INSZENIERTE TRANSPARENZ

Werden heute Java-Enterprise-Anwendungen entworfen, so muss als eine der ersten Entscheidungen die Art der Persistenz geklärt werden. Diese Frage bezieht sich dabei weniger auf den Datenspeicher – ein relationales Datenbanksystem kann stillschweigend vorausgesetzt werden – sondern auf den Umgang mit den Daten.

Für den Architekten ergeben sich zwei grundlegende Marschrichtungen: Fasst er die Daten der Anwendung als grobgranulare, wenig vernetzte Dateneinheiten auf, so wird er zu einem tabellengetriebenen Ansatz tendieren und vorrangig Entwurfsmuster einsetzen, die dem relationalen Aggregatzustand der Daten geschuldet sind, so beispielsweise Row/Record Sets, Page Iterators oder Data Access Objects. Sieht der Architekt hingegen die Daten der Anwendung als feingranulare, stark vernetzte Entitäten an, so ist er mit einem objektorientierten Ansatz besser beraten. Da die Daten trotz objektorientierter Sichtweise in einer relationalen Datenbank landen, gilt es hier eine Brücke zwischen beiden Welten zu schlagen. Der Unterschied, oft verharmlosend als Impedanz-Unterschied bezeichnet, ist jedoch wohl bekannt und technisch beherrschbar [1].

Als Vermittler zwischen beiden Welten existieren Frameworks, die ein objekt-relationales Mapping realisieren, ohne dass der Anwendungscode davon berührt wird. Aus Sicht des Anwendungsentwicklers handelt es sich bei dem Datenbestand der Anwendung also um einfache Objekte, die Attribute besitzen, zueinander Assoziationen eingehen oder Eigenschaften vererben.

Auch das Laden, Speichern, Aktualisieren oder Sperren von Objekten geschieht ohne gesonderte Mitwirkung des Anwendungsentwicklers. Objektorientiert sind ebenso Operationen auf den persistenten Objekten oder das Suchen nach persistenten Objekten in der Datenbank. Obwohl seitens der Datenbank weiterhin alles so aussieht, als ob es nach bester ERM-Manier modelliert worden wäre.

Diese enorme Abstraktion von der Speicherform der Daten bekommt man allerdings nicht zum Nulltarif. Neben architektonischen Regeln, die eingehalten werden müssen, damit sich der Zauber der Speicher-Transparenz tatsächlich einstellt, ist ein aufwändiges, explizites Mapping von Klassen auf Relationen vorzunehmen. Der hingegen oft kolportierte Performance-Verlust bei der Verwendung besagter Mapping-Frameworks, das sei nebenbei bemerkt, ist oft mehr gefühlt als real vorhanden. Aus Sicht der Gesamtanwendung ist er fast nie nachweisbar.

Sowohl hinsichtlich des Mapping-Verfahrens als auch hinsichtlich der architektonischen Spielregeln lässt sich festhalten, dass diese gut verstanden sind. Es ist sogar möglich, diese zu katalogisieren, um so einen Kanon des O/R-Mappings aufzustellen. Zu den allgemeinen Mustern lässt sich für jedes verfügbare O/R-Mapping-Framework dann präzise die konkrete Anwendung demonstrieren. Genau diese Absicht verfolgen wir mit dem Projekt OOPEX.

Hartnäckigkeit zahlt sich aus

Mit dem Projekt OOPEX, kurz für **o**bject-**o**riented **p**ersistence by **ex**ample, versuchen wir die Einstiegshürde in die Welt des O/R-Mappings ein gutes Stück tiefer zu legen [2].

Anhand von uniformen Beispielanwendungen werden die wesentlichen Merkmale eines O/R-Mappers isoliert voneinander vorgestellt. Die Beispiele sind ohne größere Aufwände ausführbar und möchten zum Ausprobieren der Technologie einladen. Sollten Sie sich beispielsweise dafür interessieren, wie das Merkmal „Fetch-Plan“ bei einem der in OOPEX behandelten Frameworks funktioniert, so können Sie sich das dazu passende Beispiel heraus greifen, anschauen, ausführen und sogar debuggen.

Allen Beispielen gemeinsam ist eine vorgefertigte Werkzeugkette, die die Ausführung der Beispiele vorbereitet. Diese kann je nach Framework die Schritte

- verifiziere die Mappings,
- transformiere den Java-Bytecode (Enhancement),
- generiere die Schema-Beschreibung als SQL DDL-Anweisungen und
- lege das Datenbankschema an

umfassen. Die Werkzeugkette wird mit Hilfe von Apache Ant [3] definiert. Damit erübrigen sich kryptische Aufrufe von Kommandozeilen-Werkzeugen oder lange Klick-Folgen durch ihre grafischen Pendants.

Neben einem schnelleren Einstieg in die Technologie, kann das OOPEX-Projekt auch dazu verwendet werden, die vorgestellten Persistenz-Frameworks gezielter zu evaluieren. Möchten Sie herausfinden, ob ein bestimmtes Framework Ihren funktionalen Anforderungen entspricht,

so können Sie sich die jeweiligen Beispiele heraussuchen und müssen dafür nicht aufwendig die Dokumentation studieren.

Auch bei nicht-funktionalen Eigenschaften, wie Performance und Skalierbarkeit können Sie so schneller zum Kern der Sache vordringen.

Ein weiterer Nutzen von OOPEX entsteht bei der Migration einer Anwendung von einem Persistenz-Framework auf ein anderes. Durch die Beispiele bekommen Sie sofort einen Blick dafür, welche der kanonischen Merkmale zur Deckung gebracht werden können und bei welchen Merkmalen größere Nacharbeit erforderlich sein wird. Auch können Sie abschätzen, ob bestehende Workarounds durch das neue Framework gegenstandslos werden. Ebenso lassen sich durch die Beispiele Fragen zum API des neuen Frameworks direkt durch Ausprobieren klären.

Insgesamt betrachtet bietet OOPEX umso mehr Nutzen je weniger die Dokumentation des jeweiligen Persistenz-Frameworks an Struktur und Inhalt besitzt. Durch scharf abgegrenzte, problemorientierte Beispiele reduziert sich so erheblich der Aufwand für Versuch-Irrtum-Annäherungen, das Durchforsten von Mailing-Listen oder das Verfolgen obsoleter Workarounds. Nebenbei bemerkt muss man leider auch feststellen, dass die wenigsten Frameworks mit übersichtlichen Beispielen ausgeliefert werden, die ohne ausführliche Anleitung ausführbar sind.

Wieso funktioniert OOPEX

Die Frage, warum ein Unterfangen wie OOPEX funktioniert kann ohne sich in framework-spezifischen Belanglosigkeiten zu verzetteln, lässt sich einfach beantworten: Zum einen hat die Modellierung von relationalen Datenbanken eine lange Tradition. In den frühen 70er Jahren des 20. Jahrhunderts stellte Edgar Codd das relationale Datenbankmodell vor, wenige Jahre später wurde der Vorläufer von SQL entwickelt und Anfang der 80er Jahre verdrängten bereits die ersten relationalen Datenbanksystem frühere Technologien. Dazu passend entwickelte sich in etwa im gleichen Zeitraum die Theorie zur Datenbank-Modellierung. Entitätstyp, Beziehungstyp, Normalform, Primärschlüssel, Fremdschlüssel, Relation, Kardinalität, Verbund und vieles mehr gehören heutzutage zum Grundwissen der Softwareentwicklung. Dieses Grundwissen kann als allgemeingültig betrachtet werden, da sich durch den Standard SQL heute jedes Datenbanksystem nach außen weitestgehend gleich präsentiert und damit datenbankspezifische Besonderheiten für die Modellierung nicht zwingend Beachtung finden müssen.

Zum anderen funktioniert OOPEX gerade deshalb, weil auch die Technologie des O/R-Mappings mittlerweile auf eine 15 jährige Tradition zurück blicken kann, angefangen mit NeXTs DBKit für Objective-C oder TopLink für Smalltalk bis hin zu den heutigen Java-Implementierungen [4][5]. Da die Konzepte einer objektorientierten Programmiersprache, bei OOPEX ist es Java, in gewisser Weise abzählbar sind, ergibt sich eine gut definierbare Erwartungshaltung an ein sinnvolles O/R-Mapping.

Bevor ich hier auf Details eingehe, sei noch ein dritter Grund aufgeführt, der für OOPEX spricht: Die Standardisierungsbestrebungen hinsichtlich des O/R-Mapping in Java (siehe Abbildung 1).

Lässt man die Fehlversuche namens Container-Managed-Persistence der frühen EJB-Spezifikationen beiseite, die sich ohnehin besser der eingangs skizzierten relationalen Betrachtungsweise zuordnen lassen, so liefert 2003 der Java Specification Request 12 mit Java Data Objects die erste nennenswerte O/R-Mapping Spezifikation für Java [6].

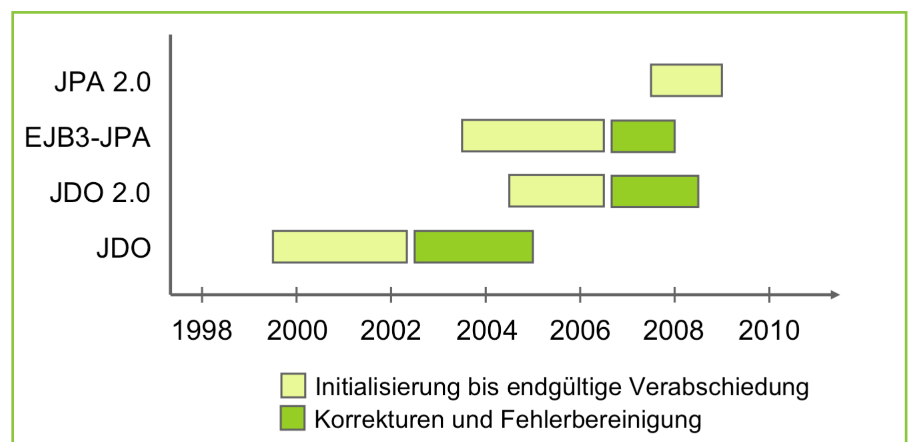

Abb. 1: O/R-Mapping-Spezifikationen für Java

Das größte Manko dieser Spezifikation besteht ironischerweise darin, dass sie keine Vorschriften darüber macht, wie das Mapping zwischen Klassen und Tabellen durch den Nutzer konkret definiert werden soll. Auch sind in der Spezifikation kaum Aussagen darüber zu finden, wie die Framework-Implementierung das Mapping zur Laufzeit vorzunehmen hat. Zwei Sachen sind

der JDO Spezifikation jedoch zu Gute zu halten: Die Persistenz muss nach dieser Spezifikation aus Sicht der Anwendung vollkommen transparent sein – Datenobjekte sind Instanzen normaler Java-Klassen, die keinen nennenswerten Einschränkungen unterliegen (POJOs). Und die Spezifikation beschreibt sowohl den Einsatz des O/R-Mappers für ein Two-Tier-Szenario als auch für ein Application-Server-Szenario mit verteilten Transaktionen. Dieser Aspekt bedeutete eine wesentliche Verbesserung, da von nun an der Persistenz-bezogene Teil einer Enterprise-Anwendung relativ gefahrlos auch außerhalb des Application-Containers getestet werden konnte.

Den Nachfolger von JDO bilden gleich zwei Spezifikationen: JSR 243 und JSR 220. JSR 243, Java Data Objects 2.0, ist die historische Weiterentwicklung des JDO-Standards und wurde Mitte 2006 verabschiedet [7]. Nicht nur wurden viele neue Features hinzugenommen, es wurden erstmalig auch die Beschreibungsmittel für das O/R-Mapping definiert. Da die JDO Spezifikationsgruppe seit Anfang an dem Irrwitz verfallen war, auch nicht-relationale Datenspeicher berücksichtigen zu müssen, definiert die JDO 2 Spezifikation neben den bekannten JDO-XML-Dateien zusätzliche ORM-XML-Deskriptoren um den Spezialfall „relationale Datenbank“ genügend abgrenzen zu können. Mehr als akademisches Gedankenspiel ist daraus nicht geworden – nicht-relationale Mapping-Frameworks, die die JDO-Spezifikation wesentlich erfüllen, habe ich in der freien Wildbahn noch nicht gesichtet.

Der zweite Nachfolger von JDO ist JSR 220 (EJB3), genauer gesprochen JSR 220-Persistence. Dieser wurde ebenso Mitte 2006 verabschiedet [8]. Landläufig wird dieser Standard auch als Java Persistence API (JPA) bezeichnet. Obwohl man auf Grund der Begleitumstände vermuten könnte, dass JPA die Fortsetzung von Container-Managed-Persistence ist, so dieser Teil der JSR 220-Spezifikation im Geiste dem JDO-Standard wesentlich näher als CMP. Dass nun zwei konkurrierende, inhaltsgleiche Spezifikationen für Java existieren, ist ein Kuriosum der Geschichte, wenn auch nur ein kurzes. JDO 2 wurde bereits still und heimlich von Sun zugunsten von JPA aufgegeben. JPA ist im Gegensatz zu JDO 2 auf objekt-relationales Mapping beschränkt und spezifiziert dieses auch mit einer hinreichenden Genauigkeit. Ebenfalls im Gegensatz zur JDO-Spezifikation sind Implementierungsdetails weitestgehend aus der Spezifikation ausgeklammert geblieben. So beschreibt JDO penibel genau, wie das Bytecode-Enhancement vollzogen werden muss und fordert sogar Kompatibilität der enhanceten Klassen zwischen verschiedenen Implementierungen. Bei JPA steht es dem Implementierer frei, ob er transparente Proxies, Instrumentierung über VM-Agenten, Class-Transformatoren, Build-time Bytecode-Enhancement oder eine andere Technologie für das transparente State-Tracking benutzt, solange die Spezifikation erfüllt bleibt. Dennoch ist JPA in der vorliegenden Form alles andere als vollständig und man kann nur hoffen, dass mit der zweiten Version dieses Standards die Lücken geschlossen werden. Der JSR 317, Java Persistence 2.0, ist bereits in Arbeit und sein Ergebnis für Ende 2008 angekündigt [9].

Obwohl neben den Standard-Frameworks auch andere, gleichermaßen bedeutende Frameworks existieren, die traditionell ein eigenes API anbieten, lässt sich doch eine gewisse Harmonisierung feststellen. Die Mitgliederlisten der Spezifikationsgruppen umfassen im Wesentlichen alle bedeutenden Framework-Hersteller, so dass es kaum verwundern kann, dass sich der Sprachgebrauch, die Architekturkonzepte, die Anwendungsmuster und sogar die Implementierungsweisen zwischen den einzelnen Frameworks immer stärker angleichen. Diese Tatsache ist dem OOPEX Projekt sehr zuträglich, da man so auf eine erläuterungsbedürftige „Universaltheorie“ des O/R-Mappings weitestgehend verzichten kann.

Die Klassifikation

Bevor ein Beispiel zum O/R-Mapping den Artikel abschließt, soll noch kurz das Klassifikationsschema von OOPEX angerissen werden. Die vollständige Klassifikation kann jederzeit auf der OOPEX-Startseite bei Google-Code eingesehen werden. Die Klassifikation unterscheidet zunächst zwischen:

- allgemeinen Anwendungsfällen,
- Feldern,
- Beziehungen,
- Vererbung,
- Abfragen und
- Optimierungen.

Innerhalb dieser Kategorien befinden sich die Eigenschaften, die das OOPEX-Projekt vorstellen möchte. Eine weitere Hierarchieebene existiert nicht. Innerhalb der allgemeinen Anwendungsfälle befindet sich beispielsweise ein einfaches „Hello-World“ Demonstrationsbeispiel. Weiterhin befinden sich dort Demonstrationen zur versteckten Objekt-Identität, zu optimistischem Offline-Locking, Attach/Detach, Objektidentität bei zusammengesetzten Primärschlüsseln oder Abfragen über nicht-committete Objekte. Die Oberkategorie „Felder“ umfasst Demonstrationen zum Mapping von Zeit-Typen, Big-Decimal-Typen, Binary-Feldern oder langen Zeichenketten. Innerhalb der Kategorie „Beziehungen“ befindet sich unter anderem ein Eintrag über unidirektionale 1-zu-N-Beziehungen mit

Join-Spalte (relationships-one_to_many_unidirectional). Dieses Beispiel ist recht interessant und soll deshalb aus der Menge aller herausgegriffen und abschließend etwas näher beleuchtet werden.

Beispiel: unidirektionale 1-zu-N-Beziehung mit Join-Spalte

Angenommen es existieren zwei Klassen: „Sportler“ und „Medaille“, beide für sich genommen vollständig existenzfähig. Ein Sportler hat keine, eine oder mehrere Medaillen. Aus Sicht der Medaille spielt es keine Rolle, welchem Sportler sie gehört. Fest steht aber, dass sie höchstens einem Sportler zugeordnet ist. Das Übertragen von Medaillen von einem Sportler auf einen anderen ist möglich, beispielsweise bei Doping-Vergehen, dies ist aber nicht die Regel. Objektorientiert würde man diesen Sachverhalt wie in Abbildung 2 modellieren.

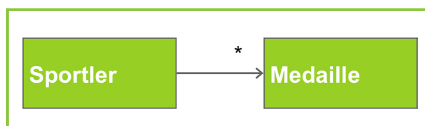

Abb. 2: Klassenstrukturdiagramm

Aus relationalen Sicht ergeben sich zwei Tabellen „SPORTLER“ und „MEDAILLE“, wobei vereinfachend für beide Tabellen ein künstlicher Primärschlüssel „ID“ angenommen wird. Über einen Fremdschlüssel „S_ID“ in der Tabelle „MEDAILLE“ auf die Spalte „ID“ der Tabelle „SPORTLER“ wird die Beziehung zwischen beiden Entitätstypen hergestellt. Abbildung 3 veranschaulicht die Zusammenhänge.

Im Nachfolgenden werde ich das Mapping mit Hilfe der Standards JDO, JDO2, JPA, JPA2 und hinsichtlich des Frameworks Hibernate näher beleuchten. In OOPEX sind noch

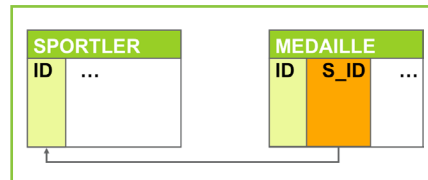

Abb. 3: Tabellenstruktur

weitere Frameworks mit proprietärem Mapping enthalten. Da aber die Mapping-Beschreibung teilweise eine derart ausladende XML-Prosa ist, muss auf eine Darstellung hier verzichtet werden.

Müsste man nun das vorgestellte Beispiel mit den Mitteln von JDO mappen, so könnte die Mapping-Beschreibung für die Klasse „Sportler“ wie in Abbildung 4 aussehen.

Die Mapping-Beschreibung ist frei erfunden, da sie Vendor-Extensions verwenden muss, um das Gewollte zu erreichen. Innerhalb von OOPEX ist kein Framework vertreten, welches dieses Mapping-Beschreibung hier verstehen kann. Den Mangel an standardisierten Beschreibungsmittel in JDO1 nahmen manche Hersteller zum Anlass, eigene Beschreibungsformen zu entwickeln. Bei Xcalia/Lido werden Angaben, die sich nicht über die jdo.dtd formulieren lassen, beispielsweise in zusätzlichen lido.xml Dateien untergebracht. Beispiel siehe Abbildung 5.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdo PUBLIC "-//Sun Microsystems, Inc.//DTD Java Data
Objects Metadata 1.0//EN"
                     "http://java.sun.com/dtd/jdo_1_0.dtd">
<jdo>
<package name="oopex.sample.model">
<class name="Sportler">
   <field name="id" primary-key="true">
      <extension vendor-name="ACME" key="field" value="basic">
         <extension vendor-name="ACME" key="column" value="ID"/>
      </extension>
   </field>
   <field name="medaillen">
      <collection element-type="oopex.sample.model.Medaille"/>
      <extension vendor-name="ACME" key="relationship"
                 value="one-to-many">
         <extension vendor-name="ACME" key="join-column"
                    value="S_ID" />
      </extension>
   </field>
   <extension vendor-name="ACME" key="table" value="SPORTLER" />
</class>
</package>
</jdo>
```

Abb. 4: Sportler.jdo

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns:xsi="[...]"
       xsi:noNamespaceSchemaLocation="lido-maping.xsd">
<package name="oopex.sample.model">
<class name="Sportler" entity="SPORTLER">
   <field name="{ID}" attribute="ID" />
   <field name="medaillen">
      <unordered-collection entity="MEDAILLE">
         <associate field="{ID}" to-attribute="S_ID"/>
      </unordered-collection>
   </field>
</class>
</package>
</project>
```

Abb. 5: lido-Sportler.xml

Mit der Spezifikation von JDO2 hielt eine detaillierte Beschreibung des O/R-Mappings Einzug in den Standard. Allerdings wurde dieser Teil der Spezifikation mangels Kundeninteresse nur von wenigen Herstellern umgesetzt. In Abbildung 6 das Mapping für Bea Kodo.

Bei Bedarf können alle Mapping-bezogenen Angaben, wie Tabellen-, Spalten- oder Sequence-Namen, in einen Datenbank-spezifischen ORM-Deskriptor ausgelagert werden (zum Beispiel Abbildung 7).

Mit der Spezifikation von EJB3 und damit JPA1 hielten Java-Annotations Einzug in den Bereich des O/R-Mappings. Statt externer XML-Dateien können nun Klassen, Felder oder Setter-Methoden direkt im Java-Quelltext mit O/R-Mapping-Informationen versehen werden. Für das hier skizzierte Beispiel macht die JPA1-Spezifikation jedoch eine kapitale Auslassung. Mit JPA-Standardmitteln ist das angestrebte Mapping nicht zu erzielen. Für unidirektionale 1-zu-N-Beziehungen sieht die Spezifikation zwingend die Verwendung einer Join-Tabelle vor, wie sie üblicherweise für M-zu-N-Beziehungen Verwendung findet. Erst wenn man die Beziehung zwischen Sportler und Medaille bidirektional gestaltet, ist auch die Verwendung einer Join-Spalte möglich. Um dennoch das gewünschte Mapping zu erzielen, werden Framework-spezifische Annotationen notwendig. Im Falle von OpenJPA sieht das Mapping wie in Abbildung 8 aus (Sportler.java).

Neben Annotationen sieht die EJB3-Persistence-Spezifikation auch die Beschreibung des Mappings in XML-Dateien vor. Allerdings ist auch hier Kuriosum zu vermelden.

```
<?xml version="1.0"?>
<jdo xmlns="http://java.sun.com/xml/ns/jdo/jdo"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://java.sun.com/xml/ns/jdo/jdo
        http://java.sun.com/xml/ns/jdo/jdo_2_0.xsd">
<package name="oopex.sample.model">
<class name="Sportler" table="SPORTLER">
   <datastore-identity column="ID" />
   <field name="medaillen">
      <collection element-type="oopex.sample.model.Medaille"/>
      <element column="S_ID" />
   </field>
</class>
</package>
</jdo>
```

Abb. 6: Sportler.jdo

```
<?xml version="1.0" encoding="UTF-8" ?>
<orm xmlns="http://java.sun.com/xml/ns/jdo/orm"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://java.sun.com/xml/ns/jdo/orm
        http://java.sun.com/xml/ns/jdo/orm_2_0.xsd">
<package name="oopex.model.sample">
<sequence name="SPORTSEQ" strategy="nontransactional"
          datastore-sequence="SPORT_SEQ" />
<class name="Sportler" table="SPORTLER">
   <datastore-identity column="ID" sequence="SPORTSEQ" />
   <field name="name" column="NAME" />
   <field name="medaillen">
      <element column="S_ID" />
   </field>
</class>
</package>
</orm>
```

Abb. 7: Sportler-oracle10.orm

```
package oopex.sample.model;
import java.util.*;
import javax.persistence.*;

@Entity
@Table(name="SPORTLER")
@SequenceGenerator(name="SPORTSEQ", sequenceName="SPORT_SEQ")
public class Sportler {

   public Person() {}

   @Id
   @GeneratedValue( strategy=GenerationType.SEQUENCE,
                    generator="SPORTSEQ")
   private long id;

   @OneToMany(cascade=CascadeType.ALL)
   @org.apache.openjpa.persistence.jdbc.
    ElementJoinColumn(name="S_ID")
   private Set<Medaille> medaillen=new HashSet<Address>();

        // Getter- und Setter-Methoden
}
```

Abb. 8: Sportler.java

buschmais

Beratung . Technologie . Innovation

```
<?xml version="1.0"?>
<entity-mappings
   xmlns="http://java.sun.com/xml/ns/persistence/orm"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation=
      "http://java.sun.com/xml/ns/persistence/orm
       http://java.sun.com/xml/ns/persistence/orm_1_0.xsd"
   version="1.0">
<sequence-generator name="SPORTSEQ" sequence-name="SPORT_SEQ"/>
<entity class="oopex.sample.Sportler" name="Sportler"
        access="FIELD">
<table name="SPORTLER" />
<attributes>
   <id name="id">
      <generated-value strategy="SEQUENCE"
                       generator="SPORTSEQ" />
   </id>
   <one-to-many name="medaillen">
      <join-column name="S_ID" />
      <cascade><cascade-all />   </cascade>
   </one-to-many>
</attributes>
</entity>
...
</entity-mappings>
```

Abb. 9: META-INF/orm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate
  Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
<class name="oopex.sample.Sportler" table="SPORTLER">
   <id name="id">
      <generator class="sequence" />
   </id>
   <set name="medaillen" cascade="persist,save-update">
      <key column="S_ID" />
      <one-to-many class="oopex.sample.model.Medaille"/>
   </set>
</class>
</hibernate-mapping>
```

Abb. 10: Sportler.hbm.xml

Während JDO1 und JDO2 Erweiterungen in Form von Vendor-Extensions zulassen, sind die Beschreibungsmöglichkeiten bei JPA auf das beschränkt, was die Spezifikation vorgibt. Im vorliegenden Fall bedeutet dies, dass eine XML-Beschreibung nur additiv aber nicht exklusiv genutzt werden kann. Die Definition der Join-Spalte muss also weiterhin durch eine Annotation im Quelltext definiert werden. OpenJPA erlaubt sich an dieser Stelle die Spezifikation großzügig auszulegen, so dass dennoch eine ausschließliche XML-Beschreibung möglich wird (Abbildung 9).

Geht man streng nach dem Spezifikationstext, sollte das Tag <join-column> innerhalb von <one-to-many> gar nicht möglich sein. Gemäß der ORM-XML Schema-Beschreibung ist es dies aber. Andere JPA-konforme Frameworks sehen die Sache weniger großzügig. Erst mit JPA2 wird die Spezifikation so erweitert, dass die hier angegebenen Beispielmappings explizit abgedeckt werden.

Abschließend bleibt noch das Mapping für Hibernate. Dies gibt sich ganz unprätentiös und liest sich wie in Abbildung 10.

Fazit

Die Beispiele verdeutlichen sehr gut, wie trotz verschiedenartiger Syntax das Wesen des Mappings bestehen bleibt. Falls Sie Lust auf Mehr bekommen haben, schauen Sie sich die kompletten Beispiele im OOPEX-Projekt an. Die Beispiele sind als Projekte in die Eclipse IDE einbindbar und können, eine konfigurierte Datebankverbindung vorausgesetzt, direkt ausgeführt werden.

TEXT: FRANK SCHWARZ

Referenzen

[1] Ch. Bauer und G. King: Java Persistence with Hibernate. Manning 2007, S. 10ff.
[2] http://code.google.com/p/oopex/
[3] http://ant.apache.org/
[4] http://en.wikipedia.org/wiki/Enterprise_Objects_Framework
[5] Oracle: A Brief History of TopLink http://www.oracle.com/technology/tech/java/newsletter/articles/toplink/history_of_toplink.html
[6] http://jcp.org/en/jsr/detail?id=12
[7] http://jcp.org/en/jsr/detail?id=243
[8] http://jcp.org/en/jsr/detail?id=220
[9] http://jcp.org/en/jsr/detail?id=317

kontakt

buschmais GbR
Leipziger Straße 93
01127 Dresden

Tel +49 (0)351 3209230
Fax +49 (0)351 32092329
info@buschmais.de
www.buschmais.de

Copyright © 2008 buschmais GbR Alle Rechte vorbehalten. Java und alle Java-basierten Markennamen sind Markennamen oder registrierte Markennamen von Sun Microsystems, Inc. in den USA und anderen Ländern. Alle anderen Marken- und Produktnamen sind Markennamen oder registrierte Markennamen des jeweiligen Eigentümers. 'buschmais' ist eine Marke der buschmais GbR. Ihre Verwendung im Geschäftsverkehr ist nur bei ausdrücklicher, schriftlicher Erlaubnis der buschmais GbR statthaft.