Masterarbeit am Institut für Informatik der Freien Universität Berlin,

Arbeitsgruppe Software Engineering

Von JDO zu JPA - Entwicklung einer Migrationsstrategie

Christian Kühl
Matrikelnummer: 4233240
christian.kuehl87@gmail.com

Eingereicht bei: Prof. Dr. Lutz Prechelt

Berlin, 12.09.2013

Zusammenfassung

Die Migration eines fest integrierten Frameworks innerhalb eines wirtschaftlich orientierten Unternehmens ist immer mit einem Risiko verbunden. Für diesen Zweck möchte die tolina GmbH für die Migration ihres Persistierungsproviders (Kodo in der Version 3.4.1), welcher momentan auf der Java Data Objects (JDO) Spezifikation aufbaut, eine Migrationsstrategie entwickelt haben. Das Ziel ist der Einsatz eines Providers, welcher auf der Java Persistence Agency (JPA) Spezifikation aufbaut. Für die Entwicklung dieser Strategie wurden die beiden Spezifikationen verglichen und die entscheidenden Unterschiede ermittelt. Zu diesen Unterschieden zählen die Konfiguration der Persistierung, die Beschreibung der zu persistierenden Klassen und der Aufbau von Queries. Für diese Unterscheidungsmerkmale wird gezeigt, wie mit Hilfe von JPA das gleiche Verhalten erreicht wird. Die entwickelte Strategie wird anhand eines vorher gewählten Prototypen getestet. Mit Hilfe der gesammelten Daten werden Abschätzungen vorgenommen, wie hoch der zeitliche Aufwand der Migration wäre und welche Risiken dabei auftreten können. Abschließend wird diskutiert, ob Tests allgemein eine Migration vereinfachen können, da Fehler eventuell frühzeitig erkannt werden.

Abstract

The migration of a framework is always a high risk for any company. The tolina GmbH is an enterprise which is developing software. They want to replace their persistence provider, named kodo, which was built regarding the Java Data Objects (JDO) spezification. The aim is the implementation of a provider which is based on the Java Persistence Agency (JPA) specification. In this theses, I have developed a strategy for a successful migration. First both specification were compared to identify the differences between them. The differences are the configuration of the specification, the configuration of persisted classes and the building and execution of queries. Next, this thesis describes how to accomplish the same behaviour by using JPA. The developed strategy was tested on a prototype. The data of the test migration is then used to make an estimation for the duration of the actual migration. Furthermore, they are used to determine which risks arise from the migration. Finally, it is discussed whether tests are capable of easing migrations by exposing possible mistakes at an early stage.

Eidesstattliche Erklärung

Ich versichere hiermit an Eides statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

12.09.2013

(Christian Kühl)

Inhaltsverzeichnis

Kapitel 1

Einleitung

Das dauerhafte Speichern von Daten ist in allen Bereichen der Wirtschaft von groser Bedeutung. Dies geschieht entweder durch Abheften von Unterlagen in Ordnern und Aktenschränken oder durch digitales Speichern in Datenbanken. In der Informatik definiert man diesen Prozess mit dem Wort Persistenz. Die tolina GmbH ist ein auf Softwareentwicklung spezialisiertes Unternhemen. Momentan gibt es zwölf Produkte, die bei Kunden im Einsatz sind. Diese Produkte speichern Einstellungen und Kundeninformationen in Datenbanken. Die Entwicklung der Produkte erfolgt komponentenbasiert, das heist es gibt viele kleine Komponenten aus denen die einzelnen Produkte zusammengesetzt werden. Somit teilen sich Produkte die gleiche Basis. Als Softwareentwickler möchte man sich nicht direkt mit der Persistenz auseinandersetzen, weshalb es Spezifikation und Provider gibt, welche das direkte Speichern übernehmen.

> *Man muß die Dinge so einfach wie möglich machen. Aber nicht einfacher.* Albert Einstein

Die tolina GmbH verwendet zur Zeit noch die Spezifikation der Java Data Objects (JDO), möchte aber möglichst einfach auf eine modernere Spezifikation wechseln. Dieser Vorgang wird als Migration bezeichnet. Zur Auswahl steht die Java Persistence API (JPA). Aber warum wechseln, wenn JDO funktioniert? Das Unternhemen verwendet momentan die kodo JDO Implementierung von Solarmetric. Diese wurden mittlerweile von BEA Systems gekauft, welche wiederum von Oracle aufgekauft wurden. Damit ergeben sich folgende unbekannte Risiken, welche das Unternehmen nicht eingehen möchte:

- Wird es weiterhin Support für das Produkt geben?
- Wie entwickelt sich die Lizenzvergabe (bei Solarmetric war es eine Lizenz je Entwickler)?

Weiterhin gibt es einige Indize dafür, dass JDO aussterben wird. Dies fast auch Herr Schwarz (vgl. [Sch08]) in seinem Artikel von 2008 zusammen. Es gäbe zu diesem Zeitpunkt überhaupt nur noch drei verschiedene Produkte auf dem Markt, welche ihrerseits keinen Support zur Verfügung stellen. Aušerdem steht der Code von kodo nicht zur Verfügung, wodurch keine eigenen Bugfixes an der vorhandenen Version vorgenommen werden können. Zuletzt ist kürzlich ein Problem mit KODO und Java 7 aufgegtreten. So zerstört KODO durch seine Bytecodeänderungen die Stackmaps von Java, sodass diese unbrauchbar sind. Da ab Version 7 jedoch davon ausgegangen wird, dass sie existieren kommt es zu Fehlern bei der Laufzeit.

Zur Migration von JDO nach JPA gibt es bisher für die tolina GmbH noch keine passende Strategie. Im Rahmen dieser Abschlussarbeit soll eine Migrationsstrategie entwickelt werden und prototypisch getestet werden. Diese Arbeit beinhaltet nicht eine Anleitung zur kompletten Migration eines Systems. Sie bietet einen Leitfaden, wie eine Migration durchgeführt werden kann. Folgend eine Beschreibung der Struktur der Abschlussarbeit.

- Kapitel 2
  In Kapitel 2 werden beide Spezifikationen vorgestellt und die ermittelten Unterschiede anhand der JDO Spezifikation detailliert beschrieben.
- Kapitel 3
  Kapitel 3 ist das Hauptkapitel meiner Arbeit und beschreibt die Migration. Dazu gehört die Ermittlung der verwendeten Kodo Extensions und die Migration dieser. Zusätzlich beschreibt dieses Kapitel die Wahl des Prototypen und des Providers. Für die in Kapitel 2 festgestellten Unterschiede wird eine prototypische Migration vorgenommen. Außerdem wird die Dauer der Migration und das Risiko für die Migration angegeben. Abgeschlossen wird das Kapitel mit dem Thema Automatisierung.
- Kapitel 4
  Kapitel 4 befasst sich mit dem Thema Qualitätssicherung. Es wird geschildert in wie weit Tests bei einer vorhandenen Migration helfen können.

- Kapitel 5
  Kapitel 5 gibt abschließend das Fazit der Arbeit ab und schafft einen Ausblick darauf, was in künftigen Arbeiten relevant sein wird.

Kapitel 2

Stand der Wissenschaft und Technik

2.1 Persistenz, Query und Co.

Persistenz: Persistenz beschreibt den Zustand eines Objektes bezüglich seiner Lebensdauer (vgl. [per]). Ist ein Objekt persistent, zum Beispiel in einer Datenbank oder auf dem Dateisystem hinterlegt, so existiert es unabhängig vom Kontext der auszuführenden Anwendung beliebig lange. Das Objekt bleibt erhalten, bis ein entsprechender Aufruf zum Entfernen des Objektes getätigt wurde.

Query: Eine Query ist eine Anfrage an eine Datenbank. Für das Schreiben von Queries wird die Structured Query Language (SQL) verwendet. Es gibt verschiedene Arten von Queries: Select Query: Zum Suchen und Finden von Informationen in Datenbanken Insert Query: Fügt neue Datensätze zu einer Tabelle hinzu Update Query: Verändert die Werte einer Tabelle Delete Query: Löscht Datensätze aus der Datenbank

Transaktion: Eine Transaktion (vgl. [LR03]) ist die Zusammenfassung mehrerer Operationen, welche dem ACID- Prinzip (atomicity, consistency, isolation, durability) folgen müssen. Die vier Eigenschaften sind folgendermaßen definiert: Atomarität (atomicity): Die Transaktion ist in sich geschlossen und kann nur als ganzes ausgeführt werden. Erst nach abgeschlossener Ausführung ist ihre Wirkung sichtbar. Konsistenz (consistency): Der Zustand der Datenbank ist vor der Ausführung in

sich konsistent und ist auch nach der Ausführung der Transaktion wieder in einem konsistenten Zustand. Lediglich während der Ausführung darf die Datenbank in einem inkonsistenten Zustand sein. Isolation (isolation): Eine Transaktion läuft immer so ab als würde sie allein ausgeführt werden. Auch nebenläufige Transaktionen dürfen sich gegenseitig nicht beeinflussen. Permanenz (durability):Das Ergebnis einer korrekt ausgeführten Transaktion bleibt in der Datenbank erhalten, solange die Datenbank besteht.

Migration: Kurz gesagt bedeutet Migration den Wechsel einer bestehenden Hard- oder auch Software auf eine neue Hard- oder Software.

Application Program Interface (API): Eine API ist eine Menge von Befehlen, Funktionen und Protokollen, welche Entwickler nutzen können, um auf das Programm zu zugreifen. Eine API stellt dabei sicher, dass alle Anbindungen an das Programm äquivalent sind.

Maven: Maven ist ein Tool zur Verwaltung von Projekten (vgl. [O'B]) . Es beinhaltet die komplette Funktionalität eines Buildtools (wie es Ant) ist und erweitert diese um neue Features. Dazu gehören zum Beispiel das Definieren von abhängigen Bibliotheken und das generieren von Testergebnissen zum Beispiel als eigene Internetseite. Beim Hinzufügen abhängiger Bibliotheken sucht Maven zuerst in seinem lokalen Repository auf der Festplatte. Findet es dort nicht die passende Abhängigkeit, so sucht es in einem definierten Repository online weiter. Bei uns ist dieses Repository der Nexus von Sonatype.

Project Object Model (POM): Das POM ist eine XML-Datei, welche Maven benötigt. Sie beschreibt deklarativ den Namen des Projektes, die Abhängigkeiten, den Build-Prozess und bietet weitere Konfigurationsmöglichkeiten für Maven.

Repository: Ein Repository ist ein Verzeichnis in dem digitale Daten verwaltet werden. Innerhalb dieser Arbeit handelt es sich immer um ein Software-Repository.

Nexus: Nexus ist ein Repository-Manager von der Firma Sonatype zur Verwaltung und Bereitstellung von Maven Projekten. Es ist ein

Proxy, welcher bei Anfrage einer Abhängigkeit, diese Anfragen weitersendet bis die entsprechende Bibliothek gefunden wurde. Anschließend landen sie im Cache, sodass eine erneute Anfrage direkt beantwortet werden kann.

Komponentenbasierte Entwicklung: Das Ziel der Komponentenbasierten Entwicklung (vgl. [kom]) ist es, unabhängige Komponenten zu entwickeln, sodass diese beliebig verwendet werden können um neue Produkte zu erstellen. Im Bezug der Softwareentwicklung ist eine Komponente ein Modul der Software. Neue Komponenten müssen erst entwickelt werden, wenn neue Anforderungen entstehen. Der Vorteil der Komponentenbasierten Entwicklung ist einerseits die Zeitersparnis beim Entwickeln neuer Produkte und andererseits die Gewährleistung der Qualität. Bei korrekter Entwicklung gibt es keinen redundanten Code. Das heißt bei eventuellen Anpassungen muss lediglich eine Codestelle verändert werden. Dies steigert die Qualität des Codes.

2.2 JDO

Abbildung 2.1 zeigt die Architektur von JDO. Welche in diesem Kapitel anhand der einzelnen Komponenten detaillierter erläutert wird.

PersistenceManagerFactory
Die PersistenceManagerFactory (vgl. [Kru02]) erzeugt neue PersistenceManager Objekte. Außerdem konfiguriert man damit das Verhalten der Persistierungsschicht der Anwendung. Dazu gehören zum Beispiel die Einstellungen für die Transaktionen. Mit Hilfe des JDOHelpers kann man sich innerhalb einer Anwendung eine PersistenceManagerFactory erstellen.

```
PersistenceManagerFactory factory = JDOHelper.//
getPersistenceManagerFactory(System.getProperties());
```

PersistenceManager
Der PersistenceManager ist das Kernstück von JDO. Er verwaltet mehrere Persistierungsobjekte und bietet Funktionen zum Erstellen und Entfernen von Objekten. Wie man Abbildung 2.1 entnehmen kann, gibt es zu jedem PersistenceManager genau ein zugehöriges Transactions-Objekt. Zusätzlich dient er als Factory zum Erzeugen von Query Objekten.

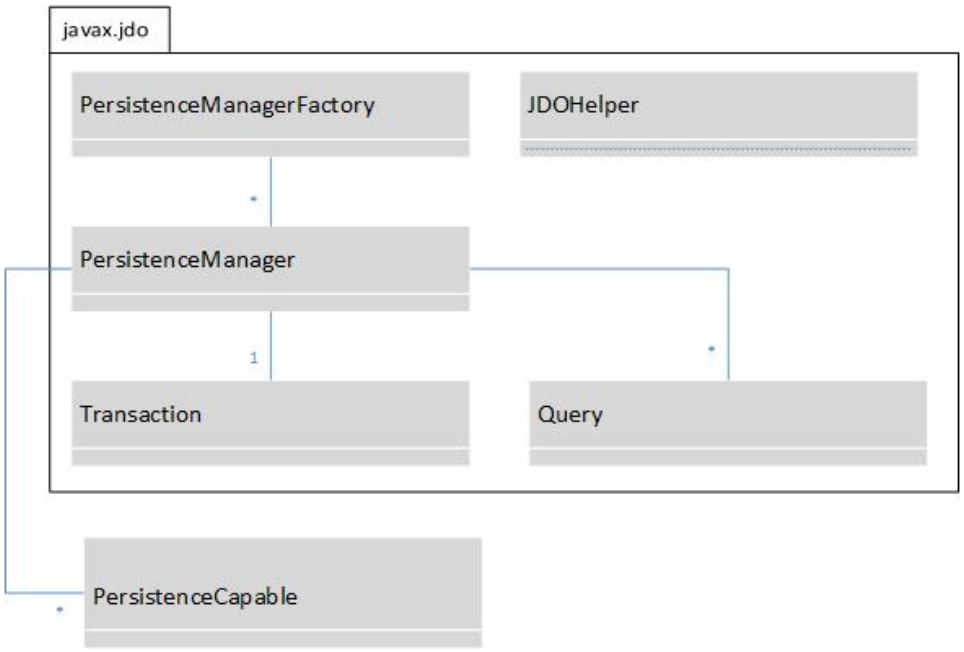

Abbildung 2.1: Architektur JDO Quelle: Developers Guide for Kodo [orac]; leicht modifiziert

PersistenceCapable
Jedes Objekt, welches gespeichert werden soll, muss das Interface PersistenceCapable implementieren. Dafür gibt es drei Möglichkeiten: direkt im Sourcecode, über einen Enhancer oder automatisch generiert durch Unterstützung von Programmen.

Transaction
Alle persistenten Operationen geschehen innerhalb einer Transaktion. Dabei werden mehrere Aktionen (erzeugen, lesen, schreiben) innerhalb einer Transaktion zusammengefasst, sodass diese entweder alle zusammen ausgeführt werden können oder durch Konflikte fehlschlagen. Damit ist die Integrität der Daten gewährleistet.

Query
JDO definiert eine eigene Query Sprache, nämlich die Java Data Objects Query Language (JDOQL). Es arbeitet im Gegensatz zu SQL nicht direkt auf den Tabellen der Datenbanken, sondern auf den Java Objekten [1]. Listing 2.1 zeigt beispielhaft die Erstellung einer Query mit JDO.

[1] siehe Kapitel 2.4.2 Query

```
Class target= Konto.class;
Extent extent = pm.getExtent (target, false);
String filter = "getKontonummer()>= 1 && getKontonummer() <= 10";
Query query = pm.newQuery (extent, filter);
Collection result = (Collection) query.execute ();
```

Listing 2.1: Beispiel zur Erstellung einer Query

Zustände eines Objektes
Die Sun Spezifikation enthält zehn verschiedene Zustände (vgl. [kod03, S. 33ff.]) für persistente Objekte. Davon sind sieben Pflicht bei einer Implementierung und drei optional. Die sieben Pflichtzustände sind:

- Transient: Ein Objekt ist transient, wenn es als persistent gekennzeichnet wurde, aber noch nicht persistent gemacht wurde. Dies geschieht zum Beispiel durch den Aufruf makePersistent() mit dem transienten Objekt als Argument. Damit wechselt es in den Zustand Persistent-New.
- Persistent-New: Ist der Zustand eines Objektes, welches innerhalb einer Transaktion das erste mal als Persistent gekennzeichnet wird.
- Persistent-Dirty : Wenn sich mindestens ein Attribut des Objektes in der aktuellen Transaktion geändert hat, so wechselt es in den Zustand Persistent-dirty.
- Hollow: Wenn die Werte eines Objektes noch nicht aus der Datenbank geladen wurden, befindet sich das Objekt im Zustand Hollow. Erst beim Zugriff auf das Objekt werden die Daten geladen. Dies ist auch bekannt als *Lazy Loading*.
- Persistent-clean: Ist der Zustand eines Objektes, solange sich keines der Attribute verändert.
- Persistent-deleted: Beschreibt Objekte, die während der Transaktion gelöscht wurden.
- Persistent-new-deleted: Wird ein Objekt innerhalb einer einzelnen Transaktion persistiert und gelöscht, dann befindet es sich in diesem Zustand. In diesem Zustand kann lediglich auf den Primärschlüssel des Feldes zugegriffen werden.

Enhancer
Enhancer bearbeiten, wie Abbildung 2.2 zeigt, die vom Compiler erstellte Klasse. Dieser Prozess wird enhancen oder auch *Byte Code Weaving* genannt. Der JDO Enhancer fügt dabei den benötigten Code zu

den persistenten Klassen hinzu. Somit müssen sich Entwickler nicht direkt mit dem PersistenceCapable Interface beschäftigen.

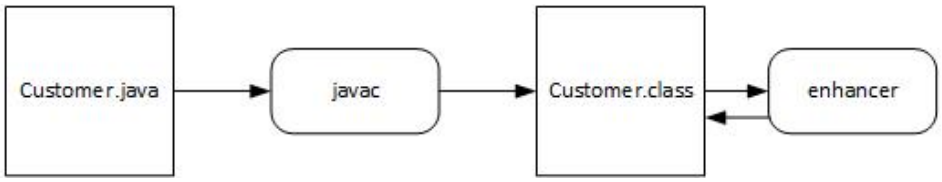

Abbildung 2.2: JDO Enhancer Quelle: Developers Guide for Kodo [orad]; leicht modifiziert

Einschränkungen von JDO
JDO unterstützt Vererbung komplett, gibt aber einige Einschränkungen bei bestimmten Vererbungen vor.

1. Persistente Klassen können nicht von bestimmten Systemklassen erben. Dazu gehören zum Beispiel java.net.Socket oder java.lang.Thread.

2. JDO kümmert sich um den Zustand von allen persistenten Felder. Dies beinhaltet das Laden des Wertes vor dem Lesen und das Kennzeichnen des Wertes als dirty nach dem Bearbeiten. Dabei gilt die Ausnahme, dass private oder static Felder nicht persistiert werden können.

3. Arrays werden nur begrenzt unterstützt. Setzt man den Wert eines Index, so muss man dies JDO direkt mitteilen zum Beispiel durch die Methode JDOHelper.makeDirty().

Lifecycle Callbacks
Möchte man zur Laufzeit Veränderungen an persistenten Klassen machen, so müssen diese entweder eines der Interfaces des InstanceCallbacks oder einen der Listener der InstanceLifecycleListener implementieren. Die Interfaces des InstanceCallbacks bieten vier Methoden an:

- Das Interface LoadCallBack bietet die Methode jdoPostLoad an. Das bedeutet, dass nach dem Speichern der persistenten Felder die Klasse benachrichtigt wird und weitere Aktionen ausführen kann. Dies wird typischerweise genutzt, um Felder zu initialisieren, welche von persistenten Feldern abhängen.

- Das Interface StoreCallback bietet die Methode jdoPreStore an. Diese wird vor dem Speichern der persistenten Felder aufgerufen.

Der Verwendungszweck ist zum Beispiel das Schreiben von Werten aus nicht persistenten Feldern (Caches) in persistente Felder.

- ClearCallback ist ein weiteres Interface und stellt die Methode jdoPreClear zur Verfügung. Diese wird vor dem Leeren von persistenten Feldern aufgerufen und dient dem Zweck Objekte, welche zum Beispiel als Cache dienten, zu löschen oder null-Verweise zu entfernen.
- Das letzte Interface ist DeleteCallback mit seiner Methode jdoPreDelete. Sie wird vor dem Wechsel in den Zustand Persistent-deleted aufgerufen und dient zum Aufräumen nicht mehr genutzer Daten.

Der Nachteil ist, dass diese Interfaces nur von Persistent-Capable Klassen implementiert werden können. Was ist aber mit Klassen, die zum Beispiel loggen sollen? Dafür gibt es die Listener des InstanceLifecycleListener. Zu ihnen gehört eine Eventklasse, InstanceLifecycleEvent, für jeden Status eines persistenten Objektes und in Interface, InstanceLifecycleListener, als Oberklasse für alle einzelnen Interfaces zum Beispiel:

- LoadLifecycleListener,
- StoreLifecycleListener und
- ClearLifecycleListener.

2.3 JPA

2.3.1 Annotationen

Mit Hilfe von Annotationen werden Klassen innerhalb von JPA als Persistent gekennzeichnet. Dabei gibt es unterschiedliche Annotationen auf die ich im Folgenden detaillierter eingehen werde (vgl. [PDBM12, S. 30ff.].

Entity
Innerhalb von JPA werden persistente Klassen als Entity bezeichnet. Somit genügt es eine Klasse mit der Annotation @Entity zu versehen um sie in der Datenbank zu speichern.

```
@Entity
public class Customer{
        private String name;
        public String getName(){};
        public String setName(){};
}
```

Properties

Der Zustand einer Klasse wird über seine Eigenschaften beschrieben. In unserem Beispiel ist es der Name. Als Standard speichert JPA alle Eigenschaften eines persistenten Objektes in der Datenbank. Lediglich die Annotation @Transient kann dies verhindern. Wird ein Attribut oder die getter-Methode des Attributes mit der Annotation versehen, so wird diese Eigenschaft nicht persistiert.

```
@Transient
private String name;
public String getName(){};
```

oder

```
private String name;
@Transient
public String getName(){};
```

Dabei ist zu beachten, dass man innerhalb einer Klasse die Varianten nicht mischen sollte. Das heißt entweder die Attribute oder ihre getter-Methoden mit der Annotation versehen. Weitere häufig verwendete Annotationen, mit denen man Attribute versehen kann, sind unter anderem @Column und @ID. Column kann genutzt werden um den Namen der Spalte und ihre Länge anzugeben.

```
@Column{ name = "NACHNAME", length=20}
private String name;
public String getName(){};
```

Id gibt typischerweise an, dass dieses Attribut der Primärschlüssel der Klasse ist. Dabei ist es verpflichtend ein Attribut der Klasse mit dieser Annotation zu versehen.

Verweise

Eine weitere Kategorie von Annotationen sind Verweise. Sie geben die Beziehung zwischen zwei Klassen an. Dazu gehören zum Beispiel @ManyToOne und @OneToMany.

```
@Entity
public class Account{
        @ManyToOne
```

```
        private Customer customer;
}
```

Dies bedeutet für unser Beispiel, dass ein Kunde mehrere Konten haben kann und ein Konto immer zu einem Kunden gehört. Die Verweise besitzen jeweils noch eigene Attribute, um die Relationen genauer zu definieren. Mit Hilfe von Verweisen können Entitys auch Listen von Entitys als Attribute besitzen. In Java ist es eine Standard Collection, welche mit der Annotation @OneToMany gekennzeichnet wird.

```
@Entity
public class Customer{
        @OneToMany
        private Collection<Account> accounts;
}
```

Embeddable

Klassen können anstelle von Referenzen auch direkt in andere eingebettet werden. Dies geschieht mit Hilfe der Annotationen @Embeddable und @Embedded. Dieses Konzept kann angewendet werden, wenn die folgenden Bedingungen zutreffen (Klasse B soll in A eingebettet werden):

- Klasse A besitzt B - zum Beispiel, eine Person besitzt ein Bankkonto.
- Wenn Klasse A gelöscht wird, kann auch B gelöscht werden. Eine Person kündigt den Vertrag mit der Bank, also kann auch das Konto entfernt werden.
- Die Klasse B wird nicht geteilt. Das heißt ein Konto gehört eindeutig einer Person.

```
@Embeddable
public class Account{
        @Column(name="KONTONUMMER")
        private int accountNumber;
        @Column(name="KONTOSTAND")
        private int money;
}
```

Die Annotation @Embeddable signalisiert dem Framework, dass die Klasse in einer anderen Klasse eingebettet werden kann. Innerhalb dieser Klasse kann man für die Attribute ebenfalls die @Column Annotation zur Festlegung der Spaltenlänge oder dem Spaltennamen verwenden.

```
@Entity
@Table(name="KUNDE")
public class Customer{
        @Embedded
        private Account myAccount;

}
```

Jetzt wurde die Klasse *Account* in der Klasse Customer eingebettet. Betrachtet man die Speicherung innerhalb der Datenbank, so erstellt JPA die Tabellen in der Form, dass die Attribute der Klasse *Account* zur Tabelle der Klasse *Customer* hinzugefügt werden. Somit besitzt die Tabelle *Kunde* jetzt zwei weitere Einträge, nämlich *Kontonummer* und *Kontostand*. Eine Tabelle mit dem Namen *Konto* wird nicht angelegt.

Dies waren einige Beispiele verschiedener Arten von Annotationen um einen kleinen Einblick in die Funktionalitäten von JPA zu geben.

2.3.2 JPA API

Wir wissen jetzt, wie man seine Klassen mit Hilfe von Annotationen so bearbeitet, dass sie persistiert werden. Wie arbeiten wir jetzt allerdings mit unseren Klassen innerhalb unserer Anwendung? Dies geschieht mit Hilfe der JPA API (vgl. [PDBM12, S. 27ff.]). Dafür wird das Package javax.persistence verwendet. Dies beinhaltet alle folgenden Klassen. Zu allererst einmal die Klasse Entity. Sie signalisiert, welche Klassen letztendlich vom Framework persistiert werden müssen. Dies geschieht, wie zuvor bereits beschrieben, durch die Verwendung der Annotation @Entity. Eine weitere notwendige Klasse ist der Entity Manager. Er stellt dabei ein klassisches Manager Objekt dar, welches eine oder mehrere Entitys verwaltet. Die API des EntityManagers bietet dabei Möglichkeiten an, Entitys zu persistieren, vom Manager zu entfernen, danach durch Queries zu suchen und letztendlich sie zu löschen. Um seinen EntityManager bei Verwendung einer Java Standard Edition (JSE) zu erzeugen, benötigt man jedoch noch zwei weitere Klassen. Die Klasse Persistence ist eine Helperklasse, die ein Entity Manager Factory Objekt erzeugen kann. Mit Hilfe dieses Objektes ist es nun möglich einen Entity Manager zu erhalten. Das ganze wird mit folgendem Code etwas ersichtlicher.

```
EntityManagerFactory entityManagerFactory=Persistence.//
createEntityManagerFactory("PersistenceUnitName");
EntityManager entityManager = entityManagerFactory.//
createEntityManager();
```

Bei der Verwendung von Java Enterprise Editions (JEE) geht das ganze etwas einfacher. Dort wird die Erzeugung eines Entity Managers wieder über eine Annotation erledigt @Resource. Mit Hilfe dieser wird das Objekt durch Injektion erzeugt.

```
@Resource
private EntityManager entityManager;
```

Ein weiterer Baustein der API ist der Persistence Context. Sobald ein Entity Manager Objekt erstellt wird, wird es mit einem Persistence Context versehen. Dieser wird somit vom Entity Manager verwaltet und verwaltet seinerseits Entitys. Der Persistence Context speichert alle Veränderungen der Entity Objekte ab, sodass der Entity Manager letztendlich entscheiden kann, ob die Transaktion sauber war und somit durchgeführt werden kann. Treten Konflikte auf, so sorgt der Entity Manager für das Zurücksetzen der Entitys. Es existieren zwei verschiedene Ausprägungen des Persistence Kontext. Diese sind Transaction-Scoped Persistence Context und Extended Persistence Context. Der Unterschied zwischen ihnen, wie sich aus den Namen schon schließen lässt, ist die Zeit, wie lange sie existieren. Ein Transaction-Scoped Persistence Context existiert solange, wie die Transaktion läuft. Nach Abschluss der Transaktion wird der Kontext entfernt, wodurch weitere Änderungen an den Entity Objekten nicht persistiert würden. Dieser Kontext wird wieder durch Injektion über eine Annotation (@PersistenceContext) am Entity Manager gesetzt.

```
@PersistenceContext(name="PersistentUnitName")
private EntityManager entityManager;
```

Der Extended Persistence Context existiert über die Transaktion hinaus. Wodurch auch Änderungen außerhalb einer Transaktion persistiert werden können. Dieser Kontext muss jedoch manuell von den Entwicklern erzeugt und verwaltet werden.

Als nächstes folgen die Transaktionen (vgl. [PDBM12, S. 149ff.]). Diese werden durch die Klasse Transactions implementiert. Diese besitzt wieder zwei Ausprägungen, abhängig davon, wer diese verwaltet. Es gibt JTA (Java Transaction API) und Resource-lokal Transaktionen. Befindet man sich in einer JEE Anwendung, so handelt es sich standardmäßig um JTA Transaktionen. Dies bedeutet: das Framework verwaltet die Transaktionen. Benötigt eine Methode eine eigene Transaktion, so kann diese über eine Annotation (@RequiresNew oder @Requires) an der Methode erzeugt werden.

Innerhalb einer JSE Anwendung wird standardmäßig die Resource-lokal Transaktion verwendet. Das bedeutet, der Entwickler kümmert

sich um das Erzeugen einer Transaktion mittels Entity Manager, dem Ausführen der Transaktion, wenn alles in Ordnung war oder dem Rollback, falls es einen Konflikt gab. Dies sieht dann folgendermaßen aus:

```
EntityTransaction userTransaction = entityManager.//
getTransaction();
  try{
    userTransaction.begin();

     // Hier passiert die Veränderung.

    // Wenn alles in Ordnung war, erfolgt der commit.
    userTransaction.commit();
    }catch(Exception exception){
        // Wenn ein Fehler aufgetreten ist,
        // erfolgt der rollback ..
        userTransaction.rollback();
    }
```

Da wir nun einen guten Einblick in die JPA API erhalten haben, betrachten wir nun die Operationen die mit Entity Objekten vollzogen werden können (vgl. [PDBM12, S. 56ff.]). Die Persistierung von Entity Objekten erfolgt wieder einmal mit Hilfe unseres Entity Managers. Durch die Methode persist(EntityObject) wird das Objekt persistiert und anschließend auch vom Entity Manager verwaltet. Dabei erfolgt zusätzlich eine Überprüfung, ob dieses Objekt nicht bereits persistiert ist. Dies erfolgt über den Unique Identifier, also den Primärschlüssel des Objektes.

```
CustomerEntity customer= entityManager.//
find(CustomerEntity.class, "12345678");
 // customer-Objekt kann null sein.
 If (customer != null){
 // Wenn nicht, dann arbeiten wir mit dem Objekt.
 }
```

Möchte man nun nach Entity Objekten innerhalb seiner Datenbank suchen, gibt es dafür unterschiedliche Anfragetechniken. Dies geschieht wie immer über unseren Entity Manager. Die erste Methode ist *find(EntityObject, id)* und bekommt die Objektklasse, welche wir suchen und die ID des Objektes, nach dem wir suchen. Wenn das Objekt gefunden wird, wird es automatisch vom Entity Manager und vom Persistent Context verwaltet. Falls das Objekt nicht gefunden wird, wird *null* als Rückgabewert geliefert. Die zweite Methode ist *getReference(EntityObject, id)* und bekommt wieder dieselben Parameter. Der Unterschied ist jedoch, dass der Zustand des Objekts, wenn es gefunden wurde, nur lazy geladen wird. Wenn das Objekt nicht gefun-

den wurde, wird eine Exception geworfen. Die dritte Methode erfolgt durch Verwendung der Query API welche später noch detaillierter besprochen wird. Als nächstes erfolgt das Entfernen von Entity Objekten aus der Datenbank. Dies geschieht mit Hilfe der Methode *remove(EntityObject)*, falls das Objekt vom Entity Manager verwaltet wird. Zu beachten ist, dass das Objekt lediglich vom Persistence Context entfernt wird, es muss noch nicht aus der Datenbank gelöscht worden sein. Dies geschieht entweder implizit zu einem späteren Zeitpunkt oder explizit über die Methode *flush()*. Die Methoden *flush()* und *refresh()* sind zwei weitere wichtige Methoden. *Flush()* sorgt dafür, dass alle Änderungen am Entity Objekt in die Datenbank geschrieben werden und *refresh()* bewirkt die Rückwärtsrichtung. Es sorgt also dafür, dass alle Änderungen an den Datenbankobjekten direkt auf das Entity Objekt übertragen werden. Zu beachten ist, dass dabei Änderungen überschrieben werden können. Jetzt fehlt nur noch die Methode, wie ein Update der Entity Objekte erfolgt. Dies geschieht über die Methode *merge(EntityObject)*. Sie ist notwendig, wenn ein Objekt vom transaktionalen Kontext verwaltet wurde, aber nach dem Beenden der Transaktion weitere Änderungen am Objekt vorgenommen wurden. Diese Änderungen sind dem Persistence Context des Entity Manager erst einmal nicht bekannt. Jedoch durch den Aufruf der Methode *merge(EntityObject)* wird das Objekt wieder vom Persistence Context verwaltet. Somit sind für ihn wieder alle Änderungen am Objekt sichtbar.

2.4 Was muss migriert werden

2.4.1 Konfiguration

Die kodo.properties ist eine Key-Value-Datei und dient zur Konfiguration von kodo (vgl. [kod03, S. 140]). Sie enthält Informationen über die Datenbankverbindung, die zu persistierenden Klassen und weitere Konfigurationen. Sie dienen dazu das Verhalten von kodo für die eigene Infrastruktur anzupassen. Eine Beispiel kodo.properties Datei kann in Listing 2.2 betrachtet werden. Auf die einzelnen Konfigurationen wird in einem späteren Kapitel genauer eingegangen.

```
com.solarmetric.kodo.impl.jdbc.AutoReturnTimeout: 10
javax.jdo.PersistenceManagerFactoryClass: kodo.jdbc.runtime.
    JDBCPersistenceManagerFactory
javax.jdo.option.ConnectionDriverName: org.hsqldb.jdbcDriver
javax.jdo.option.ConnectionPassword:
javax.jdo.option.ConnectionURL: jdbc:hsqldb:mem:tmp
javax.jdo.option.ConnectionUserName: sa
javax.jdo.option.IgnoreCache: true
javax.jdo.option.Multithreaded: true
javax.jdo.option.NontransactionalRead: true
javax.jdo.option.NontransactionalWrite: false
javax.jdo.option.Optimistic: true
javax.jdo.option.RestoreValues: true
javax.jdo.option.RetainValues: true
kodo.ConnectionFactoryProperties: MaxActive=0, MaxWait=50000
kodo.FetchBatchSize: 10
kodo.PersistenceManagerImpl: CloseOnManagedCommit=true
kodo.jdbc.SequenceFactory: db-class

kodo.jdbc.DBDictionary: de.tolina.common.jdo.HSQLDictionary
kodo.jdbc.SynchronizeMappings: buildSchema
kodo.PersistentClasses: package.Rolle
kodo.Log: log4j
```

Listing 2.2: Konfigurationsdatei für kodo

2.4.2 Query

Die Erstellung von Queries erfolgt über die JAVA basierte Sprache JDOQL (vgl. [kod03, S. 76ff.]). Sie ermöglicht Funktionen wie Filtern, Limits setzen, Objekte Ordnen, Projektionen und Vereinigungen. Beim Schreiben von Queries mit JDOQL hat man Zugriff auf alle persistenten Felder einer Klasse und auf das Schlüsselwort this. Die Verwendung von Methoden ist bis auf wenige Ausnahmen nicht unterstützt. Diese Ausnahmen sind:

- Collection.contains und Collection.isEmpty
- Map.containsKey, Map.containsValue und Map.isEmpty
- String.toUpperCase, String.toLowerCase, String.startsWith, String.endsWith, String.matches (Es werden lediglich .,.* und (?i) als reguläre Ausdrücke unterstützt), String.indexOf und String.substring
- Math.abs und Math.sqrt
- JDOHelper.getObjectId

An den zwei folgenden Beispielen werden die einzelnen Funktionen anschaulicher dargestellt.

Beispiel 1:

Query query = pm.newQuery ();	Query erstellen
query.setClass (Bank.class);	Die Klasse, auf die die Query angewendet werden soll.
query.setFilter ("customers > 1000");	Definierung eines Filters.
List results = (List) query.execute ();	Query ausführen und das Ergebis speichern.

JDOQL ermöglicht zusätzlich die Schreibweise einer Query als Single String. Diese sieht für das oben genannte Beispiel wie folgt aus:

```
Query query = pm.newQuery (Bank.class, "where customers > 1000");
List results = (List) query.execute ();
```

Beispiel 2:

Query query = pm.newQuery ();	Query erstellen
query.setResult ("customer.name as customer, money as money");	Definierung der Rückgabewerte, wenn man nicht das komplette Objekt benötigt.
query.setClass (Bank.class);	Die Klasse, auf die die Query angewendet wird.
query.setGrouping ("customer having money < :p");	Eine Gruppierung definieren.
query.setOrdering ("customer.name ascending");	Eine Ordnung definieren.
query.setRange (3, 20);	Den Bereich der Ergebnisse festlegen.
List results = (List) query.execute (new Double (500.0));	Die Query ausführen mit übergabeparameter für das Gruppieren.

Als Single String sieht das Beispiel wie folgt aus.

```
Query query = pm.newQuery ("customer.name as customer, money as money"
    + "group by customer having money < :p"
    + "order by customer.name ascending range 3,Long.MAX_VALUE");
List results = (List) query.execute (new Double (500.0));
```

2.4.3 Metadaten

JDO benötigt aus drei Gründen zu jeder persistenten Klasse Metadaten (vgl. [kod03, S. 27ff.]).

1. Um zu ermitteln, welche Klassen überhaupt persistiert werden müssen.
2. Um das Standardverhalten von JDO gegebenfalls zu verändern.
3. Um der JDO Implementation zusätzliche Informationen zur Verfügung zu stellen, welche durch reine Reflection der Klasse nicht erlangt werden können.

Die Metadaten werden in der eXtensible Markup Language (XML) geschrieben und müssen bestimmten Namenskonventionen folgen, damit die JDO Implementation sie über den Classloader finden kann. Entweder müssen die Metadaten in einer Datei mit dem Namen class-name.jdo stehen und im selben Package liegen, wobei class-name der Name der Klasse ist, auf welche sich die Datei bezieht. Oder die Informationen stehen in einer Datei mit dem Namen package.jdo, wobei package wieder der Name des Packages ist, in dem sich die Klassen befinden.

Ein komplettes Beispiel einer JDO Datei ist in Listing 2.3 zu sehen

Kommen wir nun zum Aufbau des Dokuments. Das Wurzelelement ist typischerweise das *<jdo>* Element. Die einzigen Kindelemente des *<jdo>* Elementes sind *<package>* Elemente. Diese müssen als Attribut mit dem Namen 'name' den vollständigen Namen des Packages angeben. Das *<package>* Element besitzt jetzt wiederum ein oder mehrere *<class>* Elemente und keine oder mehrere *<extension>* Elemente. Über die *<extension>* Elemente wird später noch etwas genauer gesprochen. Es folgen also die *<class>* Elemente. Jede persistente Klasse innerhalb der einzelnen Packages muss in dem entsprechenden *<package>* Element als *<class>* Element angegeben werden. Das *<class>* Element besitzt die folgenden Attribute:

- **name:** Dies ist ein Pflichtfeld und muss den Namen der Klasse enthalten. Es genügt der Name der Klasse, wenn diese in dem Package liegt, dessen *<package>* Element gerade beschrieben wird. Ansonsten benötigt man den vollständigen Klassennamen. Innere Klassen werden durch parent-class$inner-class beschrieben.

```
<jdo>
  <package name="package.checkliste">
    <class name="ChecklistenEintrag" >
      <extension vendor-name="kodo" key="jdbc-class-ind-name" value="
          metadata-value"/>
      <extension vendor-name="kodo" key="jdbc-class-ind-value" value="
          4400002"/>
    </class>
    <class name="ChecklistenReference">
      <extension vendor-name="kodo" key="jdbc-class-ind-name" value="
          metadata-value"/>
      <extension vendor-name="kodo" key="jdbc-class-ind-value" value="
          4400001"/>
      <extension vendor-name="kodo" key="detachable" value="true"/>
      <field name="checkliste">
        <collection element-type="ChecklistenEintrag"/>
        <extension vendor-name="kodo" key="element-dependent" value="true
            "/>
        <extension vendor-name="kodo" key="jdbc-field-map-name" value="
            one-many"/>
      </field>
    </class>
  </package>
</jdo>
```

Listing 2.3: Beispiel package.jdo Datei

- **persistence-capable-superclass:** Falls die Superclass dieser Klasse ebenfalls persistent ist und man will, dass JDO die Hierarchie kennt, dann kann der Name der Superclass angegeben werden.
- **identity-type:** Gibt den JDO Identitätstyp an. Gültige Werte sind: application, datastore und none. Der Standard ist datastore, wenn das Attribut objectid-class nicht gesetzt ist, ansonsten application.
- **objectid-class:** Wenn man application als identity-type verwendet, muss hier die Identitätsklasse für die persistente Klasse angegeben werden.
- **requires-extent:** Der Standardwert ist true. Falls man niemals diese Klasse abfragen will, kann man den Wert auf false setzen.

Das *<class>* Element selbst kann jetzt wieder *<extension>* Elemente und *<field>* Elemente besitzen. *<field>* Elemente repräsentieren die Felder der Klasse und sind optional. Wenn es für ein Feld einer Klasse kein *<field>* Element gibt, werden die folgenden Attribute mit Standardwerten versehen.

- **name:** Dieses Attribut ist ein Pflichtfeld und beinhaltet den Namen des entsprechenden Feldes
- **persistence-modifier:** Gibt an wie JDO dieses Feld verwalten soll. Es gibt dabei drei zulässige Werte:
  - persistent: Für Felder die persistiert werden sollen.
  - transactional: Für Felder, die bei einem Rollback mit angepasst werden sollen, aber nicht persistiert sein sollen.
  - none: Wenn JDO das Feld ignorieren soll.

  Für dieses Attribut gibt es jetzt unterschiedliche Standardwerte abhängig vom Feld:
  - Ist ein Feld static, final oder transient dann ist der Standardwert none
  - Alle primitiven Felder oder Wrapper von primitiven Feldern erhalten den Standardwert persistent.
  - Felder vom Typ String, Number, BigDecimal, BigInteger, Locale und Date haben ebenfalls den Wert persistent.
  - Vom Benutzer definierte persistence-capable Typen haben ebenfalls den Standardwert persistent

  - Alle Arrays von Typen, welche bisher erwähnt wurden haben den Wert persistent.
  - Die folgenden Container aus java.util haben ebenfalls den Wert persistent: Collection, Set, List, Map, ArrayList, HashMap, HashSet, Hashtable, LinkedList, TreeMap, TreeSet, Vector.
  - Alle übrig gebliebenen Felder erhalten den Wert none.

- **primary-key:** Erhält den Wert true, wenn die Klasse application identity verwendet und dieses Feld der Primärschlüssel sein soll. Der Standardwert ist false.

- **null-value:** Definiert das Verhalten von JDO bei Null-Werten. Der Standardwert ist none. Es gibt die Werte

  - none: JDO speichert null als Wert in der Tabelle, wenn der Wert null ist.
  - default: JDO versucht einen Standardwert für das Feld zu schreiben.
  - exception: Es wird eine Exception geworfen, wenn versucht wird null einzufügen.

- **default-fetch-group:** Alle Felder, die zur default-fetch-group gehören werden zusammen verwaltet und bei Aufrufen als Block geladen. Als Werte werden lediglich true und false akzeptiert. Das Standardverhalten ist, dass alle primitiven Datentypen, alle primitiven Wrapper und die Typen Date, BigDecimal, BigInteger, array, collection und alle Map Typen den Wert true erhalten. Der Rest erhält den Wert false.

- **embedded:** Gibt an, ob das Feld in der gleichen Tabelle gespeichert werden soll. Bei false erhält das Objekt eine eigene Tabelle. Das Standardverhalten ist das Gleiche, wie bei den default-fetch-groups.

Die einzelnen *<field>* Elemente können jetzt wieder *<extension>* Elemente haben. Felder vom Typ Array, Collection oder Map können zusätzlich ein *<array>*, *<collection>* oder *<map>* Element haben. Durch diese Elemente kann definiert werden ob die einzelnen Felder der Datentypen embedded sein sollen. Bei collection und map kann zusätzlich definiert werden, von welchem Typ die einzelnen Objekte bzw. Keys der Map sind.

2.4.4 Mapping

Die benötigten Informationen, um Klassen auf Datenbanktabellen zu mappen, werden ebenfalls in XML bereitgestellt (vgl. [kod03, S. 218ff.]. Sie ähneln vom Format her dem Aufbau der Metadaten-XML-Dateien. So gibt es wieder die Elemente *<package>*, *<class>* und *<field>* mit dem Pflichtattribut name. Das bleibt aber die einzige Gemeinsamkeit zu den Metadaten. Die Mapping Dateien bieten zusätzliche Informationen an. So erhält das *<class>* Element weitere Kind Elemente. Nämlich das Pflichtelement *<jdbc-class-map>* und die zwei optionalen Elemente *<jdbc-version-ind>* und *<jdbc-class-ind>*. Außerdem erhält das *<field>* Element ein Element *<jdbc-field-map>* zur Beschreibung des Mappings des Feldes. Diese Erweiterungen werden im folgenden detaillierter erklärt, da sie für die spätere Migration eine wichtige Rolle spielen. Ein Beispiel ist in Listing 2.4 dargestellt.

2.4.4.1 jdbc-class-map

Das Element besitzt das Pflichtattribut type, welches bestimmt, welche Mappingvariante verwendet wird. Es gibt base, flat, vertical, horizontal und custom.

- **Base:**

  Base-Class-Mapping kann nur für Klassen verwendet werden, welche nicht von anderen persistenten Klassen ableiten. Es besitzt die folgenden Attribute:

  <jdbc−class−map type=“base“table=“Customer“pk−column=“ID“/>

- **flat:** Wird zur Integration der Felder der persistenten Unterklasse in der Tabelle der Oberklasse genutzt. Als Attribut gibt es lediglich type, welches den Wert flat erhält.

  <jdbc−class−map type=“flat“/>

- **vertical:** Zur Speicherung von vererbten Feldern in der eigenen Tabelle.

  <jdbc−class−map type="vertical" table="Customer" ref−column.JDOID=
  "ACCOUNT_ID"/>

```
<mapping>
    <package name="package.checkliste">
        <class name="ChecklistenEintrag">
            <jdbc-class-map type="base" pk-column="JDOID" table="
                CHK_EINTRAG"/>
            <jdbc-version-ind type="version-number" column="
                JDOVERSION"/>
            <jdbc-class-ind type="metadata-value" column="JDOCLASS"/>
            <field name="comment">
                <jdbc-field-map type="value" column="COMMENT0"/>
            </field>
        </class>
        <class name="ChecklistenReference">
            <jdbc-class-map type="base" pk-column="JDOID" table="
                CHK_REFERENCE"/>
            <jdbc-version-ind type="version-number" column="
                JDOVERSION"/>
            <jdbc-class-ind type="metadata-value" column="JDOCLASS"/>
            <field name="checkliste">
                <jdbc-field-map type="one-many" ref-column.JDOID="
                    CHECKREFERENCE_JDOID" table="CHK_EINTRAG"/
                    >
            </field>
            <field name="_classId">
                 <jdbc-field-map type="value" column="CLASSID"/>
            </field>
            <field name="_objectId">
                 <jdbc-field-map type="value" column="OBJECTID"/>
             </field>
        </class>
    </package>
</mapping>
```

Listing 2.4: Beispiel package.mapping Datei

- **horizontal:** Sollen mehrere persistente Klassen ein Attribut verwenden, so definiert man dieses in einer Klasse, welche horizontal vererbt wird. Diese Klasse kann selbst nicht persistiert werden. Die abgeleiteten Klassen müssen alle Felder der Oberklasse mappen und persistieren.

  <jdbc−class−map type="horizontal"/>

- **custom:** Für die Verwendung eigener Mapper.

2.4.4.2 jdbc-version-ind

Ein Version-Indikator ist verantwortlich für die Versionierung von gespeicherten Objekten und hilft somit beim Aufdecken von Fehlern durch optimistisches Sperren. Dabei gibt es folgende unterschiedliche Typen:

- **version-number:** Ein Objekt erhält eine Nummer. Diese wird beim Lesen des Objektes mit geladen. Soll das Objekt wieder zurück geschrieben werden, erfolgt ein Vergleich der gelesenen Nummer mit der gespeicherten aus der Datenbank. Bei einem Konflikt gibt es ein Rollback.
- **version-date:** Im Prinzip das Gleiche, wie version-number. Jedoch handelt es sich bei dieser Nummer um einen Zeitstempel.
- **state-image:** Erzeugt beim Lesen ein Bild des Datenbankobjektes und überprüft beim Schreiben, ob dieses noch synchron ist.
- **custom:** Zur Verwendung von eigenen Versions Indikatoren.

2.4.4.3 jdbc-class-ind

Gibt für ein Datenbankobjekt an, um welche interne Klasse es sich handelt. Ist notwendig, wenn persistente Klassen voneinander ableiten. Dabei gibt es folgende unterschiedliche Typen:

- **in-class-name:** Der vollständige Name der Klasse wird in einer extra Spalte gespeichert.
- **metadata-value:** Jede persistente Klasse wird zu einer Konstante gemapped.

- **subclass-join:** Verwendet outer joins mit den Tabellen zu Subklassen, um zu ermitteln, ob sie von dieser Klasse abgeleitet sind. Dies geht jedoch nur, wenn die Datenbank outer joins unterstützt und wenn alle Klassen vertikal gemapped sind.
- **custom:** Für die Verwendung eigener Class Indikatoren.

2.4.4.4 jdbc-field-map

Der größte und für die Migration wichtigste Bereich ist das Mapping der Felder der Klassen auf die entsprechenden Attribute der Datenbanktabelle. Die Attribute für das Element sind die Folgenden:

- **type:** Gibt den Typen des FieldMappings an. Eine Liste mit Typen und kurzen Beschreibungen folgt anschließend.
- **column:** Gibt den Namen der Spalte an, welche den Wert beinhaltet und ist verpflichtend.

Die folgenden Attribute sind wichtig, wenn sich die Spalte in einer anderen Tabelle befindet.

- **table:** Der Name der Tabelle in der sich die Spalte befindet.
- **ref-column.<pk column>:** Beschreibt den Primärschlüssel der Klasse für die Relation. JDO benötigt die Informationen zum Joinen.
- **ref-constant.<column>:** Ähnlich dem *<ref-column>* Attribut, aber für den Fall, dass das Attribut eine Konstante ist.
- **ref-join-type:** Standartmäßig wird ein Inner Join durchgeführt. Möchte man einen Outer-Join durchführen müsste dieses Attribut auf outer gestellt werden.

Bei Verwendung von Relationen ändern sich folgende Attribute:

- Für 1:1 wird column zu column.<pk column> und gibt den Primärschlüssel der Tabelle an, damit JDO diese joinen kann.
- Für collection wird column zu element-column und gibt die Spalte an, welche die Werte der einzelnen Elemente enthält.

- Für n:n wird column zu element-column.<pk column> und gibt den Primärschlüssel des Objektes an, welches die Werte der einzelnen Elemente enthält.

Bei der Verwendung von Maps gibt es folgende Änderungen:

- column wird durch zwei neue Column Attribute ersetzt.
- key-column bzw. key-column.<pk column>: Gibt die Spalte der Schlüssel der Map an.
- value-column bzw value-column.<pk column>: Gibt die Spalte der Werte der Map an.

Abschließend die Liste der unterschiedlichen Fieldmapping-Typen:

- **value:** Zum direkten Mapping von primitiven Java Typen, Wrappern, Date und String.
- **blob:** Serialisiert den Wert des Feldes, welches als BLOB declariert wurde.
- **clob:** Für das Speichern längerer Strings muss der Typ clob verwendet werden.
- **byte-array:** Für das Mappen von Byte Arrays. Verwendet aber keine Serialisierung und speichert das Array direkt in die Datenbank.
- **one-one:** Für die Definierung eines 1:1 Mappings, das heißt ein Feld besitzt eine Referenz auf das Feld einer anderen Tabelle.
- **pc**[2]**:** Steht für Persistence-Capable 1:1 Mapping und wird verwendet, wenn ein Feld eine Referenz zu einem anderen unbekannten persistence-capable Feld besitzt
- **embedded:** Steht für embedded 1:1 Mapping und bedeutet, dass die Referenz des Feldes direkt als Spalte in diese Tabelle mit integriert wird. Bei Objekten werden alle Felder der Objekte integriert.
- **enum:** Speichert den Namen des enums als String in die Datenbank.

[2]Wird als Standardfall für eigene Interfaces benutzt, da bei diesen unklar ist um welche konkrete Klasse es sich handelt.

- **string-normalize:** Für das Speichern von Strings, welche vor dem Speichern oder Laden normalisiert werden sollen. Besitzt boolsche Attribute wie null-as-blank, blank-as-null, trim-leading, trim-trailing und pad zur Bearbeitung von Strings.
- **collection:** Zum Mappen von collections oder Arrays von primitiven Datentypen.
- **many-many:** Für das Speichern von Collections oder Arrays von persistenten Objekten.
- **one-many:** Für die Umsetzung einer 1:N Relation.
- **pc-collection:** Für Felder, welche ein Array oder eine Collection von unbekannten persistence-capable Objekten besitzen.
- **map:** Für das Speichern von Maps mit simplen Schlüssel-Wert Paaren.
- **n-many-map:** Wird verwendet, wenn die Schlüssel einen simplen Typ haben und die Werte persistence-capable Objekte sind.
- **many-n-map:** Die Inverse Variante von n-many-map.
- **many-many-map:** Schlüssel und Werte sind persistence-capable Objekte.
- **pc-map:** Schlüssel und Werte sind unbekannte persistence capable Objekte.
- **n-pc-map:** Schlüssel sind Objekte simplen Typs und die Werte sind unbekannte persistence-capable Objekte.
- **pc-n-map:** Die inverse Alternative zu n-pc-map.
- **pc-many-map:** Die Schlüssel sind unbekannte persistence-capable Objekte und die Werte sind persistence-capable Objekte.
- **many-pc-map:** Die inverse Variante von pc-many-map.

Kapitel 3

Die Migration

3.1 Warum jetzt? JPA 1.0 vs JPA 2.0

In der Einleitung wurde bereits die Grundmotivation für die Migration beschrieben. Warum ist man nun nicht schon früher migriert, wenn das Ziel mit JPA schon klar war? Erst durch die Entwicklung von JPA 2.0 ist der Standard für die tolina GmbH wirklich interessant geworden (vgl. [Kei09] und [wik10]). Es gab Änderungen in den Bereichen Properties, Mapping, Listen, Embedded Klassen, abgeleitete Primärschlüssel, Sperren, API und JPQL. Die einzelnen Punkte werden im Folgenden etwas genauer betrachtet. Es wurden weitere Properties der persistence.xml vereinheitlicht. So war es mit JPA 1.0 noch notwendig, für jeden Anbieter (Hibernate, EclipseLink, usw.) eigene JDBC Properties zu definieren, wodurch diese unnötig dupliziert wurden. Mit JPA 2.0 ist dies vereinheitlicht worden, sodass es nur noch einen JDBC Block mit folgenden vier Attributen

- javax.persistence.jdbc.driver,
- javax.persistence.jdbc.url,
- javax.persistence.jdbc.user und
- javax.persistence.jdbc.password gibt.

Mappings wurden erweitert, sodass es neue Möglichkeiten gibt Relationen zwischen Objekten herzustellen. Es ist nicht mehr notwendig beim Gebrauch einer @OneToMany Relation die Annotaion @JoinTable zu verwenden. Es genügt, wenn die Tabelle mit Hilfe von @JoinColumn eine Referenz auf den Fremdschlüssel erhält und diesen aktualisieren

kann. Eine weitere Änderung ist die leichtere Handhabung von Listen. Es wurde eine zusätzliche Annotation @ElementCollection eingeführt. Diese ermöglicht es, eine 1:N Relation mit einer eingebetteten Klasse oder einen Basistyp (zum Beispiel eine Liste von Strings) umzusetzen, ohne dieser Klasse ein ID Attribut zu geben oder eine inverse N:1 Relation zu definieren. Diese Objekte sind in ihrer Gebrauchsweise jedoch limitiert, da sie direkt an die Entitäten gekoppelt sind und somit nicht unabhängig persistiert oder gemerged werden können. Weiterhin wurden die Funktionalitäten von Eingebetteten Klassen erweitert. Es ist jetzt möglich diese zu schachteln und sie können Relationen definieren. Das Erstellen von Primärschlüsseln aus Relationen war mit JPA 1.0 ziemlich komplex. Es musste zuerst ein Attribut erstellt werden, welches den Primärschlüssel der Referenz darstellt. Dieses Attribut musste dann selbst mit Hilfe von @Id zur Liste der Primärschlüssel hinzugefügt werden. Mit JPA 2.0 können die Relationen selbst als Primärschlüssel definiert werden. Dabei muss einiges beachtet werden. Ist die ID der Relation der einzige Primärschlüssel der Klasse, dann wird dies direkt die ID. Ist die ID zusammengesetzt, so enthält die IDClass die Werte der Primärschlüssel und eine Referenz auf das Zielobjekt. Besitzt das Zielobjekt (in unserem Beispiel der Kunde) ebenfalls eine zusammengesetzte ID, dann enthält die IDClass der Quellklasse (in unserem Beispiel das Konto) die IDClass der Zielklasse.

JPA 1.0

```
public class Konto{
@Id int kontonummer;
@Column(name=”KUNDE_ID”)
@Id int kundennummer;

@ManyToOne
@JoinColumn(name="KUNDE_ID",insertable=false, updatable=false);
Kunde kunde;
}
```

JPA 2.0

```
@IdClass(KontoPK.class)
public class Konto{

@Id int kontonummer;

@Id @ManyToOne
Kunde kunde;
}
```

```
public class KontoPK{
int kontonummer;
int kunde;}
```

Weiterhin beinhaltete JPA 1.0 nur optimistisches Locking. Mit JPA 2.0 gibt es zusätzlich noch ein pessimistisches Locking mit einigen neuen Modi:

- OPTIMISTIC
- OPTIMISTIC_FORCE_INCREMENT
- PESSIMISTIC_READ
- PESSIMISTIC_WRITE
- PESSIMISTIC_FORCE_INCREMENT

Zusätzlich wurde die API erweitert, so hat zum Beispiel der EntityManager neue Funktionen erhalten:

- bei find und refresh kann man zusätzlich einen LockMode mit angeben
- bei find, refresh und lock kann man zusätzlich Properties übergeben
- neue Methoden:
  - void detach(Object entity)
  - <T> T unwrap(Class<T> cls)
  - getEntityManagerFactory()
  - Set<String> getSupportedProperties()
  - Map getProperties()
  - LockModeType getLockMode(Object entity)

Die Methode detach wurde bereits im JDO Kontext sehr oft verwendet, wodurch sie ein *must have* für die Vorraussetzung der Migration war. Außerdem wurde die Syntax der JPQL erweitert um die SQL Anfragen zu optimieren. Die Criteria API wurde hinzugefügt, welche das Erstellen von dynamischen Queries erleichtern soll. Wollte man bisher über JPQL dynamisch Queries erzeugen, so mussten Strings konkateniert werden. Dies ist aber sehr fehleranfällig, da die Queries erst zur Laufzeit geprüft werden können. In Anbetracht aller dieser Änderungen

muss man feststellen, dass der Funktionsumfang von JPA in der Version 1.0 zu klein war um kodo JDO zu ersetzen. Durch den erweiterten Umfang der Version 2 ist der Umstieg möglich.

3.2 Kodo Extensions - Interview mit dem technischen Leiter

Kodo Extension sind produktspezifische zusätzliche Features, welche nicht durch die JDO Spezifikation beschrieben werden. Dafür wurde eine Liste aller verfügbaren Extensions (vgl. [kod03, S. 295ff.]) zusammengestellt. Diese Liste wurde anschließend in einem Interview mit dem technischen Leiter der tolina GmbH besprochen, um einen ersten Überblick zu erhalten, welche dieser Funktionen verwendet werden. Dabei entstand die folgende Übersicht.

Name der Erweiterung	**Beschreibung**	**Verwendet oder nicht**
KodoHelper	Erweiterung des JDOHelpers z. B. zum Erstellen einer PersistenceManagerFactory aus einer Datei oder einem Stream.	wird verwendet
KodoPersistence ManagerFactory	Erweitert die standard PersistenceManagerFactory um kodo spezifische Features.	wird verwendet
KodoPersistence Manager	Erweitert den Standard PersistenceManager und bietet einem die Möglichkeit alle Features von Kodo zu verwenden. Zusätzlich erlaubt es einem den Zugriff auf Funktionen, welche erst ab JDO 2.0 zur Spezifikation gehören.	wird verwendet
JDOTransactionEvents	Mit Hilfe des KodoPersistenceManagers kann man einen kodo.event.TransactionListener registrieren. Dieser schickt Nachrichten, wenn eine Transaktion beginnt, zurückgesetzt wird, durchgeführt wird usw.	wird nicht verwendet
JDO 2.0 Preview Feature	getObjectsById()	wird verwendet

JDO 2.0 Preview Feature	attach/detach()	wird verwendet
JDO 2.0 Preview Feature	add/removeLifecycleListener	wird verwendet
JDO 2.0 Preview Feature	refreshAll (JDOException)	wird verwendet
JDO 2.0 Preview Feature	flush()	wird nicht verwendet
JDO 2.0 Preview Feature	checkConsistency()	wird nicht verwendet
JDO 2.0 Preview Feature	put/remove/getUserObject()	wird nicht verwendet
JDO 2.0 Preview Feature	setRollbackOnly()	wird nicht verwendet
JDO 2.0 Preview Feature	getObjectsById()	wird verwendet
Lifecycle Events	Ist ebenfalls ein JDO 2.0 Preview Feature und bietet einem die Möglichkeit, auf Zustandsänderungen von persistenten Objekten zu reagieren.	wird verwendet
PersistenceManager Extensions	Kodo ermöglicht es einem, seinen eigenen PersistenceManager zu implementieren.	wird nicht verwendet
KodoExtent	Erweitert die Standard Extent Klasse.	wird nicht verwendet
KodoQuery	Bietet einem viele Funktionen von JDO 2.0 und weitere kodo spezifische Features.	wird verwendet
FetchKonfiguration	Dient zur Konfiguration des Verhaltens beim Laden von Objekten. Die Einstellungen werden von der PersistenceManagerFactory zu allen erstellten Objekten vererbt.	wird verwendet

Query Extensions	Erweitert die JDOQL um weitere Funktionen. • GetColumn() • sql() Man kann eigene JDOQL Extensions schreiben, indem man das Interface kodo.jdbc.query.JDBCFilterListener verwendet. Außerdem ermöglichen sie einem eigene Query Aggregates durch das Interface kodo.jdbc.query.JDBCAggregateListener zu schreiben.	wird verwendet
Object Locking	Erweitert die die Möglichkeiten mit Locks zu arbeiten. • kodo.ReadLockLevel • kodo.WriteLockLevel • kodo.LockTimeout	wird nicht verwendet
Savepoints	Erweiteret den PersistenceManager um Funktionen zum Arbeiten mit Savepoints • void setSavepoint (String name) • void releaseSavepoint (String name) • void rollbackToSavepoint (String name)	wird nicht verwendet
Orphaned Keys	Zum Konfigurieren, was mit unbrauchbaren Fremdschlüsseln gemacht werden soll.	wird nicht verwendet

Metadata Extensions	Erweiterungen für die Definition der Metainformationen der persistenten Klassen (siehe Kapitel 2.4.2 Metadaten) • Relation: – inverse-owner – dependent • Schema: – jdbc-size – jdbc-element size, • Object-Relational Mapping: – jdbc-class-map-name • weitere: – data-cache – lock-groups – detachable	wird nicht verwendet

Die Extensions KodoHelper, KodoPersistenceManagerFactory und KodoPersistenceManager müssen verwendet werden, um überhaupt auf die Funktionalitäten von Kodo zugreifen zu können. Sie bieten aber keinerlei Mehrwehrtfunktionen an und müssen deshalb beim Evaluieren nicht weiter betrachtet werden.

Die Restlichen wurden anschließend in zwei Kategorien eingeteilt:

- Muss mit JPA ebenfalls funktionieren (*must have*)
- Sollte mit JPA ebenfalls funktionieren (*should have*)

Zur ersten Kategorie gehören die Query Extensions, Lifecycle Events und die attach/detach() Methode der Preview Features. Zur zweiten Kategorie gehören demnach die restlichen verwendeten Preview Features, KodoQuery und Fetch Konfiguration. Im weiteren Verlauf des Interviews sind zusätzliche Anforderungen entstanden. Deshalb wurden die beiden Kategorien mit weiteren Anforderungen aufgefüllt. Es

kommen Query SubSelects zur ersten Kategorie hinzu. Zur zweiten kommen Enhance und die Eigenschaft flushBeforeQueries hinzu.

3.3 Architektur

Die Architektur der einzelnen Produkte ergibt sich durch die Komponentenbasierte Entwicklung (vgl. Kapitel 2.1 „Persistenz, Query und Co“). Die Darstellung kann der Abbildung 3.1 entnommen werden.

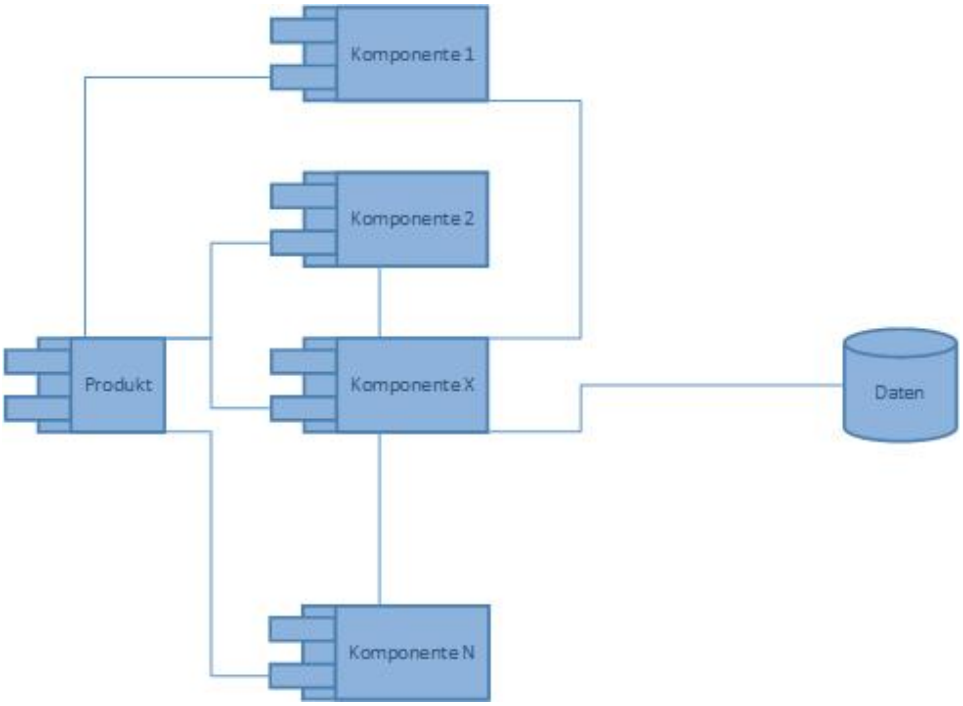

Abbildung 3.1: Architektur eines Produktes; Quelle: eigene Erhebung

Ein Produkt besteht immer aus einer Menge von Komponenten, welche die Basis der Funktionalität bereitstellen. Zusätzlich besitzt jedes Produkt seinen spezifischen Aufsatz, der es letztendlich zu einem richtigen Produkt macht. Abbildung 3.1 stellt vereinfacht die Kommunikation innerhalb eines Produktes dar. Greift man aus dem Produkt oder aus einem der Komponenten auf die Datenbank zu, so wird der Aufruf an die Komponente X weitergeleitet. Diese kümmert sich mit Hilfe von JDO um den Zugriff auf die Datenbank. Dies beinhaltet zum Beispiel das Ausführen von Queries, das Bereitstellen von Transaktionen oder auch das Persistieren bzw. Entfernen von Objekten.

3.4 Migrationsstrategien in der Theorie

3.4.1 Vollständige Migration

Wie aus dem Kapitel Architektur Abbildung 3.1 zu entnehmen ist, existiert eine strikte Trennung zwischen den Produkten und Komponenten sowie dem Zugriff dieser auf die Datenbank. Damit ist es möglich, die JDO Schicht durch JPA auszutauschen und somit den Zugriff auf die Datenbak über JPA durchzuführen. Dies beinhaltet zusätzlich alle Metadaten und Mapping Dateien (insgesamt 190 JDO Dateien, Stand 22.04.2013) zu entfernen und die entsprechenden Klassen mit den notwendigen Annotationen zu versehen. Zusätzlich müssten alle Queries umgewandelt werden.

Vorteile:

- kein zusätzlicher Wartungsaufwand
- keine späteren Arbeiten

Nachteile:

- hohes Risiko (kein Fallback)
- Produkte am Ende ihres Lebenszyklusses müssen auch umgestellt werden

3.4.2 Hybride Migration

Die hybride Lösung bedeutet, dass alle notwendigen Mapping und Metadaten Dateien gescannt werden, damit die entsprechenden Klassen mit den Annotationen versehen werden können. Zusätzlich benötigt man ein Interface, welches von JDO und JPA abstrahiert und als Schnittstelle zur Kommunikation dient. Die Klassen wissen jetzt, ob sie über JDO oder JPA mit der Datenbank kommunizieren wollen (siehe Abbildung 3.2.

Vorteile:

- bei Problemen Switch zu JDO zurück
- es muss nicht alles umgestellt werden

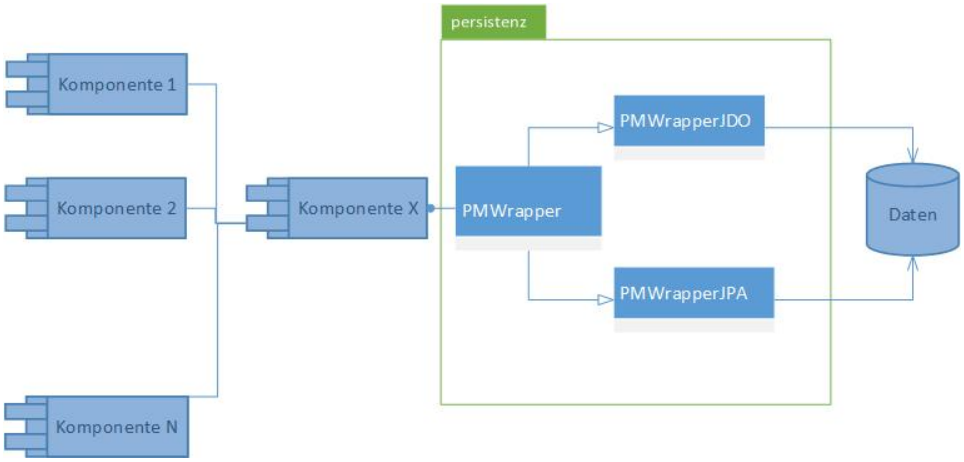

Abbildung 3.2: Schematische Darstellung einer möglichen hybriden Architektur; Quelle: eigene Erhebung

Nachteile:

- erhöhter Wartungsaufwand

3.4.3 Wahl der Migrationsstrategie

Im ersten Anlauf wird versucht die Hybride Migration durchzuführen. Der Vorteil einen Fallback zu haben wichtet gerade bei einem wirtschaftlich orientierten Unternehmen mehr, als der Nachteil des erhöhten Wartungsaufwandes. Somit wird die Migration anhand einer hybriden Migrationsstrategie beschrieben werden. Dabei ist im späteren Verlauf der Migration aufgefallen, dass es nicht möglich ist beide Frameworks zu vermischen[1], wodurch es für Neuentwicklungen notwendig ist zusätzlich zu JPA die für JDO benötigten Dateien zu erstellen. Dies führt zu einem doppelten Wartungsaufwand.

3.5 JPA Provider im Vergleich

Für die Wahl des Providers werden die größten drei in verschiedenen Kategorien miteinander verglichen. Dazu gehören Hibernate, EclipseLink und OpenJPA. Für jede Kategorie werden Punkte am Ende der Kategorie vergeben. Der Erste erhält 3 Punkte, der Zweite 2 und der

[1] vgl. Kapitel 3.10.1 Grenzen des Hybriden Ansatzes

Dritte 1. Die beiden Ersten werden anschließend in Kapitel 3.6 „Migration der Kodo Extensions“ anhand ihrer Kompatibilität zu den bereits verwendeten KodoExtensions bewertet.

3.5.1 Geschichte

Als erstes wird die Geschichte der Provider betrachtet, um sie anschließend miteinander zu vergleichen.

Hibernate wurde im Jahre 2001 entwickelt (vgl. [hisb]), um die Persistierung von Objekten zu vereinfachen. Bereits zwei Jahre später war die Entwicklung von Hibernate 2 abgeschlossen und beinhaltete viele neue Features. 2010 wurde dann Version 3.5 entwickelt, welche zusätzlich als zertifizierte Implementierung von JPA 2.0 anerkannt wurde. Wie bereits erwähnt, wurde JDO ebenfalls erst im Jahre 2003 standardisiert, weshalb Hibernate schon damals eine Alternative zu JDO war. Hibernate 5 wird momentan entwickelt und integriert JPA 2.1.

EclipseLink 1.0 wurde erst im Jahr 2008 veröffentlicht und zwar als Referenzimplementierung von JPA 2.0 (vgl. [hisa]. Ein Jahr später folgte Version 2.0 mit weiteren Implementierungen der JPA 2.0 Spezifikation. Momentan befindet es sich in der Version 2.4.

OpenJPA ist die dritte Implementierung und hat seine Wurzeln im Kodo JDO Projekt, welches im Jahr 2001 von Solarmetric entwickelt wurde (vgl. [hisc]. Wie bereits in der Einleitung erwähnt, wurde Solarmetric von Bea Systems aufgekauft, welche dann den Kern von Kodo an die Apache Software Foundation gespendet haben. Daraus wurde OpenJPA entwickelt, sodass sogar die Kodo Version 4.1 auf dem Code von OpenJPA basiert. OpenJPA 1.0 wurde im Jahr 2007 veröffentlicht und enthält Funktionen gemäß der JPA 1.0 Spezifikation. Im Jahr 2010 erfolgte dann erst die Version 2.0, welche die Funktionen gemäß der JPA 2.0 Spezifikation implementierte. Die aktuelle Version ist 2.2.1.

Somit sind Hibernate und OpenJPA die älteren und reiferen Provider. Beide besitzen Erfahrungen mit JDO. EclipseLink ist von der Entwicklung her deutlich neuer, dafür aber die Referenzimplementierung von JPA 2.0.

3 Punkte für OpenJPA, 2 Punkte für Hibernate, 1 Punkt für EclipseLink

3.5.2 Community

Als nächstes folgt die Betrachtung der Community. Das heißt zum Beispiel welche Kommunikationskanäle genutzt werden. Die Community von Hibernate nutzt dabei ein Forum, sowie Wiki, Mailing Lists, Chats, Blogs und Twitter. Als Issue Tracker wird Jira verwendet.

EclipseLink verwendet als Kanäle Wiki, Newsgroups, also ein Forum, Mailing Lists und Blogs. Als Issue Tracker wird der Standard Eclipse Issue Tracker Bugzilla verwendet.

OpenJPA verwendet lediglich MailingLists und als Issue Tracker ebenfalls Jira.

Wie man sieht, ist die Hibernate Community am breitesten gefächert, denn sie benutzt viele verschiedene Kommunikationskanäle. EclipseLink ist auch noch mit einer Vielzahl verschiedener Kanäle dabei. Lediglich OpenJPA scheint weniger Wert auf eine Vielzahl von Kommunikationskanälen zu legen. Man muss jedoch beachten, dass Hibernate und auch EclipseLink größere Projekte sind, welche eine Vielzahl von Komponenten zusätzlich zu JPA anbieten.

3 Punkte für Hibernate, 2 Punkte für EclipseLink, 1 Punkt für OpenJPA

3.5.3 Aktivität

Im Kapitel Aktivität wird betrachtet, wie lange Issues zur Bearbeitung brauchen und wie viele Issues eingestellt werden. Dabei werden nur die JPA Komponenten betrachtet.

Für den Issue Tracker wurde der Zeitraum vom 1.10.2012 bis zum 1.11.2012 betrachtet. Es wird vergleichen wie viele Tickets in diesem Zeitraum erstellt wurden und wieviele davon bis zum 30.11.2012 gelöst wurden.

In diesem Zeitraum wurde für Hibernate insgesamt ein Issue erstellt und ist bis zum 30.11.2012 noch ungelöst.

Bei EclipseLink wurden im gleichen Zeitraum 53 Issues erstellt. Drei Davon waren unpassend (Duplikat, kein EclipseLink Bug oder falsch), zwölf dieser Fehler sind bis zum 30.11.2012 behoben worden. Der Rest ist noch offen.

Bei OpenJPA wurden in diesem Zeitraum 19 Issues erstellt, von denen

fünf gelöst wurden und die restlichen 14 noch offen sind.

Somit ist die Aktivität bei EclipseLink am Höchsten, danach folgt OpenJPA und abschließend Hibernate.

3 Punkte EclipseLink, 2 Punkte OpenJPA, 1 Punkt Hibernate

3.5.4 Testprojekt

Mit Hilfe eines kleinen Testprojektes auf Maven Basis erfolgte ein weiterer Vergleich der drei Provider. Dabei handelt es sich um drei kleine Klassen. Zwischen Ihnen gibt es Relationen, sodass diese ebenfalls getestet werden können. In einer Methode sollen anschließend mehrere Objekte der Klassen erstellt und persistiert werden. Als Datenbank wurde Derby verwendet. Zuerst erfolgte die Umsetzung mit Hibernate. Dafür musste im ersten Schritt die Abhängigkeit zu Hibernate im POM registriert werden. Danach war lediglich die persistence.xml (siehe Listing 3.1) anzupassen, sodass die kleine Testmethode durchlief.

```
<persistence-unit name="JPATestProject" >
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
  <class>package.Kunde</class>
  <class>package.Konto</class>
  <class>package.Adresse</class>

  <properties>
  <property name="hibernate.dialect" value="org.hibernate.dialect.DerbyDialect
      " />

  <property name="javax.persistence.jdbc.driver" value="org.apache.derby.jdbc.
      ClientDriver" />
  <property name="javax.persistence.jdbc.url" value="jdbc:derby://
      localhost:1527/JPATest;create=true" />
  <property name="javax.persistence.jdbc.user" value="JPATest" />
  <property name="javax.persistence.jdbc.password" value="test" />
  </properties>
</persistence-unit>
```

Listing 3.1: persistence.xml für Hibernate

Der nächste Provider war OpenJPA. Es wurde wieder die entsprechende Abhängigkeit im POM hinzugefügt und anschließend die persistence.xml angepasst. Bei dem anschließenden Test trat ein Fehler auf, welcher die weitere Ausführung stoppte (siehe Listing 3.2).

```
This configuration disallows runtime optimization, but the following listed
    types were not enhanced at build time or at class load time with a
    javaagent: "
```

```
package.Kunde
package.Adresse
package.Konto".
```

Listing 3.2: Fehler bei der Ausführung mit OpenJPA

Das liegt daran, dass die Klassen nicht enhanced wurden. Das kommt daher (vgl.[Cur09]), dass die JPA Spezifikation vorsieht, dass Entity Klassen überwacht werden sollen. Die Spezifikation gibt aber nicht vor wie dies geschehen soll. Bei OpenJPA hat man sich dafür entschieden Byte Code Weaving (enhancen) zu verwenden. Das heißt, dass die Klassen vor der Ausführung des Tests noch enhanced werden müssen.

Der dritte Provider ist wie immer EclipseLink. Auch hier wurde die entsprechende Abhängigkeit im POM eingetragen. Leider gab es damit bereits Probleme, da das Repository nicht gefunden werden konnte. Nach einer Recherche hat sich herausgestellt, dass die momentane Version ein Problem mit Nexus hat.

> The current repository have some issues when used with Nexus [2] [ecl13].

Resultierend ergibt dies für Hibernate 3 Punkte, für OpenJPA 2 Punkte und für EclipseLink 1 Punkt.

3.5.5 Ergebnis

Übersichtshalber werden die Ergebnisse der Provider in den einzelnen Kategorien, sowie das Gesamtergebnis noch einmal tabellarisch dargestellt.

Kategorien	**Hibernate**	**OpenJPA**	**EclispeLink**
Geschichte	2	3	1
Community	3	1	2
Aktivität	1	2	3
Testprojekt	3	2	1
Ergebnis	9	8	7

Somit sind die Provider Hibernate und OpenJPA eine Runde weiter.

[2] Bei einem späteren Test war dieser Fehler behoben.

3.6 Migration der Kodo Extensions

Es folgt wieder eine tabellarische Übersicht der KodoExtensions und ihren äquivalenten in Hibernate bzw. OpenJPA. Die Funktionen der einzelnen Erweiterungen werden noch einmal kurz beschrieben.

Erweiterung	**Hibernate**	**OpenJPA**
getObjectsById() Das Objekt über die ID laden (vgl. Orcale JDO Doku, 8.8. JDO Identity Management).	Ist bereits in der JPA Spezifikation enthalten mit getReference(Class, ID) oder find(Class, ID) (vgl.[hib13a])	Siehe Hibernate.
attach/detach() Zum Entkoppeln und späteren Ankoppeln eines persistenten Objektes vom Persistent-Manager (vgl.[kod03, S. 306ff])	In der JPA Spezifikation enthalten durch die Methoden detach() und merge() (vgl. ObjectDB JPA Doku, Detached Entity Objects).	Siehe Hibernate.
Query SubSelects Ist das Verwenden von Queries in Queries. Select Kundennummer From Kunde Where 'Kontostand = Select max(Kunde.Kontostand)';	Die Query API von JPA ermöglicht das Ausführen von Subqueries durch das Schlüsselwort IN. Dies funktioniert sowohl dür dynamisches JPQL als auch für die Criteria API.	Siehe Hibernate.

add/removeLifecycleListener Fügt Listener zum Empfangen von Events von persitenten Objekten hinzu oder entfernt diese (vgl. [kod03, S. 20ff.]).	Die JPA Spezifikation definiert dies auf zwei verschiedene Arten (vgl. [hib13b]). Variante eins ist durch Hinzufügen der Annotation *@EntityListener(value=class)* an der Klasse. Der Value ist der Name der Klasse, welcher als Listener fungiert. Variante zwei ist das Hinzufügen einer der folgenden Annotation zu den Methoden der Klasse: • @PrePersist • @PreRemove • @PostPersist • @PostRemove • @PreUpdate • @PostUpdate • @PostUpdate	Siehe Hibernate
refreshAll (JDOException) Macht einen refresh für alle Objekte welche in dieser Exception enthalten sind.	Dies scheint ein spezielles Kodo Feature zu sein. Es wurde in keiner der Provider eine vergleichbare Funktion gefunden.	Siehe Hibernate
Lifecycle Events	siehe add/remove LifecycleListener()	siehe add/remove LifecycleListener()

KodoQuery Eine KodoQeury kann aus folgenden Teilen bestehen: • aggregates • projections • grouping • having • custom result classes • result ranges	Alle diese Methoden werden durch die JPA Spezifikation der JPQL definiert (vgl. [objb]). SELECT ... FROM ... [WHERE ...] [GROUP BY ... [HAVING ...]] [ORDER BY ...]	siehe Hibernate
FetchKonfiguration Ermöglicht das Konfigurieren von zusätzlichen Optionen beim Laden von Objekten. • *flushBeforeQueries()*	Die JPA Spezifikation bietet viele Möglichkeiten für das Konfigurieren von fetch Operationen. Der explizite Fall der flushBeforeQueries ist seit JPA 2.0 spezifiziert und benutzt dafür einen FlushModeType, welcher an einer Query gesetzt werden kann. Der Standard ist AUTO und stellt sicher, dass alle Objekte vor der Anwendung von Queries geflushed werden (vgl. [Nit10]).	siehe Hibernate

JDOQL Extensions • *getColumn()* Erlaubt das Setzen eines Filters anhand des Namens einer Tabelle und den Wert einer Zeile. `query.setFilter ("this.ext:getColumn ('LEGACY_DATA') == 'foo'");`	Über die Verwendung wurden keine expliziten Informationen gefunden.	siehe Hibernate
JDOQL Extensions • *sql()* Zur Integration von SQL Befehlen in der Filterklausel.	In Hibernate wird dies durch die Klasse Restrictions zur Verfügung gestellt. Mit Hilfe von *Restrictions.sqlRestriction()* ist es möglich SQL Statements innerhalb einer JPQL Query zu verwenden (vgl. [que]).	Es wurden keine Informationen zu dieser Funktionalität gefunden.

3.7 Umsetzung der hybriden Strategie - Architektur

Die Umstellung des Prototypen und die Ermittlung der fehlschlagenden JUnit Tests würde ohne einige Vorbereitungen zu keinen verwendbaren Aussagen führen. Alle Komponenten verwenden für den Datenbankzugriff eine zentrale Klasse PMWrapper. Diese muss im ersten Schritt für die Verwendung von JPA anstelle von JDO nachgebaut werden. Dafür wird ein Feature Branch von der entsprechenden Komponente erstellt.

Das Ziel ist die Entkernung des PMWrappers, um ihn als Delegator zu verwenden. Dieser besitzt als Attribute einen JDOPMWrapper und einen JPAPMWrapper (siehe Abbildung 3.3) und delegiert für ein übergebenes Objekt entweder an den JDOPMWrapper oder an den JPAPMWrapper. Dies ist der erste Schritt für die Umsetzung der Hybridlösung. Dafür benötigt man eine zentrale Klasse, welche keine

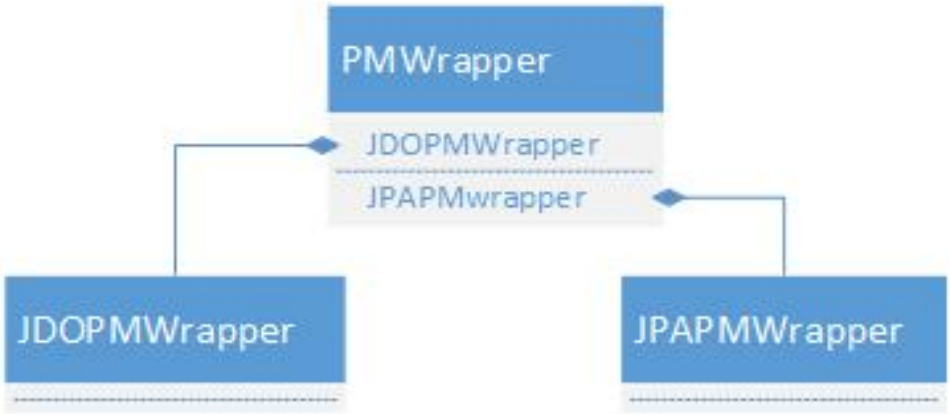

Abbildung 3.3: Anpassungen des PMWrappers; Quelle: eigene Erhebung

Abhängigkeiten zu JDO oder zu JPA besitzt. Sie bildet die Schnittstelle zur Kommunikation, wenn es um den Datenbankzugriff geht. Bei Anfragen muss sie entscheiden, ob eine Anfrage über JDO oder über JPA erfolgen soll. Wie erhält man jetzt solch eine Klasse?

1. Man hat, wie in dem oberen Beispiel beschrieben, bereits eine zentrale Klasse für den Zugriff über JDO. Diese kann anschließend verwendet werden, um die Anfragen an die entsprechenden Klassen weiter zu delegieren. Dafür erfordert es nicht einmal Code Anpassungen in den Produkten.

2. Gibt es eine solche Klasse noch nicht, wäre der erste Schritt die Erstellung eines Interface zur Ermittlung aller benötigten Methoden für den Datenbankzugriff. Anschließend implementiert eine neue Klasse dieses Interface und delegiert an die entsprechenden JDO beziehungsweise JPA Vertreter.

Dafür muss es im nächsten Schritt eine Funktionalität zur Bestimmung der Zugehörigkeit der einzelnen Objekte geben. Der PMWrapper muss ermitteln können, ob das Objekt oder die Klasse im JDO oder im JPA Kontext existiert. Ich habe mich dazu entschieden, zu überprüfen, ob eine Klasse im JPA Kontext existiert. Wenn dies nicht der Fall sein sollte, gehe ich davon aus, dass sie zum JDO Kontext gehört. Die Logik übernimmt die Methode aus Listing 3.3. Im ersten Schritt werden alle Entities aus dem Metamodel geholt. Anschließend wird mit Hilfe eines Predicate die Klasse der Entity mit der Klasse des übergebenen Objektes verglichen. Wird ein Objekt gefunden, bei dem eine Gleichheit auftritt, so gehört dieses Objekt zum JPA Kontext. Wird kein Objekt gefunden, so gehört dieses Objekt zum JDO Kontext.

```
public boolean isManagedByJPA(final Object object) {
  Set<EntityType<?>> entities = jpaPersistenceManager.getMetamodel().
      getEntities();
    Object find = CollectionUtils.find( entities , new Predicate() {
  @Override
  public boolean evaluate(Object collectionObject) {
    return ((EntityType<?>) collectionObject).getJavaType().equals(class1);
  }
    });
    if (find != null) {
  return true;
  }
    return false;
}
```

Listing 3.3: Methode welche bestimmt ob das Objekt vom EntityManager (JPA) verwaltet wird

Als nächster Schritt erfolgte die Überprüfung, ob man innerhalb einer Testmethode einen JDOPersistenceManager und einen EntityManager erzeugen kann, und mit Hilfe dieser beiden ein JDO und ein JPA Objekt in die gleiche Datenbank schreiben kann (vgl. Listing 3.4). Dafür wurde lediglich die Methode persist zu unserem bis dato fast leeren JPAPMWrapper hinzugefügt.

```
@Test
public void testIt() throws IOException {
  //JPA
  EntityManagerFactory factory = Persistence.createEntityManagerFactory("
      db_base");
  EntityManager entityManager = factory.createEntityManager();
  //JDO
  SmartProperties testdaten = new SmartProperties();
  testdaten.load("de/tolina/common/jpa/kodo.properties");
  PersistenceManager manager = KodoHelper.getPersistenceManagerFactory(
      testdaten).getPersistenceManager();
  //PMWrapper für beide
  PMWrapper pmwrapper = new PMWrapper(manager, entityManager);
  //Testobjekte
  MockJPAObject jpaObject = new MockJPAObject("test");
  MockJdoObject jdoObject = new MockJdoObject("JDOTest");



  //persistieren
  pmwrapper.persist(jpaObject);
  pmwrapper.persist(jdoObject);
}
```

Listing 3.4: Simpler Test des hybriden PMWrappers

Damit wurde gezeigt, dass es möglich ist JDO und JPA parallel laufen zu lassen und dass sie sogar auf die gleiche Datenbank zugreifen können. Als nächstes wird ein Prototyp benötigt an dem die Migration durchgeführt werden kann.

3.8 Die Wahl des Prototypen

Für die Wahl des Prototypen sind zwei Merkmale ausschlaggebend.

- Der Prototyp sollte einen Großteil der verwendeten JDO Funktionalitäten enthalten. Damit kann abschließend eine genaue Risikoabschätzung erstellt werden.
- Der Prototyp sollte eine sehr gute Testabdeckung besitzen. Dadurch können Fehler bei der Migration vielleicht schon durch JUnit Tests aufgedeckt werden.

Die Liste der möglichen Kandidaten ist in Tabelle 3.3 dragestellt.

Wie man sieht, kommen als Prototyp nur wenige Komponenten in Frage. Diese wurden in der Tabelle grün markiert. Betrachten wir diese etwas genauer.

tes_core ist momentan der Favorit, jedoch handelt es sich hierbei um eine Komponente, welche nur noch von wenigen Produkten genutzt wird und diese würden diese Abhängigkeit gerne entfernen. Damit läge der Wahl von tes_core als Prototyp lediglich der wissenschaftliche Nutzen zu Grunde. Die Umstellung dieser Komponente hätte keinen praktischen Nutzen für die Firma. Zusätzlich kann nicht gewährleistet werden, dass die genutzten JDO Funktionalitäten in tes_core ein allgemeines Set aus benutzten Funktionen darstellt.

Um sich zwischen den restlichen drei Favoriten zu entscheiden muss eine Gewichtung der Merkmale vorgenommen werden. So ist in meinen Augen die Anzahl der verwendeten JDO Funktionalitäten deutlich wichtiger als die Testabdeckung der Komponente. Damit fällt die Wahl als Prototyp auf application_base. Des Weiteren ist diese Komponente eine der zentralen Komponenten, welches jedes Produkt verwendet. Somit ist die Migration für die tolina GmbH von immenser Bedeutung. Sollte sich dabei heraus stellen, dass die verwendeten JDO Funktionalitäten trotz der deutlich höheren Anzahl an zur Verfügung stehenden Dateien nicht ausreichend genug abgedeckt sind, wird zusätzlich eine weitere Komponente prototypisch umgestellt.

Komponente	**Anzahl jdo Dateien**	**Anzahl Mapping Dateien**	**Testabdeckung in Prozent**
action_base	3	3	70,6
application_base	8	6	48,1
bo_base	1	0	34,3
browser_base	1	1	-
db_base	1	1	52,3
host_base	5	4	57,1
host_hundh	1	1	88,8
host_si_dyns	3	3	64,7
host_txbdepot	1	1	-
tes_core	11	11	45,0
util_base	2	0	68,6
util_genericfields	1	1	64,1
util_license	1	1	43,5
util_locking	1	1	53,7
util_pin	1	1	77,4
util_remoting_task	2	2	92,6
util_reporting	1	1	62,6
util_settings	2	2	68,9
util_workflow	2	2	65,0
zins_base	1	1	75,5
util_useradmin	2	2	58,8

Tabelle 3.3: Übersicht der möglichen Prototypen.

3.9 Migration anhand von Beispielen des Prototypen

3.9.1 Migration der Konfiguration

Wie bereits in Kapitel 2.4.1 „Konfiguration“ erwähnt handelt es sich bei den *kodo.properties* um eine Key-Value-Datei. Die persistence.xml ist das äquivalente JPA Gegenstück der kodo.properties. Sie dient ebenfalls zur Konfiguration von JPA, aber wird im XML Format gespeichert (vgl. Listing 3.5).

```
<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.
      com/xml/ns/persistence/persistence_2_0.xsd">
  <persistence-unit name="application_base">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <properties>
      <property name="hibernate.dialect" value="org.hibernate.dialect.
          DerbyDialect" />
      <property name="javax.persistence.jdbc.driver" value="org.apache.derby.
          jdbc.ClientDriver" />
      <property name="javax.persistence.jdbc.url" value="jdbc:derby://
          localhost:1527/JPATest;create=true" />
      <property name="javax.persistence.jdbc.user" value="JPATest" />
      <property name="javax.persistence.jdbc.password" value="test" />
    </properties>
  <!--Eine Liste aller persistenten Objekte -->
    <class>de.tolina.common.application.rollen.bo.Rolle</class>
    <exclude-unlisted-classes>true</exclude-unlisted-classes>

  </persistence-unit>
</persistence>
```

Listing 3.5: Konfigurationsdatei für JPA

Das Ziel ist es, für JPA die gleiche Konfiguration zu erhalten, wie sie für JDO durch die kodo.properties definiert wurde. Als Grundlage dient wieder das Beispiel aus Kapitel 2.4.1 „Konfiguration“, welches nochmal in Listing 3.6 dargestellt ist.

```
com.solarmetric.kodo.impl.jdbc.AutoReturnTimeout: 10
javax.jdo.PersistenceManagerFactoryClass: kodo.jdbc.runtime.
    JDBCPersistenceManagerFactory
javax.jdo.option.ConnectionDriverName: org.hsqldb.jdbcDriver
javax.jdo.option.ConnectionPassword:
javax.jdo.option.ConnectionURL: jdbc:hsqldb:mem:tmp
javax.jdo.option.ConnectionUserName: sa
```

```
javax.jdo.option.IgnoreCache: true
javax.jdo.option.Multithreaded: true
javax.jdo.option.NontransactionalRead: true
javax.jdo.option.NontransactionalWrite: false
javax.jdo.option.Optimistic: true
javax.jdo.option.RestoreValues: true
javax.jdo.option.RetainValues: true
kodo.ConnectionFactoryProperties: MaxActive=0, MaxWait=50000
kodo.FetchBatchSize: 10
kodo.PersistenceManagerImpl: CloseOnManagedCommit=true
kodo.jdbc.SequenceFactory: db-class

kodo.jdbc.DBDictionary: de.tolina.common.jdo.HSQLDictionary
kodo.jdbc.SynchronizeMappings: buildSchema
kodo.PersistentClasses: package.Rolle
kodo.Log: log4j
```

Listing 3.6: Konfigurationsdatei für kodo

Fangen wir also Schritt für Schritt damit an. Als Erstes werden die Einstellungen für die Verbindung zur Datenbank umgestellt. Dafür werden die einzelnen Key-Value Paare, durch die äquivalenten XML- Elemente ersetzt. Dies wird beispielhaft in Abbildung 3.4 für den Datenbanktreiber gezeigt. Nach diesem Muster werden alle weiteren Datenbankeinstellungen migriert. Der nächste Teil ist die Angabe der persistenten

JDO	javax.jdo.option.ConnectionDriverName:org.hsqldb. jdbcDriver </field>
JPA	<property name="javax.persistence.jdbc.driver" value=" org.apache.derby.jdbc.ClientDriver" />

Abbildung 3.4: Beispielhafte Migration der Datenbankkonfiguration

Klassen. Auch hier können die Werte aus der kodo.properties fast eins zu eins übernommen werden. Es entsteht eine Liste aus *<class>* Elementen. Zuletzt bleiben die zusätzlichen Einstellungen. Dafür wird für die am häufigsten verwendeten Einstellungen eine Liste erstellt. Alle nicht enthaltenen Einstellungen müssten in der Doku von JPA oder dem entsprechenden Provider nachgeschlagen werden. Viele der folgenden Konfigurationen für JDO wurden für JPA durch eigene Tests nachgewiesen, da oftmals keine expliziten Aussagen in der Spezifikation

von JPA oder in der Dokumentation von Hibernate gefunden wurden.

JDO/KODO	**JPA/Hibernate**
javax.jdo.option.Optimistic: **true** Definiert ob Transaktionen optimistisch vorgehen sollen. (vgl. [orab])	Der Standard sieht eine @Version Annotation an einem Attribut einer Klasse vor. Damit werden für diese Entität alle Transaktionen optimistisch arbeiten. Hibernate bietet zusätzlich eine Annotation an der Entität selber an, welche kein neues Attribut benötigt @org.hibernate.annotations.Entity(dynamicUpdate = **true**, optimisticLock = OptiticLockType.ALL) (vgl. [Whi11])
kodo.FlushBeforeQueries: **true** Gibt an, ob vor der Ausführung einer Query die lokalen Änderungen an persistenten Objekten in die Datenbank geschrieben werden sollen.	Wie bereits in Kapitel 3.6 „Migration der Kodo Extensions“ beschrieben ist der Standard in JPA, dass vor der Ausführung einer Query alle Änderungen zur Datenbank geschrieben werden. Konfigurierbar ist diese Eigenschaft durch eine Methode an der auszuführenden Query. createQuery.setFlushMode(FlushModeType)
javax.jdo.option.Multithreaded: **true** Gibt an, ob mehrere Threads zur gleichen Zeit auf die persistenten Klassen zugreifen können. Dient lediglich als Hinweis für den PersistenceManager.	Dieser Hinweis ist für JPA nicht notwendig. Deshalb gibt es dafür auch keine Konfigurationsmöglichkeit.

javax.jdo.option. NontransactionalRead: **true** Bestimmt, ob Werte aus persistenten Klassen auch ohne Transaktion gelesen werden können.	Der JPA Standard ist NonTransactionalRead, da alle persistenten Klassen beim zweiten Aufruf aus dem Cache geladen werden.
javax.jdo.option. NontransactionalWrite: **false** Bestimmt, ob Werte aus persistenten Klassen auch ohne Transaktion bearbeitet werden können.	Das Standardverhalten für JPA ist ebenfalls false.
javax.jdo.option.RestoreValues: **true** Gibt an, ob persistente Felder nach einem Rollback wieder auf den Zustand vor der Transaktion zurückgesetzt werden sollen.	Ein Test hat ergeben, dass der Standard für JPA true ist.
javax.jdo.option.RetainValues: **true** Gibt an, ob die persistenten Felder ihre Werte nach einem Commit behalten sollen.	Ein Test hat ergeben, dass der Standard für JPA true ist. Somit behalten alle Felder ihre Werte nach dem Commit.
kodo.ConnectionFactoryProperties: MaxCachedStatements=140, MaxActive=10, MaxWait=50000 Konfiguriert das Connection Pooling für die Datenbank.	Da es dafür in der JPA Spezifikation noch keine Beschreibung gibt, verwendet jeder Provider seine eigenen Varianten. So gibt es für Hibernate zum Beispiel c3po Connection-Pools. Diese sind konfigurierbar über die property: hibernate.c3p0 Jedoch ist es damit nicht möglich äquivalente Konfigurationen wie bei kodo vorzunehmen.

kodo.FetchBatchSize: 10 Gibt an, wie viele Ergebnisse in einem ResultSet liegen sollen.	Da es bei JPA und Hibernate keine ResultSet gibt, benötigt dieser auch keine Konfiguration der Größe. Eine ähnliche Konfiguration ist durch die Query Eigenschaft org.hibernate.fetchSize zu erreichen. Sie definiert die Anzahl der zu ladenden Zeilen bei einem Aufruf.
javax.jdo.option.IgnoreCache: **false** Gibt an, ob Änderungen, welche während der Transaktion an dem Objekt vorgenommen wurden, auch die auszuführende Query beeinflussen.	Nach einem Test zu urteilen, ist der JPA Standard false. Durch das Standardverhalten von FlushModeType.AUTO werden alle bestehenden Updates vor der Ausführung einer Query in die Datenbank geschrieben und beeinflussen somit die Query.

Die Gegenüberstellung hat uns gezeigt, dass eine Konfiguration von JPA in dem Maße, wie es für kodo möglich ist, nicht umgesetzt werden kann. Die ersten Tests mit einem richtigen Produkt werden Aufschluss über die Unterschiede in der Konfiguration bringen. Dadurch wird ersichtlich werden, welchen Einfluss diese auf den Betrieb des Produktes haben. Dazu gehören Performanceprobleme, wenn zum Beispiel das *NonTransactionalRead* nicht verwendet wird.

3.9.2 Migration der Metadaten

Dieses Kapitel beschreibt die Migration von den JDO Metadaten zu den JPA Annotationen. Die JDO Datei liegt üblicherweise in einem Package und definiert, welche dieser Klassen persistent sein sollen. Zusätzlich ermöglicht sie die Konfiguration der Klasse und deren Felder. Dafür wird das Beispiel aus Kapitel 2.4.3 „Metadaten“ verwendet. Dieses ist in Listing 3.7 nochmal zu sehen.

Schauen wir uns das Beispiel etwas genauer an, um die benötigten Informationen Schritt für Schritt zu extrahieren. Die erste Information betrifft das Package, in dem man sich befindet und beinhaltet dessen Namen. Darauf folgen die Informationen für die persistenten Klassen. Die erste Klasse heißt ChecklistenEintrag. Suchen wir also diese Klasse

```
<jdo>
  <package name="package.checkliste">
    <class name="ChecklistenEintrag" >
      <extension vendor-name="kodo" key="jdbc-class-ind-name" value="
          metadata-value"/>
      <extension vendor-name="kodo" key="jdbc-class-ind-value" value="
          4400002"/>
    </class>
    <class name="ChecklistenReference">
      <extension vendor-name="kodo" key="jdbc-class-ind-name" value="
          metadata-value"/>
      <extension vendor-name="kodo" key="jdbc-class-ind-value" value="
          4400001"/>
      <extension vendor-name="kodo" key="detachable" value="true"/>
      <field name="checkliste">
        <collection element-type="ChecklistenEintrag"/>
        <extension vendor-name="kodo" key="element-dependent" value="true
            "/>
        <extension vendor-name="kodo" key="jdbc-field-map-name" value="
            one-many"/>
      </field>
    </class>
  </package>
</jdo>
```

Listing 3.7: Beispiel package.jdo Datei

in dem Package und fügen die Annotation @Entity hinzu. Betrachten wir nun die beiden Elemente der Klasse. Sie gehören zusammen und geben an, dass die Klasse als Konstante unter dem Wert 4400002 in einer Spalte gespeichert wird. Dies ist notwendig, wenn andere Klassen von dieser ableiten können sollen. Für JPA gibt es für diesen Zweck die Annotation @DiscriminatorColumn und @DiscriminatorValue. Man definiert über die Column in der Oberklasse unter welchem Spaltennamen die einzelnen Values auftauchen sollen. Mit Hilfe der Annotation @DiscriminatorValue bestimmt man den Wert, welcher in der Spalte stehen soll. Dabei brauch die Annotation @DiscriminatorColumn nur in der höchsten Klasse der Hierarchie definiert werden. Damit ist die Klasse fertig umgestellt (vgl Listing 3.8).

```
@Entity
@DiscriminatorColumn(name="ChecklistenTyp")
@DiscriminatorValue(value = "ChecklistenEintrag")
public class ChecklistenEintrag{
 ...
}
```

Listing 3.8: CheckListEneintrag Metadaten hinzugefügt

Betrachten wir nun die zweite Klasse mit dem Namen ChecklistenReference. Wieder wird sie im Package gesucht und mit der Annotation @Entity versehen. Zur besseren Veranschaulichung gehen wir diesmal davon aus, dass die Klasse ChecklistenReference bereits von einer anderen Entity ableitet. Deshalb definieren wir lediglich die Annotation @DiscriminatorValue um den Eintrag für diese Klasse in der Datenbank zu bestimmen. Die darauf folgende Eigenschaft signalisiert JDO, dass diese Klasse auch außerhalb des Kontextes des PersistenceManagers existieren und bearbeitet werden kann. Sie darf also detached werden. Dies benötigt für JPA keine Konfiguration, da prinzipiell alle Klassen detachable sind. Als nächstes erfolgt eine Beschreibung für das Feld *checkliste*. JDO wird mitgeteilt, dass es sich dabei um eine Collection vom Typ ChecklistenEintrag handelt. Zusätzlich wird durch die Eigenschaft *element-dependent* definiert, dass die Werte dieser Liste entfernt werden können, wenn das Objekt selber entfernt wird. Die letzte Eigenschaft definiert, dass dies eine 1:N Relation ist. Eine äquivalente Einstellung für JPA wird durch das Hinzufügen der @OneToMany Annotation erreicht. Dieser wird ebenfalls mitgeteilt, dass es sich hierbei um ChecklistenEinträge handelt. Mit dem Attribut *orphanRemoval* wird die Eigenschaft von *element-dependent* umgesetzt. Damit ist auch die zweite Klasse fertig umgestellt (vgl. Listing 3.9). Somit ist die Migration dieser JDO Datei abgeschlossen.

```
@Entity()
@DiscriminatorValue(value = "ChecklistenReference")
public class ChecklistenReference {

  @OneToMany(targetEntity = ChecklistenEintrag.class, orphanRemoval =
      true)
  private Collection<ChecklistenEintrag> checkliste;

...
}
```

Listing 3.9: CheckListenReference Metadaten hinzugefügt

So sieht die Migration der Metadaten aus. Es folgt eine Liste mit weiteren Eigenschaften, welche in den Metadaten Dateien von JDO auftreten können.

In Abbildung 3.5 wird die Spaltengröße einer String Spalte definiert. Diese Größe lässt sich bei JPA mit der Annotation @Column und deren Attribut length konfigurieren. Jedoch ist es nicht möglich, eine unendliche Größe anzugeben, so wie es bei JDO mit -1 möglich war.

JDO	<field name="notiz"> <extension vendor−name="kodo" key="jdbc−size" value="−1"/> </field>
JPA	@Column(length = 255) **protected** String notiz;

Abbildung 3.5: Migration Größe einer String Spalte

Die folgenden Beispiele überlappen sich jetzt mit dem nachfolgenden Kapitel der Mapping Dateien, weil es für JDO notwendig ist die Typen von Listen in den Metadaten zu definieren (vgl. Kapitel 2.4.3).

Abbildung 3.6 zeigt wie für JDO die Kennzeichnung von Listen von Basistypen und einzubettenden Klassen vorgenommen werden muss. In JPA wird dieses durch die Annotation @ElementCollection abgedeckt. Entweder gibt man den Typ der Collection über das Attribut *targetClass* mit an oder man lässt es wie im unten stehenden Beispiel weg und es wird über den Typ der Collection bestimmt. Das Attri-

but *fetch* gibt an, wie dieses Feld geladen werden soll, wenn die Klasse aufgerufen wird. Der Standard dafür ist lazy und funktioniert in den meisten Fällen. Sollten jedoch entsprechende Fehlermeldungen auftauchen, sollte man den *FetchType* auf *Eager* ändern. Dann wird die Liste sofort mitgeladen.

JDO	<field name="typenLinks" **default**−fetch−group="true"> <collection element−type="Integer"/> <extension vendor−name="kodo" key="element− dependent" value="true"/> </field>
JPA	@ElementCollection(fetch = FetchType.EAGER) **private** Set<Integer> typenLinks = **new** HashSet< Integer>();

Abbildung 3.6: Migration von Listen von Basistypen und einzubettenden Klassen

In Abbildung 3.7 wird über JDO eine Collection von Notizen definiert. Es handelt sich dabei um eine bidirektionale Relation, welche von dem Attribut *fall* der Klasse Notiz verwaltet wird. Zusätzlich können die Notizen entfernt werden, wenn dieses Objekt entfernt wird. Das bisher unbekannte Attribut ist *inverse-owner*. In JPA wird dies als Attribut von Relationen unter dem Namen *mappedBy* beschrieben. Dort wird angegeben, welches Feld der abhängigen Klasse die Relation verwaltet. Dieses Attribut gibt es nur für die Relationen OneToMany und OneToOne.

3.9.3 Migration der Mappinginformationen

Kommen wir nun zu dem komplexeren der beiden Informationsquellen für persistente Klassen in JDO. Die Mapping-Dateien geben an, welche Typen hinter den persistenten Feldern stehen. Weiterhin dient sie zur Definierung von Relationen zwischen den Klassen. Es wird das bereits bekannte Beispiel aus Kapitel 2.4.4 „Mapping“ verwendet. Es ist in Listing 3.10 nochmals dargestellt. Außerdem ist dieses Beispiel die zugehörige Mapping Datei der in Kapitel 3.9.2 „Migration der Metadaten“ verwendeten JDO Datei. Somit erhalten wir am Ende zwei

JDO	<field name="notizen"> <collection element−type="Notiz"/> <extension vendor−name="kodo" key="element−dependent" value="true"/> <extension vendor−name="kodo" key="inverse−owner" value="fall"/> </field>
JPA	@OneToMany(targetEntity = Notiz.**class**, mappedBy = "fall", orphanRemoval = **true**) **protected** List<Notiz> notizen = **new** ArrayList<Notiz>();

Abbildung 3.7: Migration von bidirektionalen Relationen

komplett migrierte Klassen.

Am Anfang steht wie immer der Name des Packages und der ersten Klasse. Danach wird über den *jdbc-class-map type=„base“* definiert, dass es sich bei dieser Klasse um eine Basisklasse handelt, das heißt, dass die Klasse von keiner anderen ableitet. Weiterhin wird der Tabelle ein festgelegter Name gegeben und das Attribut JDOID wird als Primärschlüssel der Klasse definiert. Mit JPA wird der Name der Tabelle über die Annotation @Table mit dem Attribut name mitgegeben. Zur Definition des Primärschlüssels benötigt man bei JPA die Annotation @Id an dem Attribut, welches der Primärschlüssel werden soll. Anschließend wird konfiguriert, dass zusätzlich zur Klasse eine Versionsnummer mit gespeichert wird. Dies ist notwendig, um optimistischen Sperren nutzen zu können. In JPA erreicht man dies durch die Annotation @Version an einem Attribut der Klasse vom Typ int, Integer, short, Short, long, Long und java.sql.Timestamp. Zusätzlich wird ein Klassen Indikator definiert, damit der JDO Provider die Klasse zuordnen kann. Dies ist verpflichtend, wenn Klassen voneinander ableiten. Hier wird die Variante gewählt, dass die Klasse als eine Konstante in der Spalte *JDOCLASS* abgebildet wird. Dies wurde bereits im vorherigen Kapitel behandelt. Anschließend folgen die Definitionen der Felder. Bei JDO muss für jedes persistente Feld eine Angabe vorliegen. Dabei muss für jedes Feld beschrieben werden, um welchen Typ es sich handelt. Der Wert *value* gibt dabei an, dass es sich um einen Basistypen handelt. Mit Hilfe des Attributs column wird der Name der Spalte gesetzt. Vergleicht man dies jetzt mit JPA, muss man feststellen, dass

```
<mapping>
    <package name="package.checkliste">
        <class name="ChecklistenEintrag">
            <jdbc-class-map type="base" pk-column="JDOID" table="
                CHK_EINTRAG"/>
            <jdbc-version-ind type="version-number" column="
                JDOVERSION"/>
            <jdbc-class-ind type="metadata-value" column="JDOCLASS"/>
            <field name="comment">
                <jdbc-field-map type="value" column="COMMENT0"/>
            </field>
        </class>
        <class name="ChecklistenReference">
            <jdbc-class-map type="base" pk-column="JDOID" table="
                CHK_REFERENCE"/>
            <jdbc-version-ind type="version-number" column="
                JDOVERSION"/>
            <jdbc-class-ind type="metadata-value" column="JDOCLASS"/>
            <field name="checkliste">
                <jdbc-field-map type="one-many" ref-column.JDOID="
                    CHECKREFERENCE_JDOID" table="CHK_EINTRAG"/
                    >
            </field>
            <field name="_classId">
                 <jdbc-field-map type="value" column="CLASSID"/>
            </field>
            <field name="_objectId">
                 <jdbc-field-map type="value" column="OBJECTID"/>
             </field>
        </class>
    </package>
</mapping>
```

Listing 3.10: Beispiel package.mapping Datei

dort vieles ohne explizite Beschreibungen gelöst wird. Lediglich der Name der Spalte muss angepasst werden. Dies geschieht über die @Column Annotation und dem Attribut *name*. Damit ist die erste Klasse komplett abgeschlossen (vgl Listing 3.11).

```
@Entity
@Table(name ="CHK_EINTRAG")
@DiscriminatorColumn(name="ChecklistenTyp")
@DiscriminatorValue(value = "ChecklistenEintrag")
public class ChecklistenEintrag {
  @Id
  @GeneratedValue(generator = "ChecklistenEintragSequence", strategy =
      GenerationType.TABLE)
  @TableGenerator(initialValue = 1000, name = "ChecklistenEintragSequence",
      pkColumnName = "ID", table = "JDO_SEQUENCE", pkColumnValue = "
      package.ChecklistenEintrag",
      valueColumnName = "SEQUENCE_VALUE")
  private int id;

  @Column(name = "COMMENT0")
  private String comment;

  @Version
  private int version;
...}
```

Listing 3.11: CheckListEneintrag Mapping hinzugefügt

Betrachten wir nun die nächste Klasse. Dort ist vieles ähnlich. Lediglich die Relation ist neu. Hierbei handelt es sich um eine 1:N-Relation auf die Tabelle *CHK_EINTRAG* und der unter *ref-column.JDOID* angegebenen Spalte. JPA regelt Relationen, wie bereits im vorherigen Kapitel beschrieben, wieder über Annotationen. Hierfür wird die Annotation @OneToMany mit den entsprechenden Attributen, wie zum Beispiel *targetEntity*, verwendet. Der Cascade-Typ gibt an, welche Operationen an die abhängige Klasse weitergeleitet werden. Mit diesem Attribut sollte man vorsichtig umgehen, denn alle durch den Typ gekennzeichneten Operationen werden an die Relation weitergegeben. Zusätzlich wird die Annotation @JoinTable benötigt, um den Namen der Join-Tabelle anzugeben. Mit Hilfe der Attribute inverseJoinColumns und joinColumns werden die Schlüssel der Relation gekennzeichnet. Es entsteht die in Listing 3.12 dargestellte Klasse.

```
@Entity
@Table(name ="CHK_REFERENCE"
@DiscriminatorColumn(name="ChecklistenReferenceTyp")
@DiscriminatorValue(value = "ChecklistenReference")
public class ChecklistenReference {
```

```
  @Id
  @GeneratedValue(generator = "ChecklistenReferenceSequence", strategy =
      GenerationType.TABLE)
  @TableGenerator(initialValue = 1000, name = "ChecklistenReferenceSequence",
       pkColumnName = "ID", table = "JDO_SEQUENCE", pkColumnValue =
      "package.ChecklistenReference",
      valueColumnName = "SEQUENCE_VALUE")
   private int id;
  @JoinTable(name = "CHK_EINTRAG", inverseJoinColumns = @JoinColumn(
      name = "ChecklistenEintrag_jpaid"), joinColumns = @JoinColumn(name
      = "ChecklistenReference_jpaid"))
  @OneToMany(targetEntity = ChecklistenEintrag.class, cascade =
      CascadeType.PERSIST)
  private Collection<ChecklistenEintrag> checkliste;
  @Column(name = "CLASSID")
  private String _classId;
  @Column(name = "OBJECTID")
  private String _objectId;
...}
```

Listing 3.12: CheckListenReference Mapping hinzugefügt

Somit wurden alle Klassen des package erfolgreich migriert.

Folgend wieder eine Liste mit weiteren verwendeten Mapping Informationen.

Abbildung 3.8 zeigt die Beschreibung eines Feldes, welches für JDO ein clob ist. Dafür wird in JPA die Annotation @Lob verwendet. Die gleiche Annotation wird auch für BLOB Typen verwendet.

JDO	<field name="notiz"> <jdbc−field−map type="clob" /> </field>
JPA	@Lob **private** String notiz;

Abbildung 3.8: Migration von BLOB und CLOB

Abbildung 3.9 zeigt die Definition des Feldes *typenLinks* als Collection. Der Inhalt der Collection steht in der Spalte *ELEMENT* der Tabelle *ROLLE_TYPENLINKS*. In JPA benötigt man dafür einige Annotationen. Mit Hilfe von @ElementCollection erfolgt die Beschreibung, dass

es sich um eine Collection handelt. @CollectionTable hilft bei der Konfiguration für die Tabelle der Collection und über @Column wird wieder der Spaltenname der Tabelle mitgegeben.

JDO	<field name="typenLinks"> <jdbc−field−map type="collection" element− column="ELEMENT" ref−column.JDOID=" JDOID" table="ROLLE_TYPENLINKS"/> </field>
JPA	@ElementCollection() @CollectionTable(name = "ROLLE_TYPENLINKS") @Column(name = "ELEMENT") **private** Set<Integer> typenLinks = **new** HashSet< Integer>();

Abbildung 3.9: Migration von angepassten Listen

3.9.3.1 Relationen

Wie eine 1:N Relation migriert wird, wurde bereits an der Klasse ChecklistenReference gezeigt. Als nächstes folgt eine 1:1 Relation (vgl. Abbildung 3.10, welche in JDO über den type-Value *one-one* definiert wird und zusätzlich die Information benötigt, über welche Spalte der Join mit der Klasse funktioniert. In JPA wird dies über die Annotation @OneToOne abgebildet. Diese beinhaltet wieder die Klasse der Relation. Die Bezeichnung der Join Spalte wird mit Hilfe der Annotation @JoinColumn beschrieben.

Eine Besonderheit unter den Relationen ist die N:N Relation (vgl. Abbildung 3.11. Diese benötigt für JPA eine @JoinTable Annotation. Mit ihr kann man definieren, wie die gemeinsame Tabelle heißen soll und wie die beiden Primärspalten heißen.

JDO	<field name="parentInfo"> <jdbc−field−map type="one−one" column. JDOIDX="PARENTINFOX"/> </field>
JPA	@OneToOne(targetEntity = ActionExecutionInfo.**class**, cascade = CascadeType.PERSIST) @JoinColumn(name="PARENTINFOX") **private** ActionExecutionInfo parentInfo;

Abbildung 3.10: Migration von 1:1 Relationen

JDO	<field name="rollen"> <jdbc−field−map type="many−many" element− column.JDOIDX="UA_IDX_ROLLE" ref− column.JDOIDX="UA_IDX_BENUTZER" table="UA_REL_BENUTZER_ROLLE"/> </field>
JPA	@ManyToMany() @JoinTable(name = "UA_REL_BENUTZER_ROLLE", joinColumns = @JoinColumn(name = " UA_IDX_BENUTZER", nullable = **false**), inverseJoinColumns = @JoinColumn(name = " UA_IDX_ROLLE", nullable = **false**)) **private** List<Rolle> rollen = **new** ArrayList<Rolle>();

Abbildung 3.11: Migration von N:N Relationen

3.9.3.2 Primärschlüsselvergabe

Betrachten wir noch einmal genauer den Primärschlüssel der Klasse. Dieser wurde in den Beispielen zusätzlich mit der Annotation @GeneratedValue versehen. Dadurch wird dem Provider mitgeteilt, dass es die ID automatisch vergeben soll. Es gibt dabei die Strategien AUTO, IDENTITY, SEQUENCE und TABLE. Der Standard ist AUTO und bedeutet, dass der Provider abhängig von der zugrundeliegenden Datenbank die Strategie wählt. Dies kann zu Problemen in der Performance führen, da der Provider alle Strategien nacheinander durchprobiert, bis eine funktioniert. Wird die Strategie IDENTITY gewählt, so wird der Primärschlüssel beim Einfügen in der Datenbank vergeben. Der Nachteil ist, dass der Primärschlüssel auch erst nach dem Einfügen im Code zur Verfügung steht. Somit kann es leicht passieren, dass man mit dieser Strategie NullPointerExceptions bekommt. SEQUENCE sorgt dafür, dass zur Erstellung der Primärschlüssel eine Sequenz verwendet wird und mit TABLE wird für die Vergabe des Primärschlüssels eine eigene Tabelle verwendet. Abbildung 3.12 zeigt die Verwendung der einzelnen Varianten.

In den Beispielen wurde immer TABLE als Generatortyp verwendet. Dies kommt daher, dass der Standard für JDO, so wie es tolina GmbH verwendet, ebenfalls einen Table Generator verwendet. Dabei wird eine Tabelle mit dem Namen *JDO_SEQUENCE* verwendet. Als Primärschlüssel verwendet JDO den Pfad der Klasse (package plus Klassenname) und nennt diese Spalte *ID*. Der Name der Spalte für die generierten Werte ist *SEQUENCE_VALUE*. Somit ist es mit dieser Generierung möglich die gleiche Primärschlüsselvergabe für JDO und JPA zu erhalten. Das einzige Problem bei der Konfiguration ist der Startwert. Da vorher nicht bekannt ist, wie viele IDs JDO bereits vergeben hat, sollte als Startwert ein möglichst großer Wert genommen werden, da es sonst zu Konflikten bei der Primärschlüsselvergabe kommen kann.

3.9.4 Migration der Query

3.9.4.1 Theorie

JDOQL ist, wie bereits in Kapitel 2.4.2 „Query“ erwähnt, eine Java basierte Query Sprache. Das Ziel dieses Kapitels ist es Schritt für Schritt zu zeigen, wie aus einer JDOQL eine äquivalente JPQL werden kann. Dafür gibt es bei der Nutzung von JPA zwei verschiedene Ansätze. Der erste Ansatz ist das Schreiben von dynamischen Queries. Dies ist

AUTO	@Id @GeneratedValue **private int** id;
IDENTITY	@Id @GeneratedValue(strategy=GenerationType. IDENTITY) **private int** id;
SEQUENCE	@Entity @javax.persistence.SequenceGenerator(name="SEQ_STORE", sequenceName=" my_sequence") **public class** Store { @Id @GeneratedValue(strategy=GenerationType. SEQUENCE, generator="SEQ_STORE") **private int** id;
TABLE	@Id @GeneratedValue(generator = " ChecklistenReferenceSequence", strategy = GenerationType.TABLE) @TableGenerator(initialValue = 10000, name = " ChecklistenReferenceSequence", pkColumnName = "ID", table = " JDO_SEQUENCE", pkColumnValue = " package.ChecklistenReference", valueColumnName = " SEQUENCE_VALUE") private int id;

Abbildung 3.12: Generierung von Primärschlüsseln

in meinen Augen eine schnelle Variante zum Schreiben einer Query, jedoch bietet sie nicht viele Möglichkeiten zur Verallgemeinerung. Die zweite Variante ist die Benutzung der JPA-Criteria API. Sie ähnelt, im Gegensatz zur JPQL, wieder der bekannten Java Syntax und bietet bessere Alternativen zur Verallgemeinerung. Betrachten wir nun alle Funktionen von JDOQL (siehe Abbildung 3.13) (vgl. [objа])).

Zur Veranschaulichung ist in Abbildung 3.14 ein Beispiel zur Gegenüberstellung von JDOQL, dynamisches JPQL und Criteria Query dargestellt. Es wird eine Query erstellt, welche alle Banken findet, welche mehr als 1000 Kunden haben.

Nicht nur bei der Erzeugung und Verwendung von Queries existieren Unterschiede zwischen JDOQL und JPQL. So ist die Verwendung von relationalen (vgl. Abbildung 3.15) und logischen (vgl. Abbildung 3.16) Operatoren ebenfalls unterschiedlich.

Zusätzlich besitzt JPQL Operatoren wie zum Beispiel: IS [NOT] NULL, [NOT] BETWEEN, [NOT] IN, IS [NOT] EMPTY, [NOT] MEMBER [OF] und [NOT] LIKE. Daraus wird erkennbar, dass die JDOQL die java basierten Operatoren verwendet, um es java Entwicklern möglichst einfach zu machen.

Betrachten wir als nächstes Funktionen zum Aggregieren des Ergebnisses. Dazu gehören Funktionen wie zählen, summieren, Mittelwert bilden, Maximum und Minimum bestimmen. Zusätzlich zeigen die Beispiele in Abbildung 3.17 noch die Verwendung der *Having Klausel* JDOQL (vgl. [oraa]) verwendet dafür Funktionen, welche auf das Ergebnis angewendet werden. Dynamische JPQL verwendet diese Funktionen direkt integriert im Query String. Die Criteria API verwendet für das Aggregieren den CriteriaBuilder und modifizierte select-Befehle.

Folgend betrachten wir das Definieren von Reichweiten in Queries. JDOQL verwendet dafür eine Methode setRange(), welche als Parameter den Startwert und den Endwert bekommt. Hierbei sind die Criteria API und dynamisches JPQL gleich, da die Konfiguration der Reichweite erst auf die erstellte Query angewendet wird. Mit Hilfe der Methode setFirstResult() wird angegeben, welches das erste Elemente sein soll und durch die Methode setMaxResults() wird angegeben, wie viele Ergebnisse maximal geholt werden sollen.

Funktion	**JDOQL**	**dynamisches JPQL**	**Criteria API**
Query erzeugen	pm.newQuery()	em.createQuery ()	CriteriaBuilder cb = em. getCriteriaBuilder(); CriteriaQuery<?> query = cb.createQuery();
Selektion	q.setClass()	*from* Klausel in der Query	q.from()
Filter benutzen	q. setFilter ()	*where* Klausel in der Query	q.where()
Parameter definieren	q.declare Parameters()	q.setParameter ()	q.setParameter()
Typen von Variablem definieren	q. declareVariables ()	Es wird versucht den Typen der Variable aus dem Kontext heraus zu ermitteln.	cb.parameter(Class, name)
Ergebnisse sortieren	q.setOrdering()	*Order By* Klausel in der Query	q.orderBy()
Query ausführen und das Ergebnis erhalten	q.execute()	q.getResultList ()	em.createQuery(query). getResultList()

Abbildung 3.13: Vergleich der Queries

JDOQL	Query query = pm.newQuery (Bank.**class**); query. setFilter ("customers > numberOfCustomers"); query.declareParameters(“Integer numberOfCustomers”); List results = (List) query.execute (**new** Integer (1000));
dynamisches JPQL	String queryString = "SELECT b FROM Bank b WHERE b.customers > :customers" Query query = entityManager.createQuery(queryString); query.setParameter("customers", Integer.valueOf (1000)); List<Bank> resultList = (List<Bank>) createQuery.getResultList();
Äquivalente Criteria Query	CriteriaBuilder cb = em.getCriteriaBuilder(); CriteriaQuery<Object> query = cb.createQuery(Bank.**class**); Root<Bank> root = query.from(Bank.**class**); *//Filter bauen* ParameterExpression<Integer> customers = cb. parameter(Integer.**class**); Path<List<Customer>> numberOfCustomers = root.get("customers"); Predicate condition = cb.gt(cb.size(numberOfCustomers), customers) query. select (root); query.where(condition); TypedQuery<Bank> typeduery = em.createQuery (query); typedQuery.setParameter(customers, Integer. valueOf(1000)); List<Bank> resultList = createQuery. getResultList();

Abbildung 3.14: Query zum Vergleich von JDOQL und Criteria Query

Relationaler Operator	**JDOQL**	**JPQL**
Gleichheit	==	=
Ungleichheit	!=	<>

Abbildung 3.15: Relationale Operatoren im Vergleich

Logischer Operator	**JDOQL**	**JPQL**
Und	&&	AND
Oder	||	OR
Nicht	!	NOT

Abbildung 3.16: Logische Operatoren im Vergleich

JDOQL	query.setResult(“count(**this**)”); query.setResult(“sum(customers)”); query.setResult(“min(customers)”); query.setResult(“max(customers)”); query.setResult(“avg(customers)”); q.setGrouping ("bank having max(customers) > 1000");
dynamisches JPQL	em.createQuery("SELECT COUNT(b)... em.createQuery("SELECT SUM(b.customers)... em.createQuery("SELECT MIN(b.customers)... em.createQuery("SELECT MAX(b.customers)... em.createQuery("SELECT AVG(b.customers)... em.createQuery(“...HAVING COUNT(b)>1”)
Äquivalente Criteria Query	query.select(cb.count(root)); query.select(cb.sum(root.get(“customers”)); query.select(cb.min(root.get(“customers”)); query.select(cb.max(root.get(“customers”)); query.select(cb.avg(root.get(“customers”)); query.having(cb.gt(cb.count(root),1);

Abbildung 3.17: Aggergationsfunktionen im Vergleich

Das Beispiel in Abbildung 3.18 sorgt dafür, dass alle Zeilen zwischen 20 und 60 als Ergebnis der Query zurückgegeben werden.

JDOQL	query.setRange(20, 60);
dynamisches JPQL	query.setFirstResult(20) query.setMaxResults(40)

Abbildung 3.18: Definieren von Reichweiten im Vergleich

Da es für JPA nicht die Funktionalität der *Result Sets* gibt, kann man diese Funktionen nutzen um ein *Paging* über die Ergebnismenge zu legen.

3.9.4.2 Umsetzung

Produkte und Komponenten möchten, ohne davon zu wissen entweder JDOQL Queries bauen, wenn sich die benötigten Objekte im JDO Kontext befinden oder aber JPQL Queries bauen, wenn sie sich im JPA Kontext befinden. Dies wird am besten wieder einmal durch einen allgemeinen Builder (vgl Abbildung 3.19) erreicht, ähnlich des zuvor erstellten PMWrappers. Dafür benötigt man eine Klasse, die unabhängig vom Kontext die einzelnen Teile einer Query erstellt. Erst wenn die Query ausgeführt werden soll, wird überprüft, zu welchem Kontext das Objekt oder die Objekte gehören. Ist der Kontext bekannt, wird aus den abstrakten Teilen, welche vorher vom Builder gebaut wurden, die entsprechende Query erstellt. Die Ermittlung des Kontexts erfolgt mit Hilfe des enthaltenen PMWrappers.

Im ersten Schritt wurde eine Klasse FilterExpression angelegt. Sie repräsentiert den Vergleich zweier Werte. Der Operator wird als enum definiert, um unabhängig von den Unterschieden zwischen JDOQL und JPQL synchrone Operatoren haben zu können (vgl. Listing 3.13).

```
new FilterExpression("Kontonummer", Operator.EQUAL, x);
new FilterExpression("Geld", Operator.GREATER_EQUAL, money);
```

Listing 3.13: Beispiele von FilterExpressions

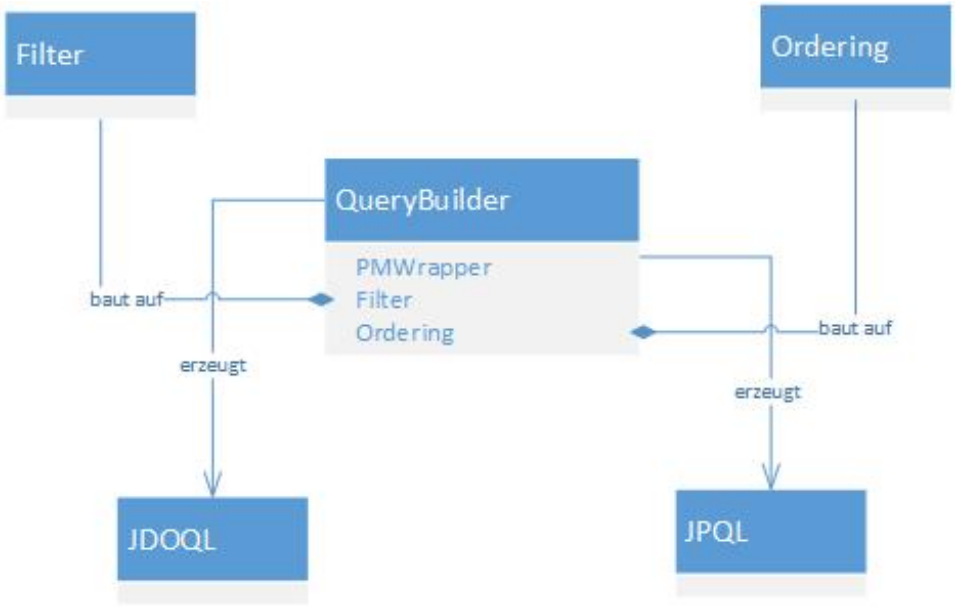

Abbildung 3.19: Schematische Darstellung einer möglichen Architektur eines hybriden QueryBuilders

Die Klasse Filter benutzt jetzt diese FilterExpressions, um sie gemäß der Anforderungen aneinander zu ketten. Dafür wird eine Baumstruktur verwendet. Durch die Methoden *andCondition* oder *orCondition* (vgl. Listing 3.14), welche als Parameter eine *FilterExpression* enthalten, werden neue Filter erstellt und als Kinder des aktuellen Filters angehangen.

```
Filter .andCondition(FilterExpression)//
.andCondition(FilterExpression)//
.orCondition(FilterExpression);
```

Listing 3.14: Erstellung neuer Filter durch Verwendung der Methoden andCondition und orCondition.

Abschließend besitzt die Klasse *Filter* die beiden unterschiedlichen Methoden zum Bauen eines Filters für JDOQL und eines Filters für JPQL. Dafür wird Rekursion verwendet, welche die Baumstruktur mittels Tiefensuche durchläuft und den Filter aufbaut. Im Falle von JDOQL ist dies ein String und für JPQL wird die Criteria API, aufgrund der besseren Flexibilität und der Typsicherheit verwendet, um ein Predicate zu erstellen (vgl. Abbildung 3.20.

Die erstellten Filter können anschließend als Parameter für die entsprechenden Funktionen übergeben werden. Für die JDO-Query ist es *query.setFilter()* und für die JPA-Query ist es *query.where()*.

JDOQL	"(vorname == _a0 && nachname == _a1) || (vorname == _a2 && nachname == _a3)"
Criteria Query	cb.or(cb.and(cb.equals(root.get(“vorname”),_a0), root.get(“nachname”),_a1)),cb.and(cb.equals(root.get(“vorname”),_a2),root.get(“nachname”),_a3))

Abbildung 3.20: Erstellung eines Filters

Betrachten wir als Nächstes das Ordering, das angibt, wie die einzelnen Ergebnisse sortiert werden sollen. Dafür wird, wie bereits für den Filter, eine eigene Klasse Ordering erstellt. Die Erzeugung eines Ordering Objektes ist in Listing 3.15 zu sehen. Es beinhaltet den Namen des Feldes, sowie die Art der Sortierung (aufsteigend oder absteigend).

```
/** Standardkonstruktor zur Angabe eines Feldnamens und einer Sortierrichtung,
   * nach der sortiert werden soll. */
  public Ordering(String fieldname, OrderDirection direction) {
    this.direction = direction;
    this.fieldname = fieldname;
  }
```

Listing 3.15: Erstellung eines Ordering Objektes

Der QueryBuilder besitzt jetzt wiederum eine Methode zum Hinzufügen eines Orderings und erstellt intern eine Liste von Orderings. Außerdem existieren die beiden Methoden, um aus der Liste der Orderings einerseits ein korrektes Ordering für JDOQL zu erzeugen und andererseits ein korrektes Ordering für JPQL zu erzeugen (vgl. Abbildung 3.21).

Somit ist die Architektur geschaffen, um unabhängig von JPA oder JDO einen Filter und ein Ordering für Queries zu erstellen. Die zugrundeliegende Architektur ermöglicht es, den Builder beliebig zu erweitern zum Beispiel um Gruppierungen hinzuzufügen.

3.9.4.3 Query Erweiterungen

Zur einfacheren Verwendung von Queries wurden einige Erweiterungen für JDO geschrieben. Dazu gehören die folgenden Erweiterungen:

JDOQL	“vorname asc, nachname desc”
Criteria Query	List<Order> listOfOrdering = **new** ArrayList< Order>(); listOfOrdering.add(cb.asc(root.get(“vorname”))); listOfOrdering.add(cb.desc(root.get(“nachname”))) ;

Abbildung 3.21: Vom Builder erzeugtes Ordering

- simpleMatch
- trim und
- notNull

Die simpleMatch Erweiterung wurde im Vergleich zu den beiden anderen deutlich häufiger verwendet, weshalb ich lediglich die Umstellung dieser Erweiterung erklären werde. Mit Hilfe dieser Erweiterung wurden Java Regular Expressions verwendet, um ein Datenbankeintrag mittels *Like* Abfrage zu vergleichen. Dafür wurde die Java Regular Expression vor der Ausführung in die Datenbank Regular Expression transformiert. Zusätzlich gab es Einstellungsmöglichkeiten, zum Beispiel die Nichtbeachtung von Groß- und Kleinschreibung oder Umlauten. Zu den Umlauten ist anzumerken, dass lediglich für Umlaute auch die Variante ohne Umlaute getestet werden soll, aber nicht umgekehrt. Das Beispiel in Abbildung 3.22 verdeutlicht dies.

Diese simpleMatch Ausdrücke wurden dem JDO Filter hinzugefügt. Der erste Schritt der Umstellung war das Hinzufügen eines neuen Operators. Es wird nur einer benötigt, da mir beim Scannen der Projekte kein simpleMatch aufgefallen ist, welches die Groß- und Kleinschreibung beachten wollte. Somit entstanden im Builder vier neue Methoden. Für die Varianten mit oder ohne Umlaute, jeweils eine für Oder-Verknüpfungen und eine für Und-Verknüpfungen. In Listing 3.16 werden die Funktionen für die Und-Verknüpfung dargestellt.

```
public QueryLanguageBuilder<E> andSimpleMatch(String ausdruck, String
    wert) {
    return andSimpleMatch(ausdruck, wert, false);
}
```

Beispiele	**Was wird gefunden**
"name1.simpleMatch(nachame,'true','true')" Finde alle Namen die gleich dem Nachnamen sind. Ignoriere dabei die Groß- und Kleinschreibung und überprüfe beide Schreibweisen für Umlaute.	Unter der Annahme nachname=mueller: Müller (wird gefunden) Mueller (wird gefunden) mÜLler (wird gefunden) Unter der Annahme nachname=müller: Müller (wird gefunden) Mueller (wird nicht gefunden) mÜLler (wird gefunden)
"name1.simpleMatch('∗le∗','true')" Finde alle Namen, welche ein "le" enthalten. Ignoriert wird dabei die Groß- und Kleinschreibung.	Köhler (wird gefunden) Müller (wird gefunden) Lenz (wird gefunden) Schmidt (wird nicht gefunden)

Abbildung 3.22: Beispiele der simpleMatch Extension

```
public QueryLanguageBuilder<E> andSimpleMatch(String ausdruck, String
    wert, boolean checkUmlaute) {
  wert = SimpleMatchExtension.createPatternForDB(wert.toCharArray(),
      checkUmlaute);
  if (checkUmlaute) {
    this.andCondition(newSubFilter().andCondition(ausdruck, Operator.
        SIMPLE_MATCH, wert, null)//
    .orCondition(ausdruck, Operator.SIMPLE_MATCH, SimpleMatchExtension.
        replaceUmlaute(wert), null));
  } else {
    this. filter .andCondition(ausdruck, Operator.SIMPLE_MATCH, wert, null
        );
  }
  return this;
}
```

Listing 3.16: Verwendung des simpleMatch Operators für Und-Verknüpfungen

Die Umwandlung von Java Regular Expressions zu den entsprechenden Datenbank Ausdrücken erfolgt jetzt bereits beim Aufruf dieser Methoden. Zusätzlich wurden bei Verwendung der Umlaute-Funktion intern bei der Ausführung der simpleMatch Erweiterung zwei Like-Ausdrücke gebaut: einen für jede Schreibweise. Dies wird jetzt ebenfalls sofort erstellt, damit für die Verwendung von JPA beide als Argumente für die

Query zur Verfügung stehen. Der Builder erstellt daraus wieder die entsprechenden Filter für JDOQL und für die Criteria Query. So könnten jetzt auch die anfangs erwähnten anderen beiden Erweiterungen hybridfähig gemacht werden.

3.10 Erkenntnisse aus der Migration

3.10.1 Metadaten und Mapping

Die Beispielmigration am Prototypen war wichtig, um weitere Erkenntnisse über JPA zu erlangen. So stellte sich raus, dass es nicht zwei Entities mit dem gleichen Namen geben darf. Es gab einen Fall, in dem eine Klasse Notiz in einem Package lag und ein Package tiefer gab es ein weiteres Mal eine Klasse Notiz. Verziert man beide lediglich mit der Annotation @Entity, so bekommt man zur Laufzeit einen Fehler vom Provider mitgeteilt. Dieser weist darauf hin, dass das Entity Notiz zweimal existiert. So muss eines von beiden über das Attribut “name” ein anderer Name gegeben werden.

Ein weiterer Fehler zur Laufzeit wies mich darauf hin, dass ich bei einer Relation lediglich eine Seite der Relation definiert hatte. So gab es eine Klasse Vorgang, welche eine Liste von Notizen und Wiedervorlagen besaß. Die einzelnen Notizen und Wiedervorlagen kannten ihre Vorlagen, aber sie besaßen nicht die entsprechende Annotation. Wie entstand dieser simple Fehler nun? Die Ursache war, dass für diese Klassen lediglich eine JDO Datei mit den Metadaten der einzelnen Klassen vorlag. Die zugehörige Mapping Datei liegt in den einzelnen Produkten, da sie diese Klassen unterschiedlich mappen möchten. Somit existierten innerhalb der Komponente keine Informationen darüber, dass Notizen und Wiedervorlagen ihre Vorlagen kennen. Dieses Vorgehen wird bei einigen Klassen angewendet, wodurch dabei besonders aufmerksam vorgegangen werden muss.

Bei der Verwendung von Vererbungen ist ein weiterer Fehler aufgefallen. Bevor man den Primärschlüssel einer neuen Klasse definieren will, sollte man sich die Oberklasse anschauen. Verwendet diese die Strategie **SINGLE_TABLE** so werden alle Attribute der Hierarchie in einer Klasse gespeichert. Dies bedeutet, dass die Primärschlüsseldefinition der Oberklasse angewendet wird. Bei den beiden anderen Strategien **JOINED** und **TABLE_PER_CLASS** muss wieder ein Primärschlüssel definiert werden.

3.10.2 Query

Durch die Umsetzung der Migration an dem Prototypen bin ich darauf gestoßen, dass meine Annahme bezüglich der möglichen Operatoren in den Filtern der Queries falsch war. Innerhalb des Prototypen wurde vermehrt der Operator contains gefunden. Dieser wird verwendet, um zu überprüfen, ob eine Collection des persistenten Objektes diesen Eintrag enthält oder ob der Wert des Objektes in einer Collection enthalten ist. Im Beispiel in Abbildung 3.23 ist der Kunde unser persistentes Objekt. Es soll einmal überprüft werden, ob der Name eines bestimmten Kunden in der übergebenen Liste ist und das andere Mal ob in der Liste aller Kundennamen ein gewisser Name auftaucht.

JDOQL	Namen.contains(Kunde.name) Kunde.namen.contains(name)
dynamisches JPQL	root.get(“name”).in(namen) cb.isMember(name,root.get(“namen”)

Abbildung 3.23: Beispiel zur Umsetzung des Contains Operators

Zusätzlich wurden durch die Migration kleine Fehler an den Buildern festgestellt. Diese waren nicht null-safe. Wenn zum Beispiel kein Filter oder kein Ordering erstellt wurde. Wenn dieser Filter an die Query übergeben wurde, gab es zur Laufzeit vom Provider eine NullPointer-Exception bei der Ausführung der Query. Der Builder wurde angepasst, sodass er auch mit null-Werten umgehen kann.

Bei der Migration des zweiten Prototypen tauchte wieder eine neue Funktionalität der JDOQL auf. Dies war eine toLowerCase-Methode für Datenbankobjekte. Damit wurde ein Case Insensitiver Vergleich simuliert. Um unabhängig zu bleiben, musste eine allgemeine Lösung zum Builder hinzugefügt werden. Diese Funktion wurde durch einen neuen Operator für den Filter umgesetzt.
EQUALS_IGNORE_CASE macht intern genau diese Simulation für JDO und JPA. Die Umsetzung des Operators in den entsprechenden Filter für JDOQL und JPQL ist in Abbildung 3.24 dargestellt.

JDOQL	" first .toLowerCase() ==second.toLowerCase()"
dynamisches JPQL	cb.equal(cb.lower(first), cb.lower(second))

Abbildung 3.24: Beispiel zur Umsetzung des EQUALS_IGNORE_CASE Operators

3.11 Hindernisse

3.11.1 Multimodule-Projekte

Anfangs unterlag ich der Annahme, dass es Probleme bei Multimoduleprojekten geben könnte, da für jedes Projekt eine eigene persistence.xml angelegt werden muss und man somit unterschiedliche EntityManager erhalten würde. Betrachtet man aber diese Multimoduleprojekte etwas genauer, so gibt es unter ihnen immer nur ein HauptProjekt, welches die persistenten Klassen enthält und mit ihnen arbeitet. Somit reicht es für dieses Projekt, eine persistence.xml anzulegen.

3.11.2 Sind Annotationen überschreibbar?

Bei der Migration musste ich feststellen, dass es für einige Klassen eine JDO Datei, aber keine Mapping Datei gab. Der Grund dafür ist, dass die Produkte diese Klassen selbst mappen möchten. Also einen eigenen Tabellennamen, eigene Spaltennamen und auch andere Relationen verwenden möchten. Damit dies mit JPA weiter funktioniert muss es eine Möglichkeit geben, dass die Produkte die Annotationen überschreiben können.

Herr Müller beschreibt in seinem Buch, dass die Verwendung von Mapping Dateien das Überschreiben von einigen Annotationen ermöglicht (vgl. [PDBM12] Seite 246). Um das Problem nachzustellen, wurde ein kleines Testszenario erstellt. Zuerst wurden die Klassen über Annotationen definiert und persistiert. Anschließend wurde über die orm.xml eine dieser Klassen neu definiert. Dabei wurde der Tabellenname geändert.

Anschließend wurde der Test zur Persistierung erneut ausgeführt. Als Resultet war nun zusätzlich die neue Tabelle ebenfalls in der Datenbank. Damit wurde gezeigt, dass die JPA Annotationen durch eine orm.xml überschrieben werden können.

3.11.3 Dateninkompatibilität

Für die Migration der vorhandenen JDO-Tabellen zu JPA muss eine Lösung gefunden werden. Das Problem hierbei ist, dass JDO innerhalb der Tabellen weitere Spalten anlegt. Diese entstehen durch die Konfiguration in den Mapping- und Metadateien. Wenn es konfiguriert ist, wird eine Spalte für den Klassen-Indikator (jdoclass) angelegt, eine für den Versions-Indikator (jdoversion) und eine für den Primärschlüssel (jdoid). Diese Spalten müssen für JPA migriert werden. Die Spalte für den Primärschlüssel muss durch den neuen Primärschlüssel von JPA ersetzt werden. Der Version-Indikator muss durch die Spalte der Annotaion @Version und der Klassen-Indikator muss durch die Discriminator Spalte ersetzt werden. Dies muss für jeden Eintrag in der Tabelle vorgenommen werden.

3.12 Kostenabschätzung

3.12.1 Dauer der Migration

Während der Migration bin ich schrittweise von Package zu Package vorgegangen und habe sie nacheinander umgestellt. Dabei wurde für jedes Package notiert, wie viele Klassen angepasst werden mussten, ob es eine Metadaten und Mapping Datei gab und wie lange die ganze Umstellung gedauert hat. Anschließend an jedes Package folgte das Korrigieren der fehlgeschlagenen JUnit Tests. Auch dafür wurde die Zeit notiert. Die Resultate der Migration sind in Tabelle 3.5 dargestellt.

Dauer insgesamt in Minuten:	594
Dauer Umstellung der Klassen in Minuten:	210
Dauer JUnit Tests in Minuten:	264
Dauer Rest in Minuten:	120
Anzahl Klassen:	35
Anzahl Klassen mit Metadaten:	35
Anzahl Klassen mit Mapping:	24

Tabelle 3.5: Dauer der Migration des Prototypen

Betrachtet man die Resultate genauer, so sieht man, dass die meiste Zeit für das Korrigieren der JUnit Tests verwendet wurde. Dies lag unter anderem daran, dass viele Tests noch mit JUnit 3 liefen. Ich benötigte jedoch JUnit 4 Tests, um meine erstellte JPARule verwenden zu können, damit die Tests einen Datenbankzugriff auf eine in-memory Datenbank vollführen können. Jedoch wird dieses Problem auch in weiteren Komponenten und auch Produkten auftreten. Zur Kategorie Rest gehören die Compilefehler, welche durch Änderungen in anderen Komponenten entstanden sind und auch die Anpassungen der Querys. Diese mussten auf den Builder umgestellt werden. Bricht man die Dauer für die Klassen runter auf eine einzelne Klasse so kommt man auf 6 Minuten je Klasse. Diese Zahl können wir jetzt für die Hochrechnungen aller Komponenten verwenden. Für die Hochrechnungen von Produkten ist sie weniger aussagekräftig, da die zu migrierenden Klassen in Produkten einen viel größeren Umfang haben. Dies lässt sich durch eine kleine Anpassung besser approximieren. Dafür betrachtet man die Anzahl der Attribute der Klasse und baut diese ebenfalls mit in die Rechnung ein.

3.12.2 Hochrechnungen Komponenten

Für die Hochrechnungen der Komponenten wird die Tabelle aus Kapitel 3.8 "Die Wahl des Prototypen"wiederverwendet. Anhand unser gerade ermittelten Zeiten werden die Spalten Dauer Klassen, Dauer JUnit, Dauer Rest und Dauer insgesamt hinzugefügt. Zur Berechnung der Zeiten benötigt man jetzt noch die durchschnittliche Anzahl der Klassen pro Package/JDO-Datei. Dabei wird davon ausgegangen, dass der Prototyp repräsentativ dafür ist. Es ergibt sich ein Wert von 4,375 Klassen pro Package/JDO-Datei. Bei einer ungeraden Zahl wird immer aufgerundet. Dabei ergeben sich die in Tabelle 3.7 dargestellten Werte.

Damit ergibt sich eine Zeit von 56,6 Stunden für die Komplettumstellung aller Komponenten auf JPA. Hinzu kommt noch ein unbekannter Zeitfaktor für einmalige Änderungen am QueryBuilder oder PMWrapper. Dieser Zeitfaktor umfasste beim Prototypen 318 Minuten. Unter der Annahme, dass der gewählte Prototyp repräsentativ ist, dürften die Änderungen durch andere Komponenten kleiner werden.

Komponente	**Anzah jdo Da-teien**	**Anzah Klas-sen**	**Dauer Klas-sen**	**Dauer JU-nit**	**Dauer Rest**	**Dauer ins-ge-samt**
action_base	3	14	84	106	49	239
application_base	8	35	210	264	121	595
bo_base	1	5	30	38	18	86
browser_base	1	5	30	38	18	86
db_base	1	5	30	38	18	86
host_base	5	22	132	166	76	374
host_hundh	1	5	30	38	18	86
host_si_dyns	3	14	84	106	49	239
host_txbdepot	1	5	30	38	18	86
tes_core	11	49	294	370	169	833
util_base	2	9	54	68	31	153
util_genericfields	1	5	30	38	18	86
util_license	1	5	30	38	18	86
util_locking	1	5	30	38	18	86
util_pin	1	5	30	38	18	86
util_remoting_tasl	2	9	54	68	31	153
util_reporting	1	5	30	38	18	86
util_settings	2	9	54	68	31	153
util_workflow	2	9	54	68	31	153
zins_base	1	5	30	38	18	86
util_useradmin	2	9	54	68	31	153

Tabelle 3.7: Geschätzte Dauer der Migration aller Komponenten in Minuten

Komponente	Anzahl Klassen	Dauer Klassen	Dauer JUnit	Dauer Rest	Dauer insgesamt
erwartete Zeit	9	54	68	31	153
tatsächliche Zeit	5	46	107	35	188

Tabelle 3.9: Vergleich zwischen erwarteter und tatsächlich benötigter Zeit

3.12.3 Prüfung der Ergebnisse

Zur Überprüfung meiner Hochrechnungen führe ich eine zweite Migration an einer kleineren Komponente durch, um zu überprüfen, wie genau meine Hochrechnungen sind. Dafür wird die Komponente util_useradmin verwendet. Diese besitzt zwei JDO Packages und hat damit eine erwartete Bearbeitungszeit von 153 Minuten. Ein Vergleich der tatsächlichen Werte mit den erwarteten erfolgt in Tabelle 3.9.

Das Ergebnis ist, dass die benötigte Zeit für die Umstellung etwas länger war als die erwartete. Die Gründe dafür liegen bei den JUnit Tests. Erstens ist die Testabdeckung der Komponente 10 Prozent höher als die des Prototypen, wodurch eine größere Chance besteht, dass mehr Persistenztests durchgeführt werden. Der zweite Grund liegt an der Komponente selbst. Sie kümmert sich um die Benutzerverwaltung, weshalb gerade bei ihr viele Persistenztests notwendig sind. Bei dieser Komponente gab es ebenfalls wieder Funktionen, welche für JPA neu entwickelt werden mussten. Sie sind aber wie erwartet mit 130 Minuten bedeutend geringer ausgefallen, als bei dem Prototypen.

Somit habe ich gezeigt, dass meine Hochrechnungen zutreffen.

3.12.4 Hochrechnungen Produkte

Wie bereits erwähnt, wird die Formel für Produkte nicht funktionieren. Daher benötigt man eine Anpassung, welche die Anzahl der zu persistierenden Attribute mit einbezieht. Dies ist notwendig, da eventuell alle Attribute bearbeitet wurden, indem sie zum Beispiel einen angepassten Spaltennamen haben. Die aus dem Prototypen gemessene Zeit von 6 Minuten zerfällt dabei in einen festen Anteil von 3 Minuten, darunter fallen die Standardannotationen @Entity oder @Embeddable, sowie Tabellennamen und Primärschlüssel. Übrig bleibt ein variabler Anteil von 3 Minuten für die Attribute und Vererbung. Teilt man die umge-

Komponente	Anzahl Klassen	Anzahl an Attributen im Schnitt	Dauer je Klasse	Dauer Klassen	Dauer JUnit	Dauer Rest	Dauer insgesamt
application_base	35	2	6	210	264	121	595
Produkt	19	10	18	342	144	66	552

Tabelle 3.10: Hochrechnung für die Dauer von Produkten

stellten Attribute des Prototypen durch die Anzahl seiner persistenten Klassen, so erhält man einen Durchscnitt von 2 Attributen je Klasse. Somit gehe ich davon aus, dass man 1,5 Minuten zur Umstellung eines Attributes benötigt. Daraus entstehen die in Tabelle 3.10 dargestellten Werte für Produkte.

Wie man sieht, steigt, trotz geringerer Anzahl der Klassen, die benötigte Zeit für das Umstellen aller Klassen. Die Zeit für die JUnit Tests und den Rest bleibt gleich, da die Anzahl der Attribute keinen Einfluss darauf hat. Mit Hilfe der gewonnenen Daten und erstellten Formeln kann nun mit einer gewissen Varianz die benötigte Zeit der Umstellung für ein Produkt angegeben werden. Nicht beachtet in diesen Zeiten sind die fachlichen Tests der einzelnen Produkte für das Verhalten, welches nicht durch JUnit Tests abgedeckt ist.

3.13 Risikoabschätzung

3.13.1 Grenzen des hybriden Ansatzes

Am Anfang der Migration hatte ich die Annahme, dass es möglich sei, JPA und JDO komplett hybrid und transparent voneinander zu verwenden, dass sie von einander erben könnten und untereinander Attribute voneinander sein können. Jedoch wurde diese Annahme zu dem Zeitpunkt hinfällig, an dem eine persistence capable superclass auf JPA umgestellt wurde. Beim nächsten Ausführen der Tests tritt eine kodo Exception auf, dass für die persistence capable superclass keine Mapping Informationen vorliegen. Somit führte ich eine Reihe von Tests mit JPA und JDO Objekten durch, welche von einander erben oder Attribute untereinander sind.

Die Resultate sind, dass JDO Objekte nicht als Attribute in JPA Objekten vorkommen dürfen, wenn man diese persistieren will. Definiert man eine Relation zwischen ihnen, so erwartet JPA, dass das JDO Objekt eine Entität ist und die Anwendung startet nicht. Umgekehrt, also beim EInbinden eines JPA Objekts in ein JDO Objekt, würde die Anwendung starten. Jedoch würde JDO das JPA Objekt als BLOB in der Datenbank speichern, da es keine Informationen zu der abhängigen Klasse besitzt. Erbt ein JPA Objekt von einem JDO Objekt, so gibt es dem ersten Anschein nach keine Probleme, jedoch landen später lediglich die Daten der eigenen Attribute in der Datenbank. Die Attribute der Oberklasse werden verworfen. Dreht man die Angelegenheit um, so kommt es wie vorher bereits beschrieben zu dem Fehler, dass JDO keine Informationen zu der JPA Oberklasse besitzt.

Der Konsens daraus ist, dass nur eine komplett hybride Architektur funktionieren kann, weil nur dann für jede persistente Klasse einerseits die Informationen für JPA über Annotationen definiert sind und andererseits die Informationen für JDO über die JDO und Mapping Dateien definiert sind. Damit wird ein Produkt, solange es JDO intern verwendet, auch die JDO persistenten Klassen der Komponenten verwenden. Erst wenn es intern zu JPA migriert, wird es die JPA Komponenten verwenden können. Somit muss auch bei Neuentwicklungen immer die Entscheidung getroffen werden, ob diese ebenfalls für die älteren Produkte zur Verfügung gestellt werden sollen. Dieser Ansatz funktioniert jedoch nicht problemlos, denn KODO verwendet Auto-Enhancement. Somit passt KODO für alle Klassen, für die es JDO Dateien findet, an und fügt Attribute zu diesen hinzu. Diese Attribute will JPA jetzt ebenfalls persistieren, da jedoch eines dieser Attribute vom Typ Objekt ist, kommt es zu einem Fehler, da JPA nicht entscheiden kann, welchen Datenbanktypen dieses Attribut bekommen kann. Eine Lösung für das Problem bietet wieder die Funktionalität zur Überschreibung der Attribute. Man legt eine orm.xml Datei an, welche alle Entitäten enthält, welche auch für KODO bekannt sind und fügt die Eigenschaft *Transient* zu allen KODO spezifischen Attributen hinzu (vgl. Listing 3.17).

```
...
<entity class="package.ChecklistenEintrag">
    <attributes>
   <transient name="jdoDetachedState"/>
    </attributes>
  </entity>
<entity class="package.ChecklistenReference">
    <attributes>
   <transient name="jdoDetachedState"/>
    </attributes>
```

```
  </entity>
...
```

Listing 3.17: Kennzeichnung der Kodo Attribute als transient für JPA

Auch andersherum muss eine Anpassung vorgenommen werden. KODO möchte von sich aus jedes Attribut einer Klasse persistieren, welches nicht mit transient gekennzeichnet wurde. Da für JPA der Primärschlüssel als neues Attribut hinzugefügt werden muss, muss auch die entsprechende JDO Datei angepasst werden. Es wird ein neuer Eintrag für das Feld hinzugefügt, welcher KODO mitteilt, dieses Feld zu ignorieren (vgl. Listing 3.18).

```
...
<field name="jpaId" persistence-modifier="none" />
...
```

Listing 3.18: Kodo ignoriert die jpaid bei der Persistierung

3.13.2 Risiken und ihre Auswirkungen

Dieses Kapitel soll eine Übersicht über die unterschiedlichen Risiken für die Migration darstellen. Dafür erfolgt eine Unterteilung in verschiedene Kategorien anhand ihrer Gefahr für die Migration.

Kategorie A beschreibt dabei die kleinen Risiken, welche die Migration um einen Faktor von 1,2 verlängern. Dazu gehören JUnit 3 Tests, weil diese erst auf JUnit 4 Tests umgestellt werden müssen, damit diese die Rule zur Persistierung verwenden können. Die Migration eines JUnit 3 Tests zu einem JUnit 4 Test ist jedoch nicht immer ganz trivial, wodurch dies auch Auswirkungen auf die JPA Migration hat. Ebenfalls zur Kategorie A gehört eine vermehrte Anzahl von JUnit Tests für die Persistenz, da jeder einzeln umgestellt werden muss und somit, wie bereits in Kapitel 3.12.3 “Prüfung der Ergebnisse” beschrieben, die Migration verlangsamen. In diesem Kapitel hat man gut gesehen, wie die gesamte Migration 20 Prozent länger gedauert hat, weil die benötigte Zeit für die JUnit Tests deutlich über dem Erwarteten lag.

Kategorie B beschreibt die größeren Gefahren, welche die Migration um einen Faktor von 1,6 verlangsamen. Dazu gehören Weiterentwicklungen am QueryBuilder, weil eine Funktion verwendet wird, welche bisher am Builder vorbeigearbeitet hat. So ist es bisher in beiden Prototypen vorgekommen, dass der Builder erweitert werden musste. Bei dem ersten Prototypen dauerte die Migration dadurch 1,55 mal länger und bei dem zweiten Prototypen 1,68 mal länger.

3.14 Automatisierung

3.14.1 Was kann man automatisieren?

Automatisierung spielt gerade bei dieser Migration eine große Rolle, da es wiederkehrende Muster gibt, welche bei jeder zu migrierenden Komponente oder Produkt auftreten. Das Muster ist hierbei die Umstellung der Klassen auf JPA. Dafür erfolgt ein Sichten der JDO- und Mapping-Dateien und anschließend das Hinzufügen der äquivalenten Annotationen. Dies ist eine meist simple Aufgabe, welche relativ viel Zeit in Anspruch nehmen kann und zusätzlich sehr fehleranfällig ist. Somit wäre es sinnvoll diesen Schritt zu automatisieren. Zusätzlich kann das Erstellen der persistence.xml, sowie der orm.xml automatisiert werden.

3.14.2 Funktionsweise

Das entwickelte Tool scannt dabei alle Packages in einem übergebenen Projekt nach JDO- und Mappingdateien. Diese werden anschließend von einem Parser eingelesen. Dabei wird aus jedem XML-Element eine Klasse und aus den Attributen des Elements werden die Attribute der Klasse. Das entstandene Model kennt alle persistenten Klassen und ihre Felder mit den entsprechenden Mappinginformationen des Package. Als nächstes werden alle Klassen des Package überprüft. Befindet sich einer der Namen in der Liste der persistenten Klassen, so werden zu dieser Annotationen hinzugefügt. Dazu gehören die Annotationen @Entity, @Table, @Id, @Column, @Lob, @Temporal und @Enumerated für die einzelnen Attribute. Die benötigten Informationen für die Annotationen werden aus dem Model gelesen, da sie bereits für JDO definiert wurden. Zusätzlich wird zu der Klasse ein Attribut “jpaid” hinzugefügt, um einen Ersatz für den meist automatisch erstellten Primärschlüssel “JDOID” zu erhalten. Umgesetzt wird dies mit Hilfe von Abstract Syntax Trees (AST) von Java. Diese ermöglichen es, den Programmcode zur Laufzeit zu scannen und zu diesem Baum weitere Elemente, wie zum Beispiel Annotationen, hinzuzufügen (vgl. Listing 3.19).

```
...
final ListRewrite listRewrite = rewriter.getListRewrite(astRoot,
    CompilationUnit.TYPES_PROPERTY);
final NormalAnnotation entityAnnotation = ast.newNormalAnnotation();
entityAnnotation.setTypeName(ast.newName("Entity"));
entityAnnotation.values().add(createStringAnnotationMember(ast, "name", clazz.
    getName()));
```

```
...
MemberValuePair createStringAnnotationMember(final AST ast, final String
      name, final String value) {
   final  MemberValuePair memberValuePair = ast.newMemberValuePair();
   memberValuePair.setName(ast.newSimpleName(name));
   final  StringLiteral  stringLiteral  = ast.newStringLiteral();
   stringLiteral . setLiteralValue (value);
   memberValuePair.setValue(stringLiteral);
   return memberValuePair;
}
..
```

Listing 3.19: Erstellen einer @Entity-Annotation mit AST

Somit entsteht eine Klasse, bei der lediglich die Relationen noch ergänzt werden müssen.

3.14.3 Nutzen

Durch die Automatisierung ist es möglich, bei der Migration der Komponenten ungefähr 33 Prozent der Zeit einzusparen. Bei den Produkten ist es wiederum abhängig davon, wie viele Attribute die Klassen besitzen, aber dort kann man, wie bereits im Kapitel 3.12.4 “Hochrechnungen Produkte” beschrieben, bis zu 60 Prozent der Zeit sparen. Zusätzlich verschwinden durch die Automatisierung Flüchtigkeitsfehler, die bei solch einfachen Aufgaben häufiger auftreten. Ein weiterer positiver Effekt der Automatisierung ist die erleichterte Nachbesserung bei erkannten Fehlern. Wird festgestellt, dass eine der JPA Annotationen doch nicht das gleiche Verhalten aufweisen, wie ihr JDO Pendant, so muss lediglich das Tool angepasst werden. Mit diesem angepassten Tool ist es jetzt möglich alle fehlerhaften Klassen auf einmal zu bearbeiten.

3.15 Resultierende Migrationsstartegie für die tolina GmbH

Nachdem alle hoffentlich notwendigen Informationen gesammelt wurden, kann nun die Migrationsstrategie genauer beschrieben werden.

- Schritt 1 Migration der Komponenten
  Da die Verwendung beider Spezifikationen innerhalb eines Produktes zu vielen unvorhersehbaren Fehlern führen kann, muss als erster Schritt eine Migration der Komponenten vorgenommen werden.

Dafür muss die hybride Version des PMWrappers im pom verwendet werden und die Abhängigkeit zu Hibernate muss hinzugefügt werden. Als nächstes sollten die Kompilierungsfehler beseitigt werden. Anschließend kann das Tool zur Automatisierung verwendet werden. Dies erstellt einem die persistence.xml, die orm.xml und verziert die persistenten Klassen mit den Annotationen @Entity, @Table, @Column, @Lob, @Temporal und @Enumerated für die Attribute und erstellt einen Primärschlüssel mit den Annotationen @Id und @GeneratedValue. Danach müssen von Hand die Relationen zu den persistenten Klassen hinzugefügt werden. Zusätzlich soll JDO das neue Feld ignorieren. Als letztes kann man sich um das Korrigieren der JUnit-Tests kümmern. Diese Vorgänge müssen für alle Komponenten wiederholt werden.

- Schritt 2 Migration des Referenzprojektes
  Nach der Migration der Komponenten folgt die Umstellung des Referenzprojektes. Dies ist ein Core Projekt der Firma, welches als Basis für neue Projekte verwendet wird. Somit wird sichergestellt, dass alle neu erstellten Projekte bereits die JPA Varianten der Komponenten benutzen.

- Schritt 3 Migration der Ableger des Referenzprojektes
  Als nächster Schritt erfolgt die Migration der aktiven Ableger des Referenzprojektes. Da die Basis bereits umgestellt ist und zu diesem Zeitpunkt schon einige Zeit fehlerfrei gelaufen sein wird, kann man anschließend diese Ableger umstellen.

- Schritt 4 Migration der Produkte
  Als letzter Schritt folgt die Migration aller Produkte, in denen noch aktiv Entwicklung stattfindet. Dies sind oft alte Produkte, bei denen die Migration, aufgrund starker Verwurzelung mit JDO, schwierig werden könnte. Bis zu diesem Zeitpunkt sollten jedoch alle Risiken durch die vorherigen Migrationen bekannt sein, sodass auch diese Migration fehlerfrei funktioniert. Produkte, welche zwar ausgeliefert sind, aber nur noch gewartet werden, müssen selbst entscheiden, ob sie migrieren wollen.

Für die Migration in den Schritten 2, 3 und 4 gelten die gleichen Schritte, wie bei Schritt 1.

Kapitel 4

Qualität

4.1 Hilft eine gute Testabdeckung bei der Migration?

Tests helfen dabei verschiedene Kategorien von Fehlern bei der Migration aufzudecken. Dazu gehören folgende:

- Fehler bei der Umstellung der Entitäten
- Fehler bei der Umstellung von Utility Klassen
- Fehler in der Architektur
- GAPs, also was muss noch entwickelt werden (Test Driven Migration)

Diese sollen in den nachfolgenden Kapiteln genauer beschrieben werden.

4.1.1 Fehler in Entity

Die meisten Fehlkonfigurationen von Entitäten wären auch ohne Tests aufgefallen, da diese bereits bei der Initialisierung des EntityManagers auffallen. Dazu zählen Fehler wie, dass keine ID für die Klasse existiert oder dass es keinen Standardkonstruktor gibt. Auch Fehler in Relationen - es gab für eine Relation nicht die entsprechende Rückreferenz - wären ohne Tests aufgefallen. Aber es gab durchaus Fehlkonfigurationen, welche erst durch das Ausführen der Tests aufgefallen sind. So

ist es bei der Verwendung von Relationen durchaus nützlich, das Attribut *cascade* mit zu definieren. Es gab Testfälle in denen ein Objekt und die Objekte seiner Relation erzeugt wurden. Anschließend sollte das Objekt persistiert werden, was jedoch zu einem Fehler führte, da innerhalb der Datenbank versucht wurde, die Relation zu den abhängigen Objekten herzustellen. Diese waren aber zu diesem Zeitpunkt noch nicht persistent. Durch das Definieren des Attributs *cascade* werden die gerade erstellten Objekte der Relation ebenfalls persistiert, wenn das Objekt selbst persistiert wird. Daraus resultierte, dass ich dieses Attribut nahezu für jede Relation definiert hatte. Beim zweiten migrierten Prototypen gab es jedoch einige Tests, welche das Löschen von Objekten von Relationen oder das Vorhandensein eines Objekts nach dem Löschen des zugehörigen Objekts überprüften. Dadurch wurden Fehler entdeckt, welche darauf hinwiesen, dass das Attribut *cascade* oft falsch verwendet wurde. In diesem Rahmen hatte ich mich dazu entschlossen mich noch einmal genauer mit diesem Attribut zu beschäftigen, da das Verhalten nicht immer das gewünschte war.

4.1.2 Fehler im Builder und PMWrapper

Bereits nach der ersten Änderung, also dem Umsetzen des Delegate Design Pattern, auf dem PMWrapper schlugen die ersten Tests fehl. Dadurch entstand zum Beispiel ein neuer Konstruktor, welcher ein übergebenes persistententes Objekt erhält. In JDO ist es möglich, zu jedem Objekt den zugehören PersistenceManager zu bekommen. Da diese Funktionalität in Tests und auch im Produktivcode häufig verwendet wurde, musste dieser neue Konstruktor entstehen. Weiterhin deckten JUnit Tests die Fehler des QueryBuilders bezüglich Null-Safe auf.

4.1.3 Architekturfehler

Wie bereits beschrieben hatte ich bis zur Umsetzung des Prototypen die Annahme, dass JDO und JPA auch miteinander funktionieren können - in einer Hybriden Welt. Durch das Ausführen der JUnit Tests, welche Funktionalitäten aus Komponenten verwendet hatten, die noch nicht auf JPA umgestellt waren, bemerkte ich, dass diese Annahme falsch sein könnte.

Resultierend habe ich einige kleine JUnit Tests geschrieben, welche diese Ergebnisse validieren sollten. Es stellte sich heraus, dass die Ergebnisse korrekt waren und meine Annahme somit falsch war. Ohne JUnit Tests wären diese Fehler erst deutlich später aufgefallen.

4.1.4 Test Driven Migration

Ein weiterer Nutzen der JUnit Tests galt dem Verstehen von vorhandenen Funktionalitäten. So verstand ich zum Beispiel erst durch einen entsprechenden JUnit Test den kompletten Umfang des QueryBuilders. Gerade das verschachteln der FilterANfragen des Builders war für mich anfangs unklar. Durch das Ausführen der Tests konnte ich sehen welche Filter durch welche BuilderAufrufe entstanden. Damit konnte die Funktionalität für JPA entwickelt werden. Damit kommen wir auch zum letzten und in meinen Augen wertvollsten Verwendungszweck für JUnit Tests; und zwar das Herausfinden von GAPs. So erfuhr ich durch das Ausführen der JUnit Tests, welche Funktionalitäten von JDO verwendet wurden. Es gab jedesmal Exceptions, wenn versucht wurde, JDO Operationen auf JPA Entitäten anzuwenden. Damit konnten diese Lücken in der Entwicklung gefunden und beseitigt werden. Entweder wurde etwas Äquivalentes für JPA entwickelt oder aber es wurde versucht über einen Delegator intern wieder zu entscheiden, ob JPA oder JDO verwendet werden soll. So entstand zum Beispiel auch der vielmals erwähnte QueryBuilder, welcher unabhängig davon funktioniert, ob es sich um JDO oder JPA handelt. Auch spätere Erweiterungen des Builders wurden erst durch JUnit Tests identifiziert. Diese Vorgehensweise würde ich als TestDrivenMigration bezeichnen.

4.2 Gewährleistung der Funktionalität

Vor der Anpassung von JDO Funktionalitäten erfolgte als erstes eine Überprüfung, ob es für diese Funktion bereits einen Test gab. Falls dieser noch nicht existierte, musste ein neuer geschrieben werden, um sicherzustellen, dass die Funktionalität auch nach der Umstellung noch gegeben war. Anschließend folgte die Umstellung, sodass JPA das gleiche Verhalten aufwies, welches abschließend durch einen Test sichergestellt wurde. Ein Beispiel hierfür war die simpleMatch Methode des QueryBuilders. Diese funktionierte für JDO, war jedoch in der existierenden Form für JPA nicht nutzbar. Somit wurde ein Test entwickelt, welcher die Funktionalität von simpleMatch sicherstellt. Dadurch wur-

den Fehler bei den notwendigen Refactorings schnell aufgedeckt. Ein weiteres Beispiel war die Umstellung des bereits existierenden Query-Builders für JDO auf einen allgemeinen Query-Builder für JDO und JPA. Dabei machte ich von den bereits zahlreich existieren JUnit Tests für den JDO Teil Gebrauch, um äquivalente Tests für den JPA Teil zu schreiben. Damit konnte ich gleichzeitig gewährleisten, dass die Query für JDO und für JPA die gleichen Ergebnisse liefert.

Kapitel 5

Nach der Migration ist vor der Migration

In dieser Arbeit wurde gezeigt, wie eine Migration von JDO nach JPA vorgenommen werden kann. Dafür erfolgte eine Analyse der Unterschiede beider Spezifikationen. Es wurden Ideen genannt, welche Migrationsstrategien verfolgt werden können. Es erfolgte eine detaillierte Beschreibung der hybriden Migrationsstrategie sowie eine prototypische Umsetzung dieser. Dabei wurden die zu migrierenden Funktionalitäten (Konfiguration, Metadaten, Mapping und Queries) ausführlich beschrieben und mit zahlreichen Beispielen erklärt. Die Kosten der Migration wurden tabellarisch zusammengefasst und es erfolgte eine Beschreibung der Risiken für die Migration, welche diese verlangsamen könnte. Probleme des hybriden Ansatzes wurden detailliert erklärt und festgestellt, dass sie die Migration lediglich erschweren, aber nicht verhindern. Es wurde etwas über Automatisierung gesprochen. Was automatisiert werden kann, wie es automatisiert werden kann und was diese Automatisierung letztendlich bringen kann. Abschließend wurden Tests betrachtet. Einerseits, wie Tests bei der Migration helfen können und andererseits das Schreiben von neuen Tests, um auch die hybriden Funktionalitäten sicher zu stellen.

Was ist jetzt mit den eingangs gestellten Forschungsfragen? Die Fragen waren:

- Ist eine Migration von JDO zu JPA möglich?
- Welche Strategien gibt es?
- Was sind die Kosten und Risiken der Migration?

- Helfen vorhandene Tests bei der Migration?

Es wurde mit Hilfe eines Prototypen gezeigt, dass die Migration von JDO zu JPA möglich ist. Dabei wurden zwei Migrationsstrategien evaluiert und die für die tolina GmbH passende hybride Methode ausgewählt. Anhand dieser Methode wurden die Kosten der Migration genannt und Risiken, welche die Migration verlangsamen, erwähnt. Wie in Kapitel 4 dargestellt, helfen Tests bei der Migration, vorrausgesetzt sie testen die Persistenz.

Neue interessante Forschungsfragen wären zum Beispiel:

- Wie einheitlich sind eigentlich die verschiedenen JPA Provider?
- Warum setzen sich nur OpenSource Lösungen durch?
- Ist JDO wirklich "tot" oder kommt als es als OpenSource nochmal zurück?

In künftigen Studien wäre das Testen weiterer Migrationsstrategien lohneswert. Außerdem wäre es interessant zu überprüfen, ob eine schwierigere zugrunde liegende Architektur vielleicht ein großes Hindernis für eine Migration ist. Wie sieht das Ganze eigentlich im Java-EE Bereich aus, in dem JPA wirklich viele Vorteile bietet?

Die Schwächen dieser Arbeit liegen an dem starken Bezug zur tolina GmbH. Damit wird hauptsächlich kodo als zugrundeliegender JDO Provider betrachtet und weniger die JDO Spezifikation. Außerdem wurde die hybride Migrationsstrategie gewählt, da es Produkte gibt, für die keine Migration mehr vorgenommen werden soll. Gerade bei kleineren Firmen kann es wesentlich effizienter sein, alles direkt auf JPA umzustellen. Dadurch, dass die tolina GmbH hauptsächlich im Java-SE Umfeld tätig ist, wurde die Unterstützung von JPA für Java-EE fast gar nicht betrachtet. Dazu gehören automatische Transaktionssteuerung, Generierung von EntityManagern und Injektion dieser in den benötigten Klassen (vgl. Kapitel 2.2). Weiterhin wurde lediglich gezeigt, dass und wie es möglich ist mit JPA das gleiche Verhalten zu erreichen wie mit JDO. Somit wurde das existierende Verhalten nicht in Frage gestellt, obwohl JPA vielleicht bessere Alternativen bildet.

Wie sieht es in Zukunft mit Migrationen aus? Dies hängt davon ab, wie stark die Verwendung von JDO noch ist. Meiner Meinung nach wird es da noch einige Unternehmen geben, welche längerfristig gesehen nach JPA migrieren wollen. Aber der Inhalt dieser Arbeit ist beliebig anwendbar auf weitere Migrationen von Persistenzschichten. So wird

es vielleicht irgendwann einen neuen Standard geben und es wird eine Migration von JPA zu diesem neuen Standard benötigt. Falls dies der Fall sein sollte, ermöglicht die bereits existierende Architektur einen leichteren Umstieg auf neue Standards.

Literaturverzeichnis

[Cur09] Rick Curtis. What is enhancement anyway?, 2009. Erreichbar unter http://websphere persistence.blogspot.de/2009/02/openjpa-enhancement.html; besucht am 07.02.2013.

[ecl13] Eclipselink/maven, 2013. Erreichbar unter http://wiki.eclipse.org/EclipseLink/Maven; besucht am 07.02.2013.

[hib13a] Hibernate jpa doku, chapter 3. persistence contexts, 2013. Erreichbar unter http://docs.jboss.org/hibernate/orm/4.1/devguide/en-US/html/ch03.html#d5e836; besucht am 08.02.2013.

[hib13b] Hibernate jpa doku, chapter 6. entity listeners and callback methods, 2013. Erreichbar unter http://docs.jboss.org/hibernate/orm/4.0/hem/en-US/html/listeners.html; besucht am 08.02.2013.

[hisa] Eclipselink - geschichte. Erreichbar unter http://de.wikipedia.org/wiki/EclipseLink; besucht am 14.04.2013.

[hisb] Hibernate(java) - history. Erreichbar unter http://en.wikipedia.org/wiki/Hibernate_(Java)#History; besucht am 14.04.2013.

[hisc] Openjpa - what is the history of openjpa? Erreichbar unter http://openjpa.apache.org/faq.html; besucht am 14.04.2013.

[Kei09] Mike Keith. What's new and exciting in jpa 2.0, 2009. Erreichbar unter http://jazoon.com/portals/0/Content/ArchivWebsite/jazoon.com/jazoon09/download/presentations/8461.pdf; besucht am 12.04.2013.

[kod03] Solarmetric kodo™ jdo 3.4.1 developers guide, 2003.

[kom] Komponentenbasierte entwicklung. Erreichbar unter `http://de.wikipedia.org/wiki/Komponentenbasierte_Entwicklung`; besucht am 31.01.2013.

[Kru02] Jacek Kruszelnicki. Persist data with java data objects, part 2, 2002. Erreichbar unter `http://www.javaworld.com/jw-04-2002/jw-0412-jdo.html?page=1`; besucht am 15.02.2013.

[LR03] P. Levi and U. Rembold. *Einführung in die Informatik für Naturwissenschaftler und Ingenieure*. Hanser Fachbuchverlag, 2003.

[Nit10] KL Nitin. The jpa 2 enhancements every java developer should know, 2010. Erreichbar unter `http://www.developer.com/java/ent/the-jpa-2-enhancements-every-java-developer-should-know.html`; besucht am 08.02.2013.

[O'B] Marc O'Brien. Maven: The complete reference, kapitel 1.1: Maven... what is is? Erreichbar unter `http://books.sonatype.com/mvnref-book/reference/`; besucht am 05.07.2013.

[obja] Object db jdo doku- jdoql. Erreichbar unter `http://www.objectdb.com/database/jdo/manual/chapter7`; besucht am 1.03.2013.

[objb] Objectdb jpa doku, jpa query structure (jpql / criteria). Erreichbar unter `http://www.objectdb.com/java/jpa/query/jpql/structure`; besucht am 08.02.2013.

[oraa] Oracle kodo doku kapitel 10.8. query aggregates and projections. Erreichbar unter `http://docs.oracle.com/cd/E13189_01/kodo/docs316/ref_guide_aggregates_and_projections.html#ref_guide_aggregates_avg`; besucht am 14.03.2013.

[orab] Oracle kodo doku kapitel 2.5. jdo standard properties. Erreichbar unter `http://docs.oracle.com/cd/E13189_01/kodo/docs316/ref_guide_conf_jdo.html`; besucht am 16.05.2013.

[orac] Oracle kodo doku kapitel 3. jdo architecture. Erreichbar unter `http://docs.oracle.com/cd/E13189_01/kodo/docs40/full/html/jdo_overview_arch.html`;.

[orad] Oracle kodo doku kapitel 4. persistencecapable. Erreichbar unter `http://docs.oracle.com/cd/E13189_01/kodo/docs40/full/html/jdo_overview_pc.html`;.

[PDBM12] Harald Wehr Prof. Dr. Bernd Müller. *JAVA PERSISTENCE API 2*. Hanser Fachbuchverlag, 2012.

[per] Itwissen-persistenz. Erreichbar unter `http://www.itwissen.info/definition/lexikon/Persistenz-persistence.html`; besucht am 21.03.2013.

[que] Advanced query options (hibernate). Erreichbar unter `http://what-when-how.com/hibernate/advanced-query-options-hibernate/`; besucht am 14.02.2013.

[Sch08] Frank Schwarz. Jdo-jpa-migrationsstrategien, 2008. Erreichbar unter `http://www.buschmais.de/2008/08/04/jdo-jpa-migrationsstrategien/`; besucht am 31.01.2013.

[Whi11] Jim White. Hibernate optimistic lock without a version, 2011. Erreichbar unter `http://www.intertech.com/Blog/hibernate-optimistic-lock-without-a-version-or-timestamp/`; besucht am 16.05.2013.

[wik10] Java persistence/what is new in jpa 2.0?, 2010. Erreichbar unter `http://en.wikibooks.org/wiki/Java_Persistence/What_is_new_in_JPA_2.0%3F`; besucht am 13.04.2013.