# Die Migration von Hibernate nach OpenJPA: Ein Erfahrungsbericht

Uwe Hohenstein, Michael C. Jäger

Corporate Research and Technologies, Siemens AG
Software & Engineering, Architecture (CT SE2)
Otto-Hahn-Ring 6
D-81730 München
{uwe.hohenstein | michael.c.jaeger}@siemens.com

**Abstract:** Persistenzframeworks sind komplexe Softwarelösungen für die Speicherung und das Auffinden von Objekten in relationalen Datenbanken. Immer wichtiger für die Weiterentwicklung solcher Frameworks werden Erfahrungen, die mit ihrem Einsatz gesammelt werden. Aus diesem Grund soll der vorliegende Beitrag die Architektur eines hochverfügbaren Systems mit firm-realtime Anforderungen vorstellen und Erfahrungen erläutern, die bei der Migration eines zugrunde liegenden Persistenzframeworks gesammelt wurden. Die Migration betrifft eine Telekommunikationsmiddleware, welche zunächst das weit verbreitete objekt/relationale Persistenzframework Hibernate eingesetzt hat. Da Hibernate mit einer Patentklage in den USA konfrontiert wurde, entstand eine negative Auswirkung auf jegliche Produkte, die Hibernate enthalten und im US-Markt ausgeliefert werden. Aus diesem Grund wurde der Austausch von Hibernate durch das Alternativprodukt OpenJPA für die Telekommunikationsmiddleware forciert.
Diese Arbeit stellt die Migration von Hibernate nach OpenJPA vor, nennt die wichtigsten Herausforderungen sowie deren Lösungen und zeigt dabei, dass trotz konzeptioneller Ähnlichkeiten bedeutende Unterschiede zwischen Hibernate und OpenJPA in der Verwendung bestehen, welche teilweise den Erfolg der Migration in Frage stellten.

## 1 Einleitung

Persistenzframeworks vereinfachen den Zugriff aus einer objektorientierten Programmiersprache wie Java auf ein relationales Datenbanksystem (DBS). 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 dieser Objekte speichern kann. 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 objektobjektorientierten API, ohne SQL formulieren zu müssen.

Insbesondere das Java-basierte Persistenzframework Hibernate [Hib] hat in der Vergangenheit in der Java-Szene zunehmend an Beliebtheit gewonnen [Be05]. Der Reiz von Hibernate liegt sicherlich darin, dass es ein frei verfügbares Open-Source-Projekt ist, das

eine Vielzahl von DBSen unterstützt. Neben den etablierten Produkten wie DB2, Oracle, Sybase oder SQL Server kann Hibernate auch mit den üblichen Open-Source-Produkten wie PostgreSQL oder MySQL und auch weniger bekannten Systemen wie Interbase problemlos arbeiten. Ein weiterer Grund für die Beliebtheit sind sicherlich die geringen Performanzeinbussen gegenüber einer direkten Datenbankanbindung mit JDBC. Aus technischer Sicht kommt hinzu, dass Hibernate erlaubt, hochgradig Einfluss auf das Verhalten und die Performanz nehmen zu können. Eine mangelnde Einflussnahme wird häufig beim konkurrierenden Standard Java Data Objects [JDO] und entsprechenden Produkten bemängelt [MR02,BaS05].

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 unter Benutzung einer Datei, in der die Beziehungen zwischen Objekten und Tabelleneinträgen definiert sind (vgl. [HOD+00]). 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 als ein Problem für Red Hat erscheint, entpuppt sich 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.

Somit wurde dies auch zu einem Problem für Siemens Enterprise Communications (SEN), welches Hibernate in deren Middleware namens OpenSOA [Str07] integriert. OpenSOA implementiert eine Plattform für Dienste, mit denen eine Service-orientierte Architektur (SOA) bei der Entwicklung von 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. Um die Bewegungsfreiheit von SEN auf dem amerikanischen Markt zu erhalten bzw. auszubauen und um Produkte dem Kunden weiterhin ohne Risiken anzubieten, entschloss sich SEN, Hibernate durch ein anderes Persistenzframework zu ersetzen. Neben der Patenklage bestand ein weiterer Grund in den kritischen Bedingungen zur verwendeten GNU Lesser General Public License (LGPL).

Aus diesen rechtlichen Rahmenbedingungen entstand nun der Bedarf für die Migration. Die vorliegende Arbeit erläutert dieses Vorhaben und liefert dabei Einsichten über den tatsächlichen Grad der Kapselung einer Datenbank unter Verwendung eines Persistenzframeworks. Der nächste Abschnitt 2 stellt das OpenSOA-Projekt vor und liefert die technischen Hintergründe für die Migration. Abschnitt 3 geht auf die Persistenzarchitektur von OpenSOA ein. Das generelle Vorgehen bei der Migration aus organisatorischer Sicht wird in Abschnitt 4 erläutert, während Abschnitt 5 wichtige Aspekte der Migration aus technischer Sicht vorstellt. Abschnitt 6 präsentiert dann eine Auswahl von speziellen Problemen, die zu Beginn der Migration nicht erwartet wurden; diese Probleme machen deutlich, wie weitreichend der Austausch eines Persistenzframeworks sein kann. Analog dazu erläutert Abschnitt 7 die dazugehörigen Lösungen. Die Arbeit endet mit einer Zusammenfassung der zusammengetragenen Erkenntnisse.

## 2 Die OpenSOA-Software

OpenSOA umfasst eine Dienstplattform, die auf Basis einer OSGi-Laufzeitumgebung [OSGi] implementiert ist. Da die OSGi-Spezifikation nicht alle Funktionen einer Service-orientierten Architektur abdeckt, wurden proprietäre Erweiterungen erstellt oder bereits erhältliche Software hinzugenommen. Besonderes Augenmerk wurde hierbei auf die Anforderungen an Performanz und Skalierbarkeit für Kommunikationsanwendungen gelegt. Hierbei dient ein OGSi-Container als leichtgewichtige Basis. Um darauf basierend beispielsweise Replikationstransparenz in einem Verbund von Computern herzustellen, wurde ein zu OSGi konformer Life Cycle Manager entwickelt, der den Zugriff auf Komponenten über mehrere Container hinweg innerhalb des Verbunds ermöglicht („Distributed OSGi"). Weitere Ergänzungen umfassen beispielsweise eine Infrastruktur für Nachrichten, Fehlermanagement und ein Persistenzsystem. Letzteres wird gesondert im folgenden Kapitel behandelt.

OpenSOA realisiert in einer Service-orientierten Architektur (SOA) Basisdienste, die genereller Bestandteil von Kommunikationsanwendungen sein können. Beispiele hierfür sind eine Benutzerverwaltung, die Erfassung von Präsenszeiten usw. Kommunikationsanwendungen können ein E-Mail Client, eine Web-Anwendung, die dem Benutzer die Verwaltung des eigenen Telefons ermöglicht, oder auch eine Administrationsumgebung für PBX-Vermittlungssysteme (Private Branch Exchange) sein. Die implementierte Architektur gliedert sich in folgende Schichten (Abbildung 1):

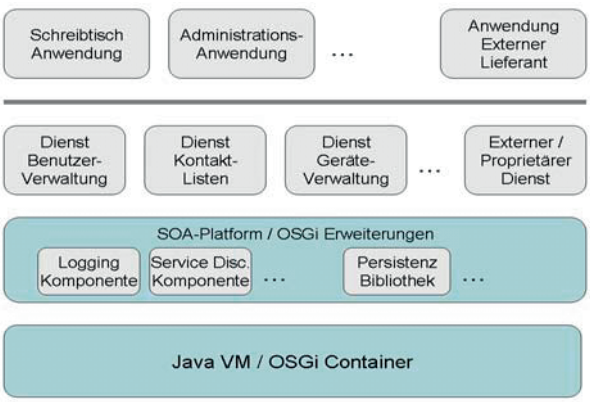


Abbildung 1: Überblick Architektur von OpenSOA

Auf Basis einer Java-Laufzeitumgebung kommt ein OSGi-Container zum Einsatz. Dieser wird durch weitere Software, proprietär oder kommerziell erhältlich, zu einer SOA-Plattform ergänzt. Auf dieser SOA-Plattform werden Dienste bereitgestellt, die von einer oder mehreren Kommunikationsanwendungen genutzt werden.

Für die Entwicklung von Anwendungen bietet OpenSOA den Vorteil der Bereitstellung einer Produkt-Plattform, die die Implementierung von Kommunikationsanwendungen durch Wiederverwendung dieser Basisfunktionalität erleichtert und effizienter gestaltet.

Auf der Seite der Anwendung ist darüber hinaus nicht nur die Wiederverwendung im Rahmen der Entwicklung von eigenen Produkten geplant – auch Anwendungen fremder Hersteller sollen Basisdienste nutzen können, um zusätzlich eigene Dienste auf der Plattform bereitzustellen.

Vom Umfang her ist die Implementierung der SOA, also die Erweiterungen zur derzeit existierenden OSGi-Implementierung, deutlich weniger umfangreich als beispielsweise eine Laufzeitumgebung, welche die JEE-Spezifikation implementiert. Betrachtet man zusätzlich die Basisdienste, die Standardfunktionen für Kommunikationsanwendungen implementieren, ergibt sich von der Anzahl der Java-Klassen her ein ähnlicher Umfang wie bei gängigen JEE-Implementierungen (ca. 6000 bis 7000 Klassen bei OpenSOA, Der JBoss AS 4 von Red Hat umfasst ca. 6300 Klassen).

## 3 Architektur der Persistenzsystems in OpenSOA

Der vorangegangene Überblick über die Architektur von OpenSOA ist nicht auf das Persistenzsystem eingegangen, das in klassischen 3-Schichten Architekturen eine der drei Schichten konstituiert. Für einen Dienst auf OpenSOA ist es nicht zwingend notwendig, Daten persistent zu speichern. Viele Basisdienste nutzen allerdings das Persistenzsystem zum Beispiel für die Verwaltung von Kontaktlisten oder um die Kontakte und Telefonnummern eines Teilnehmers speichern. OpenSOA und dessen Dienste sind in der objektorientierten Programmiersprache Java implementiert und als DBS kommen relationale Systeme zum Einsatz. Wie eingangs erwähnt, wird bei OpenSOA das Persistenzframework Hibernate für die Kopplung verwendet. Zudem wird ein Connection Pool verwendet, da die Verwaltung von parallel laufenden Verbindungen zur Datenbank erwartet wird. Daher kommuniziert das Persistenzframework nicht direkt mit dem Datenbanktreiber, sondern mit einem Connection Pool. In einer OpenSOA-Installation werden ca. 200-300 gleichzeitige Verbindungen verwaltet.

Ein Persistenzframework deckt viele Aspekte ab und mittlerweile kann dessen Implementierung als intuitiv komplex erachtet werden. Dementsprechend hat der Entwickler viele Möglichkeiten einen bestimmten Sachverhalt zu realisieren. Darüber hinaus bietet es viele Konfigurationsmöglichkeiten, um auf spezielle Anforderungen für Performanz oder Integrität der Daten Einfluss zu nehmen. Um diese Punkte für die Entwicklung von Diensten auf OpenSOA zu standardisieren, wurde eine Bibliothek implementiert, die im Wesentlichen zwei Bereiche abdeckt:

1. Eine Art Programmier-Template erledigt das Anfordern einer Datenbankverbindung, die Aufrufe für das Öffnen und Schließen einer Transaktion, das Verhalten bei Ausnahmefällen während der Anfragebearbeitung oder eine Wiederholung der Anfrage bei Verbindungsproblemen.
2. Des Weiteren implementiert die Bibliothek ein zentralisiertes Vorgehen zur Konfiguration des Persistenzframeworks. Auf diese Weise kann die Installation der Software leichter automatisiert werden.

Insgesamt besteht das gesamte Persistenzsystem, auf das der Entwickler eines OpenSOA Dienstes zugreift, aus mehreren Schichten, die sich wie folgt anordnen:

1. Eine **Template-Bibliothek** für standardisierte Nutzung der Abbildungssoftware und einheitliche Konfiguration: Diese Bibliothek fasst wiederkehrende Aufgaben wie Session- und Transaktions- wie auch Fehlerbehandlung zusammen uns ist ein proprietärer Teil von OpenSOA ist somit eine Art API für den Entwickler.
2. Die **Hibernate Software**, ein Persistenzframework, welches die Abbildungsfunktion implementiert: Dieses Softwareprodukt wurde durch die Template-Bibliothek gekapselt. Zur Programmierung der Persistenzlogik wird Hibernate innerhalb der Templateaufrufe benutzt, also um Daten zu speichern oder abzurufen.
3. Ein **Connection Pool**, der die Verbindungen verwaltet, die durch den JDBC Treiber zur Datenbank aufgebaut werden, wird durch Hibernate (bzw. OpenJPA) genutzt. Grundsätzlich hat der Entwickler keinen Kontakt zu dieser Schicht.
4. Ein **JDBC-Treiber**, der die Kommunikation mit der Datenbank über eine Java-API ermöglicht. Auch wenn ein Entwickler primär mit der Template-Bibliothek und Hibernate arbeitet, kann ein direkter Zugriff auf JDBC für spezielle Aufgaben, die Hibernate nicht anbietet wie z.B. Anfragen mit Optimizer Hints, nötig sein.
5. Das **Datenbanksystem:** Als DBSe werden MySQL, PostgreSQL und SOLID unterstützt. SOLID besitzt hierbei die besondere Rolle, dass es eine Hot Standby Installation unterstützt, die bereits von Siemens PBX-Lösungen genutzt wird. In OpenSOA wie auch in anderen Java Anwendungen erfolgt die Interaktion mit dem DBS grundsätzlich über den JDBC Treiber.

Abbildung 2 fasst die Bestandteile der Persistenz-Architektur graphisch zusammen. Die grauen Pfeile in der Abbildung deuten an, über welche Schichten der Architektur ein Dienst auf andere Teile der SOA-Infrastruktur in OpenSOA zugreift. Diese Architektur ist damit auch Ausgangspunkt der Ersetzung von Hibernate durch OpenJPA.

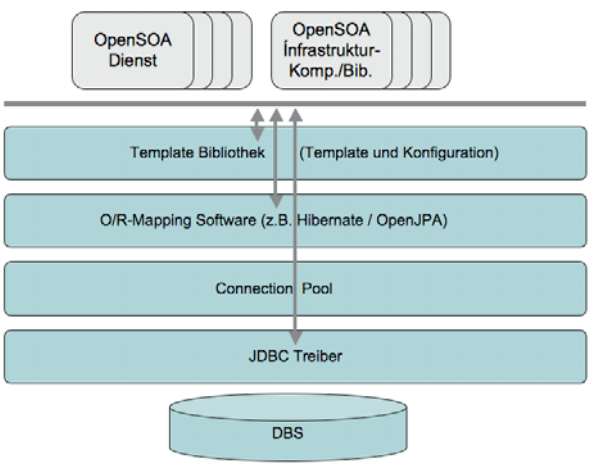


Abbildung 2: Architektur des Persistenzsystems in OpenSOA

## 4 Allgemeine Ersetzungsstrategie

Nachdem die Migration beschlossen wurde, erfolgte das Vorgehen nach einem iterativen, heuristischen Phasenmodell, das sich in folgende sechs Aktivitäten gliedert:

**1. Projektdefinition:** Im Rahmen der Projektdefinition wurde zügig ein geeigneter Ersatz für Hibernate ermittelt. Grundsätzlich existiert eine große Auswahl verschiedener Frameworks für die O/R-Abbildung, die auch für den Einsatz in OpenSOA geeignet wären. Die Überlegungen, die zu einer schnellen Entscheidung führten, sind im folgenden Abschnitt dargelegt. Zusammenfassend ist die Wahl auf OpenJPA gefallen, welches aus dem Produkt Kodo von BEA Systems hervorgegangen ist.

**2. Grobplanung:** Nachdem der Entscheidung für ein Produkt, wurden im Detail die kritischen Punkte zusammengestellt, die durch ein Proof-of-Concept zu verifizieren sind. Es musste auch bedacht werden, dass JPA-Implementierungen fehlerhaft oder unzureichend dokumentiert sind, da der Standard JPA wie auch das Produkt OpenJPA in der Version 0.9.7 zum damaligen Zeitpunkt relativ neu waren. Dadurch gliederte sich die Grobplanung in zwei Teile: Zum einen wurde auf Basis der Produktauswahl das generelle Vorgehen definiert. Zum anderen mussten die kritischen Punkte systematisch zusammengetragen werden, um diese möglichst frühzeitig ausräumen zu können bzw. im negativen Fall den Schwenk auf eine andere Abbildungssoftware zu ermöglichen.

**3. Testmigration:** Anschließend wurde die zentrale Template-Bibliothek in Verbindung mit dem umfangreichsten OpenSOA-Dienst umgesetzt. Die Testmigration diente zur Überprüfung der zuvor als kritisch ermittelten Punkte und ging auch darüber hinaus: 1) Um die Umstellung auf OpenSOA in der Vorbereitung möglichst umfangreich abzusichern, sollte OpenJPA über die kritischen Punkte hinaus evaluiert werden. Dies wurde mit der Testmigration erfüllt: Es stellten sich neue Probleme heraus, die bei der Planung nicht bedacht wurden. 2) Ein weiterer Vorteil lag darin, dass ein Überblick darüber gewonnen werden konnte, welche Arbeiten im Detail zu bewerkstelligen sind. Dadurch wurde die Feinplanung der Umstellung präziser und die Automatisierung möglicher Schritte konnte evaluiert werden. 3) Schließlich wurde mit der Umstellung der Template-Bibliothek ein wichtiger Teil der Migration bereits geleistet, da im günstigen Fall die einzelnen Dienste und Systembestandteile das Persistenzsystem mittels der Template-Bibliothek nutzen.

**4. Feinplanung:** Auf Basis der vorangegangen Schritte wurde die Feinplanung der Migration durchgeführt. Dieser Schritt kann als heuristische Iteration zum zweiten Schritt gesehen werden. Wie eingangs erwähnt, stellt OpenSOA eine umfangreiche Software dar, mit der Folge, dass die einzelnen Bestandteile nicht zu einem Stichtag umgestellt werden konnten. Parallel dazu vollzog sich die Weiterentwicklung von OpenSOA, so dass bei einer Umstellung von ein bis zwei Wochen pro Subsystem die Arbeit der Entwickler hätte für Wochen unterbrochen werden müssen, es sei denn, man würde massiv auseinander gelaufene Versionen der Quelltexte später manuell zusammen führen, was sehr aufwändig wäre. Daher musste ein Plan erstellt werden, nach dem die Dienste schrittweise umzustellen waren. Dieser Plan resultierte aus den Abhängigkeiten der einzelnen Subsysteme untereinander. Darüber hinaus musste das Vorgehen mit den betroffenen Entwicklern kommuniziert werden.

**5. Migration:** Anschließend erfolgte die Migration schrittweise. Um die „normale“

Entwicklungsarbeit nicht zu unterbrechen, konnten nicht alle Komponenten zu einem singulären Termin umgestellt werden. Somit wurde in Kauf genommen, dass Entwicklungszwischenstände von OpenSOA eine gemischte Umgebung, teils auf OpenJPA, teils auf Hibernate basierend, nutzten. In dieser Zeit liefen Dienste in der OGSi-Laufzeitumgebung parallel jeweils mit Hibernate und OpenJPA.

**6. Verifikation:** Zur Verifikation dienten die verschieden gestuften Softwaretests im Entwicklungsprojekt von OpenSOA. Dies sind Unit-Tests auf Ebene von Klassen, eine sogenannte Functional Test Suite, die bestimmte Use Cases implementiert, sowie Integrationstests, bei denen OpenSOA mit den darauf entwickelten Anwendungen kombiniert und getestet wird. Darüber hinaus existierten Performanztests, die unterschiedliche Anforderung an die zu verarbeitende Last simulieren, und Feldtests, in denen eine OpenSOA-basierte Anwendung im Hause für die alltägliche Kommunikation eingesetzt wird. Die Verifikation des Funktionsumfangs und damit der erfolgreiche Abschluss der Migration scheinen zunächst trivial: Die funktionalen Tests müssen bei einer OpenJPA-basierten Version ebenso fehlerfrei abschneiden wie bei der Hibernate-Version. Die Herausforderung der Verifikation der Migration bestand jedoch darin, dass sich OpenSOA noch in der Weiterentwicklung befand. An dieser Stelle wird der heuristisch, iterative Charakter des Migrationsvorhabens deutlich.

## 5 Technische Aspekte der Migration

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 viele Produkte einem gängigen Standard folgen und somit einen erneuten Wechsel des Produkts ermöglichen würden. Sollte ein Produkt von der Patentklage eingeholt werden, besteht die Möglichkeit, mit moderatem Aufwand auf ein Alternativprodukt auszuweichen. Standardkonformen Produkten ist 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 in Frage. Dabei ist JPA im Vergleich zum JDO-Standard aktueller [MR02] und derzeit auch aus folgenden Gründen die bessere Wahl:

- JPA ist ein Teil des Standards für EJB3-Applikationsserver, er erlaubt also eine leichtere Migration von Stand-alone-Applikationen in EJB3-Applikationen und zurück. JPA-konforme Produkte sind benutzbar unter Java SE und Java EE.
- JPA beinhaltet nach eigenen Aussagen das „Beste“ aus Hibernate, TopLink, JDO und EJB2.1-Container-Managed Persistence. Gerade am JDO-Standard und an JDO-Produkten ist häufig Kritik geübt wurden, da Ersterer zu wenig mächtig ist und Zweitere viele (notwendige) nicht-standardkonforme Erweiterungen aufweisen.
- Für eine Ersetzung von Hibernate ist sicherlich relevant, dass JPA eine gewisse Hibernate-Nähe aufweist, insbesondere bei der Anfragesprache JPQL.

Im Projekt fiel die Wahl für einen Hibernate-Ersatz somit auf ein JPA-konformes Pro-

dukt, im genauen auf OpenJPA, welches als Projekt der Apache Software Foundation entwickelt wird. Es wird als Open Source Software entwickelt und vertrieben und basiert auf dem Produkt Kodo von BEA Systems. Im Jahr 2006 hat sich BEA entschlossen, den Grossteil des Produkts Kodo als Apache Software Projekt 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 Software Lizenz vertrieben wird. Diese bietet Vorteile gegenüber der LGPL von Hibernate, wenn es um den Umgang mit Erweiterungen geht. Zum anderen ist die Standardkompatibilität bedeutend, auch wenn Hibernate einen JPA-kompatiblen Aufsatz bietet, aber ansonsten proprietäre APIs bietet. Diese potenzielle Austauschbarkeit ist für industrielle Anwendungen ein wichtiges Argument.

### 5.1 Hibernate Grundlagen

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 etc. 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 ausschließlich mit XML-Spezifikationen verwendet. Die Mapping-Dateien `Customer.hbm.xml` und `Order.hbm.xml` in Abbildung 3 beschreiben die Abbildung der Java-Klassen aus Abbildung 4.

```
<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>
```

```
<class name="Order table="O">
  <id name="oid">
  ...
</class>
```

Abbildung 3: Beispiel einer hbm-Mapping-Datei

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

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

Abbildung 4: Beispiel für dazu bezügliche Java-Klassen

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 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(oi,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"/>`

vereinbart. Hierdurch erhält Tabelle `O` eine Spalte `custId` zur Repräsentation der Beziehung, die als Fremdschlüssel auf den Primärschlüssel `cid` von `C` verweist. Hibernate ermöglicht dabei eine umfangreiche Steuerung der Abbildung von Klassen und Beziehungen auf Tabellen, wie [Pl04] systematisch aufzeigt. Das folgende Quelltextbeispiel zeigt, wie mit der Hibernate API programmiert wird.

```
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();
```

Nach dem Öffnen einer Session, d.i. im Prinzip eine Verbindung zum DBS, 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 [HoB06]. 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 die Behandlung von Ausnahmefällen wurde hier verzichtet.

### 5.2 Das Grundprinzip der Migration

Die Handhabung ist in OpenJPA recht ähnlich:

```
EntityManagerFactory emf =Persistence.createEntityManagerFactory("DB1");
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();
```

Die Programmierung erfolgt mit 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. Daher ist eine direkte Umsetzung der Hibernate-Schnittstelle einfach möglich.

Bei einer Umstellung von Hibernate nach OpenJPA können die zu persistierenden Java-Klassen unverändert bleiben. Allerdings müssen die Mapping-Dateien umgestellt werden, da die XML-Formate eine unterschiedliche Syntax aufweisen. Eine XSLT-Transformation zur Überführung ist allerdings recht einfach zu erstellen. Kleinere konzeptionelle Unterschiede bestehen. Z.B. müssen transiente Attribute einer Klasse explizit als `<transient …/>` in der Mapping-Datei gekennzeichnet werden.

Im Grunde gibt es zwischen Hibernate und OpenJPA sehr viele Gemeinsamkeiten in grundlegenden Konzepten mit geringen syntaktischen Unterschieden. Das trifft auch für einige Besonderheiten wie Notifizierungen von Datenbankereignissen oder ein Konzept zu, dass die Austauschbarkeit von DBSen ermöglicht, aber dennoch proprietäre Konzepte eines einzelnen DBS effizient nutzt. In Hibernate kümmert sich ein spezielles Dialect-Konzept um die bestmögliche Umsetzung der Hibernate-Features auf adäquate DBS-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 DBSe, wobei beide SOLID nicht unterstützen.

Auch wenn viele Unterschiede durch einen Wrapper-Ansatz abgefangen werden können, treten doch ein paar Probleme auf, die eine weiter gehende Behandlung erfordern.

## 6 Probleme bei der Migration

Im Folgenden richtet sich der Fokus auf diverse Probleme, die Verzögerung bei der Umstellung von Hibernate auf OpenJPA bereiteten, wobei die Ergebnisse von [ViS07] hier ausgelassen werden. Die Erkenntnisse gelten im Prinzip auch für den JPA-Standard.

### 6.1 Unterschiede bei der Konfiguration

In Abschnitt 5.1 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 von `buildSessionFactory()` gelesen und verarbeitet wird. Diese Konfigurationsdatei enthält typische Datenbankparameter wie die Datenbank-URL, den Treiber, Benutzer, Passwort und verweist mit `<mapping>` auf die persistenten Klassen. Sie kann über einen expliziten Pfad referenziert werden.

```
<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>
<property name="c3p0.max_size"> 50 </property> <!—c3p0 Connection Pool -->
<mapping resource="Customer.hbm.xml"/>         <!—Mapping-Dateien -->
```

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 etc.) vereinbaren. Hier steht auch ein Verweis auf die Mapping-Datei orm.xml bzw. die annotierten Klassen.

```
<persistence-unit name="DB1" 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>
```

OpenJPA erwartet die Dateien persistence.xml und orm.xml in einem META-INF-Verzeichnis innerhalb des classpath. Das ist nicht immer vorteilhaft. Da im Projekt OpenJPA in einem OSGi-Container benutzt wird, wird die Datenbank-Konfiguration Bestandteil der Deployment-JAR-Datei. Somit wird das projektinterne Deployment-Prinzip verletzt, dass das DBS, 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, was auf dem Deployment-Rechner nicht möglich ist. Somit musste nach Möglichkeiten gesucht werden, DBS-spezifische Daten außerhalb des Deployments zu verwahren, aber dennoch die OpenJPA-Initialisierung zu gewährleisten.

### 6.2 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 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 Abbau zeitaufwändig ist, verwaltet Hibernate einen Connection Pool, 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 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 ist die Property `openjpa.ConnectionDriverName` statt `value="com.mysql.jdbc.Driver"` auf `value="org.apache.commons.dbcp.BasicDataSource"` 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"/>
```

Dieser Unterschied ist zunächst einmal nicht dramatisch, wirkt sich aber später noch beim Failover (siehe Abschnitt 6.7) negativ aus. Problematisch sind auch die Unterschiede bei den Konfigurationsparametern des Pools wie Initialgröße, minimale und maximale Größe und deren Bedeutung. Diese Parameter haben unterschiedliche Wirkung in Hibernate und OpenJPA. Der erste Versuch, DBCP 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.

### 6.3 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 Objekt erzeugt wird, das zunächst temporär ist, 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 mit `update()`. Hier behandelt OpenJPA das Objekt ebenfalls als neu, so dass sich das DBS mit einer Eindeutigkeitsverletzung beklagt.

### 6.4 Unterschiede bei Anfragen

Obwohl die Hibernate Query Language HQL und OpenJPA's 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. Ein Vergleich mit `!=` ist in OpenJPA als `<>` zu notieren. Statt `SELECT COUNT(*) FROM Customer` ist 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'` heißt es `SELECT c FROM Customer c WHERE c.name='Ms.Marple'`. Bei der Migration trat dabei das Problem auf, dass im GUI Bedingungen der Form `Feld=Wert` zusammengestellt werden, die direkt als `Feld=Wert` 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. Das Lazy/Eager Fetching und die daraus resultierende Performanzproblematik ist ohnehin ein komplexes Thema.

Performanzkritisch ist auch, dass `DELETE FROM Customer c WHERE c.name='Ms.Marple'` in OpenJPA nicht funktioniert, weder mit noch ohne Variable `c`. OpenJPA produziert folgende Anfrage mit Selbstbezug, die in den meisten DBSen verboten ist:

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

### 6.5 Löschkaskadierung

Hibernate bietet eine flexible Steuerung, Datenbankoperationen wie das Löschen oder Speichern über Beziehungen zu kaskadieren. Eine `cascade`-Option ist für alle Beziehungsarten anwendbar. Zu kaskadierende Operationen sind `save-update` (Einfügen und Speichern), `delete` (Löschen), `all` (`save-update` und `delete`) sowie spezielle Formen

`delete-orphan` und `all-delete-orphan`: Während mit `cascade="delete"` für die `Customer-Order`-Beziehung vereinbart wird, dass mit einem `Customer`-Objekt 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`-Objekt 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, mit Ausnahme von `delete-orphan`. Insofern muss die Migration einen adäquaten Ersatz bereitstellen.

### 6.6 Schlüssel-Generierung und Objektidentität

Jede persistente Java-Klasse erfordert einen Identifikator, der in Hibernate-Mappings mit `<id>` ausgezeichnet ist (vgl. Abb. 3) und Objekte der Klasse eindeutig identifiziert. Er wird auch als Primärschlüssel der zugrunde liegenden Tabelle benutzt. Hibernate unterstützt mehrere Strategien, die über `<generator>` ausgewählt werden. Beispielsweise kann Hibernate typische Mechanismen der DBSe wie Sequenzgeneratoren (z.B. SOLID) oder Autoinkrement-Spalten (z.B. MySQL) nutzen, um Schlüsselwerte zu belegen, die dann als Strategien `sequence` bzw. `identity` wählbar sind. Da die OpenSOA-Applikationen mehrere DBSe unterstützen müssen, insbesondere MySQL, SOLID und PostgreSQL, ist ein abstrakter, vom DBS unabhängiger Mechanismus nötig. Schließlich ist das Ziel eines O/R-Frameworks die Unabhängigkeit von DBSen, was eigentlich auch für Mapping-Dateien gelten sollte. Hibernate verfügt zu diesem Zweck über eine `native`-Strategie, die, je nachdem, was das zugrundeliegende DBS 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 in OpenSOA natürlich zu Wertekonflikten bei bestehenden Datenbanken, da bereits durch Sequenzen oder Autoinkrement-Spalten vergebene Werte höchstwahrscheinlich nochmals mit Hi/Lo erzeugt werden. Im Prinzip kann man DBS-spezifische Mapping-Dateien pflegen, die je nach gewähltem DBS direkt `sequence` oder `identity` festlegen. Ein modellgetriebener Ansatz kann hier helfen, eine entsprechende Mapping-Datei zu erzeugen. Da die Mapping-Datei, wie bereits in Abschnitt 6.1 angesprochen, Bestandteil des Deployment-JARs sein muss, steht das wiederum im Konflikt zur Deployment-Strategie, die fordert, dass es ein DBS-unabhängiges Deployment gibt, so dass beim Kunden das DBS eingestellt werden kann. Eine Änderung der Deployment- und Installationsprozedur wäre wiederum sehr aufwändig. Das Problem wird noch dadurch verstärkt, dass sich einige OpenJPA-Konzepte nur als Annotation vorliegen, wie z.B. `delete-orphan`. Einerseits ist es mühselig, `delete-orphan` händisch zu implementieren, insbesondere wenn im Objektmodell Kaskaden über mehrere Stufen gehen. Nutzt man andererseits die `delete-orphan`-Option mit einer Annotation, so sind mehrere Quellcode-Varianten vorzuhalten, da die Mapping-Annotationen Bestandteil des Quellcodes sind. Die fehlende Unterstützung von `native` ist für die Migration daher schwerwiegend.

Des Weiteren bietet der JPA-Standard im Gegensatz zu Hibernate keine UUID-Generierung (weltweit global eindeutiger Identifier) und keine `increment`-Strategie (inkrementiere den höchsten Schlüsselwert der Tabelle). Auch hierfür sind Lösungen nötig.

### 6.7 Failover

Eines der DBSe, dass von OpenSOA zu unterstützen ist, ist SOLID. SOLID ist zwar weniger bekannt, aber dennoch im Bereich der Telekommunikation häufig vertreten, weil es ein interessantes Failover-Konzept bietet. So lassen sich zwei SOLID-Server installieren, ein Primärer und ein Sekundärer, deren Datenbanken sich synchronisieren. Stürzt der Primärserver ab, wird der Sekundärserver sofort zum Primärserver, der den Betrieb übernimmt. Damit Applikationen unabhängig davon sind, wer gerade Primärserver ist, müssen sie eine spezielle *dual-node* URL der Form `jdbc:solid://node1:1315,node2:1315/myusr/mypw` verwenden. Diese spezifiziert beide Server, auf node1 und node2. Dieses Failover-Konzept ist für das Projekt sehr bedeutend.

Da Hibernate keine Probleme mit der speziellen URL hatte, konnte man davon ausgehen, dass auch OpenJPA diese URL einfach zum JDBC-Treiber durchreicht. Bei der Testdurchführung stellte sich jedoch heraus, dass das Failover nicht funktionierte, es konnten überhaupt keine Verbindungen zur Datenbank aufgebaut werden. Tiefergehende Recherchen ergaben, dass die dual-node URL von OpenJPA verstümmelt wurde: Nur der erste Teil `jdbc:solid://comp1:1315` erreichte den SOLID-Server. Der Grund lag darin, dass OpenJPA alle Verbindungseigenschaften als `openjpa.ConnectionProperties` subsumiert, die URL, den Namen der Treiberklasse etc.:

```
Properties prop = new Properties();
String jpa = "Url=jdbc:solid://comp1:1315,comp2:1315/myusr/mypw,
              DriverClassName=solid.jdbc.SolidDriver, ...";
prop.setProperty("openjpa.ConnectionProperties", jpa);
EntityManagerFactory emf = provider.createEntityManagerFactory("db",prop);
```

Während der Analyse von `openjpa.ConnectionProperties` nimmt OpenJPA das Komma als Separierungssymbol und leitet somit die folgenden Einheiten ab:

```
Url=jdbc:solid://comp1:1315
comp2:1315/usr/pw
DriverClassName=solid.jdbc.SolidDriver
```

Da die zweite „dieser Einheiten“ nicht der Form *`property=value`* genügt, ignoriert OpenJPA sie schlichtweg; die URL wird zu `jdbc:solid://comp1:1315` gekürzt. Zur Lösung des Problems ist offensichtlich das Verhalten von OpenJPA so zu ändern, dass die URL intakt bleibt, was im Prinzip auf eine Änderung des Quellcodes hinausläuft.

### 6.8 Performanz

Das Lazy/Eager-Fetching-Prinzip ist für die Performanz von O/R-Applikationen sehr bedeutsam [BaH07]. Hibernate und OpenJPA unterscheiden sich hierbei konzeptionell als auch syntaktisch sehr stark. Generell sind die Möglichkeiten des JPA-Standards wesentlich geringer, was OpenJPA teilweise 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. Im Zusammenspiel von OpenJPA und OSGi trat dabei das Problem auf, dass aufgrund eines Class Loading Problems Standard-Java-Klassen nicht gefunden wurden. Der Query Compilation Cache musste abgeschaltet werden.

## 7 Lösung der Probleme

Die zentrale Template-Bibliothek kann einen Teil der Migration abnehmen. Z.B. sind die in Abschnitt 6.1 erwähnten Konfigurationsprobleme dort zentral lösbar. Das Problem wird derart behoben, dass die Datenbankverbindungsdaten in eine externe properties-Datei ausgelagert und vom Template verarbeitet werden. Die Konfiguration ist somit nicht mehr Bestandteil des Deployments. Auch die Werte für den Connection Pool sind in der properties-Datei enthalten. Das Problem 6.2, geeignete Belegungen zu finden, lässt sich anhand der Dokumentation nicht lösen. Hier hilft nur das Ausprobieren und teilweise eine Analyse des Quellcodes, um die Wirkung der Parameter zu verstehen.

Ein zweiter zentraler Baustein bei der Migration war das Wrapping. Um die Umstellung von Hibernate auf OpenJPA auf möglichst wenige Quellcode-Änderungen zu reduzieren, wurde das Hibernate-API in einem eigenen Java-Package beibehalten, allerdings auf der Grundlage von OpenJPA implementiert. Um Hibernate auszutauschen, war lediglich der Import auf das neue Package zu legen.

Für alle Probleme, die dennoch zur Übersetzungszeit oder Laufzeit auftraten, mussten adäquate Lösungen bereitgestellt werden, die sich wie folgt kategorisieren lassen:

1. Benutzung nicht-standardkonformer OpenJPA-Erweiterungen: OpenJPA bietet einige proprietäre Erweiterungen an, die Hibernate-Funktionalität nachbilden. Obwohl diese die prinzipielle Austauschbarkeit von OpenJPA gefährden, wurde von ihnen Gebrauch gemacht, um Zeit bei der Migration zu sparen.
2. Änderung der Dictionary-Klasse: Wie auch Hibernate konzentriert OpenJPA die Abhängigkeit von DBSen in Dictionary-Klassen, die sich um die effiziente Umsetzung auf proprietäre Datenbankkonzepte kümmern.
3. Anwendungsprogrammierung: Obwohl der Wrapping-Ansatz syntaktische Unterschiede weitestgehend beheben konnte, musste in die Programmierung eingegriffen werden, um fehlende Hibernate-Funktionalität nachzuprogrammieren.
4. Nutzung der Aspekt-Orientierung: Für einige schwerwiegende Probleme wie das Failover reichten die vorangegangenen Mittel nicht aus. Mit der aspektorientierten Sprache AspectJ stand ein mächtiger Mechanismus bereit, diese Probleme schnell und elegant zu lösen.

### 7.1 Nicht-standardkonforme OpenJPA-Erweiterungen

Wie in Abschnitt 6.3 erwähnt, sieht der JPA-Standard keine UUID-Schlüsselgenerierung vor, OpenJPA bietet dennoch einen proprietären Mechanismus als Erweiterung der AUTO-Strategie: `<generated-value strategy="AUTO" generator="uuid-hex"/>`. Leider funktioniert das in der Version 0.9.7 von OpenJPA, mit der die Migration begann, nicht mit XML-Mapping-Dateien. Man kann aber auf folgende Annotation ausweichen:

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

Ebenso stellt OpenJPA einen nicht-JPA-konformen Extra-Mechanismus bereit, der eine `delete-orphan`-Kaskadierung über eine Annotation `@ElementDependant` (aber wieder ohne XML-Äquivalent) ermöglicht. Diese wurde benutzt, um aufwändige, händische Programmierung zu vermeiden.

### 7.2 Änderung der Dictionary-Klasse

Ein Persistenzframework sollte die Austauschbarkeit von DBSen ermöglichen, aber dennoch die Möglichkeit haben, proprietäre Konzepte effizient zu nutzen. Diesem Anspruch wird in OpenJPA durch ein spezielles Dictionary -Konzept Rechnung getragen, dass sich um die bestmögliche Umsetzung von OpenJPA auf adäquate DBS-spezifische Konzepte kümmert. Das performanzkritische DELETE-by-Query-Problem (vgl. 6.4) lässt sich über eine geänderte Dictionary-Klasse lösen, indem das DBS-spezifische-Dictionary so abgeändert wird, dass eine relationale SQL-Anweisung ohne Selbstbezug generiert wird:

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

### 7.3 Anwendungsprogrammierung

Teilweise musste in die Anwendungsprogrammierung eingegriffen werden, um semantische Unterschiede auszugleichen oder fehlende Hibernate-Funktionalität auszuprogrammieren. Ein Beispiel hierfür ist die fehlende UUID-Schlüsselgenerierung, die alternativ zur `@GeneratedValue`-Annotation (siehe 7.1) auch durch explizite Initialisierung mit einem UUID-Generator realisiert werden kann. Das Fehlen des `increment`-Mechanismus' ist ebenfalls nicht so dramatisch, da sich dahinter nur eine `SELECT MAX`-Anfrage verbirgt. Syntaktische Unterschiede in den Anfragesprachen und das Beheben von Performanzproblemen wurden ebenfalls in der Programmierung behoben.

### 7.4 Aspekt-Orientierte Programmierung

#### Simulation der `native`-Schlüsselgenerierung

Wie bereits erwähnt, ist es aus Gründen des projektinternen Deployments nicht möglich, verschiedene XML-Mapping-Dateien, eins für jedes DBS mit der jeweiligen Strategie `sequence` oder `identity`, bzw. bei Benutzung von Annotationen sogar mehrere Varianten von Java-Klassen vorzuhalten.

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 DBS 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 der Einsatz von AspectJ [La03] beschlossen.

Der folgende Aspekt modifiziert das OpenJPA-Verhalten entsprechend.

```
@Aspect
```

```
public class KeyGenerationAspect {
  private String db = null;
  private static final int JPA_STRATEGY_SEQUENCE = 2;
  private static final int JPA_STRATEGY_IDENTITY = 3;
  @Before("execution(*
           org..PersistenceProviderImpl.createEntityManagerFactory(..))
       && args(.., p)")
  public void determineDBS(final java.util.Properties p) {
    String str = p.getProperty("openjpa.ConnectionProperties");
    if (str.contains("Solid"))           db = "SOLID";
    else if (str.contains("mysql"))      db = "MYSQL";
    else if (str.contains("postgresql")) db = "POSTGRES";
  }
  @Around("call(*
           org.apache.openjpa.meta.FieldMetaData.getValueStrategy(..))
       && !within(com.siemens.ct.aspects.*)")
  public Object useAppropriateStrategy(final JoinPoint jp) {
    FieldMetaData fmd = (FieldMetaData) jp.getTarget();
    int strat = fmd.getValueStrategy();
    if (db.equals("SOLID") && strat==OPENJPA_STRATEGY_IDENTITY){
      fmd.setValueSequenceName("system");
      return JPA_STRATEGY_SEQUENCE;
    } ... // analog für "MYSQL"
    return strat;
} }
```

`KeyGenerationAspect` ist eine normale Java-Klasse, die mit `@Aspect` zu einem AspectJ-Aspekt wird. Trotz der Nutzung von AspectJ konnte die Entwicklungsumgebung mit Eclipse weiterhin mit einem gewöhnlichen Java-Compiler – also ohne AspectJ-Plugin AJDT – betrieben werden. Dieser Punkt war für das Projekt sehr bedeutsam.

Der Aspekt implementiert zwei Methoden: Die Erste `determineDBS` bestimmt das DBS und die Zweite `useAppropriateStrategy` ändert die Strategie, sofern nötig. Beide Methoden tauschen den DBS-Typ mittels einer lokalen Variablen `db` aus. Dadurch, dass die Methode `determineDBS` mit `@Before` annotiert ist, wird sie zu einem Before-Advice, deren Code *vor* sogenannten Joinpoints ausgeführt wird. Die Joinpoints legen fest, wo der Aspekt eingreifen soll. Diese Stellen sind als Zeichenkette in der Annotation spezifiziert: Jede Ausführung einer Methode `PersistenceProviderImpl.CreateEntityManagerFactory` mit einem `Properties`-Parameter wird abgefangen. Obwohl die Parameterliste "`(..)`" Methoden mit beliebigen Parametertypen fängt, schränkt `args(..,p)` implizit die Methoden auf solche ein, die einen `Properties`-Parameter haben, und bindet gleichzeitig eine Variable `p` an diesen Parameter. Die Variable erscheint auch in der Methodensignatur, wodurch der Zugriff auf Parameterwerte innerhalb des Methodenrumpfs möglich wird: Mit `p.getProperty("openjpa.ConnectionProperties")` kann die kommaseparierte `ConnectionProperties`-Liste erfragt und der Typ des DBSs ermittelt werden. Das Ergebnis wird in einer Aspekt-internen Variablen `db` abgelegt.

Die `useAppropriateStrategy` Methode benutzt das Wissen über den DBS-Typ und wechselt von der `identity` zur `sequence` Strategie im Falle SOLID. Der Aspekt kann somit als einfacher Mechanismus genutzt werden, Daten unter Advices auszutauschen, selbst wenn die abgefangenen Methoden bislang keinerlei Bezug aufweisen, z.B. weil sie in verschiedenen JARs liegen. Der `@Around` Advice `useAppropriateStrategy` ersetzt die ursprüngliche OpenJPA-Logik von `getValueStrategy` und entscheidet anhand des

DBS-Typs und des Ergebnisses von `fmd.getValueStrategy()`, die Strategie zu ändern. Diese Logik wird an allen aufrufenden Stellen aktiv. Der Parameter `jp` liefert Kontextinformation über den Joinpoint, insbesondere das Objekt `jp.getTarget()`, dessen Methode aufgerufen wird. In diesem Fall muss es ein `FieldMetaData`-Objekt `fmd` sein. Zu beachten ist, dass die Klausel `!within(...)` Rekursionen vermeidet: Sie verhindert, dass Aufrufe von `getValueStrategy` innerhalb des Aspekts selbst abgefangen werden.

In der Tat ist das eine Art von Quellcode-"Patching": Das Verhalten von 3rd-Party-Software, hier OpenJPA, wird modifiziert, allerdings ohne direkten Einfluss auf den Quellcode zu nehmen. Zu bemerken ist, dass hierfür der Quellcode nicht vorliegen muss. Der definierte Aspekt wirkt auf JAR-Dateien, in diesem Fall entsteht also eine neue openjar.jar Datei, welche die Aspektlogik enthält. Für den build-Prozess bedeutet das, dass nur ein zusätzlicher Schritt zur Erzeugung der JAR-Datei(en) nötig ist.

Dieses Vorgehen ist wesentlich einfacher als in OpenJPA eine neue `native` Strategie einzuführen. Hierdurch wären mehrere Bestandteile von OpenJPA betroffen wie die XML-Parser für Mapping-Dateien, die Analyse von Annotationen, die interne Verwaltung der Strategien als Metadaten und das Erzeugen der SQL-Operationén.

**Behebung des Failover-Problems**

Ein weiterer Aspekt löst das sehr kritische Failover-Problem elegant und schnell. Ein `@Around`-Advice fängt die Ausführung der „fehlerhaften" `parseProperties`-Methode ab und ersetzt sie durch eine korrigierende Logik:

```
@Around("execution(public static Options
       org.apache.openjpa.lib.conf.Configurations.parseProperties(String))
       && args(props)")
public Object parseProperties(final String s){
  Options opts;
  Analysiere den properties-String s korrekt und baue die URL
  korrekt zusammen;
  return opts;
}
```

Leider löst der Aspekt nur einen Teil des Failover-Problems: Von nun an wird zwar die URL korrekt zum JDBC-Treiber durchgelassen, aber der Failover vom Primär- zum Sekundärserver findet nicht statt. Der Grund hierfür liegt darin, dass OpenJPA die wichtige Verbindungsoption `solid_tf_level` verschluckt, die für jede Datenbankverbindung gesetzt sein muss. In OpenJPA kann man derartige Optionen im `prop`-String (siehe 6.7) mitgeben, es werden aber nur OpenJPA-Optionen, also solche, die mit"`openjpa.`" beginnen, zum JDBC-Treiber durchgelassen; andere werden schlichtweg ignoriert.

Ein weiterer Aspekt fügt jedem `SolidDriver.connect(...,Properties)`, also jeder Verbindungsanforderung, die `solid_tf_level` Option hinzu, indem er die Ausführung dieser Methode abfängt und den `Properties`-Parameter modifiziert:

```
@Before("execution(*solid.jdbc.SolidDriver.connect
                   (..,String,..,Properties,..)) && args(url, prop)")
public void addSolidTfLevel(String url,Properties prop) {
    if (url!=null && url.contains("solid"))
       prop.setProperty("solid_tf_level","1");
}
```

`(..,String,..,Properties,..)` spezifiert die relevanten Parameter. Die `args`-Klausel bindet entsprechende Variable `url` and `prop` an diese. Die Variable `url` ist nötig, um das DBS zu identifizieren, während `prop` zum Setzen der `solid_tf_level`-Option benutzt wird. Auch in diesem Fall wird eine externe JAR-Datei, der SOLID-JDBC-Treiber, modifiziert, deren Quellcode nicht verfügbar ist!

Erwähnenswert ist darüber hinaus, dass der Einsatz von Aspekten auch geholfen hat, das Failover-Problem in kurzer Zeit zu identifizieren. Folgender Aspekt übernimmt eine spezielle Form des Tracings, indem er nur relevante Ausgaben produziert:

```
@Before("execution(* *.*(..,String,..))")
public void myTrace(final JoinPoint jp) {
  Object[] args = jp.getArgs();
  for (Object param : args) {
   if (param instanceof String
   && param != null &&((String) param).contains("jdbc:solid:"))
       System.out.println("* In: " + jp.getSignature()
                          + "-> " + param.toString());
} } }
```

Dieser `@Before`-Advice fängt alle Ausführungen (`execution`) aller Methoden ab (die Wildcard `*` in `(* *.*)` bedeutet beliebiger Rückgabetyp, beliebige Klasse und Methode), die einen String-Parameter (`(..,String,..)`) besitzen, und prüft, ob der String eine SOLID URL enthält. Wenn dem so ist, wird die URL ausgegeben. Der Parameter `JoinPoint jp` liefert dabei Kontextinformation über den Joinpoint, z.B., `jp.getSignature` gibt die Signatur der abgefangenen Methode aus und `jp.getArgs()` die übergebenen Parameterwerte. Somit lässt sich der Übergang von einer korrekten zu einer abgeschnittenen URL sehr schnell in `Configurations.parseProperties` bestimmen:

```
* In: Options org.apache.openjpa.lib.conf.Configurations.parseProperties
  (String)
-> DriverClassName=solid.jdbc.SolidDriver,Url=
  jdbc:solid://host1:1315,host2:1315/usr/pw,defaultAutoCommit=false,initia
  lSize=35,maxActive=35,maxIdle=35,minIdle=10,minEvictableIdleTimeMillis=6
  0000,timeBetweenEvictionRunsMillis=60000,defaultTransactionIsolation=4
* In: boolean solid.jdbc.SolidDriver.acceptsUR L(String)->
  jdbc:solid://host1:1315
```

## 8 Zusammenfassung

Diese Arbeit hat gezeigt, dass die Migration von Hibernate nach OpenJPA mit moderatem Aufwand möglich ist. In dem vorgestellten Projekt kommt entgegen, dass das Persistenzframework in einer Template-Bibliothek gekapselt ist, so dass durch deren Umstellung bereits ein Großteil der Arbeiten an zentraler Stelle erledigt wird. Die Erfahrung zeigt auch, dass eine zügige Projektfindung in Verbindung mit einem heuristischem, iterativen Vorgehen zu einem fundierten und erfolgreichen Ergebnis führt.

Nichtsdestotrotz soll diese Arbeit betonen, dass eine Reihe von kniffligen Problemen auftraten, z.B. das Fehlen der nativen Schlüsselgenerierung oder das Failover-Problem, die im Vorfeld nicht ohne weiteres zu erwarten waren. Die verwendeten Lösungsstrategien sollten hierbei verdeutlichen, dass sehr unterschiedliche Ansätze den gewünschten Erfolg bringen können. Darüber hinaus sind Probleme aufgetreten, die an

sich keine besondere Verbindung mit der eigentlichen Domäne aufweise. Als Beispiel sei das Problem von OpenJPA mit der Ladereihenfolge von Klassenpfaden aufgrund eines „Split Packages“ im OSGi-Umfeld zu nennen (siehe Abschnitt 6.8). Dennoch stellte sich keines der Probleme als unüberwindbar hinaus, so dass die erfolgreiche Migration als Empfehlung für den Einsatz von OpenJPA gewertet werden kann.

Trotz der speziellen Natur einzelner Probleme, muss natürlich abschließend betont werden, dass der maßgebliche und grundlegende Erfolgsfaktor für die Migration die umfangreiche Testinfrastruktur 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.

## Referenzen


[BaH07] J. Bartholdt, U. Hohenstein: Caching und Transaktionen in Hibernate für Fortgeschrittene. JavaSpektrum 2007

[BaS05] M. Bannert, M. Stephan: JDO mit Kodo - ein Praxisbericht. JavaSPEKTRUM, Sonderheft CeBIT 2005

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

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

[EFB01] T. Elrad, R. Filman, A. Bader (eds.): Theme Section on Aspect-Oriented Programming. CACM 44(10), 2001

[Hib] Hibernate Reference Documentation. http://www.hibernate.org /hib_docs/v3/ reference/en/html/

[HoB06] U. Hohenstein, J. Bartholdt: "Performante Anwendungen mit Hibernate" in JavaSPEKTRUM 2/06

[JDO] JSR-000012 JavaTM Data Objects Specification. http://jcp.org/aboutJava/ communityprocess/first/ jsr012/

[JPA] Java Persistence API. http://java.sun.com/javaee/technologies/persistence.jsp

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

[La03] R. Laddad: AspectJ in Action. Manning Publications Greenwich 2003

[MR02] K. Müllner, K.-H. Rau, S. Schleicher: Java-basierte Datenhaltung: Ein kritischer Vergleich (Teil 1). JavaSPEKTRUM 9/10 2002

[Ne94] Jon Udell: Next's Enterprise Objects Framework, Byte Magazine, July 1994

[PaT06] S.E. Pagop, M. Tilly: “Caching Hibernate“, JavaMagazin 3/4 2006

[Pl04] M. Plöd: Winterschläfer: Objketrelationales Mapping mit Hibernate. Javamagazin 8/2004

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

[HOD+00] 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

[ViS07] D. Vines, K. Sutter: Migrating Legacy Hibernate Applications to OpenJPA and EJB 3.0. http://www.ibm.com/developerworks/websphere/techjournal/0708_vines/0708_vines.html

# Demo-Programm