL transformiert werden, was zu erheblichen Performanzeinbußen führt. Im Zusammenspiel von OpenJPA und OSGi trat dabei ein Class-Loading-Problem auf. Ein sogenanntes „Split Package“ (`javax.transaction.*`) wurde mit reduziertem Interface aus der JRE anstelle des mitgegebenen Packages genommen. Um den Query Compilation Cache zu verwenden und um die Bootloader Delegation aus Kompatibilitätsgründen zu erhalten, musste hierfür das Classloading-Verhalten für das in OpenJPA enthaltene OSGi-Bundle verändert werden.

Zusammenfassung

Grundsätzlich ist die Migration von Hibernate nach OpenJPA mit moderatem Aufwand möglich. Dem vorgestellten Projekt kommt entgegen, dass das Persistenzframework in einer Template-Bibliothek gekapselt ist, sodass durch deren Umstellung bereits ein Großteil der Arbeiten, wie die andere Form der Konfiguration oder das Problem mit dem Query Compilation Cache, an zentraler Stelle erledigt werden kann. Die Erfahrung zeigt auch, dass eine zügige Projektfindung in Verbindung mit einem heuristischen, iterativen Vorgehen zu einem fundierten und erfolgreichen Ergebnis führt [HJ09a].

Trotzdem traten bei der Migration eine Reihe von kniffligen Problemen auf, z. B. das Fehlen der nativen Schlüsselgenerierung oder das Query-Compilation-Cache-Problem, die im Vorfeld nicht ohne Weiteres zu erwarten waren. Für alle Probleme, die zur Übersetzungszeit oder Laufzeit auftraten, mussten adäquate Lösungen bereitgestellt werden, die sich wie folgt kategorisieren lassen:

- Benutzung nicht-standardkonformer OpenJPA-Erweiterungen: OpenJPA bietet einige proprietäre Erweiterungen an, die Hibernate-Funktionalität wie delete-orphan-Kaskadierung nachbilden. Obwohl diese die prinzipielle Austauschbarkeit von OpenJPA gefährden, wurde von ihnen Gebrauch gemacht, um Zeit bei der Migration zu sparen.
- Änderung der Dictionary-Klasse: Ebenso wie Hibernate konzentriert OpenJPA die Abhängigkeit von Datenbanksystemen in Dictionary-Klassen, die sich um die effiziente Umsetzung auf proprietäre Datenbankkonzepte kümmern. Manche Probleme ließen sich mit einer eigenen Dictionary-Klasse lösen.
- Anwendungsprogrammierung: Teilweise musste in die Programmierung eingegriffen werden, um fehlende Hibernate-Funktionalität nachzuprogrammieren.
- Nutzung der Aspektorientierung: Für einige schwerwiegende Probleme, unter anderem auch tief im Innern von OpenJPA, reichten die vorangegangenen Mittel nicht aus. Mit der aspektorientierten Sprache AspectJ stand ein mächtiger Mechanismus bereit, diese Probleme schnell und elegant zu lösen [HJ09b].

Die verwendeten Lösungsstrategien verdeutlichen, dass sehr unterschiedliche Ansätze den gewünschten Erfolg bringen können. Dabei traten einige Probleme auf, die an und für sich keine besondere Verbindung mit der eigentlichen Domäne „Persistenz“ aufweisen. Als Beispiel sei das Problem von OpenJPA mit der Ladereihenfolge von Klassenpfaden aufgrund eines „Split Package“ im OSGi-Umfeld zu nennen. Dennoch stellte sich keines der Probleme als unüberwindbar hinaus, sodass die erfolg-

reiche Migration als Empfehlung für den Einsatz von OpenJPA gewertet werden kann.

Trotz der speziellen Natur einzelner Probleme muss abschließend betont werden, dass der maßgebliche und grundlegende Erfolgsfaktor für die Migration die umfangreiche Testinfrastruktur für OpenSOA war. Generell gilt, dass das Aufspüren von Problemen aufwändiger ist als deren Lösung. Ohne die vorhandenen Tests wäre eine Verifikation der Migration unmöglich gewesen, d. h. die Lauffähigkeit der OpenJPA-basierten Variante von OpenSOA wäre nicht nachweisbar gewesen. Dies unterstreicht ein weiteres Mal die Wichtigkeit von Softwaretests.

Literatur

[Be05] U. Bettag, Und ewig schläft das Murmeltier: Persistenz mit Hibernate, in: JavaSPEKTRUM, Sonderheft CeBIT, 2005

[BS05] M. Bannert, M. Stephan, JDO mit Kodo – ein Praxisbericht, in: JavaSPEKTRUM, Sonderheft CeBIT, 2005

[De05] F. von Delius, JDO und Hibernate: zwei Java-Persistenztechnologien im Vergleich, in: JavaMAGAZIN, 6/2005

[HB06] U. Hohenstein, J. Bartholdt, Performante Anwendungen mit Hibernate, in: JavaSPEKTRUM, 3/06

[HJ09a] U. Hohenstein, M. C. Jaeger, Die Migration von Hibernate nach OpenJPA: Ein Erfahrungsbericht, in: 13. GI-Fachtagung Datenbanksysteme für Business, Technologie und Web, März 2009, Münster

[HJ09b] U. Hohenstein, M. C. Jaeger, Using Aspect-Orientation in Industrial Projects: Appreciated or Damned?, in: Proceedings of the 8th ACM international conference on Aspect-Oriented Software Development, März 2009, Charlottesville, Virginia, USA

[HODC00] R. Heubner, G. Oancea, R. Donald, J. Coleman, Object Model Mapping and Runtime Engine for Employing Relational Database with Object Oriented Software, United States Patent 6,101,502, Appl No. 09/161,028, August 2000

[KB04] G. King, C. Bauer, Hibernate in Action, Manning, 2004

[La03] R. Laddad, AspectJ in Action: Practical Aspect-Oriented Programming. Manning, Greenwich, 2003

[Ne94] J. Udell, Next's Enterprise Objects Framework, in: Byte Magazine, Juli 1994

[Pl04] M. Plöd, Winterschläfer: Objektrelationales Mapping mit Hibernate, in: JavaMAGAZIN, 8/2004

[PT06] S.E. Pagop, M. Tilly, Caching in Hibernate: Teil 1 und Teil 2, in: JavaSPEKTRUM 3 und 4, 2006

[Str07] W. Strunk, The Symphonia Product-Line, in: Java and Object-Oriented (JAOO) Conference, Arhus, Denmark, 2007

[VS07] D. Vines, K. Sutter, Migrating legacy Hibernate applications to OpenJPA and EJB3.0, `www.ibm.com/developerworks/websphere/techjournal/0708_vines/0708_vines.html`, 2007

Dr. Uwe Hohenstein arbeitet als Senior Research Scientist im Zentralbereich Corporate Technology (CT) der Siemens AG auf dem Gebiet datenbankbasierter Softwarearchitekturen. Er ist Koautor der im dpunkt.Verlag erschienenen Oracle-Bücher „Effiziente Anwendungsentwicklung mit objektrelationalen Konzepten“ und „Webanwendungen entwickeln mit Oracle9i – Java, XML, JDBC & SQLJ, Oracle9i Application Server“.
E-Mail: uwe.hohenstein@siemens.com.

Dr. Michael C. Jaeger arbeitet als Research Scientist im Zentralbereich Corporate Technology (CT) der Siemens AG im Bereich Architekturen verteilter Systeme. Seine Doktorarbeit verfasste er über Algorithmen zur Optimierung von Geschäftsprozessen unter Berücksichtigung nicht-funktionaler Anforderungen. Zuvor erwarb er einen Abschluss im Fach Technische Informatik an der Technischen Universität Berlin.
E-Mail: michael.c.jaeger@siemens.com.