

**Evaluation von Java Data Objects 2.0 -
Objektpersistenz durch O/R-Mapping**

Diplomarbeit

vorgelegt von

Christian Hedemann
aus Mönchengladbach

geboren am: 11.08.1978
Matrikel-Nr.: 558474

Hochschule Niederrhein
Fachbereich Wirtschaftswissenschaften
Studiengang Wirtschaftsinformatik

Wintersemester 2005 / 2006

Referent: Prof. Dr. Dietmar Abts
Korreferent: Prof. Dr. Berthold Stegemerten

Gliederung

ABKÜRZUNGSVERZEICHNIS	6
VORWORT	8
1 EINLEITENDES.....	10
1.1 Einführung.....	10
1.1.1 Über diese Diplomarbeit	10
1.1.2 Verwendete Software	11
1.2 Problemstellung.....	12
2 HISTORISCHE ENTWICKLUNG.....	13
2.1 Entwicklung der Datenbanken	13
2.1.1 Einleitung	13
2.1.2 Dateien	13
2.1.3 Hierarchisches Datenbankmodell.....	13
2.1.4 Netzwerkdatenbanken	14
2.1.5 Relationale Datenbanken	15
2.1.6 Objektrelationale Datenbanken	15
2.1.7 Objektdatenbanken.....	16
2.1.8 Sonderfall O/R-Mapping.....	17
2.2 Entwicklung der APIs.....	17
2.2.1 Einleitung	17
2.2.2 Klartext / Binärmodus	17
2.2.3 Serialisierung.....	17
2.2.4 JDBC	18
2.2.5 EJB	18
2.2.6 Hibernate	19
2.2.7 JDO	20
2.3 Zusammenfassung.....	22
3 JAVA DATA OBJECTS 2.0.....	23
3.1 Entwicklungsprozess.....	23
3.2 Anwendungsbeispiele / Architekturen	25
3.2.1 Architekturen mit Fokus auf der JVM	25
3.2.1.1 Eine JVM und ein bis viele PersistenceManager	25
3.2.1.2 2nd Level Cache.....	27
3.2.2 Datenspeicherzugriff	29
3.2.2.1 Direkter Zugriff auf das Dateisystem.....	29
3.2.2.2 Zugriff über JDBC	30
3.2.2.3 Zugriff über einen Resource Adapter.....	30
3.2.3 Systemarchitekturen mit JDO Applikationen	30
3.2.3.1 Einzelplatzrechner mit lokalem Datenspeicher.....	30

3.2.3.2 JDO Applikation in einem Webserver / als Webservice.....	31
3.2.3.3 Verschiedene Umsetzungen für Applikationsserver.....	32
3.3 Unterschiede zu JDBC.....	34
3.3.1 In der Syntax	34
3.3.2 In der Performance.....	35
3.3.3 In dem Arbeitsaufwand.....	36
3.3.4 In dem Datenmodell.....	38
4 JDO 2.0 IM DETAIL.....	40
4.1 Installation	40
4.2 Vorbereitung.....	40
4.2.1 Interfaces	40
4.2.1.1 Übersicht	40
4.2.1.2 API	41
4.2.1.3 Exceptions	42
4.2.2 Konfiguration	44
4.3 Persistente und flüchtige Klassen	45
4.4 O/R-Mapping.....	46
4.4.1 Einleitung	46
4.4.2 Class Mapping.....	47
4.4.3 Zweittabellen.....	48
4.4.4 Eingebettete Objekte	49
4.4.5 Serialisierte Objekte	50
4.4.6 Constraints.....	50
4.4.7 Vererbung.....	51
4.4.8 Interfaces	52
4.4.9 Objekte (FCOs)	52
4.4.10 Arrays	53
4.4.11 1:1 Beziehungen.....	54
4.4.12 Sammlungen.....	54
4.4.12.1 Allgemeines.....	54
4.4.12.2 1:N Collection	55
4.4.12.3 1:N Set.....	56
4.4.12.4 1:N List.....	56
4.4.12.5 1:N Map	57
4.4.13 N:1 Beziehung.....	57
4.4.14 M:N Beziehung	57
4.4.15 Compound Identity Relationship	58
4.4.16 Voneinander abhängige Felder	59
4.5 Bytecode-Enhancer	60
4.6 Persistence-Manager.....	60
4.6.1 Einleitung	60
4.6.2 Create	61
4.6.3 Read.....	62
4.6.4 Update	62
4.6.5 Delete	62

4.6.6 Callback-Methoden und Listener	63
4.7 JDOQL	64
4.7.1 Die Default-Fetch-Group	64
4.7.2 Extents	65
4.7.3 Queries	65
4.7.3.1 Allgemeines.....	65
4.7.3.2 Query-Objekte und deren Methoden.....	67
4.7.3.3 Single String Queries	67
4.7.3.4 SQL Queries.....	68
4.7.3.5 Named Queries.....	68
4.8 Identität.....	68
4.8.1 Datastore-Identity.....	68
4.8.2 Application-Identity	70
4.8.3 Nondurable-Identity	71
4.9 Besondere Zugriffsmechanismen.....	71
4.9.1 Cache.....	71
4.9.2 Nicht-transaktionaler Zugriff	72
4.9.3 Optimistische Transaktionen.....	73
5 NACHSATZ EJB 3.0 / JDO 2.0	74
5.1 Persistenzmodell EJB 2.0.....	74
5.2 Warum noch JDO?	77
5.3 Der Streit um EJB 3.0 und JDO 2.0	78
6 EVALUATION.....	81
6.1 Die aktuelle Situation um JDO 2.0	81
6.2 Ergebnisse wirtschaftlich gesehen	84
6.2.1 Die Faktoren Personal, Kosten und Zeit	84
6.2.2 Investitionssicherheit.....	86
6.2.3 Risiken.....	87
6.2.4 Chancen.....	87
7 FAZIT	89
ANHANG	90
A Lebenszyklen der Klassen innerhalb JDO.....	90
B JDO 2.0 Metadaten DTD	92
C JDOQL Backus-Naur-Form.....	101
E Verbindungen zwischen JDO-Interfaces.....	109
F Vererbungsbaum der JDO-Exceptions.....	110

LITERATURVERZEICHNIS.....	111
VERSICHERUNG	113

Abkürzungsverzeichnis

API	Application Programming Interface
Aufl.	Auflage
BLOB	Binary Large Object
BMP	Bean Managed Persistence
CAD	Computer Aided Design
CMP	Container Managed Persistence
CPU	Central Processing Unit
CRUD	Create, Read, Update, Delete
DBMS	Datenbankmanagementsystem
DFG	Default Fetch Group
DTD	Document Type Definition
EIS	Enterprise Information System
EJB	Enterprise Java Beans
EJBQL	Enterprise Java Beans Query Language
ER	Entity Relationship
FCO	First Class Object
FUD	Fear, Uncertainty, Doubt
H.	Heft
HQL	Hibernate Query Language
I/O	Input / Output
ID	Identifikator
IDE	Integrated Development Environment
JAR	Java Archive
Java EE	Java Enterprise Edition
Java ME	Java Micro Edition
Java SE	Java Standard Edition
JDBC	Java Database Connectivity
JDO	Java Data Objects
JDOQL	Java Data Objects Query Language
Jg.	Jahrgang
JNDI	Java Naming and Directory Interface
JPOX	Java Persistent Objects
JSP	Java Server Pages
JVM	Java Virtual Machine
lv.	Level
O/R-Mapping	Object-Relational-Mapping
ODBMS	Objektdatenbankmanagementsystem
ODMG	Object Data Management Group
ORDBMS	Objektrelationales Datenbankmanagementsystem
ORM	Object-Relational-Mapping
PKW	Personenkraftwagen
POJO	Plain Old Java Object
RDBMS	Relationales Datenbankmanagementsystem
RI	Reference Implementation
S.	Seite
SPI	Service Provider Interface
TCK	Test Compatibility Kit
UUID	Universally Unique Identifier
VM	Virtual Machine

XML	Extensible Markup Language
XML-DD	Extensible Markup Language Deployment Descriptor

Vorwort

Knapp fünfeinhalb Jahre ist es nun her, dass die Expertengruppe zum Java Specification Request 12 zum ersten Mal zusammentrat und die Arbeit an der Spezifikation für Java Data Objects in Angriff nahm. Seit diesem Zeitpunkt hat sich viel verändert, sowohl JDO selbst betreffend, als auch im Umfeld dieses Frameworks. JDO legte einen mäßigen Start hin, verfehlte lange Zeit das öffentliche Interesse, kam unter stillem Protest mancher Mitglieder des „Executive Committee for Java SE/EE“ in die Version 2.0, weitere Konkurrenten betraten den Markt und in manchen Foren weiß man aufgrund der dort gewählten Worte und Argumente nicht direkt, ob dort eine Diskussion über Persistenzframeworks stattfindet, oder ob es sich um den ewigen Kampf „Windows gegen Linux“ dreht.

Verfolgt man die Geschichte dieser Programmierschnittstelle genauer, wird einem schnell klar, dass diese Entwicklung erst dann verstanden werden kann, wenn die nähere und weitere Umgebung von JDO in die Betrachtungen miteinbezogen wird.

Ist man jedoch erst so tief in die Materie mit seinem Grundwissen über Java und Datenbanken eingestiegen, stellen sich einem viele Fragen. Eine der häufigsten Fragen, die mich bei der Erkundung von Java Data Objects begleitet hat, war das „warum?“, das selbst in der Fachwelt nicht explizit beantwortet wird.

Allen voran Fragen wie:

- Warum nicht direkt eine andere Datenbank verwenden?
- Warum ist Java Data Objects nicht populärer?
- Warum sollte man JDO benutzen, wenn die ersten zwei Fragen nicht zureichend geklärt sind?

Auf den folgenden Seiten werde ich auf die oben stehenden Fragen ausführlich eingehen und dem Leser dieser Arbeit eine Anleitung über die Funktionsweise des Frameworks liefern, aufzeigen weshalb die Verwendung von JDO eine ernste Überlegung wert ist und wieso eine andere Datenbank die entstehenden Probleme in der Softwareentwicklung unter Umständen nur verschiebt. Eine bestimmte Pressemeldung seitens Sun

Microsystems Inc. erschien nur auf den ersten Blick als vernichtender Schlag und letztendlich war eine eigenständig entworfene Datenbank für Computer- und Videospiele das Fundament für den Prototyp einer Anwendung, der als „Proof of Concept“ dieser Diplomarbeit diente. Aufgrund der Verwendung des Quelltextes für die Beispiele und deren ausführliche Erklärung in den jeweiligen Kapiteln wurde auf einen Abdruck der Dateien im Anhang verzichtet. Die Quelldateien und ein Diagramm des Datenmodells befinden sich auf der beigelegten CD.

1 Einleitendes

1.1 Einführung

1.1.1 Über diese Diplomarbeit




Der Inhalt dieser Arbeit zielt im ersten Teil darauf ab, dem Leser einen Überblick über die bisherige Entwicklung der Datenbanken in Unternehmen bis zur Gegenwart zu vermitteln und den damit verbundenen Problemen, denen sich Softwareentwickler bei der Erstellung neuer Programme stellen müssen.

Im anschließenden Teil folgt eine Darstellung eines typischen Entwicklungsprozesses in der Benutzung von JDO mit einer ausführlichen Beschreibung der Syntax des Frameworks und der Verwendung im Kontext einer Applikation. Der Schwerpunkt dieser Arbeit bildet der Vergleich mit JDBC, da hier die Unterschiede in der Konzeption am gravierendsten ausfällt. Andere Lösungen unterscheiden sich zu JDO weniger in den Grundlagen, sondern in einigen Details der Umsetzung und dort um so mehr.

Gegen Ende folgt eine Gegenüberstellung von EJB und JDO, deren Geschichte und deren Anwendungsbereich eng miteinander verwoben sind und dies nicht zuletzt wegen einiger aktueller Ereignisse, die hauptsächlich die Enterprise Edition betreffen.

Den Schluss der Arbeit bildet eine wirtschaftliche Bewertung JDOs in den Bereichen „Personal, Kosten und Zeit“, „Investitionssicherheit“, „Risiken“ und „Chancen“.

Für die Diagramme in dieser Diplomarbeit gilt folgende Legende:

-  Austausch von Daten über Methodenaufrufe
-  Weg der Objekte aus dem Datenspeicher in die Anwendung
-  Datenaustausch zwischen Client und Server (RMI, HTTP, etc.)

Die gestrichelten Teile der Diagramme 3.1 und 3.2 sind als optional zu verstehen.

Syntax und Quellcode sind im Schriftsatz *Courier New* abgedruckt.

1.1.2 Verwendete Software

Die verwendeten Programme und Klassen für die Beispieldatenbank sind:

DBMS:	MySQL 5.0.16	(www.mysql.com)
IDE:	Eclipse 3.1	(www.eclipse.com)
	Eclipse JPOX Plugin 1.1.0-beta-2	(www.jpox.org)
Java Runtime:	Version 1.5.0_04	(java.sun.com)
Java Klassen:	JPOX 1.1.0-beta-5	(www.jpox.org)
	JPOX Enhancer 1.1.0-beta-5	(www.jpox.org)
	Sun JDO-2.0-snapshot7	(www.jpox.org ¹)
	log4j 1.2.12	(jakarta.apache.org/log4j)
	bcel 5.1	(jakarta.apache.org/bcel)
	MySQL Connector Java 3.1.2	(www.mysql.com)
Webserver:	Tomcat 5.5.9	(tomcat.apache.org)
Hilfsprogramme:	phpMyAdmin 2.6.3-pl1	(www.phpmyadmin.net)
	fabForce DBDesigner 4	(www.fabforce.net)
	Sysdeo Eclipse Tomcat Launcher Plug-In 3.1	(www.sysdeo.com)

Auf einen Applikationsserver habe ich verzichtet, da ich zur Durchführung des Beispiels Wert auf den kleinsten, gemeinsamen Nenner gelegt habe und die verwendete Konstellation (Tomcat, Java SE und MySQL) auf den meisten kleineren und mittleren Webhosting-Angeboten Verwendung findet. Diese Verbindung ist mehr als ausreichend für die Anwendung und wird auch vom Verständnis JDOs nicht ablenken, oder nur entsprechend spezialisierte Java EE Programmierer ansprechen.

Die Wahl der IDE fiel auf Eclipse, weil dieses Programm unter Java-Entwicklern eine weite Verbreitung gefunden hat und bei der Erstellung von JDO-Applikationen nützliche Funktionen über optionale Plug-Ins bereithält. FabForces DBDesigner ist eines der wenigen frei erhältlichen Programme zur visuellen Erstellung von Datenmodellen und wurde auf die Belange von MySQL zugeschnitten. Bedauerlicherweise ist die Entwicklung vor einigen Jahren eingestellt worden und das Nachfolgeprodukt MySQL Workbench hat ein Produktivstadium noch nicht erreicht.

¹ Im Bereich der Paketabhängigkeiten befindet sich dort ein Link zum Download des Snapshots. Auf der eigentlichen Projektseite existiert noch keiner.

Sun Microsystems hat nur die Reference Implementation von JDO 1.0 erstellt und für Version 2.0 die Erstellung des JDO JAR, des TCK und der RI der Apache Software Foundation übertragen. Wegen zeitlicher Engpässe wurde dort jedoch die Entwicklung der RI nicht als Apache Projekt durchgeführt¹, sondern an die Entwickler von Java Persistent Objects (www.jpox.org) weitergereicht, deren Arbeit ein Fork des TriActive-JDO Projekts (bis 2003 die vorherrschende JDO-Implementation) ist.²

1.2 Problemstellung

Der De-facto-Standard in der Programmierung von datenbankgestützten Applikationen ist bei Java gerade im Bereich der Ausbildung noch der direkte Weg über JDBC und SQL.

Die damit verbundenen, unterschiedlichen Arten der Datenhaltung – einerseits mengenorientiert bei den Datenbanken und andererseits objektorientiert im Bereich der Applikationen – sind im Laufe der Zeit angewachsen, wobei es den Softwareentwicklern bei jedem Projekt erneut viel Mühe abverlangt, die Kluft dazwischen zu schließen.

Dieses Prozedere ist zeitintensiv und fehleranfällig.

Vereinzelt haben Firmen Lösungen mit unterschiedlichem wirtschaftlichen Erfolg, Umfang und Ansatz entwickelt, doch ein übergreifender Standard als solcher wurde bisher für Java ME/SE/EE noch nicht gebildet.

Insofern ist die Fragestellung, ob ein Einsatz von JDO 2.0 bei Projekten (seien es Neuentwicklungen oder Portierungen auf dieses Framework) sich spürbar in den Bereichen Kosten und Zeit niederschlägt. Gerade was die Schließung des „impedance mismatch“ zwischen relationalen Datenbanken und modernen Programmiersprachen angeht.

Zusammenfassend könnte man die provokante Frage stellen: „Ist nun endgültig die Zeit gekommen, um sich von JDBC zu verabschieden?“

Und natürlich die Zukunft betreffend: „Wie hell wird der Stern JDO in ein paar Jahren strahlen?“

¹ Russell, C. et al.: FrontPage – Jdo Wiki
<http://wiki.apache.org/jdo/>, 10.11.2005

² Java Persistent Objects JDO – JPOX History
<http://www.jpox.org/docs/history.html>, 15.12.2005

2 Historische Entwicklung

2.1 Entwicklung der Datenbanken

2.1.1 Einleitung

In diesem Kapitel wird zuerst angesprochen, in welcher Art von Datenspeichern Daten gesichert werden, gefolgt von den einem Programmierer zur Verfügung stehenden Hilfsmitteln und einer tabellarischen Gegenüberstellung der Vor- und Nachteile dieser Hilfsmittel. Eine vollständige und umfassende Auseinandersetzung würde hier zu umfangreich werden, daher beschränke ich mich auf eine Auswahl der zur Zeit geläufigsten Lösungen und deren markantesten Eigenschaften.

2.1.2 Dateien

Dateien sind die einfachste und älteste Art Daten dauerhaft zu speichern. Der Vorteil bei dieser Speicherung besteht in der Einfachheit der Handhabung bei kleinen Datenmengen mit niedriger Änderungsfrequenz und der direkten Kontrolle über das, was letztendlich gespeichert werden soll. Darunter fallen z.B. Konfigurationsdaten, oder Datenmengen, die aus einem System extrahiert wurden und durch die eigentliche Applikation keiner Änderung mehr unterworfen werden (Logdateien, Reports, Statistiken, etc.).

Leider erschöpft sich dieser Vorteil sehr schnell bei größeren Datenmengen in Bezug auf bspw. Zugriffsgeschwindigkeit, oder bei zusätzlichen Features (wie Datensicherheit, Datenschutz, usw.), welche durch den Programmierer zusätzlich in vielen Programmzeilen implementiert werden müssen.

Natürlich sind Dateien auch heute noch der Grundstock sämtlicher DBMS, nur hat der Benutzer keine direkte Gewalt mehr über die Speicherung der Daten in diese Dateien.

2.1.3 Hierarchisches Datenbankmodell

Dieses Datenbankmodell hat seine Wurzeln in den frühen Datenbankmodellen der 50er und 60er Jahre des 20. Jahrhunderts und wurde in der Zwischenzeit durch andere Datenbanksysteme abgelöst.¹ Das Modell zeichnet sich aus durch die Speicherung der Daten in einer Baumstruktur mit einem Einstiegspunkt (Root) und mehreren

¹ Hierarchisches Datenbankmodell – Wikipedia
http://de.wikipedia.org/wiki/Hierarchische_Datenbank, 25.10.2005

Kindelementen, die wiederum die Rolle eines Elternelements übernehmen können und somit ebenfalls Kindelemente beherbergen.

Ein Vorteil hierin besteht demnach in der schnellen Erreichbarkeit eines Elements vom Wurzelement ausgehend. Leider besteht keinerlei Möglichkeit Bedingungen wie m:n-Beziehungen abzubilden oder innerhalb einer Ebene direkt zu navigieren: Im ungünstigsten Fall muss der komplette Weg über das Wurzelement zurücknavigiert werden, um in einen anderen Zweig zu gelangen.

Dieses hierarchische Modell findet zwar als reine Datenbank keine Anwendung mehr, aber in Bereichen wie der Organisation von Daten im Hauptspeicher oder bei der Erstellung von Indexen über Dateneinträge wird es immer noch eingesetzt.

Mit der wachsenden Beliebtheit von XML erlebt dieses Datenbankmodell einen erneuten Frühling, der sich jedoch nur auf dieses Segment der semantisch behafteten Datenhaltung beschränkt.

2.1.4 Netzwerkdatenbanken

Die Zeit der 70er bis in die frühen 80er Jahre gehörte den Netzwerkdatenbanken¹ bis diese durch relationale Datenbanken und steigende Leistungsfähigkeit der Hardware in den Hintergrund gedrängt wurden. Sie stellten im Prinzip ein „aufgeweichtes“ hierarchisches Datenbankmodell dar: Die Einschränkung, dass ein Datenknoten nur eine Verbindung zu einem Elternelement haben darf, wurde aufgelöst mit dem Ergebnis, dass die ursprüngliche Datenstruktur eines Baumes nun einem Netz gleicht. Auf diese Weise ist die Navigation innerhalb der Datenbank grundlegend vereinfacht, was Abbildungen von m:n-Beziehungen ermöglicht.

Dennoch sind Netzwerkdatenbanken im Auge des Benutzers stärker präsent als die modernen DBMS, da die Datenaufbereitung oftmals in Form von Text und Zahlen (im weitesten Sinne) verbunden mit Querverweisen erfolgt. Beispiele seien hier das Internet oder das „semantische Web“.

¹ Netzwerkdatenbankmodell – Wikipedia
<http://de.wikipedia.org/wiki/Netzwerkdatenbankmodell>, 06.06.2005

2.1.5 Relationale Datenbanken

Relationale Datenbanken basieren auf Edgar F. Codd's Datenmodell, welches er 1970 veröffentlichte¹. Die Datenhaltung erfolgt in Tabellen, deren Datensätze einen eindeutigen Wert zur Identifikation (Primärschlüssel) haben. Die Datensätze sind hierbei die Zeilen und die Spalten sind Attribute – eine Zelle dieser Tabelle ist folglich die Ausprägung eines Attributes eines Datensatzes. Verbindungen zwischen den jeweiligen Tabellen werden über sog. Fremdschlüssel vollzogen. Fremdschlüssel sind Primärschlüssel anderer Tabellen, die in den aktuellen Datensatz importiert werden. Durch die Haltung in Tabellen und eine reglementierte, schrittweise Vorgehensweise bei der Erstellung eines relationalen Datenmodells (Normalisierung) wird ein mengenorientiertes Gebilde erschaffen ohne Redundanzen unter den gespeicherten Daten.

Das relationale Datenmodell hat sich bis in die Gegenwart gehalten: Die dazugehörigen DBMS bieten einen standardisierten Befehlssatz namens SQL und wurden ständig fortentwickelt: Einerseits in die Richtung zusätzlicher Features und höherer Performance und andererseits in die später „entstandenen“ objektrelationalen Datenbanken.

Leider hat das relationale Datenmodell seine zeitlichen Wurzeln unter den prozeduralen Programmiersprachen und ist nur mangelhaft an die Bedürfnisse der aktuellen Programmiersprachengeneration angepasst worden (denn das würde oftmals eine Verletzung der Normalisierung bedeuten). Selbst heute stellt z.B. die Speicherung von Interfaces in einem RDBMS ohne Verletzung der Regeln der Normalisierung die Entwickler vor große Probleme.

Geschäftsdaten können mit Hilfe der RDBMS normalerweise problemlos gespeichert werden, aber die Lücke zur Objektorientierung wurde stetig größer.

2.1.6 Objektrelationale Datenbanken

Aus dem relationalen Datenmodell erwuchs das objektrelationale Datenmodell als eine Art Erweiterung. ORDBMS verwenden den Befehlssatz SQL3, der abwärtskompatibel zu dem der RDBMS namens SQL2 / SQL-92 ist, und bieten eine Möglichkeit diese Lücke zwischen

¹ Abts, D.; Müller, W.: Grundkurs Wirtschaftsinformatik, Braunschweig, Wiesbaden 2001, S. 166

Programmierung und Datenspeicherung ein Stück zu schließen. Die Neuerungen wären hier z.B. die Erstellung eigener Datentypen, oder die Speicherung von Methoden in der Datenbank. Doch auch dieser Ansatz schließt die Lücke nicht ganz: Zwar wird die generelle Unterstützung der objektorientierten Paradigmen besser, aber man ist immer noch an Tabellen gebunden.

Ein oft gebrauchtes Beispiel für von ORDBMS profitierenden Anwendungen sind Geoinformationssysteme bei denen Koordinatenobjekte mit anderen Daten wie Straßennamen in Verbindung gesetzt werden – wohl nicht zuletzt weil die bekannteste objektrelationale Datenbank im Open-Source Bereich „PostgreSQL“ geometrische Datentypen direkt unterstützt.

2.1.7 Objektdatenbanken

Die völlige Schließung der Lücke bilden hingegen Objektdatenbanken (oftmals auch als objektorientierte DBMS bezeichnet, obwohl das DBMS nicht zwangsweise objektorientiert sein muss). Die Datenstruktur im Hauptspeicher wird unverändert übernommen und im Datenspeicher abgelegt. Dieser völlig neue Ansatz verwirft auch das Tabellenmodell und die damit verbundene Abfragesprache SQL. Ursprünglich waren Objektdatenbanken reine Datenspeicher für komplexe Objektmodelle, die lediglich schnell gesichert und geladen werden mussten, während die eigentliche Bearbeitung in der Applikation vollzogen wurde (bspw. ein CAD-Programm). Im Laufe der Zeit sind noch Abfragemöglichkeiten hinzugekommen, die direkt in der Datenbank durchgeführt werden können – einerseits per Navigation und andererseits per Suchanfrage. Leider sind hier im Anfangsstadium Möglichkeiten verschenkt worden, da jeder Hersteller seine eigene Abfragesprache entwarf, die zu anderen Herstellern nicht kompatibel war. Eine ins Leben gerufene Vereinigung zur Standardisierung von Objektdatenbanken auf Benutzerseite, die Object Data Management Group (www.odmg.org) schuf daraufhin das Pendant zu SQL, die „Object Query Language“ (kurz OQL) nebst entsprechendem Interface. Die ODMG wurde im Jahr 2001 aufgelöst und die Arbeit im Bereich Javaanbindung von Objektdatenbanken der Expertengruppe von JDO 1.0 übergeben.¹

¹ Object Data Management Group: ODMG Home Page
<http://www.odmg.org/>, 15.12.2005

2.1.8 Sonderfall O/R-Mapping

Der Sonderfall O/R-Mapping stellt keine Lösung auf Seiten der Datenbanken oder der Datenmodelle dar, sondern ist eine Middleware-Lösung des Problems.

Die Daten der Applikation in Form von Objekten werden einem Persistenzlayer übergeben und dort für die Speicherung in einer relationalen Datenbank transformiert und letztendlich dem RDBMS übergeben.

Auf diese Weise kann das Know-how für die Speicherung von Objekten gebündelt und Fertiglösungen angeboten werden. Diese Layer haben alle gemeinsam, dass ihnen per Konfigurationsdateien die Struktur des Objektmodells und ggf. des zugrundeliegenden Datenmodells bekannt gegeben werden muss.

Der Verlust an Performance wird hierbei hauptsächlich durch mehr Flexibilität und einen Zeitgewinn in der Entwicklung ausgeglichen.

2.2 Entwicklung der APIs

2.2.1 Einleitung

Nun stellt sich die Frage, auf welche Art und Weise die Daten an die Zielorte gebracht werden können. Die folgende Aufstellung stellt einen Auszug gebräuchlicher Verfahren dar:

2.2.2 Klartext / Binärmodus

In Verbindung mit der Speicherung in eine Datei wird oftmals die Speicherung im Klartext oder Binärmodus verwendet. Die Operationen um dies zu bewerkstelligen beschränken sich auf die direkten Dateioperationen (Datei erstellen, öffnen, schließen, löschen) und auf Operationen zum Lesen und Schreiben in den offenen Dateistream. Auf diese Weise können auch nachträglich Werte per (Hex-)Editor verändert werden.

2.2.3 Serialisierung

Eine Technik zur Speicherung oder Übertragung von Objekten bildet die Serialisierung, in der ein Objekt samt seinen referenzierten Objektbäumen in eine serielle Datenfolge ausgeschrieben wird. Die Voraussetzung, um diese Technik verwenden zu können ist das Implementieren des Interfaces „Serializable“ in das entsprechende Objekt.

Generell ist es möglich auf diese Weise auch Objekte in BLOB-Feldern relationaler Datenbanken zu speichern und damit kleineren Problemen in der Programmierung aus dem Weg zu gehen.

Die Nachteile sind jedoch vielfältig: Nach einer Serialisierung muss ein Objekt erst deserialisiert werden, bevor man es erneut verwenden oder durchsuchen kann. Auch ist eine dauerhafte Speicherung in BLOB-Feldern eines RDBMS nicht ratsam, weil oftmals nur der Zustand eines Objekts gesichert werden muss. Auf diese Weise wird auf jeden Fall Speicherplatz verschwendet, indem die Objektbäume des zu speichernden Objektes ebenfalls gesichert werden (ggf. entstehen Redundanzen).

2.2.4 JDBC

Der aktuelle Standard in der Programmierung von RDBMS ist JDBC.

Es wird über einen Connector eine Verbindung zur Datenbank aufgebaut und die Daten werden unter Zuhilfenahme der Datenbanksprache direkt manipuliert, bzw. aus der Anwendung übertragen. Bei dieser

Vorgehensweise muss der Programmierer das Objektmodell kennen, das zugrundeliegende Datenmodell und die für ihn wichtigen Zusammenhänge zwischen den Daten. Weiterhin die Programmiersprache der Anwendung und auch die Sprache der Datenbank, wobei er in letzterer Optimierungen vornehmen können sollte. Ebenfalls ist der Programmierer verantwortlich für sämtliche Belange der Konsistenz der Daten, oder eventueller Redundanzen, hat dafür aber sämtliche Befehle des Datenbank direkt zur Verfügung. Er ist also Programmierer und Administrator der jeweiligen Datenbank in einer Person.

Diese totale Kontrolle wird teuer erkaufte: Nicht umsonst haben mittlerweile alle großen RDBMS Werkzeuge zur grafischen Abfrageerstellung und in umfassenden Abfragen können sich schnell Fehler einschleichen, welche schwer zu entdecken sind.

2.2.5 EJB

Aus dem Bereich der Java Enterprise Edition stammt die Idee der Enterprise Java Beans.

„JavaBeans Komponenten, kurz Beans, sind wiederverwendbare Softwarekomponenten, die visuell mit einem Builder-Tool manipuliert

werden können.“¹ Der Unterschied von Enterprise Java Beans ist, dass EJBs die Geschäftslogik auf einem Applikationsserver darstellen und Bereiche wie Datenhaltung oder die Darstellung über andere Programmteile abgewickelt werden. In dem Zusammenhang ist es noch wichtig, den Container zu erwähnen: Der Container ist Teil des Applikationsservers und er stellt den Rahmen (den „Behälter“) für die EJBs dar. Nun wird, was die Datenhaltung angeht, zwischen CMP und BMP unterschieden.

Die „Bean Managed Persistence“ (BMP) benötigt zusätzliche Klassen im Container, Konfigurationsdaten und Methoden für die Umsetzung der Speicherung, die in der jeweiligen Bean selbst vorhanden sein müssen. Die Bean ist für die Speicherung ihrer Daten also selber zuständig. Bei der „Container Managed Persistence“ (CMP) werden diese Aufgaben aus der Bean in den Container verlagert und durch diesen zentral verwaltet.

Für sich genommen bieten EJBs keinerlei Persistenzlösung an, sondern lediglich ein sehr umfangreiches Rahmengerüst, in das der Anwendungsentwickler seinen Code zur Persistenz der Beans einbettet.

2.2.6 Hibernate

Ungefähr zur gleichen Zeit wie die finale Spezifikation zu JDO kam Hibernate Mitte 2002 als Open-Source Persistenz-Framework auf den Markt¹, hat sich mittlerweile in die Version 3 fortentwickelt und einen großen Bekanntheitsgrad erreicht.

Hibernate verfolgt den Ansatz des objektrelationalen Mappings und verwendet eine Mischung aus:

- Reflection während der Laufzeit (automatisches Auslesen der Getter- und Settermethoden einer Klasse)
- der Verwendung von XML-Dateien für das Mapping (pro Klasse eine eigene Datei)
- einer an SQL angelehnten Abfragesprache namens HQL (Hibernate Query Language)

¹ Sun Microsystems, Inc.: JavaBeans FAQ: General Questions
http://java.sun.com/products/javabeans/faq/faq_general.html, 15.12.2005

¹ King, G.: Announcing Hibernate 1.0 Open Source O/R Persistence Tool
http://www.theserverside.com/news/thread.tss?thread_id=14314, 06.07.2002

- einer SessionFactory und zugehörigen Sessions in denen die Transaktionen ablaufen

Dazu kommen noch zusätzliche Features wie ein Cache, der in der Applikation einer Datenbank vorgeschaltet wird, transparente Persistenz (Objekte werden erst in den Speicher geladen, wenn diese auch vom Programm verwendet werden), oder einer breiten Unterstützung von relationalen Datenbanken.

Letztere Unterstützung war auch der Anstoß für die Entwicklung Hibernates: Daten aus Objekten konnten nicht bequem und sicher in den weit verbreiteten relationalen Datenbanken untergebracht werden, weshalb diese Datenbanken auch die einzigen Datenspeicher sind, die Hibernate unterstützt.

2.2.7 JDO

Java Data Objects ist ein Persistenz-Framework, das als Spezifikation das Licht der Welt erblickte. Dies bedeutet, dass JDO nicht als „ein Produkt“ erhältlich ist, sondern als ein Standard durch eine Expertengruppe unter der Schirmherrschaft Sun Microsystems entworfen wurde. Dieser Standard ist für jedermann frei einsehbar und als Produkt umsetzbar und deshalb gibt es sowohl mehrere kommerzielle, als auch Open-Source JDO

Implementationen. Als einzige Restriktion ist es nicht erlaubt, ein Persistenzframework zu programmieren, welches lose an die Spezifikation angelehnt ist, und dieses dann als JDO-Implementation zu vertreiben.

Die o.g. Expertengruppe setzt sich aus verschiedenen Industrievertretern zusammen, wobei sich zwischen JDO 1.0 und JDO 2.0 ihre

Zusammensetzung grundlegend geändert hat: War bei JDO 1.0 noch der Fokus auf Vertretern der objektorientierten Seite (Objektdatenbanken, Programmiersprachen, etc.), hat sich dieser mit Version 2.0 auf die Belange der relationalen Datenbanken verschoben (RDBMS, Middleware, etc.).

Dieses späte Einlenken wurde JDO lange Zeit mit der Begründung am Markt vorbei entwickelt worden zu sein vorgeworfen und am Ende stand die Fertigstellung von JDO 2.0 auf der Kippe, als die Stimmen immer lauter wurden, JDO mit EJB zusammenzulegen.

Zu guter letzt ist JDO 2.0 nun doch noch als selbstständiger Standard veröffentlicht worden und die Kernfeatures sind:

- komplette Unabhängigkeit von dem unterliegenden Datenspeicher, sofern dieser von der Umsetzung unterstützt wird („Write once, store anywhere.“)
- 100% Kompatibilität zwischen den Umsetzungen verschiedener Hersteller, sofern keine herstellerspezifischen Erweiterungen verwendet werden
- der Bytecode-Enhancer, der nach dem Übersetzen der persistenten Klassen spezielle Getter- und Setter-Methoden zwischen den eigentlichen Methoden einfügt
Der zusätzliche Aufwand, der bei Hibernate durch Reflection immer während der Laufzeit verursacht wird, wird also nach der Kompilierung fest in der Klasse verankert
- die Verwendung von XML-Dateien für das Mapping (eine zentrale Datei oder Aufsplittung in mehrere kleinere nach Paketzugehörigkeit aufgeteilte Dateien und spezielle Mapping-Dateien)
- eine an Java angelehnten Abfragesprache namens JDOQL (alternativ kann auch SQL verwendet werden)
- eine PersistenceManagerFactory und zugehörige PersistenceManager in denen die Transaktionen ablaufen

Dazu kommen noch zusätzliche Features wie Cachefähigkeiten, transparente Persistenz (Objekte werden erst in den Speicher geladen, wenn diese auch vom Programm verwendet werden), in XML-Dateien ausgegliederte Queries (wenn sich eine Abfrage ändert, muss nur diese Datei angepasst werden, ohne den Quellcode zu berühren), oder Erweiterbarkeit des Frameworks um herstellerspezifische Metadaten.

2.3 Zusammenfassung

Abschließend ist zusammenzufassen, dass Ansätze wie die Speicherung als Textdatei, oder die Serialisierung nur für sehr spezielle Fälle der dauerhaften Datenspeicherung geeignet sind. JDBC bietet da eine bessere Lösung in Verbindung mit einem Datenspeicher und bietet völlige Kontrolle über diesen, aber dies auf Kosten einer einfachen Programmerstellung. Erst Persistenzframeworks stellen die benötigte Verbindung aus bequemer Entwicklung, Wartbarkeit und Unabhängigkeit vom Datenspeicher dar.

	Text / binär	Serialisierung	JDBC	EJB 2.0 (CMP)	Hibernate	JDO 2.0
Komplexität der API	+	+	-	-	0	0
Geschwindigkeit	-	-	+/0 ¹	+	+	+
Portabilität	-	-	-	0	0	+
Ausnutzung von Datenspeicher- funktionen	/	/	+	/	+	+
zeitlicher Aufwand	-	0	-	0	+	+
Komfort	-	0	-	0	+	+
Fehleranfälligkeit	-	0	-	0	+	+
Migrationsaufwand	/	/	-	0	+	+
Anpassungsaufwand bei Änderung des Datenmodells	-	-	-	0	+	+
Anpassungsaufwand bei Änderung des Datenspeichers	/	-	-	0	+	+

Tabelle 2.1: Matrix zur Verwendung der Softwarelösungen

¹ Zweiter Wert wegen Abwertung aufgrund fehlendem Cache und hohen Antwortzeiten bei Vollast

3 Java Data Objects 2.0

3.1 Entwicklungsprozess

Der Entwicklungsprozess einer JDO-Anwendung besteht im Grunde genommen immer aus einem der drei Schritte¹

- Forward Engineering
- Reverse Engineering
- Bridge Mapping

Ebenfalls ist eine Kombination der Schritte möglich, je nachdem ob bereits ein Datenmodell besteht, oder welchen Anforderungen das Datenmodell, bzw. das Objektmodell gerecht werden soll.

In dem verwendeten Beispiel der Spieledatenbank gab es bereits ein Datenmodell, das vor der Applikation Anwendung im Bereich eher prozedural programmierter Webanwendungen fand. Insofern war das Datenmodell nach den Grundsätzen der Normalisierung entworfen.

Forward Engineering jedoch ist der entgegengesetzte Ansatz zur Erstellung einer Applikation. Zuerst wird ein Objektmodell entworfen und dieses dann in einer relationalen Datenbank abgebildet, wobei das Ergebnis meist nicht den Grundsätzen relationaler Datenmodelle genügt. Die zugrunde liegende objektorientierte Programmierung schlägt sich auf das Datenmodell durch und so können z.B. plötzlich Tabellen angelegt werden, die mehrere Objekte beinhalten. Es können auch „überflüssige“ Hilfskonstrukte auftauchen, die zwar aus Sicht des Objektmodells „logisch“ erscheinen möchten, aber aus Sicht eines Datenbankadministrators keinen Sinn ergeben: m:n-Beziehungen werden in zwei 1:n Beziehungen mit jeweils einer Tabelle aufgelöst, oder Tabellen mit zusammengesetzten Primärschlüsseln erhalten einen losgelösten Primärschlüssel als Objekt-ID.

Reverse Engineering hat ein Datenmodell als Grundstock und erzeugt davon ausgehend das Objektmodell. Dieses Vorgehen ist das schnellste, um bei einer Neuentwicklung der Applikation einen bestehenden Datenstock zu migrieren.

¹ Jordan, D.; Russell, C.: Java Data Objects, Sebastopol 2003, S. 71f.

Leider ist das entstehende Datenmodell aus der Welt der relationalen Datenmodelle entsprungen: Zwar sind Beziehungen zwischen den Tabellen mit Hilfe von Schlüsseln in dem DBMS verankert (sofern der Tabellentyp dies unterstützt), aber auf Seiten der objektorientierten Programmiersprache ist zusätzlich wichtig, welche Art von Sammlung diese Relation darstellt: Es können in Java an der Stelle bspw. eine Collection, ein Set, eine List oder eine Map stehen. Ebenso ist in der Datenbank nicht gespeichert, von welcher Seite die Relation hauptsächlich navigiert wird (oder ob beide Objekte zwingend voneinander wissen müssen). Gerade in Webapplikationen wird oftmals nur in eine Richtung navigiert und falls der Benutzer die entgegengesetzte Richtung beschreiten möchte, wird er dies per erneuter Anfrage verdeutlichen.

Diese Ungewissheit wird durch das Risiko einer zweiten Abfrage in Kauf genommen und so die Rechenleistung um die Population von Sammlungen und das Laden und Übertragen von nicht genutzten Daten vermindert.

Bridge Mapping geht davon aus, dass sowohl ein Daten-, als auch ein Objektmodell besteht, welche durch O/R-Mapping Strategien miteinander verbunden werden müssen.

Dieser Ansatz wird zwar beiden Welten gerecht, aber durch die getrennte Entwicklung können sich hier andere Probleme einschleichen. Es wäre möglich, dass Objekte definiert werden, die keine Tabelle als Gegenstück in der Datenbank haben, oder andersrum. Dennoch ist die Wichtigkeit dieses Schrittes als „letzter Schliff“ für ein reibungsloses Zusammenspiel nicht zu unterschätzen, sofern man keine Abstriche beim Objektmodell oder beim Datenmodell machen möchte – gerade im Bezug auf bestehende Anwendungen oder einem komfortablen Objektmodell.

In dem verwendeten Beispiel benutzte ich zuerst die Technik des Reverse-Engineering, um ein erstes, rohes Objektmodell zu erhalten. Dieses wurde an die Belange der Applikation angepasst und das Ergebnis per Bridge-Mapping in die bestehende Datenbank abgebildet.

Reines Reverse- und Forward-Engineering sind recht einfache Vorgehensweisen, die sich schnell automatisieren lassen und rasch Ergebnisse liefern. Der zeitaufwändige Prozess an dieser Stelle ist das

Bridge-Mapping, da kein Modell erstellt oder angepasst wird, sondern eine Punktlandung auf dem Datenmodell stattfinden muss. In dem Schritt offenbaren sich spätestens typische objektorientierte Muster, die sich nur schwer auf relationale Datenmodelle abbilden lassen wie z.B. Interfaces (Kapitel 4.2.1), Implementationen von Sammlungen (Kapitel 4.4.12.1), oder Abweichungen von der Verwaltung der Primärschlüssel (Kapitel 4.4.1).

3.2 Anwendungsbeispiele / Architekturen

3.2.1 Architekturen mit Fokus auf der JVM

3.2.1.1 Eine JVM und ein bis viele PersistenceManager

Diese erste vorgestellte Version einer Architektur ist auch diejenige, welche am ehesten implementiert werden kann: Innerhalb einer Virtual Machine befindet sich ein PersistenceManager mit seinem Cache und eine Applikation. Wichtig zu erwähnen ist hier, dass der Programmierer von der Existenz des 1st Level Cache nicht betroffen ist – er verwendet die Objekte, als ob sie in seiner Anwendung wären. Der PersistenceManager befindet sich über JDO in Kontakt mit einem Datenspeicher (siehe Abbildung 3.1, oberes Drittel).

Natürlich ist es auch möglich, innerhalb einer Virtual Machine für mehrere Applikationen (Öffnung eines neuen Fensters zum Beispiel) ihren jeweils eigenen PersistenceManager zu definieren. In diesem Falle wird für jeden weiteren sein eigener 1st Level Cache erstellt, um auch eine saubere Trennung für die Durchführung von Transaktionen zu erzielen (Abbildung 3.2, mittleres Drittel).

Auf die Spitze kann dies noch getrieben werden, wenn innerhalb einer VM verschiedene Implementationen von JDO verwendet werden, die auf unterschiedliche Datenspeicher zugreifen können. Die Übereinkunft zur binären Kompatibilität gewährt, dass die gleichen Klassen von JDO Implementationen verschiedener Hersteller genutzt werden können, ohne erst neu kompiliert werden zu müssen (Abbildung 3.2, unteres Drittel).

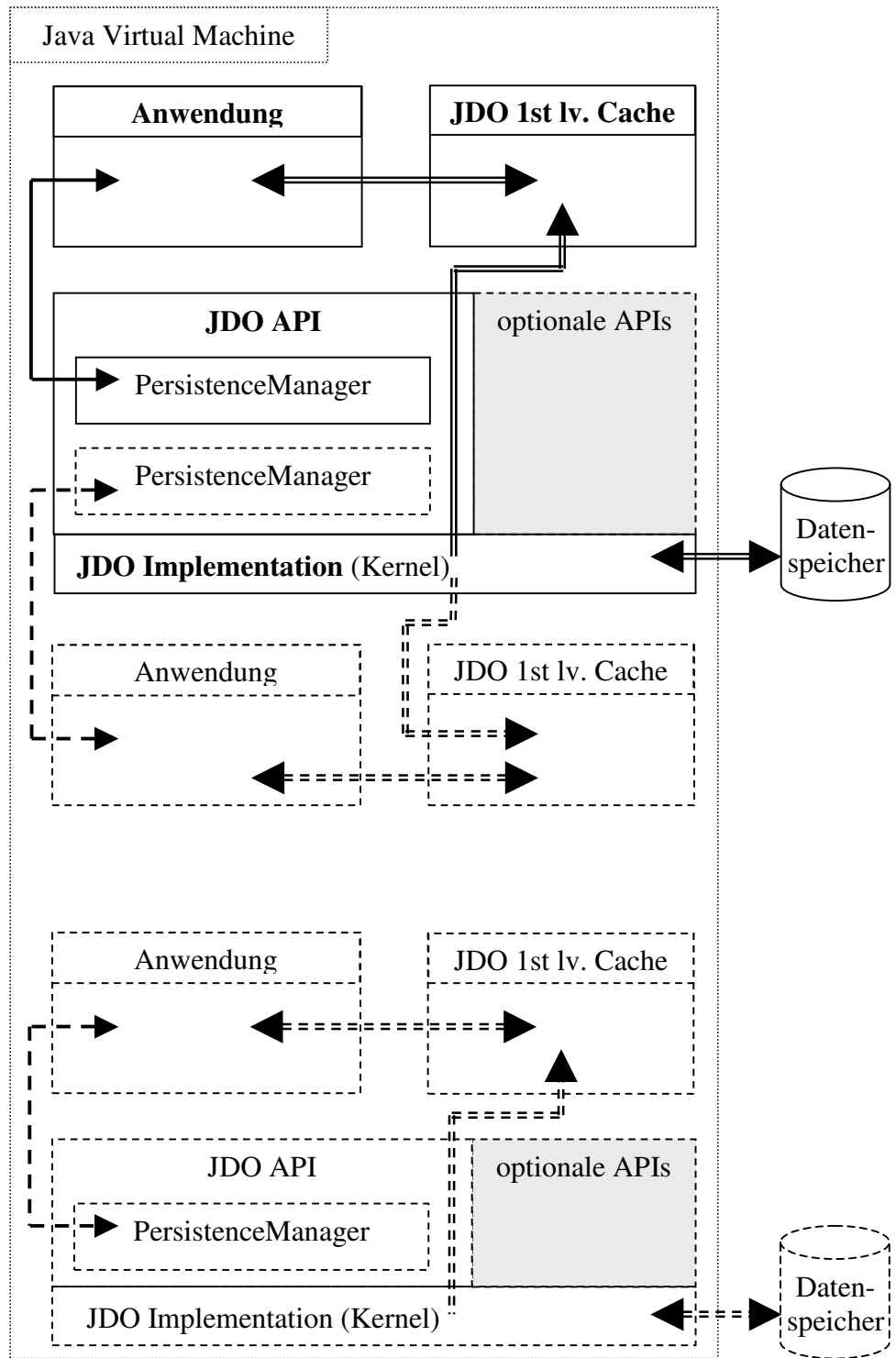


Bild 3.1: Eine JVM und ein bis viele PersistenceManager

3.2.1.2 2nd Level Cache

Neben dem eigentlichen Cache, der dem jeweiligen PersistenceManager untersteht, kann der Hersteller sich noch dazu entscheiden einen 2nd-Level Cache bereitzustellen (oder einer entsprechenden Schnittstelle für Dritthersteller), welcher allen PersistenceManagern einer Implementation vorgeschaltet ist. Ein Performancevorteil kann sich jedoch hier nur ergeben, wenn der 2nd-Level Cache größer ausfällt, als der 1st-Level Cache (da sonst deren Inhalte identisch wären), oder mehrere PersistenceManager existieren, welche auf diesen zugreifen können.

Außerdem können die Instanzen der 2nd-Level Caches über mehrere JVM hinweg koordiniert auf einen gemeinsamen Datenspeicher zugreifen. In diesem Falle schlagen sich die Änderungen eines Cache auf die anderen durch, ohne dass das Objekt dort neu geladen werden muss (siehe Abbildung 3.2).

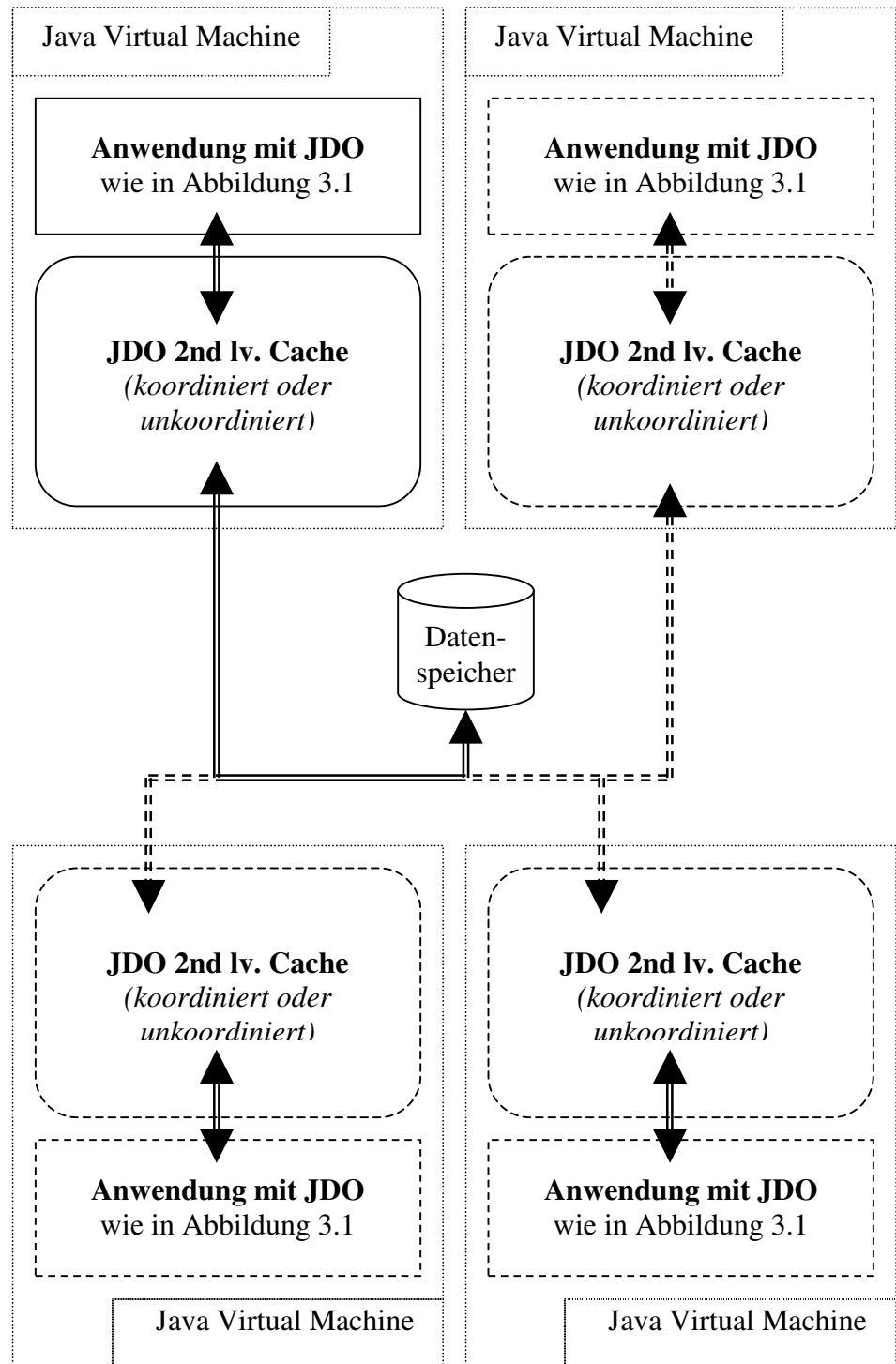


Bild 3.2: Mögliche Architektur mit 2nd Level Caches

3.2.2 Datenspeicherzugriff

3.2.2.1 Direkter Zugriff auf das Dateisystem

Der direkte Zugriff auf das Dateisystem bringt gewisse Einschränkungen mit sich. Aus diesem Grund wird der direkte I/O-Zugriff über die entsprechenden Java-Bibliotheken nur für einfache Datenspeicher wie binäre Dateien, XML-Dateien oder ähnliche Datenspeicher verwendet. Da die Hersteller eine solche Unterstützung jeweils neu entwickeln müssen, sind einfache Dateien dort auch mit einer niedrigen Priorität versehen; die Relevanz von RDBMS ist einfach höher.

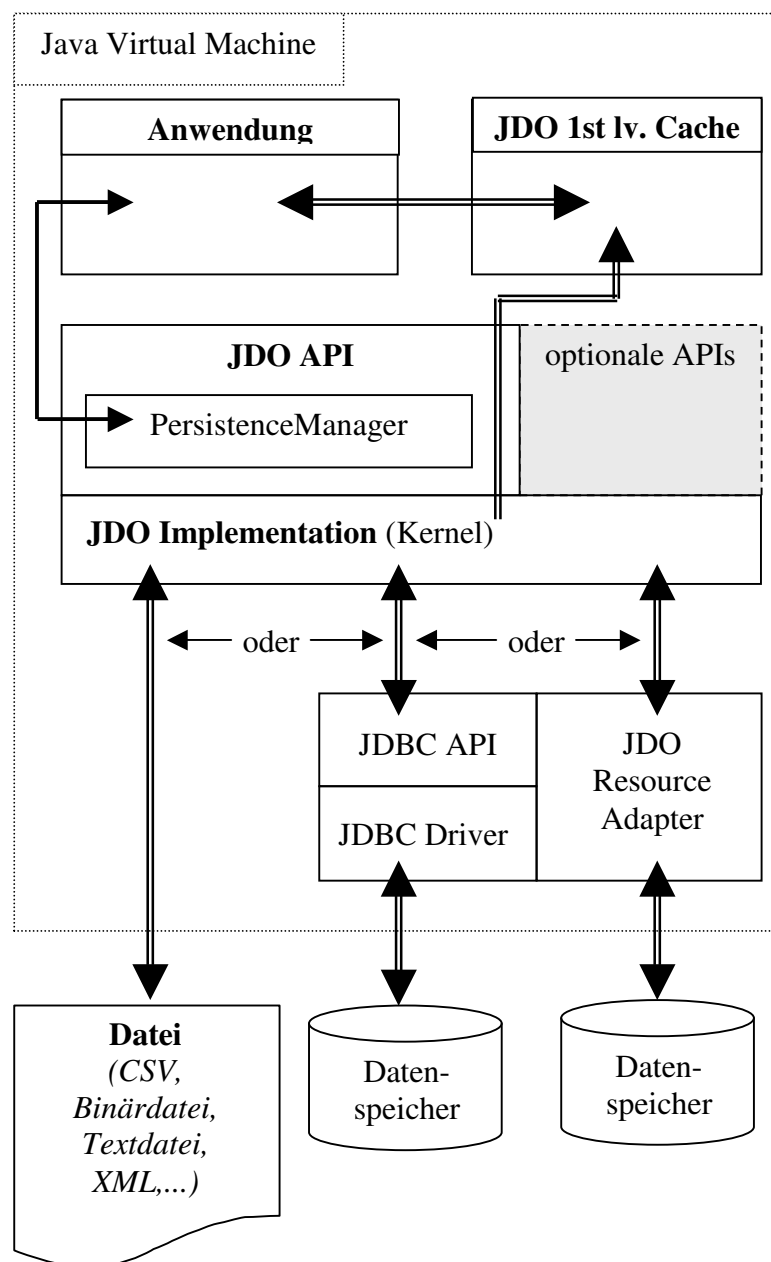


Bild 3.3: Mögliche Anbindungsarten eines Datenspeichers

3.2.2.2 Zugriff über JDBC

Der nächste Schritt, die Funktionen einer JDO-Implementation zu erweitern, wäre als Basis JDBC zu verwenden. Auf diese Weise wird wieder ein standardisierter Weg beschritten, um Daten in ein RDBMS unterzubringen und diese Schnittstelle bietet sich dafür an. JDBC ist ausgereift und bietet eine große Auswahl an Datenbanken an, die einen Treiber für JDBC bereitstellen.

In diesem Fall kommuniziert JDO nur indirekt mit dem Datenspeicher und reicht Befehle an die unterliegende JDBC-Schicht weiter (Abbildung 3.3, Mitte).

3.2.2.3 Zugriff über einen Resource Adapter

Besonders im Bereich der Java Enterprise Edition sind nicht nur relationale Datenbanken anzutreffen, sondern z.B. auch anders gestaltete EIS, die auch einen andersartigen Zugriffsweg als JDBC haben können.

Im Rahmen der Java EE Connector Übereinkunft liefern solche Hersteller dann einen sog. „Resource Adapter“, der sich ähnlich wie JDBC unterhalb JDOs einbindet und in einem proprietären Protokoll mit dem Datenspeicher kommuniziert (Abbildung 3.3, rechts).

Zwar bekommt der Programmierer davon bis auf die Auswahl des Adapters nichts mit – er kann aber diesen Adapter über den PersistenceManager weiter konfigurieren.

3.2.3 Systemarchitekturen mit JDO Applikationen

3.2.3.1 Einzelplatzrechner mit lokalem Datenspeicher

Auf den ersten Blick scheint eine Einzelplatzlösung für diese Technik nicht die geeignete Umgebung zu sein. Seine wahre Stärke zeigt JDO im Zusammenhang mit einer zentralen Datenhaltung und vielen Datenzugriffen, aber auch hier sind Anwendungsmöglichkeiten angesiedelt: Zwar werden die wenigsten Nutzer ein DBMS auf einem Rich Client ansiedeln wollen, aber kleinere, auf Geschwindigkeit ausgelegte DBMS könnten hier punkten.

Auch denkbar wäre es, ausführliche Konfigurationsdateien bequem per JDO anzusteuern und dabei den Cache abzuschalten, um Arbeitsspeicher zu schonen. Andererseits könnte JDO auch mit Java ME und Dateizugriff

kombiniert werden, um auch kleinste Applikationen mit eigener Datenhaltung auf mobilen Endgeräten zu erstellen.

3.2.3.2 JDO Applikation in einem Webserver / als Webservice

JDO als Persistenz-Lösung innerhalb einer Webanwendung ist eine der Haupteinsatzmöglichkeiten des Frameworks. Die Einbettung in JSP-Seiten oder Servlets geht ohne Umwege vonstatten und die Darstellung und die Datenhaltung werden auf verschiedene Rechner verteilt. Interessant ist, dass auf dem Server JDO im Gegensatz zu JDBC für eine bessere Ausnutzung der Ressourcen sorgt (siehe Kapitel 3.3.2: „Unterschiede zu JDBC in der Performance“).

Gleiches gilt für Webservices, nur wird dort der Zugriff auf die Logikschicht über ein anderes Protokoll abgewickelt (Abbildung 3.4).

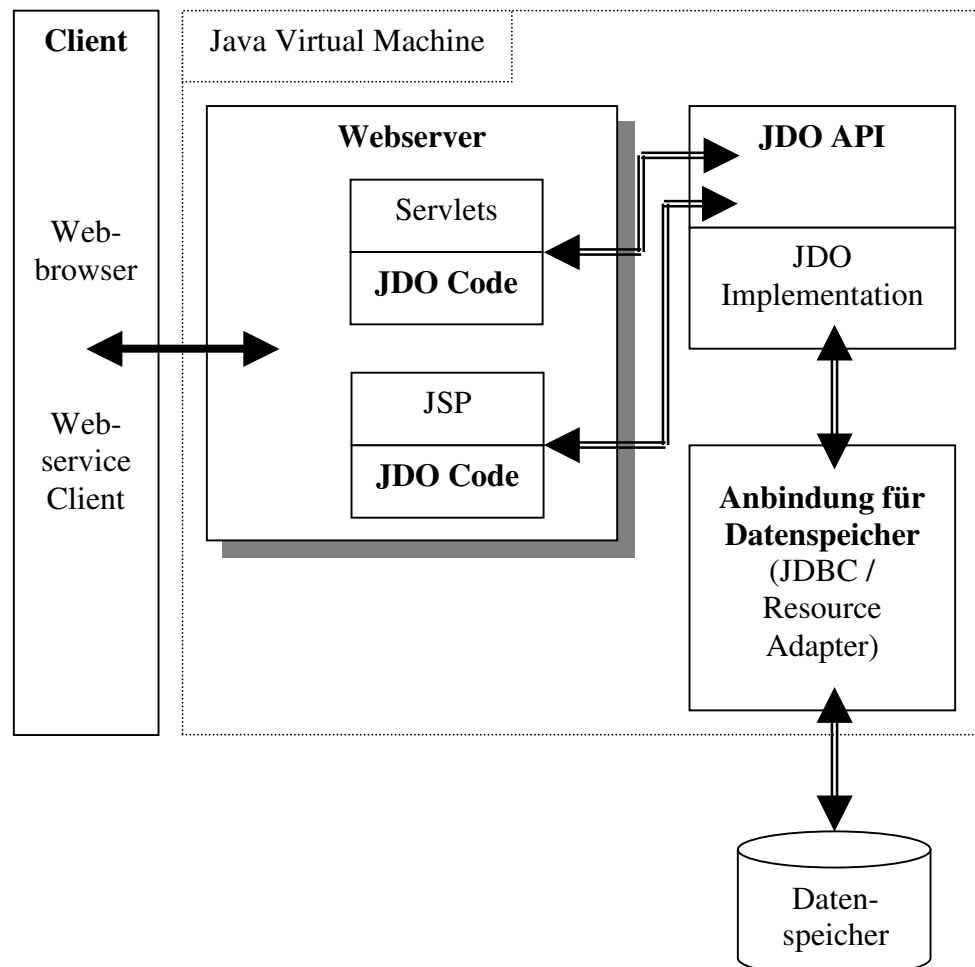


Bild 3.4: Einsatzmöglichkeit mit einem Webserver

3.2.3.3 Verschiedene Umsetzungen für Applikationsserver

Im Feld der Applikationsserver kann JDO vielfältig eingesetzt werden: Die Beans werden entweder direkt über einen passenden Client angesprochen, oder über einen Webserver als Mittler.

Enterprise Java Beans bieten bereits vordefinierte Mechanismen für die Persistenz der Beans an: Bei der Bean Managed Persistence kann JDO durch die Bean direkt verwendet werden und so selber die Speicherung ihrer Daten umsetzen (Abbildung 3.5, obere drei Beans).

Bei der Container Managed Persistence hingegen übernimmt der Container die Speicherung der Daten und versteckt die Speicherungslogik vor den Beans. Wie diese Logik letztendlich auszusehen hat, ist nicht fest definiert - aus diesem Grunde ist es auch nur verständlich, dass einige Anbieter von Java EE Application Servern, deren Firmen-Know-how in anderen Bereichen liegt, sich für die Umsetzung von CMP eines Persistenzframeworks wie bspw. JDO oder Hibernate bedienen.

Einen Sonderfall bildet innerhalb der EJB-Thematik das Entwurfsmuster der Fassade: Eine Session Bean kann mit Hilfe von JDO zu einer Fassade ausgebaut werden und die Steuerung der Persistenz anderer Beans übernehmen (Abbildung 3.5, untere zwei Beans).

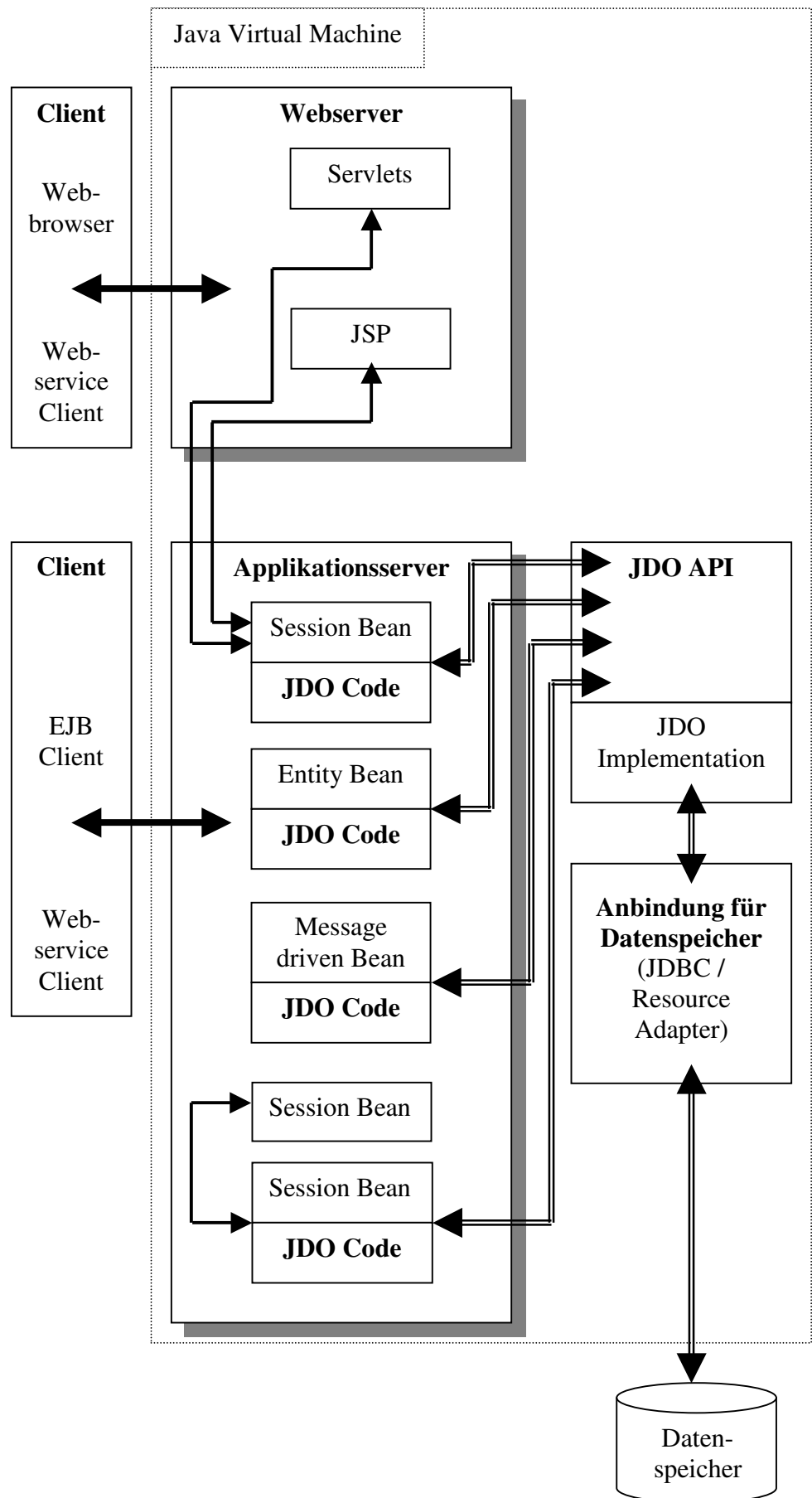


Bild 3.5: Einsatzmöglichkeiten für Applikationsserver

3.3 Unterschiede zu JDBC

3.3.1 In der Syntax

Schon aus dem Grund, dass ein Framework die Persistenz für den Programmierer übernimmt, schrumpft für diesen bereits die Anzahl der geschriebenen Zeilen innerhalb eines Projekts.

In einem Fall hatten Douglas Barry und Torsten Stanienda ein typisches Übungsbeispiel mit dem Vorgänger von JDO 1.0, der ODMG Java Binding, programmiert und kamen mit ca. 25% des JDBC-Codes aus (reines Forward Engineering ohne irgendwelche Anpassungen).¹

Folgend ist ein Beispiel inklusive Overhead für die Applikation selber. Die eigentlichen drei Zeilen für die Abwicklung einer Transaktion (in diesem Falle dem Speichern eines Objektes mitsamt seiner referenzierten Objekte) stehen zwischen dem Begin und Commit der Transaktion tx. Bei einer Umsetzung per JDBC müsste für jede Referenz ein eigener Insert-Befehl geschrieben und durch den Programmierer die Reihenfolge der SQL-Statements berücksichtigt werden.

```
// Import der Schnittstelle
import javax.jdo.*;

// Variablen für die Anwendung
private PersistenceManagerFactory pmf;
private PersistenceManager pm;
private Transaction tx = null;

// Vorbereitung für die Anwendung
pmf =
JDOHelper.getPersistenceManagerFactory(props);
pm = pmf.getPersistenceManager();
tx = pm.currentTransaction();

try {
    // Objekt(e) speichern
    tx.begin();
    pm.makePersistent(meinObjektInklusiveReferenzbaum);
    tx.commit();
} catch {JDOException e} {
    // Abwicklung der Ausnahme
} finally {
    if (tx.isActive())
        tx.rollback();
}
```

¹ Barry, D.; Stanienda, T.: Solving the Java Object Storage Problem. In: Computer, 31. Jg., H. 11, 1998, S. 33 – 40

3.3.2 In der Performance

Das Thema Performance und Benchmarking im Zusammenhang mit Datenbanken ist oftmals begleitet von Kritik an diesen Benchmarks, da diese gerne einzelne Operationen mit gestufter Häufigkeit durchführen und dadurch keinerlei Bezug zu einer produktiven Umgebung haben in der sich CRUD-Operationen mit unterschiedlicher Häufigkeit abwechseln. Verallgemeinernd lässt sich im voraus jedenfalls logisch herleiten, dass einzelne Befehle über JDBC schneller abgewickelt werden, jedoch weitere Fähigkeiten wie das Cachen von Daten (und damit Objekten) oder das Laden von Objekten bei Zugriff durch den Programmierer implementiert werden müssen. Persistence Frameworks wie Hibernate oder JDO haben diese „Sprintkraft“ nicht – schon alleine deshalb nicht, weil sie oft JDBC als Basis für Datenbankzugriffe verwenden. Dafür haben sie einen längeren Atem bei Daten, die sich bereits im Cache befinden, und bei komplexen Objektstrukturen. Es kann also ein gleichmäßigerer Verlauf der Last über die Zeit erwartet werden. Die Zeit, die Hibernate hier mit Reflection zusätzlich braucht, wird bei JDO in den Kompilierungsvorgang ausgelagert. Ein weiteres Problem wird noch bei einem Vergleich aufgeworfen, in dem JDO eine Rolle spielt: Unterschiedliche Implementationen von JDO können und werden unterschiedliche Ergebnisse erzeugen. Aufgrund der Auslegung als Spezifikation wäre der Vergleich so aussagefähig wie der zwischen einem Motor eines bestimmten Herstellers und einem beliebigen anderen. Unter diesen Gesichtspunkten sind die Benchmarks von PolePosition mit Vorsicht zu genießen, auch wenn sie Tendenzen aufzeigen:

Barcelona Benchmark	Write	Read	Query	Delete
db4o/4.5.200	0,07	0,23	0,33	0,42
Hibernate/hsqldb	0,36	2,38	175,99	1,66
Hibernate/mysql	2,18	4,77	2,21	7,33
JDBC/MySQL	0,99	2,46	0,55	2,75
JDBC/HSQldb	0,11	0,10	222,57	0,29
JDO/VOA/mysql	1,00	1,00	1,00	1,00

Tabelle 3.1: Barcelona Benchmark mit 100 Selects auf 30.000 Objekte (Zeiten relativ zur JDO-Implementation „Versant Open Access“)¹

¹ PolePosition: PolePosition Benchmark Results
<http://polepos.sourceforge.net/results/PolePosition.pdf>, 20.12.2005

Einen weiteren Test mit höherem Realitätsbezug vollzog Kris Van Echelpoel nach der Entwicklung einer Applikation für Apotheker²: Während in einem Test mit einem Client und einem Server die Lösung mit JDBC 2-3x schneller war, als die verwendete JDO 1.0 Implementation und anfangs die Füllung des Cache im zeitlichen Verlauf gut zu erkennen war, fiel ein erneuter Test mit 50 Clients auf einem Server zugunsten JDOs aus. Der Geschwindigkeitsunterschied ging auf 25% zurück und die Übertragungsspitzen bei JDBC von 50-60 Sekunden Wartezeit entfielen völlig. Zudem war auf Seiten des Servers die Auslastung der CPU mit 70-100% besser gegenüber JDBC mit 20-50%. JDBC schaffte es nicht die CPU für die ankommende Flut der Anfragen zu belegen und deshalb wuchs die Antwortzeit des Servers in wirtschaftlich indiskutable Bereiche an.

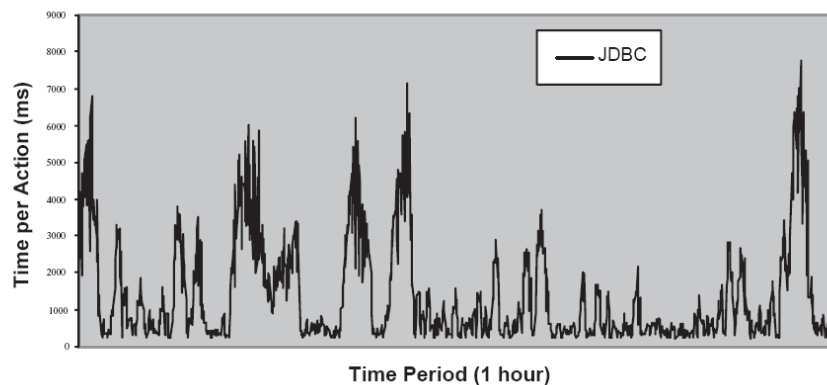


Bild 3.8: Ausführungszeiten mit JDBC in Van Echelpoels 2. Benchmark¹

3.3.3 In dem Arbeitsaufwand

Arbeitsaufwand teilt sich in Schulung, Umsetzung und Wartung.

Bei der Schulung selbst sind im direkten Vergleich zu anderen Techniken wohl die wenigsten Einsparungen zu vollziehen, genauer gesagt es sind zuerst Investitionen zu tätigen. Dabei ist der Aufbau der Sprache und der Metadateien sehr einfach strukturiert und man kommt beim Einsatz schnell zu Ergebnissen. Die Vertiefung dieser Kenntnisse gerade im Bereich des Bridge-Mapping kann dabei etwas mehr Zeit in Anspruch nehmen (was weniger eine Eigenschaft der Sprache ist, als mehr des Prozesses selbst). Kommerzielle Hersteller bieten reguläre Lehrgänge im Zeitumfang von ein

² Van Echelpoel, K.: Java Data Objects: A Revolution in Java Databanking?, Master Thesis, Antwerpen 2002, S. 59 – 67

¹ Van Echelpoel, K.: Java Data Objects: A Revolution in Java Databanking?, Master Thesis, Antwerpen 2002, S. 64

bis drei Tagen an kombiniert mit einer Beratung zur Verwendung des Produkts.

In der Umsetzung kann im Gegensatz zu anderen Techniken die Teilung von reiner Programmierung der Applikation und deren Datenspeicherung sehr gut vollzogen werden. Die Programmierer schreiben die Anwendung mit einfachen Java-Objekten (POJOs), einem allgemeinen Rahmengerüst zur Verwaltung der Transaktionen und müssen ihre Objekte lediglich mit dem Bytecode Enhancer und XML-Dateien anreichern. Die Abfragesprache JDOQL mit ihrer Java-ähnlichen Syntax erleichtert zudem die Eingewöhnungsphase für die Programmierer.

Auf der Datenspeicherseite können mit Hilfe von verschiedenen XML-Dateien die Anpassungen für das Mapping parallel geschrieben und die Datenbank entworfen werden. Das Grundgerüst für das Datenmodell oder das Objektmodell erzeugen die Tools mancher Anbieter automatisch. Mit Hilfe dieser Tools kann man auch testen, ob das Mapping dem Modell in der Datenbank entspricht, oder ob Nachbesserungen nötig sind.

In der Wartung und Aufrechterhaltung des Betriebs der Anwendung greifen unterschiedliche Mechanismen dem zuständigen Personal unter die Arme: Ein ganz besonderes Feature ist bspw. die vollkommene Kompatibilität von per JDO angereicherten Klassen zwischen unterschiedlichen Herstellern: Standard-Tags in den Metadateien können von jedem Bytecode Enhancer und von jeder Implementation unterschiedlicher Hersteller ohne Einschränkung verwendet werden – eine erneute Anreicherung ist nicht nötig.

Existieren herstellerspezifische Tags in den Metadateien, werden diese von Implementationen anderer Hersteller ignoriert. Wurden Klassen mit diesen Tags angereichert, werden diese Erweiterungen im Bytecode von anderen Implementationen ebenfalls ignoriert und immer der kleinste, gemeinsame Nenner verwendet: Die Standard-Tags.

Ebenso ist es möglich, den Datenspeicher zu wechseln mit einer Änderung der Eigenschaften des PersistenceManagers und (sofern zutreffend) der Datei für das O/R-Mapping. Alternativ können auch mehrere solcher Dateien für das Mapping gleichzeitig koexistieren. Insofern wäre es möglich durch die Änderung der Properties-Datei (zentrale Konfigurationsdatei) für den PersistenceManager oder per Mausklick in der Anwendung ohne

Rücksicht auf die Architektur des Datenspeichers diesen zu wechseln: Zum Beispiel von einer binären Datei zu einem RDBMS zu einem ODBMS zu einem EIS binnen wenigen Sekunden ist alles möglich. Die alten Datenbestände müssen zwar trotzdem migriert werden, doch kann dies ebenfalls mit Hilfe von JDO in Angriff genommen werden: Ist einmal die Anbindung zum neuen Datenspeicher vorhanden, können die alten Objekte wie im alten Programm per JDO geladen und ebenso wie im alten Programm in den neuen Datenspeicher abgelegt werden. Der Umfang des Migrationprojekts beschränkt sich also auf die Erstellung eines Tools, das diese zwei Aktionen mit sämtlichen Hauptobjekten durchführt – eine manuelle Übertragung inklusive dem Erlernen der Sprache und Eigenschaften des neuen Datenspeichers entfallen. Solche Projekte rangierten in der Vergangenheit je nach Datenkomplexität zwischen mehreren Wochen bis Monaten oder wurden aus monetären oder technischen Gründen erst gar nicht angefangen.

Die Technik der Named Queries gibt weitere Flexibilität in der Wartung der Applikation bei einer Änderung im Datenspeicher. Den Abfragen wird im Quellcode ein Name gegeben und die eigentliche Syntax in einer XML-Datei ausgelagert, die auch nachträglich mit einem Texteditor angepasst werden kann ohne lange im Quelltext zu suchen, oder diesen direkt an mehreren Stellen fehleranfällig per Copy & Paste oder automatischer „Ersetzen“ Funktion auszutauschen.

3.3.4 In dem Datenmodell

Die Belange von Datenbankadministratoren, alter Unternehmenssoftware mit alten Datenbeständen und objektorientiert denkenden Programmierern geht bisweilen stark auseinander.

Während erstere beiden Gruppen aus Gründen des manuellen Eingreifens (im Ausnahmefall) oder des maschinellen Eingreifens (im Regelfall) auf die Regeln der Normalisierung angewiesen sind, möchten Programmierer gerne relationale Datenbanken als „Blackbox“ gebrauchen und das DBMS der Applikation überlassen.

Abgesehen von der Vergabe eindeutiger Bezeichner für Objekte, auf die nur die JDO Implementation Zugriff hat, oder Primärschlüssel, die der Programmierer selber als eigenständige Klassen erstellen muss, stellen selbst einfache Relationen ein Repertoire an Auswahlmöglichkeiten dar.

Im Datenmodell werden Relationen durch Schlüssel (im Falle 1:n oder 1:1) oder durch Schlüssel und Hilfstabellen (im Falle m:n) dargestellt. Im Objektmodell ist jedoch nicht jede Verbindung sinnvoll und u.U. mit einem großen Datenaufkommen verbunden, auf welches jedoch nie zugegriffen wird. Insofern sind im Klassendiagramm für die Erstellung des Mappings noch zwingend die Navigationsrichtungen einzuzeichnen. Nur so ist es der Person, die das Mapping erstellt, ersichtlich, ob bspw. in einer 1:n-Beziehung die Sammlung auf der 1-Seite weggelassen werden kann, oder ob eine m:n-Beziehung im ER-Modell als unidirektionale 1:n-Beziehung mit Hilfstabelle umgesetzt werden muss.

In diesem Punkt hat JDO 2.0 die größten Änderungen erfahren: nämlich das Mapping. War es bei JDO 1.0 ausschließlich über Herstellererweiterungen möglich ein Mapping zu erstellen, sind diese Befehle nun in die Spezifikation mit eingeflossen.

4 JDO 2.0 im Detail

4.1 Installation

Für die Installation benötigt man die unter Kapitel 1.1.2 aufgelistete Software. Der DBDesigner und phpMyAdmin sind für das Nachvollziehen der Beispielaufgaben nicht zwingend nötig. Sie dienen durch ihre Verwendung nur der besseren Administrierbarkeit der Tabellen, bzw. der grafischen Veranschaulichung eben dieser.

Zusätzlich zur Einrichtung der Laufzeitumgebung, des Webserver und der Datenbank, wird Eclipse in ein Verzeichnis des Entwicklungsrechners installiert und das JPOX Plug-In in das Plug-In Verzeichnis entpackt. Falls Eclipse nach einem erneuten Start keinen Eintrag namens „JPOX Eclipse Plug-In“ für diese Erweiterung unter dem Menüeintrag Window→Preferences generiert, hilft ein einmaliger Aufruf des Programms mit dem Kommando „Eclipse -clean“.

Nun müssen im Unterpunkt „Classpath“ des Plug-Ins noch die JARs eingetragen werden, die zur Benutzung JDOs nötig sind, und mit einem Rechtsklick auf das Projekt im Package Explorer die JPOX Unterstützung im Unterpunkt „JPOX“ aktiviert werden. Im gleichen Unterpunkt kann man danach auch den automatischen Start des Bytecode Enhancers nach jedem erneuten Kompilieren des Projekts an- und abstellen.

4.2 Vorbereitung

4.2.1 Interfaces

4.2.1.1 Übersicht

Die API im Paket `javax.jdo` sind die, mit denen sich der Entwickler am häufigsten befassen muss und diese sind wie folgt aufgliedert:

- Extent
- FetchPlan
- InstanceCallbacks
- PersistenceManager
- PersistenceManagerFactory
- Query
- Transaction
- JDOHelper
- und einige spezielle JDO Exceptions

Die Pakete `javax.jdo.datastore` und `javax.jdo.identity` beinhalten Methoden für Spezialfälle innerhalb einer Applikation, wenn bspw. der 2nd-

Level Cache genauer konfiguriert werden muss, oder wenn Identitätsklassen für einfache Datentypen benötigt werden. Da diese Pakete den regulären Betrieb JDOs nur selten berühren, wird auf eine genauere Beschreibung in dieser Arbeit verzichtet.

Das Paket `javax.jdo.listener` wird im Kapitel 4.6.6 behandelt.

Die SPI ist unter `javax.jdo.spi` zu finden und adressiert die Hersteller von Applikationsserver Containern und JDO Implementationen, weshalb sie hier nicht erläutert wird.

4.2.1.2 API

Der PersistenceManager ist das Herzstück der Verwaltung der persistenten Daten und deren Lebenszyklen. Er bietet Methoden zur Erstellung von Query und Transaction Objekten an.

Die PersistenceManagerFactory ist die Verbindung der JDO Implementation zum Datenspeicher. Hier werden sämtliche Konfigurationen der Implementation und der Datenspeicherverbindung verwaltet. Weiterhin stellt die Fabrik PersistenceManager her.

Ein Transaction Objekt grenzt innerhalb eines PersistenceManagers eine Transaktion zwischen dem Begin und dem Commit (bzw. Rollback) vom Rest der Applikation ab. Auch kann mit dem Objekt die Transaktion selbst beeinflusst werden.

Ein Extent Objekt ist die Sammlung aller Instanzen einer Klasse und – falls gewünscht – deren Unterklassen. Diese Sammlung kann Schritt für Schritt iteriert, oder für Abfragen benutzt werden.

Ein Query Objekt stellt die eigentliche Abfrage eines Datenbestands dar in der die Suchformel, Beschränkungen des Ergebnis und eventuell verwendete Variablen gespeichert werden.

Das InstanceCallbacks-Interface sorgt für den Aufruf von benutzerdefinierten Methoden beim Erreichen eines neuen Abschnitts im

Lebenszyklus einer Instanz. Für jeden Abschnitt ist eine eigene optional verwendbare Callback Methode definiert.

Mit Methoden des Interfaces FetchPlan können Fetch Groups aktiviert werden.

Das JDOHelper Interface stellt eine lose Sammlung von statischen Hilfsmethoden zur Verfügung, die sonst keinem anderen Interface zugeordnet werden konnten. Beispiele wären hier die Erstellung einer PersistenceManagerFactory unter Verwendung eines Properties-Objekt oder das Abfragen der Zustände von Instanzen.

4.2.1.3 Exceptions

Aus der Wurzel `java.lang.RuntimeException` geht die `JDOException` hervor, die als Basisklasse für sämtliche JDO Ausnahmen fungiert. Die `JDOException` hat eine Methode namens „`getFailedObject()`“ mit deren Hilfe das Objekt erreicht werden kann, welches die Ausnahme ausgelöst hat. Sollten hingegen mehrere Objekte involviert sein, wird für jedes Objekt eine eigene Ausnahme generiert, die in einer zusätzlichen Ausnahme eingebettet werden. Wird eine Ausnahme geworfen, veranlasst JDO automatisch ein Rollback der Transaktion.

Die weiteren Ausnahmen gliedern sich in einem Baum mit zwei Ästen auf, wie in Anhang D zu sehen ist. Die Aufgliederung fügt den Ausnahmen keinerlei weitere Eigenschaften hinzu, sondern dient lediglich der besseren Klassifikation, womit in der Applikation eine einfachere Verzweigung in der Fehlerbehandlung stattfinden kann.

`JDOCanRetryException` ist die Basisklasse für Ausnahmen, deren zur Ausnahme führender Quellcode wiederholt werden kann.

`JDOUserException` ist die Basisklasse für Ausnahmen, deren Quelle in der Applikation selbst liegen und wiederholt werden kann.

`JDODetachedFieldAccessException` tritt auf, wenn auf nicht geladene Felder einer abgekoppelten Instanz zugegriffen wird.

JDONullIdentityException wird geworfen, wenn eine Primärschlüsselklasse erstellt werden soll und der Wert des übergebenen Parameters null ist. Bei der Verwendung von flüchtigen Instanzen kann dies passieren, da jene keine Identität zu dem Zeitpunkt haben.

JDOUnsupportedOptionException ist eine Unterklasse, die bei Verwendung einer durch die Implementation nicht unterstützten Option geworfen wird.

JDOUserCallbackException tritt auf bei Fehlern innerhalb der Ausführung von Callbacks und Listnern – also bei Methoden, die aufgerufen werden sobald sich der Zustand eines Objektes ändert.

JDODataStoreException ist die Basisklasse für Ausnahmen, deren Quelle im Datenspeicher liegt und wiederholt werden kann.

JDOObjectNotFoundException wird geworfen, wenn auf ein Objekt zugegriffen wird, das im Datenspeicher nicht vorhanden ist.

JDOFatalException ist die Basisklasse für Ausnahmen, deren zur Ausnahme führender Quellcode nicht wiederholt werden kann. Die zugehörige Transaktion sollte also kein zweites Mal verwendet werden.

JDOFatalUserException wird geworfen, wenn im Quellcode eine Ausnahme auftritt, die nicht wiederholt werden kann.

JDOFatalInternalException ist eine Ausnahme, welche die Implementation selbst betrifft und dem Hersteller gemeldet werden sollte. Eine Behebung des Fehlers durch den Programmierer ist nicht möglich – höchstens eine Umgehung.

JDOFatalDataStoreException ist die Basisklasse für Ausnahmen, deren Herkunft im Datenspeicher und somit außerhalb der Reichweite der Applikation liegt. Beispiele wären hier Connection Timeouts, oder Fehler in der Datenübertragung.

JDOOptimisticVerificationException wird benutzt beim Fehlschlagen einer Überprüfung der Daten in einer optimistischen Transaktion. Eine optimistische Transaktion setzt die Instanzen erst bei Änderung in einen transaktionellen Kontext und prüft die Daten zum Zeitpunkt des Commit.

4.2.2 Konfiguration

Bevor ein PersistenceManager durch seine Fabrik erzeugt und verwendet werden kann, muss diese konfiguriert werden. Diese Optionen erhält sie wahlweise über eine Properties-Instanz oder nach der Erzeugung über die entsprechenden Setter-Methoden.

Die von JDO 2.0 verwendeten Standardoptionen sind folgende:

`javax.jdo.PersistenceManagerFactoryClass`

Der Name der Implementation der PersistenceManagerFactory (in diesem Fall „org.jpox.PersistenceManagerFactoryImpl“).

`javax.jdo.option.ConnectionDriverName`

Der Name des JDBC-Treibers (für das Beispiel in dieser Arbeit „com.mysql.jdbc.Driver“)

`javax.jdo.option.ConnectionURL`

Die URL für die Verbindung zum Datenspeicher (für die verwendete MySQL-Datenbank: „jdbc:mysql://localhost/cogdb“)

`javax.jdo.option.ConnectionUserName`

Der Name des Benutzers unter dem sich JDO anmelden soll.

`javax.jdo.option.ConnectionPassword`

Das Passwort für das Benutzerkonto.

`javax.jdo.option.DetachAllOnCommit` (Defaultwert: false)

Sollen nach einem Commit sämtliche Objekte der Transaktion von JDO abgekoppelt werden (bspw. um diese an eine Applikation zu verschicken, die außerhalb der Reichweite der JDO-Implementation liegt)?

`javax.jdo.option.IgnoreCache` (Defaultwert: false)

Soll die Funktion des Caches abgeschaltet werden?

`javax.jdo.option.Multithreaded` (Defaultwert: false)

Läuft der PersistenceManager in einer Multithread-Applikation?

`javax.jdo.option.NontransactionalRead` (Defaultwert: false)

Sollen nichttransaktionale Lesevorgänge erlaubt werden?

`javax.jdo.option.NontransactionalWrite` (Defaultwert: `false`)

Sollen nichttransaktionale Schreibvorgänge erlaubt werden?

`javax.jdo.option.Optimistic` (Defaultwert: `false`)

Sollen optimistische Transaktionen verwendet werden?

`javax.jdo.option.RetainValues` (Defaultwert: `false`)

Sollen sämtliche Variablen persistenter Instanzen nach einem Commit im Cache verbleiben?

`javax.jdo.option.RestoreValues` (Defaultwert: `false`)

Bei einem Wert von „`true`“ werden sämtliche Werte einer Instanz bei einem fehlgeschlagenen Rollback auf ihre Anfangswerte zurückgesetzt.

Bei einem Wert von „`false`“ löscht die Implementation diese Werte und holt sie erst bei erneutem Zugriff aus dem Datenspeicher.

`javax.jdo.option.Mapping`

Name der ORM Metadata Mappingdatei. Wird bspw. ein Wert „`mysql`“ eingesetzt, sucht die Implementation erst eine Datei namens „`Klassenname-mysql.orm`“ und falls diese Datei nicht existiert, „`package-mysql.orm`“. Wird der Wert nicht gesetzt, wird angenommen, dass sämtliche relevanten Informationen in der Datei mit den JDO Metadaten stehen.

`javax.jdo.mapping.Catalog`

Präfix für sämtliche Tabellen des RDBMS, sofern Kataloge durch die Software unterstützt werden.

`javax.jdo.mapping.Schema`

Präfix für sämtliche Tabellen des RDBMS, sofern Schemata durch die Software unterstützt werden.

Eine vollständige Liste aller von der Implementation unterstützten Optionen kann mit Hilfe der Methode „`supportedOptions()`“ der `PersistenceManagerFactory` ausgegeben werden. Die Defaultwerte sind dann über `getOptionname()` derselben erreichbar.

4.3 Persistente und flüchtige Klassen

Generell werden in Anwendungen die Klassen zwischen „persistenten Klassen“ und „flüchtigen Klassen“ (transient) unterschieden. Hierbei spielt es weniger eine Rolle, ob die persistente Klasse auch im Datenspeicher gesichert wird, oder nicht. Aus diesem Grund verwendet die JDO-

Spezifikation auch den Begriff „persistence-capable“ für die erstere Art von Klassen.

Um also „persistence-capable“ zu werden, muss eine Klasse den Default-Konstruktor besitzen, in den Metadaten verzeichnet sein und zusammen mit diesen Metadaten den Bytecode Enhancer durchlaufen. Während dieses Vorgangs schaltet der Bytecode Enhancer spezielle Getter- und Settermethoden zwischen die Schnittstelle der Klassen und deren Variablen mit deren Hilfe JDO seine Aufgaben erledigen kann.

Die Mindestvoraussetzungen für die Metadaten sind neben der reinen Angabe der Klasse auch die der Elemente der Sammlungen innerhalb der jeweiligen Klassen (List, Map, etc.). Weitere Angaben beziehen sich dann noch auf optionale Angaben wie z.B. der Ausnahme einiger Felder von der Speicherung, oder eine andere Zuordnung zu der Default-Fetch-Group. Doch auch flüchtige Klassen können – auch wenn sie nicht gespeichert werden – in einen transaktionellen Modus gelangen, damit deren Werte nach einem Rollback zurückgesetzt werden.

Abschließend ist noch zu erwähnen, dass die Metadaten nicht nur zur Zeit des Vorgangs der Erweiterung von Klassen benötigt werden, sondern auch während des regulären Betriebs der Anwendung.

4.4 O/R-Mapping

4.4.1 Einleitung

Das Mapping wird für relationale Datenbanken verwendet und entweder in der Datei „package.jdo“ integriert oder in „package.orm“ bzw. „package-NameDerMappingdatei.orm“ abgelegt.

Für JDO 2.0 werden folgende Doctypes benutzt.

JDO-Datei:

```
<!DOCTYPE jdo PUBLIC "-//Sun Microsystems, Inc.//DTD Java  
Data Objects Metadata 2.0//EN"  
"http://java.sun.com/dtd/jdo_2_0.dtd">
```

Das Wurzelement, das die Datei umschließt heißt in diesem Fall <jdo> und endet mit </jdo> wie es in XML üblich ist.

ORM-Datei:

```
<!DOCTYPE orm PUBLIC "-//Sun Microsystems, Inc.//DTD Java  
Data Objects Mapping Metadata 2.0//EN"  
"http://java.sun.com/dtd/orm_2_0.dtd">
```

Für ORM-Dateien startet die Datei mit <orm> und endet mit </orm>.

Neben den Angaben, wie welches Feld in welcher Spalte abgelegt wird, können noch spezielle Angaben zu den Beschränkungen des Datenspeichers benutzt werden (damit bspw. die Länge eines Strings schon vor dem Speichern in einem zu kleinen Varchar-Feld geprüft wird), oder genauere Angaben zu Bereichen wie inhaltlichen Beschränkungen (unique) oder Indexen. Das Standardmapping für viele Datentypen ist bereits in der Spezifikation grob festgelegt, doch kann davon leicht abgewichen werden (wahlweise durch Speicherung von String in Char- oder Varchar-Spalten). Gerade beim Mapping von Beziehungen ist es wichtig, wie diese abgebildet werden – auch hier hilft diese Datei für die richtige Erstellung von Hilfstabellen oder deren Vermeidung, um z.B. alten Datenbeständen gerecht zu werden.

Die in den Kapiteln verwendeten Beispiele sind größtenteils konstruiert und finden sich nicht in der gezeigten Form in der Spieledatenbank wieder.

4.4.2 Class Mapping

Einfaches Mapping von Klassen stellt das kleinste Problem in der Verwendung JDOs dar, da hier lediglich eine Klasse mit einfachen Datentypen in eine Tabelle übertragen wird.

Folgende Klasse dient nun als ein Beispiel:

```
public class Esrb {
    private String name;

    private Esrb() {}

    public Esrb(String name) {
        this.name = name;
    }

    // weitere Methoden...
}
```

Dann reicht die folgende Zeile in der Datei package.jdo aus:

```
<class name="Esrb" />
```

Nun noch mit dem Bytecode Enhancer erweitert, ist die Klasse „persistence-capable“ und das Mapping wird der Implementation überlassen.

Mit Hilfe der ORM-Datei, können danach genauere Angaben gemacht werden:

```
<class name="Esrb" table="cogdb_esrb">
    <datastore-identity column="esrbID" />
    <field name="name">
        <column length="15" />
    </field>
</class>
```

Jede dieser Angaben ist optional und beschreibt lediglich, wie die Daten in das zugrundeliegende RDBMS gespeichert werden. Im vorliegenden Fall wird für die Klasse Esrb die Tabelle „cogdb_esrb“ verwendet. Die Primärschlüssel der Tabelle verwaltet das RDBMS automatisch (in MySQL über einen autoincrement) in der Spalte „esrbID“. Die Angabe, dass Datastore-Identity verwendet wird und JDO automatisch nach das beste Verfahren zur Erzeugung eines Primärschlüssels wählt, wird ebenso wie die Speicherung des Feldes „name“ in der Spalte „name“ implizit per Standardwert festgelegt. Lediglich die Längenbeschränkung von 15 Zeichen muss in einem eigenen Tag angegeben werden.

Denkbare Änderungen wären z.B. die Änderung des Spaltennamen mit „name=„einNeuerName““ im Column-Tag, oder mit „jdbc-type=„char““ eine Änderung auf diesen fixen Spaltentyp (Standardwert ist hier Varchar). Eine vollständige Auflistung aller möglichen Optionen befindet sich in Anhang B in Form der DTDs der JDO- und ORM-Metadaten.

4.4.3 Zweittabellen

Manchmal kann es nützlich sein, wenn die Felder einer Klasse im RDBMS nicht in einer einzigen Tabelle, sondern in zwei getrennten Tabellen aufbewahrt werden. Denkbar wären aus Teilobjekten zusammengesetzte Objekte, die jedoch im RDBMS getrennt aufbewahrt werden sollen. Dazu ein konstruiertes Beispiel: Die Tabelle Hardware beschreibt verschiedenste Konfigurationen von PC-Systemen. Ein Teilaspekt dieser Klasse wäre also mit folgendem Quellcode beschrieben.

```
public class Hardware {
    private String cpuName;
    private float takt;
    private float ram;

    // Konstruktoren und Methoden
}
```

Die Angabe in der Datei `package.jdo` entspricht dem letzten Beispiel, nur dass der Name anders lautet. Ein Unterschied zeigt sich jedoch im Mapping:

```
<class name="Hardware" table="cogdb_hardware">
  <datastore-identity column="hardwareID" />
  <join table="Cpu" column="cpuID" />
  <field name="cpuName" table="cpu" />
  <field name="takt" table="cpu" />
  <field name="ram" />
</class>
```

Die letzte Angabe über das Feld „ram“ kann weggelassen werden, da einfache Datentypen immer gesichert werden (sofern nicht anders angegeben). Das oben stehende Mapping ist ausreichend, um jede gespeicherte Instanz auf beide Tabellen zu verteilen, doch geht diese Verwobenheit etwas weiter, als bei einer üblichen Relation: Für jede Instanz, die in „cogdb_hardware“ gesichert wird, wird eine Zeile in der Tabelle „cogdb_cpu“ angelegt.

4.4.4 Eingebettete Objekte

Die Verwendung eingebetteter Objekte stellt eine Vorgehensweise dar, die sehr stark an objektorientierte Verhaltensmuster angelehnt ist und sich mit den Prinzipien der Normalisierung nicht vereinbaren lässt. Allgemein gilt, dass solche Objekte in einer einzigen Tabelle gespeichert und nicht über Relationen aufgeschlüsselt werden.

Zwei Auswahlmöglichkeiten sind möglich:

1. Über mehrere Klassen eingebettete Objekte teilen sich eine Tabelle.
2. Eingebettete Objekte von Sammlungen werden in eine Hilfstabelle ausgelagert ohne eine eigene Tabelle für ihre Klasse zu erhalten.

Ein Beispiel für die erste Option:

```
public class Hardware {
    private Cpu cpu;
    private float ram;

    // Konstruktoren und Methoden
}

public class Cpu {
    private String name;
    private float takt;
    private float ram;

    // Konstruktoren und Methoden
}
```

Die JDO-Metadaten zeigen wieder keine Besonderheiten, nur die ORM-Metadaten ändern sich wieder:

```
<class name="Hardware" table="cogdb_hardware">
  <datastore-identity column="hardwareID" />
  <field name="cpu">
    <embedded null-indicator-column="cpuName">
      <field name="name" column="cpuName" />
      <field name="takt" column="takt" />
    </embedded>
  </field>
  <field name="ram" />
</class>
```

```

<class name="Cpu" table="cogdb_cpu">
  <datastore-identity column="cpuID" />
  <field name="name" />
  <field name="takt" />
</class>

```

Mit Hilfe dieser Einstellungen wird eine Instanz der Klasse Cpu in die Tabelle „cogdb_hardware“ geschrieben, falls sie in der Klasse Hardware verwendet wird. Das Attribut „null-indicator-column“ sorgt dafür, dass beim Auslesen der Datenbank die Felder auf „null“ gesetzt werden, sobald die angegebene Spalte ebenfalls einen „null“-Wert besitzt.

Auf eine Darstellung eines Beispiels der zweiten Möglichkeit verzichte ich an dieser Stelle, da ich der Meinung bin, dass ein eingebettetes Objekt u.U. sinnvoll verwendet werden kann, aber Sammlungen solcher Objekte in Hilfstabellen zum RDBMS-Pendant von Spaghetticode führen.

4.4.5 Serialisierte Objekte

Natürlich ist es einem Entwickler immer freigestellt, Objekte oder Sammlungen von Objekten in einer serialisierten Form im Datenspeicher innerhalb eines BLOB-Feldes zu sichern – generell kann davon jedoch aus Gründen, die in Kapitel 2 verzeichnet sind abgeraten werden.

Als Beispiel dienen die zwei Klassen aus Kapitel 4.4.4 mit folgendem Mapping:

```

<class name="Hardware" table="cogdb_hardware">
  <datastore-identity column="hardwareID" />
  <field name="cpu" serialized="true" />
  <field name="ram" />
</class>

```

Mit der Option „serialized="true““ können Cpu-Objekte serialisiert gespeichert werden mit den damit verbundenen Problemen: Abfragen auf solche Objekte sind nicht mehr möglich.

4.4.6 Constraints

Constraints werden durch die Datenbank umgesetzt und im Mapping festgelegt, damit die Implementation davon in Kenntnis gesetzt wird. Ob ein solche Einschränkung in der Applikation selbst sichergestellt ist, obliegt dem Programmierer und fällt nicht in den Zuständigkeitsbereich JDOs.

Die Umsetzung der Constraints kann auf zwei Arten geschehen:

1. Aus Sicht des Feldes / Objektmodells
2. Aus Sicht der Klasse / Datenmodells

Der erste Weg sieht generell folgendermaßen aus:

```
<class>
  <field>
    <index name="indexName" />
  </field>
</class>
```

Der zweite Weg so:

```
<class>
  <index name="indexName">
    <column name="spaltenname" />
  </index>
  ...
</class>
```

Das Schlüsselwort „index“ kann durch „unique“ ersetzt werden, doch für Fremdschlüssel (*foreign-key*) funktioniert nur der erste Weg und für Primärschlüssel (*primary-key*) nur der zweite.

4.4.7 Vererbung

JDO 2.0 unterstützt drei verschiedene Ansätze zur Umsetzung von Vererbungshierarchien innerhalb einer Anwendung:

1. New-Table

Für jede Klasse (auch abstrakte Klassen) wird eine eigene Tabelle angelegt und die Primärschlüssel spezialisierter Klassen beinhalten die Primärschlüssel ihrer Elternklassen – je tiefer die Vererbungshierarchie also geht, desto größer wird dadurch der Aufwand ein einzelnes Objekt zusammensetzen.

2. Superclass-Table

Mit dem Wert „*superclass-table*“ werden sämtliche Datenfelder in der Hierarchie nach oben weitergereicht und von der dortigen Klasse gesichert. Dadurch entstehen sehr wahrscheinlich einzelne Tabellen mit einer großen Spaltenanzahl, was die Übersichtlichkeit beeinträchtigen kann. Andererseits ist die Performance der Abfragen bei passender Verwendung der DFG die beste, weil Joins komplett entfallen. Zusätzlich zum Inheritance-Tag muss noch ein Discriminator-Tag unterhalb dessen eingesetzt werden, um den Typ der Klasse innerhalb der Datenbank zu vermerken.

3. Subclass-Table

Die Felder von per „*subclass-table*“ definierten Klassen werden in der Vererbungshierarchie eine Stufe tiefer gereicht und in den Tabellen der dieser Klassen abgelegt. Hierbei kann es zu doppelter

Datenhaltung kommen, wenn in der gleichen Hierarchiestufe mehrere Kindklassen zu finden sind. Dieses Attribut eignet sich z.B. sehr gut für die Speicherung der Felder einer abstrakten Klasse, die aufgrund ihrer Natur nie instanziiert wird.

Eine Angabe der Strategie im Inheritance-Tag muss immer angegeben werden, weil durch JDO kein Standardwert gesetzt wird.

4.4.8 Interfaces

Ein Problem im O/R-Mapping sind Interfaces, die durch fehlende Unterstützung im relationalen Datenmodell nicht direkt in eine solche Datenbank übertragen werden können. Zur Speicherung gibt es unterschiedliche Ansätze wie z.B. bei TopLink, in denen der Tabelle neben dem eigentlichen Fremdschlüssel noch ein Feld für den Datentyp der referenzierten Klasse (Tabelle) im Klartext genannt wird; doch dies ist kein SQL-Standard.

Ein anderer Ansatz wäre es, jeder möglichen Klasse, die das Interface verwendet, eine eigene Spalte zu geben und den Wert der Spalten aller Klassen, die nicht referenziert werden, auf „null“ zu setzen. Hier zeigt sich dann, wie gut eine Datenbank mit dem Nullwert umgehen kann. Zur Minimierung dieser Nullwerte gibt es auch herstellerspezifische JDO-Extensions, die der Implementation genau angeben, welche Werte zu erwarten sind. Es muss also nicht für alle Klassen, die das Interface benutzen, eine Spalte angelegt werden.

Eine Klasse, die eine Schnittstelle in der Spieledatenbank verwendet, besitzt den Namen „System“:

```
<class name="System">
  <implements name="net.cogdb.database.data.Gegenstand" />
  ...
</class>
```

Außerdem müssen Klassen, die ein Interface als Variablentyp verwenden, im Field-Tag das Attribut „persistence-modifier="persistent"“ stehen haben, weil Interfaces nicht implizit persistent sind, wie es bspw. bei einfachen Datentypen der Fall ist.

4.4.9 Objekte (FCOs)

Die Unterstützung von Objekten als Feldtyp funktioniert analog zu der Umsetzung der Interfaces. Der einzige Unterschied ist das Fehlen des Implements-Tags, da keine Schnittstellen involviert sind.

4.4.10 Arrays

Arrays werden optional unterstützt und die Umsetzung ist nur schwach reglementiert, weshalb an dieser Stelle erstmals kein implementationsübergreifender Einsatz gewährt werden kann.

Zur Veranschaulichung dienen hier wieder drei fiktive Beispiele, wie die Verbindung zwischen den Klassen „Titel“ und „Spiel“ umgesetzt werden kann:

1. Serialisiertes Array

```
<class name="Titel">
  <field name="spielumsetzungen" serialized="true">
    <array />
  </field>
</class>
```

2. Array in einer Hilfstabelle (Join-Table)

```
<class name="Titel" table="cogdb_titel">
  <field name="spielumsetzungen"
    table="cogdb_spielumsetzungen">
    <array />
    <join column="spielID" />
    <order column="reihenfolge" />
  </field>
</class>
```

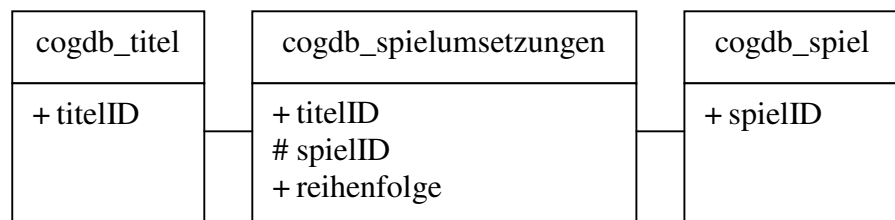


Bild 4.1: Datenmodell „Array mit Hilfstabelle“

Die Umsetzung ähnelt einer 1:n-Beziehung mit einer List-Sammlung, nur dass hier keine Relation, sondern ein Array Anwendung findet.

3. Fremdschlüssel im Element des Arrays

```
<class name="Titel" table="cogdb_titel">
  <field name="spielumsetzungen"
    table="cogdb_spielumsetzungen">
    <array />
    <element column="spielID" />
    <order column="reihenfolge" />
  </field>
</class>
```

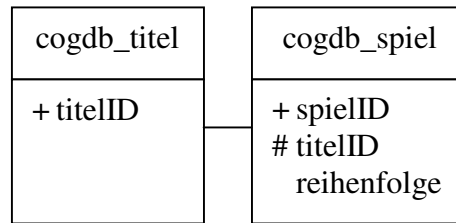


Bild 4.2: Datenmodell „Array mit Fremdschlüssel“

Auch hier ähnelt die Umsetzung wieder einer List-Sammlung. In diesem Falle der Umsetzung einer solchen 1:n-Relation mit Hilfe eines Fremdschlüssels.

4.4.11 1:1 Beziehungen

Diese Art von Beziehung ist kurz erklärt: Ein Objekt referenziert ein anderes und speichert diese Referenz in einem Feld. In der Applikation stellt der Entwickler sicher, dass diese Verbindung auch eindeutig ist, während in der Datenbank diese Aufgabe ein Unique-Constraint auf der betreffenden Spalte übernimmt.

Das Mapping für das Feld mit der Referenz fällt einfach aus, da neben einfachen Datentypen die Relationen ebenfalls im Datenspeicher abgelegt werden, sofern sie von einer persistenten Klasse stammen.

```

<class name="Spiel" table="cogdb_spiel">
  <field name="esrb" column="esrbID" />
</class>
  
```

Dieses Mapping reicht für eine unidirektionale 1:1-Beziehung aus. Für eine bidirektionale 1:1-Beziehung muss in der referenzierten Klasse ein Feld samt Mapping existieren, welches auf die erste Klasse verweist.

4.4.12 Sammlungen

4.4.12.1 Allgemeines

Wie im letzten Kapitel beschrieben, zählt es sich bei der Erstellung des Mappings aus, wenn der Programmierer in Kenntnis der Navigationsrichtungen ist. Ob eine Relation der Form 1:n oder von der anderen Seite betrachtet n:1 ist, spielt zwar im ER-Modell keine Rolle, aber es schlägt sich direkt ins Datenmodell durch und kann ein unnötig erhöhtes Datenaufkommen verursachen (gespeicherte Daten, Overhead im Mapping, oder erhöhtes Nachrichtenaufkommen zwischen Threads).

Wichtig an dieser Stelle ist, dass auf jeden Fall in den Metadaten der Datei `package.jdo` die Elementtypen der Sammlung angegeben werden und das

übergeordnete Interface der Sammlung („Collection“ oder „Map“). Was übrig bleibt, ist die Anpassung an den Datenspeicher durch O/R-Mapping und Angaben, welche Daten durch welche Klasse verwaltet werden sollen. Die Klassen „Zubehoer“ und „Ware“ decken beide Syntaxvarianten ab:

```

<class name="Zubehoer">
  ...
  <field name="zubehoerkompatibilitaet">
    <collection element-type="Zubehoergruppe" />
  </field>
</class>

<class name="Ware">
  ...
  <field name="inhalt">
    <map key-type="Inhalt" value-type="java.lang.Byte" />
  </field>
</class>

```

4.4.12.2 1:N Collection

Es gibt beim Mapping von Sammlungen zwei Auswahlmöglichkeiten: Einerseits die Verwendung von Fremdschlüsseln und andererseits die Verwendung einer Hilfstabelle (Join-Table). Wurde die Wahl getroffen, muss noch festgelegt werden, ob die Navigationsrichtung im Objektmodell uni- oder bidirektional implementiert worden ist.

Im Beispiel der Klassen „Spiel“ und „Thema“ wäre folgendes Mapping für eine Hilfstabelle anzuwenden:

```

<class name="Spiel" table="cogdb_spiel">
  <datastore-identity column="spielID" />
  ...
  <field name="schnitte" table="cogdb_schnitte">
    <join>
      <column name="spielID" />
    </join>
    <element>
      <column name="themaID" />
    </element>
  </field>
  ...
</class>

<class name="Thema" table="cogdb_thema">
  <datastore-identity column="themaID" />
  ...
</class>

```

Das Ergebnis wären folgende drei Tabellen innerhalb der Datenbank:

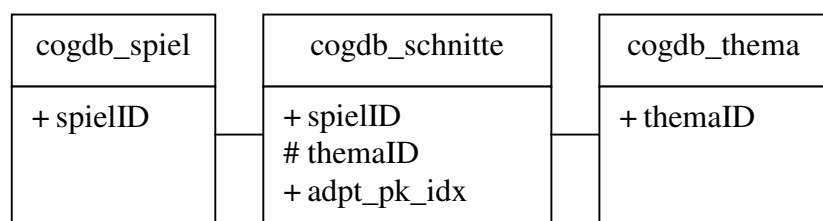


Bild 4.3: Datenmodell „1:n-Collection mit Hilfstabelle“

Wird eine Umsetzung per Fremdschlüsselbeziehung benötigt, ist das

Mapping einfacher:

```
<class name="Spiel" table="cogdb_spiel">
  <datastore-identity column="spielID" />
  ...
  <field name="schnitte" table="cogdb_schnitte" />
  ...
</class>

<class name="Thema" table="cogdb_thema">
  <datastore-identity column="themaID" />
  ...
</class>
```

Das Ergebnis wären folgende zwei Tabellen innerhalb der Datenbank:

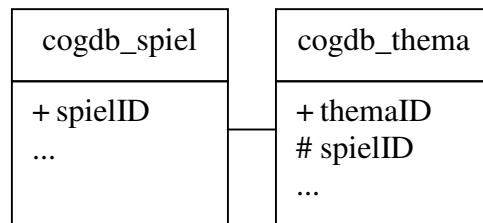


Bild 4.4: Abbildung „1:n-Collection mit Fremdschlüssel“

Für das bidirektionale Mapping wird lediglich das Attribut „mapped-by“ auf der 1-Seite der Relation im Field-Tag gesetzt, wobei der Wert den Namen des Feldes beinhalten muss, der im Element die besitzende Klasse referenziert. Auf diese Weise wird in JDO angegeben, dass das Mapping des Elements durch seine Klasse selber durchgeführt wird.

Die Tabellen in der Datenbank sind bei beiden Navigationsrichtungen gleich, nur muss in der Java-Klasse ein Feld für die Referenz angelegt sein.

4.4.12.3 1:N Set

Der einzige Unterschied im Vergleich zur Collection besteht hier in der Verwendung eines Sets als Sammlungstyp in der entsprechenden Java-Klasse. Sämtliche Anweisungen für das Mapping bleiben gleich und aufgrund der Verwendung eines Sets sind doppelte Einträge in der Sammlung nicht möglich. Aus diesem Grund entfällt auch die Indexspalte in der Hilfstabelle.

4.4.12.4 1:N List

Eine Liste ist eine Collection mit einer Erweiterung: Die Reihenfolge der Elemente ist von Bedeutung. Elemente dürfen auch mehrfach vorkommen, wobei durch die Verbindung mit dem Platz des Elements innerhalb der Liste jedes einzigartig ist.

Neben der Benutzung des Sammlungstyps „List“ in der jeweiligen Klasse ist noch ein weiteres Tag unterhalb des Field-Tags ansprechbar: `<order>` mit dessen Hilfe das Attribut „name="spaltenname"“ für den Listenplatz gesetzt werden kann.

4.4.12.5 1:N Map

Maps in 1:n-Beziehungen sind etwas komplexer in den möglichen Erscheinungsformen als die restlichen Sammlungen. Die Kombination aus Schlüssel und Wert können mit einer, zwei oder keiner persistenten Klasse bevölkert werden. Zusätzlich kann sich der Programmierer noch für eine Umsetzung mit Hilfstabelle entscheiden oder Fremdschlüssel verwenden.

Bei der Verwendung einer Hilfstabelle muss unterhalb des Field-Tags der Sammlung ein Join-Tag eingefügt werden. Die Spaltennamen für den Schlüssel und den Wert können in einem Column-Tag unterhalb des Key- bzw. Value-Tags auf der gleichen Hierarchieebene wie das Join-Tag gesetzt werden.

Werden Fremdschlüssel eingesetzt, muss das „mapped-by“ Attribut in bidirektionalen Beziehungen gesetzt werden wie in Kapitel 4.4.12.2 beschrieben. Außerdem muss unterhalb des Field-Tags der Sammlung in einem Key- und / oder Value-Tag das Attribut „mapped-by“ für die entsprechende Klasse eingefügt werden.

4.4.13 N:1 Beziehung

Eine n:1-Beziehung ist aus Sicht der Klasse quasi eine aufgeweichte 1:1-Beziehung, die immer unidirektional ist und der auf der Gegenseite keine Sammlung von Objekten gegenüber steht.

Das Mapping ist demnach wie in Kapitel 4.4.11 durchzuführen, und soll die Beziehung bidirektional gestaltet werden, wird das Mapping aus Kapitel 4.4.12 benutzt.

4.4.14 M:N Beziehung

Hierbei zeigt sich unter allen Möglichkeiten im Mapping am stärksten, wie sehr sich das Datenmodell und das Objektmodell unterscheiden können: Die m:n-Beziehungen werden im ER-Modell als zwei 1:n-Beziehungen mit einer Hilfstabelle abgebildet. Innerhalb des Mappings hingegen hat der

Programmierer die Wahl, ob dies auch für das Datenmodell übernehmen will, oder ob zwei voneinander unabhängige Hilfstabellen Anwendung finden sollen.

Für die Sammlungen List und Set gibt es zwei getrennte Mappings, während bei Maps der Programmierer eine Wahl in der Implementation zwischen diesen beiden hat. Nach den vorhergehenden Kapiteln wird an dieser Stelle auf eine ausführliche Auflistung der Syntax und der zu erwartenden Tabellen verzichtet, weil m:n-Beziehungen lediglich eine Kombination dieser bereits dargestellten Techniken ist.

- Set

Sets verwenden eine eindeutige Beziehung ohne doppelte Einträge in ihren Sammlungen. Daher ist auch nur eine Hilfstabelle von Nöten, die durch das Einfügen eines Join-Tags unterhalb des Field-Tags in der einen Klasse erreicht wird und zusätzlich einem „mapped-by“ Attribut im Field-Tag der anderen Klasse.

- List

Eine List in einer m:n-Beziehung muss nicht zwingend in beiden Listen die gleiche Reihenfolge beinhalten, weshalb hier zwei Hilfstabellen verwendet werden. Unterhalb des Feldes der jeweiligen Klasse wird mit einem Join-Tag eine Hilfstabelle angelegt.

- Map

Diese Sammlung lassen dem Programmierer die Wahl, ob sie mit einer oder zwei Hilfstabellen erstellt werden sollen. Die Verfahrensweisen sind analog zu den jeweils o.g. Beispielen.

4.4.15 Compound Identity Relationship

Die Compound Identity Relationship ist ein Sonderfall im Bereich der Application Identity: Zwei Objekte stehen miteinander in Verbindung, wobei der Primärschlüssel des einen ein Teil des Primärschlüssels des anderen Objektes ist.

Für die Umsetzung müssen beide Objekte „Application Identity“ benutzen und einen bestimmten Aufbau aufweisen. Als Beispiel dient hier ein abstraktes Beispiel (Bestellung und Bestellposten), da sich in der Spieledatenbank keine passenden Klassen befinden:

- Die Primärschlüsselklasse der Bestellung muss als Variable die Bestellnummer beinhalten und die Variable muss denselben Namen wie in der eigentlichen Instanz haben.
- Die Instanz des Bestellpostens hat eine Artikelnummer und eine Referenz auf das Bestellsobjekt. In der zugehörigen Primärschlüsselklasse befindet sich ein Feld mit demselben Namen und Inhalt wie die Artikelnummer und ein Feld mit dem Variablentyp der Primärschlüsselklasse der Bestellung und dem Namen des Bestellobjektes.

Hier der zugehörige Quellcode:

```
class Bestellung {
    long bestellnummer;
    set<Bestellposten> artikel;
    ...
}

class BestellungId {
    long bestellnummer; //gleicher Name wie in obiger Klasse
    ...
}

class Bestellposten {
    Bestellung bestellung;
    long artikelnummer;
    ...
}

class BestellpostenId {
    BestellungId bestellung; // Typ und Name sind wichtig!
    long artikelnummer; //gleicher Name wie in obiger Klasse
    ...
}
```

4.4.16 Voneinander abhängige Felder

In einer Verbindung von Objekten ist es manchmal nicht sinnvoll, dass ein Objekt weiter existiert, wenn das andere gelöscht wird. Die Felder können also bereits in JDO in eine entsprechende Abhängigkeit gesetzt werden. Sollte also in der Verbindung der Klassen „Titel“ und „Spiel“ ein Titel gelöscht werden, wäre es möglich, dass die Umsetzungen des Spiels alleine nicht mehr existieren können.

Mit dem Attribut „dependent="true"“ im Field-Tag, bzw. „dependent-element="true"“ im Collection-Tag werden die betroffenen Datensätze aus der Tabelle „Spiel“ gelöscht. Der Standardwert für diese Attribute ist „false“ und daher braucht es nur bei Verwendung gesetzt werden.

Vor unvorsichtigem Gebrauch muss jedoch gewarnt werden: JDO prüft nicht, ob noch weitere Objekte eine Referenz auf das abhängige Objekt bestehen. Es wird direkt gelöscht.

4.5 Bytecode-Enhancer

Der Bytecode-Enhancer ist ein externes Java-Programm und kann per Hand aufgerufen werden, durch einen Build-Task (Ant, Make, etc.), oder direkt aus der IDE, sofern eine Unterstützung in Form eines Plug-In, o.ä. vorliegt.

Die Syntax zum Aufruf des JPOX JDO Bytecode Enhancers ist folgende:

```
java -cp classpath org.jpox.enhancer.JPOXEnhancer [options] [jdo-files]
```

wobei folgende Optionen zulässig sind:

- d <Zielpfad> : Zielverzeichnis für erweiterte Klassen
- checkonly : Status der Klassen (erweitert / nicht erweitert)
- verify : Prüfung der Klassen
- v : Ausführliche Ausgabe (Verbose Output)

Der Classpath muss folgende Java Archive und Dateien enthalten:

- jpox-enhancer.jar
- jpox.jar
- bcel.jar
- jdo.jar
- log4j.jar
- die verwendeten Klassen
- die Dateien mit den Metadaten

4.6 Persistence-Manager

4.6.1 Einleitung

Die hauptsächlichen Abläufe für CRUD-Operationen laufen innerhalb einer Transaktion ab. Bevor jedoch eine Transaktion erstellt werden kann, muss ein PersistenceManager-Objekt von einer PersistenceManagerFactory erstellt werden. Eine PersistenceManagerFactory kann mehrere dieser Manager haben und jeder Manager besitzt exakt eine Transaktion.

Eine gewöhnliche Verwendung JDOs sieht folgendermaßen aus:

```
import java.io.*;
import javax.jdo.*;

public class EineKlasse {
    private PersistenceManagerFactory pmf;
    private PersistenceManager pm;
    private Transaction tx;

    public EineKlasse() {}

    public void eineMethode() {
        try {
            InputStream pStream = new FileInputStream("datei");
            Properties props = new Properties();
            jdoproperties.load(pStream);
            pmf = JDOHelper.getPersistenceManagerFactory(props);
            pm = pmf.getPersistenceManager();
            tx = pm.currentTransaction();
            tx.begin();
            // Logik innerhalb der Transaktion
            tx.commit();
        } catch (JDOException e) {
            // Behandlung der Ausnahme
        } finally {
            if (tx.isActive())
                tx.rollback();
            pm.close();
        }
    }
}
```

Die Erstellung einer PersistenceManagerFactory kann auf unterschiedliche Weisen geschehen:

- über eine Properties-Objekt wie im Beispiel oben
- ohne irgendwelche Parameter
- über einen String, der auf eine Ressource verweist mit einem Properties-Objekt
- über einen Ort per JNDI

Die folgenden vier Kapitel behandeln die Anweisungen innerhalb des Transaktionsrahmens, der die eigentlichen Aufgaben zur Persistenz beinhaltet.

4.6.2 Create

JDO unterstützt zwei Arten zur Speicherung von Objekten, doch vorerst muss die zu sichernde Klasse „persistence-capable“ sein und mit den zu sichernden Daten gefüllt werden.

Die erste Art ist die explizite Persistenz bei der die zu speichernden Klassen in einer Methode dem PersistenceManager als Parameter übergeben werden:

```
pm.makePersistent (einObjekt);  
pm.makePersistentAll (einObjektArray);  
pm.makePersistentAll (eineCollection);
```

Dadurch werden die übergebenen Objekte oder das einzelne Objekt direkt dem PersistenceManager als zu speichernd vorgemerkt. Die Speicherung selbst wird erst zum Zeitpunkt des Commit vollzogen.

Die zweite Art speichert Objekte per Erreichbarkeit. Es ist keine eigene Methode damit verbunden, sondern diese Technik markiert sämtliche referenzierte Objekte der Parameter samt deren Referenzen als zu sichernd. So kann also ein kompletter Objektbaum gesichert werden nur unter Angabe des Wurzelements. Ist jedoch im Objektbaum ein flüchtiges Objekt vorhanden, wird ab diesem Element die Sicherung in dem Ast unterbrochen – Referenzen flüchtiger Objekte werden generell nicht gesichert und damit können auch nachfolgende Objekte nicht gesichert werden, egal ob diese persistence-capable sind, oder nicht.

4.6.3 Read

Leseoperationen werden ausschließlich über Extents (Kapitel 4.7.2) und Queries (Kapitel 4.7.3) durchgeführt und in diesen Kapiteln genauer beschrieben.

4.6.4 Update

Eine Änderung der Daten eines Objekts funktioniert ohne zusätzliche Methoden: Das Objekt wird durch JDO aus dem Datenspeicher gelesen, in der Applikation geändert und zum Zeitpunkt des Commit werden die Änderungen permanent übernommen.

4.6.5 Delete

Vergleichbar mit dem Sichern von Objekten im Datenspeicher gibt es drei Methoden zur Löschung derselben:

```
pm.deletePersistent (einObjekt);  
pm.deletePersistentAll (einObjektArray);  
pm.deletePersistentAll (eineCollection);
```

Sofern jedoch keine abhängigen Felder definiert wurden, oder kaskadierende Löschungen im Fremdschlüssel des Datenspeichers definiert wurden, werden referenzierte Objekte nicht gleichzeitig aus der Datenbank entfernt.

4.6.6 Callback-Methoden und Listener

Es gibt drei durch persistente Klasse implementierbare Interfaces, die Methoden bereitstellen, welche bei bestimmten Änderungen des Objektzustands aufgerufen werden:

1. InstanceCallbacks

Diese Methoden betreffen eine Zustandsänderung der Instanz.

a. `jdoPostLoad()`

Führt Code nach dem Laden aus dem Datenspeicher aus, wobei innerhalb dieser Methoden kein vermittelter Zugriff auf die Felder der Instanzen gewährt ist. Es sollten also keine Variablen außerhalb der DFG angesprochen werden.

b. `jdoPreClear()`

Führt Code vor dem Zurücksetzen der Werte einer Instanz nach einem Commit aus. Der Zugriff erfolgt nicht durch die Implementation und unterliegt so denselben Beschränkungen wie bei der vorherigen Methode (mit der Ausnahme, dass bereits Werte geladen worden sein können).

c. `jdoPreStore()`

Führt Code vor dem Sichern in den Datenspeicher aus. Der Zugriff wird an dieser Stelle über JDO vermittelt und Änderungen der Variablen werden in den Datenspeicher übernommen.

d. `jdoPreDelete()`

Führt Code vor dem Löschen einer Instanz aus. Der Zugriff wird wieder über JDO vermittelt und mit Hilfe der Methode kann bspw. ein „Cascading Delete“ softwareseitig umgesetzt werden.

2. AttachCallback

a. `jdoPreAttach()`

b. `jdoPostAttach(Object attached)`

Diese beiden Callback-Methoden werden aufgerufen bevor, bzw. nachdem ein Objekt vom PersistenceManager einer Applikation angekoppelt wurde.

3. DetachCallback

- a. `jdoPreDetach()`
- b. `jdoPostDetach(Object detached)`

Die Callback-Methoden des DetachCallback-Interface werden aufgerufen bevor, bzw. nachdem ein Objekt vom PersistenceManager einer Applikation abgekoppelt wurde.

Weiterhin gibt es noch acht Interfaces, die Methoden für Events auf Ebene des PersistenceManagers bereitstellen und diesem entweder per Methode „`addInstanceLifecycleListener(InstanceLifecycleListener listener, java.lang.Class[] classes)`“ bekannt gegeben werden, oder dessen Fabrik übergeben werden können.

Die Interfaces sind folgende:

- AttachLifecycleListener
- ClearLifecycleListener
- CreateLifecycleListener
- DeleteLifecycleListener
- DetachLifecycleListener
- DirtyLifecycleListener
- LoadLifecycleListener
- StoreLifecycleListener

Der Zustand „Dirty“ wird erreicht, wenn Variablen einer von JDO verwalteten Instanz innerhalb einer Transaktion geändert werden.

4.7 JDOQL

4.7.1 Die Default-Fetch-Group

Die Default-Fetch-Group stellt die Gruppe von Feldern dar, die bei der Instanzierung eines Objektes durch die JDO-Implementation auf jeden Fall aus dem Datenspeicher geladen werden.

Darunter fallen:

- Felder des Primärschlüssels
- Primitive Variablentypen
- `java.util.Date`
- Folgende Typen aus `java.lang`: Boolean, Byte, Short, Character, Integer, Long, Float, Double, String, Number und Object
- `java.math.BigDecimal` und `java.math.BigInteger`

Eine Anpassung der Gruppe wird bei Leistungsoptimierung immer als erster Schritt angewandt: Per Angabe auf Ebene des Field-Tags kann das Attribut „`default-fetch-group`“ auf „`true`“, oder „`false`“ gesetzt werden.

4.7.2 Extents

Extents sind die Sammlungen aller gesicherten Instanzen eines bestimmten Objektes in einem Datenspeicher und wahlweise dessen Unterklassen.

Der Befehl, um einen Extent zu bekommen, lautet:

```
pm.getExtent(eineKlasse.class);  
pm.getExtent(eineKlasse.class, false);
```

Erster Befehl ist mit zweitem gleichzusetzen, nur dass dann anstatt „`false`“ „`true`“ eingesetzt werden muss. Der zweite Parameter gibt an, ob

Unterklassen auch mit eingeschlossen werden sollen, oder nicht.

Zur Verwendung des Extents erlangt man mit der Methode „`iterator()`“.

Genauere Abfragen werden mit Hilfe von Queries vollzogen, die auf den Extent angewendet werden.

4.7.3 Queries

4.7.3.1 Allgemeines

Queries sind die Abfragen innerhalb der Applikation. Sie können vorkompiliert werden, um schneller zu reagieren und werden über Parameter mit Variablen ausgestattet – bei einer Änderung eines Parameters einer Abfrage muss diese also nicht jedes Mal neu kompiliert werden, wie es z.B. bei JDBC der Fall ist.

Die Erstellung einer Query kann auf vier Arten vollzogen werden, die Hersteller durch eigene, zusätzliche Abfragesprachen erweitern können.

Die folgenden vier Kapiteln behandeln:

1. Query-Objekte und deren Methoden
2. Single String Queries
3. SQL Queries
4. Named Queries

Jedoch haben alle Queries einige Bestandteile gemeinsam:

- Parameter

Parameter sind Bestandteile eines Filters, die aus der Applikation übergeben werden. Sie sind in Where-Klauseln auf der rechten Seite des Gleichheitszeichens.

- Variablen

Sollen Sammlungen durchsucht werden, beinhaltet der Filter zwei Elemente: Zuerst wird geprüft, ob ein Element einer Klasse überhaupt in der Sammlung vorhanden ist und danach wird dieses Element gegen einen Parameter geprüft.

Ein Beispiel ist:

```
WHERE (spiel.schnitte.contains(thema) && thema.name_de == \"christliche Symbole\")
```

Zur Wahrung der Kompatibilität muss die Contains-Klausel auf der linken Seite des „&&“ stehen und der Test auf Gleichheit auf der rechten Seite. Der Ausdruck sollte in Klammern gesetzt werden (wie im Beispiel zu sehen ist) weil ansonsten in komplexeren Filtern die Gefahr besteht, dass der Filter durch Nichtbeachtung der Priorität der Operatoren in der falschen Reihenfolge abgearbeitet wird.

- Import von Namensräumen

Ist der Namensraum eines verwendeten Variablentyps der Anwendung zum Zeitpunkt der Ausführung der Abfrage nicht bekannt, wird eine Ausnahme geworfen. Dies kann umgangen werden, indem mit Hilfe einer Importanweisung der Typ bekannt gegeben wird.

- Optionen zur Änderung des Ergebnisses

Unter diesem Punkt sind diverse Operationen zusammengefasst, welche das Ergebnis einer Abfrage nachträglich bearbeiten, wie z.B. die Zusammenfassung unter bestimmten Gesichtspunkten (`group by`), die alphabetische Sortierung (`order by`), oder eine Limitierung des Ergebnisses auf eine bestimmte Breite an Datensätzen (`range`, was analog zum SQL-Statement `limit` funktioniert).

4.7.3.2 Query-Objekte und deren Methoden

Die einzige Möglichkeit in JDO Version 1.0 Abfragen zu erstellen, war über die Methoden der Query-Objekte die gewünschten Strings zu übergeben.

Es wurden immer mehrere Anweisungen benötigt, um eine Anfrage an die Datenbank zu stellen und der Ablauf war immer der gleiche:

1. Abfrageziel festlegen
Ziele können eine Klasse, ein Extent oder das Ergebnis einer anderen Abfrage sein.
2. Filter schreiben
Per String müssen die gewünschten Bedingungen übergeben werden.
3. Query-Objekt erstellen
Durch die passende Methode des PersistenceManagers wird ein Query-Objekt erzeugt.
4. Weitere Optionen festlegen
Es können z.B. durch weitere Methodenaufrufe nähere Angaben zum Namensraum der verwendeten Klassen gemacht werden, oder Variablen im Filter deklariert werden.
5. Query ausführen und in gewünschte Sammlung casten

Ein Beispiel für eine solche Abfrage wäre folgende:

```
Extent e = pm.getExtent(net.cogdb.database.data.Plattform);
Query q = pm.newQuery(e);
q.setOrdering("name ascending");
Collection allePlattformen = (Collection)q.execute();
```

4.7.3.3 Single String Queries

Single String Queries sind ab JDO 2.0 neu hinzugekommen, wie die restlichen Arten von Queries, die ab hier besprochen werden.

Im Gegensatz zu den vielen Methodenaufrufen der vorigen Vorgehensweise sind hier alle Bestandteile innerhalb eines einzigen Strings zusammengefasst, der dem Query-Objekt übergeben wird.

Der Aufbau des Strings ist fest vorgegeben:

```
SELECT [UNIQUE] [<Ergebnis>] [INTO <Ergebnis-Klasse>]
  [FROM <Kandidaten-Klasse> [EXCLUDE SUBCLASSES]]
  [WHERE <Filter>]
  [VARIABLES <Deklarationen der Variablen>]
  [PARAMETERS <Deklarationen der Parameter>]
  [IMPORTS <Deklarationen der Importe>]
  [GROUP BY <Gruppierung>]
  [ORDER BY <Sortierung>]
  [RANGE <Start>, <Ende>]
```

Das Schlüsselwort „Unique“ hat die Bedeutung, dass maximal eine Instanz zurückgegeben wird und die „Into“-Klausel castet die Elemente des Ergebnisses in eine andere Klasse.

Nach der Erstellung können diese Werte noch über die entsprechenden Methoden des Query-Objektes geändert werden.

4.7.3.4 SQL Queries

Etwas, was von vielen Programmierer in JDO 1.0 vermisst wurde, war die Möglichkeit über das Framework direkte SQL-Befehle an die Datenbank zu übermitteln. Früher war dies zwar auch durchführbar mittels einer Umgehung JDOs mit direkten JDBC-Aufrufen, aber nun ist es auch über JDO möglich.

Die Syntax zur Erstellung einer solchen SQL-Abfrage ist diese:

```
Query q = pm.newQuery("javax.jdo.query.SQL", dieSqlSyntax);
```

Als zweiter String kann jede beliebige SQL-Anweisung übertragen werden.

4.7.3.5 Named Queries

Named Queries verwenden innerhalb der Applikation einen Platzhalter für die Abfrage, die dann innerhalb der Metadaten abgelegt wird (sowohl die JDO- als auch ORM-Metadaten sind für Named Queries geeignet).

Ein Query-Objekt kann mit dieser Syntax erzeugt werden:

```
Query q =  
    pm.newNamedQuery(net.cogdb.database.data.Plattform,  
        "allePlattformen");
```

Innerhalb der Metadaten für die Klasse „Plattform“ muss noch dieses Element eingetragen werden:

```
<query name="allePlattformen"  
    language="javax.jdo.query.JDOQL">  
    <![CDATA[SELECT FROM net.cogdb.database.data.Plattform  
        GROUP BY name ascending]]>  
</query>
```

Der Vorteil an dieser Vorgehensweise ist, dass die Syntax der Abfragen nachträglich ohne Neukompilierung der Anwendung geändert werden kann.

4.8 Identität

4.8.1 Datastore-Identity

Hierbei übernimmt JDO oder die Datenbank sämtliche Vergaben der Primärschlüssel und versteckt die entsprechende Spalte vor der Anwendung. Datastore-Identity stellt zwar die einfachste Möglichkeit zur Verwendung

JDOs dar, aber in einigen Fällen verletzt sie das relationale Datenmodell in einem vertretbaren Bereich: Tabellen, in denen sich der Primärschlüssel regulär ausschließlich über mehr als zwei Fremdschlüssel zusammensetzt (also keine einfachen m:n-Beziehungen zwischen zwei Klassen), erhalten so als Ersatz einen bedeutungslosen Primärschlüssel zugeteilt. Es obliegt hierbei dem Benutzer einen Unique-Constraint auf die Spalten der Fremdschlüssel zu legen, um die Einzigartigkeit zu gewährleisten. Folgende Mechanismen zur Primärschlüsselvergabe werden unterstützt:

- Native

Die Implementation wählt den am besten passenden Mechanismus aus.

- Increment

Die Implementation verwaltet einen Generator zur Erzeugung eindeutiger Schlüssel in Form automatisch inkrementierender Zahlen, die in einer separaten Tabelle gesichert werden.

- Autoassign

Die „auto increment“-Funktion des Datenspeichers wird verwendet, sofern er die unterstützt. Es wird verfahren, wie bei der „Increment“-Option, nur dass die Datenbank die Schlüsselvergabe verwaltet.

- Identity

Die „identity“-Funktion des Datenspeichers wird verwendet, sofern er diese unterstützt. Die Arbeitsweise ist inhaltsgleich mit Autoassign und unterscheidet sich nur im Schlüsselwort des DBMS.

- Sequence

Verwendet eine Sequenz-Funktion des Datenspeichers, sofern er sie unterstützt. Bei einer Sequenz kann der Programmierer selber die Grenzen und Schritte bei der Schlüsselvergabe festlegen.

- UUID-String

JDO erstellt UUIDs, die als Primärschlüssel gespeichert werden.

- UUID-Hex

JDO erstellt UUIDs im Hex-Format, die als Primärschlüssel gespeichert werden.

4.8.2 Application-Identity

Das Gegenteil zur Datastore-Identity ist die Application-Identity. Hier obliegt es dem Programmierer Klassen für den Primärschlüssel anzulegen.

Ähnlich wie bei EJB 2.0 gibt es Mindestvoraussetzungen für die Primärschlüsselklasse:

- Sie muss als „public“ deklariert sein.
- Sie muss das Interface „Serializable“ implementieren.
- Sie braucht einen Konstruktor, der keine Argumente übernimmt.
- Sämtliche nicht als „static“ ausgezeichnete Variablen müssen serialisierbar sein. Folgende Variablentypen werden von allen Implementationen unterstützt: String, Date, Byte, Short, Integer, Long, Float, Double, BigDecimal und BigInteger.
- Alle serialisierbaren, nicht statischen Variablen müssen „public“ sein.
- Sie muss Variablen mit denselben Namen und Typen wie jedes zum Primärschlüssel zugehörige Feld ihrer Klasse haben, die nicht als „static“ ausgezeichnet sind.
- Die Methoden „equals()“ und „hashCode()“ müssen alle zum Primärschlüssel zugehörigen Felder beinhalten.
- Wenn sie innerhalb ihrer zugehörigen Klasse definiert wird, muss sie als „static“ sein.
- Die Methode „toString()“ muss überschrieben werden und einen String ausgeben, der als Parameter für einen Konstruktor verwendet werden kann.
- Ein Konstruktor muss erzeugt werden, der als Parameter einen String (oder einen String und eine Klasse) übernimmt und eine Instanz der Primärschlüsselklasse zurückgibt, die den gleichen String mit „toString()“ erzeugt, welchen sie als Parameter bekam.

Insofern darf der Programmierer dafür sorgen, dass die Einzigartigkeit seiner Instanzen gewährleistet ist. Denkbar wäre, dass der Primärschlüssel eine Auftragsnummer ist und man auf diese Weise ein wenig Speicher in der Datenbank sparen möchte oder es mag andere Gründe wie Performance, Datenbestände aus Altsystemen, etc. haben.

4.8.3 Nondurable-Identity

In manchen Fällen kann es vorkommen, dass Instanzen selber keine Identität besitzen müssen. Die Gewichtung liegt in diesen Anwendungen auf einer sehr schnellen Verwendung dieser Daten mit wenig bis gar keinem Verwaltungsaufwand, wie es z.B. in Logfiles der Fall ist.

Der Zugriff auf diese Art von Daten braucht nicht besonders schnell vonstatten zu gehen und mehrfache Einträge sind geduldet oder schließen sich prinzipiell aus. Ist dies der Fall, kann Nondurable-Identity eine Lösung sein – leider ist deren Anwendungsbereich aufgrund der o.g.

Voraussetzungen beschränkt und so gibt es auch nur wenige Hersteller, die diese Technik umgesetzt haben.

Folgendes ist jedenfalls zu beachten:

- Nachdem eine Transaktion beendet wird, ist die Objektidentität der Instanz und die Instanz selbst nicht mehr ansprechbar.
- Eine Instanz kann nicht zwischen mehreren PersistenceManagern ausgetauscht werden, da sie keine Identität hat.
- Durch Abfragen nacheinander generierte Instanzen der gleichen Objekte aus dem Datenspeicher werden zwar die gleichen Inhalte haben, aber unterschiedliche Identitäten.
- Der Befehl „makePersistent“ wird ausgeführt, auch wenn das Objekt bereits im Datenspeicher vorhanden sein sollte.

Für die Umsetzung der ObjektID-Klassen gilt dies:

- Sie muss als „public“ deklariert sein.
- Sie braucht einen Konstruktor, der keine Argumente übernimmt.
- Alle Felder müssen „public“ sein.
- Die Variablentypen müssen serialisierbar sein.

4.9 Besondere Zugriffsmechanismen

4.9.1 Cache

JDO unterstützt einen zweistufigen Cache in Form eines eigenen, nicht abschaltbaren 1st-Level Caches und einer Schnittstelle für einen 2nd-Level Cache, der optional ist.

Da der 2nd-Level Cache nicht fester Bestandteil JDOs ist, sind folgend nur die Methoden des 1st-Level Caches beschrieben:

- `refresh(Object obj)`
- `refreshAll()`
- `refreshAll(Object[] objs)`
- `refreshAll(Collection objs)`
- `evict(Object obj)`
- `evictAll()`
- `evictAll(Object[] objs)`
- `evictAll(Collection objs)`

Generell stehen dem Programmierer zwei Methoden in je vier unterschiedlichen Ausführungen zur Verfügung: `Refresh` und `Evict`.

Mit dem Befehl „Refresh“ können die Variablen ein oder mehrerer Objekte aktualisiert werden und mit dem Befehl „Evict“ werden ein oder mehrere Objekte aus dem Cache als nicht mehr benötigt markiert. Die letztendliche Löschung aus dem Cache wird durch die Implementation geregelt und der genaue Zeitpunkt ist ähnlich Javas Garbage Collector nicht voraussagbar.

4.9.2 Nicht-transaktionaler Zugriff

Der nicht-transaktionale Modus wird für den schnellen Zugriff auf den Datenspeicher zum Lesen von Daten benutzt, deren Konsistenz innerhalb der Applikation eine untergeordnete Rolle spielt.

Über zwei Flags kann nicht-transaktionaler Zugriff aktiviert werden:

- `javax.jdo.option.NontransactionalRead`
- `javax.jdo.option.NontransactionalWrite`

Nicht-transaktionales Lesen von Daten eignet sich für Situationen, in denen das oben genannte Kriterium zutrifft. Die Umsetzung ist denkbar einfach: Das Flag muss auf den Wert „true“ gesetzt werden, wonach der erzeugte `PersistenceManager` kein Transaktionsobjekt mehr benötigt.

Nicht-transaktionales Schreiben hingegen ist ein wenig irreführend formuliert, weil zwar außerhalb einer Transaktion geschrieben werden kann, aber die Änderungen nie in den Datenspeicher übertragen werden – sie betreffen nur den Cache. Der Sinn hinter dieser Funktion ist die Schaffung einer Möglichkeit für den Programmierer, Änderungen im Datenbestand manuell in den Datenspeicher übertragen zu dürfen und so z.B. seine eigenen Cache-Funktionen zu schreiben.

4.9.3 Optimistische Transaktionen

Der klassische Weg zur Durchführung einer Transaktion beinhaltet die Sperrung aller in der Transaktion involvierten Tabellen ab dem Startzeitpunkt bis zum Zeitpunkt des Commits. Auf diese Weise wird sichergestellt, dass die Daten in der Zwischenzeit nicht durch andere Benutzer geändert werden.

Diese Vorgehensweise ist bei JDO der Standard und sehr effektiv, sofern die gleichen Daten einer hohen Änderungsfrequenz unterliegen, oder die Befehle innerhalb der Transaktion maschinell abgearbeitet werden.

Bei optimistischen Transaktionen treffen diese Fälle nicht zu: Kann davon ausgegangen werden, dass sich die Zugriffe gleichmäßig auf alle Datensätze verteilen, oder die Datensätze generell nicht oft geändert werden, oder die Laufzeit einer Transaktion sogar abhängig von den Reaktionen des Benutzers ist, wird der klassische Weg ineffektiv.

Die Lösung zur Steigerung der Performance ist das Auslesen der benötigten Daten außerhalb einer Transaktion und eine Sperrung der Tabellen erst zum Zeitpunkt des Commit – auf diese Weise wird die Datenbank nicht unnötig lange gesperrt. Einer unvorhergesehenen Änderung des involvierten Datenbestands wird dadurch begegnet, indem vor der Speicherung noch einmal geprüft wird, ob sich die Inhalte in der Datenbank geändert haben. Ist dies der Fall, wird eine Ausnahme geworfen.

Dieses Feature kann über die Methode „`setOptimistic(true)`“ der `PersistenceManagerFactory` gesetzt werden, bzw. über die Properties-Klasse bei der Erstellung einer solchen Fabrik.

5 Nachsatz EJB 3.0 / JDO 2.0

5.1 Persistenzmodell EJB 2.0

Wie bereits in Kapitel 2.2 beschrieben, sind Enterprise Java Beans auf die Umgebung eines Applikationsservers und dessen Container angewiesen, welcher den Rahmen (den „Behälter“) für die Beans liefert.

Was die Datenhaltung dieser Programmteile angeht, wird zwischen den Techniken BMP und CMP unterschieden. Folgend sind die beiden Persistenzmodelle von EJB 2.0 nur kurz angesprochen, da mit dem Erscheinen der Version 3.0 sich die Entwicklung in eine komplett andere Richtung entwickeln wird, wie später in diesem Abschnitt nachzulesen ist.

Die „Bean Managed Persistence“ (BMP) benötigt für jede zu speichernde Bean eine zusätzliche Bean für die Verwaltung des Primärschlüssels und einige Konfigurationsdaten in XML-Format und Methoden für die operative Umsetzung der Speicherung, die in der jeweiligen Bean selbst vorhanden sein müssen. Die Bean ist für die Speicherung ihrer Daten also selber in vollem Umfang zuständig.

Benötigt werden:

- eine separate Klasse, die als Primärschlüssel fungiert
- Methoden für
 - o den Verbindungsaufbau
 - o das Suchen im Datenspeicher
 - o das Laden aus dem Datenspeicher
 - o das Speichern in den Datenspeicher
 - o das Löschen aus dem Datenspeicher
 - o den Verbindungsabbau
- Angaben im Deployment Descriptor (XML-Konfigurationsdatei für den Applikationsserver) mit Angaben über
 - o die Primärschlüsselklasse
 - o die verwendete Datenbank (zwecks Autorisierung durch den Container)
 - o den Persistenz-Typ der Bean (in diesem Fall „Bean“ also Bean Managed Persistence)

Eine Vorgabe, mit welchen Hilfsmitteln diese Methoden geschrieben werden sollen, existiert nicht. Es ist dem Entwickler also freigestellt, ob er z.B. eine direkte JDBC-Verbindung an eine relationale Datenbank wählt oder in eine Datei auf einem Datenträger schreibt oder ein fertiges Persistenz-Framework einsetzt.

Bei der „Container Managed Persistence“ (CMP) werden all diese Aufgaben aus der Bean in den Container verlagert und durch diesen zentral verwaltet. Dies ist für die Erstellung der Beans mit weniger Aufwand verbunden und auch in der Wartung freundlicher.

Änderungen gegenüber der BMP sind:

- die Primärschlüsselklasse entfällt
- die Bean wird abstrakt
- die Methoden für die Variablen werden abstrakt und die zu speichernden Variablen entfallen in den Beans (der Container verwaltet diese)
- sämtliche Methoden zum Datenspeicher (siehe oben) bleiben leer
- es ist verboten, eine Methode im Container namens `findByPrimaryKey()` zu implementieren, dafür können aber beliebig viele `findBy`-Methoden in der Form `findByVariablenname()` für die Variablennamen erstellt werden
- im Deployment Descriptor werden dem Container der Persistenztyp und die zu verwaltenden Felder bekannt gegeben innerhalb spezieller CMP-Anweisungen
- der Persistence Descriptor benötigt zusätzlich noch Angaben über die Datenbank und das zu verwendende Mapping

CMP und BMP können in einem Container koexistieren, was ein Plus an Flexibilität verspricht.

Eine dritte Lösung wäre noch eine Session Bean zu erschaffen, die das Entwurfsmuster einer Fassade umsetzt und die Kommunikation mit dem Datenspeicher übernimmt und einkommende Daten bspw. in ein RDBMS sichert.

Für sich genommen bietet die Bean Managed Persistence nur ein sehr umfangreiches Rahmengerüst, in das der Anwendungsentwickler seinen Code zur Persistenz der Beans einbettet. Die Container Managed

Persistence hingegen geht einen ersten Schritt in Richtung Persistenz-Framework wie Hibernate und JDO (nicht zuletzt, weil diese Frameworks bei manchen Herstellern hinter der CMP stehen), doch bleiben in den Beans immer noch Reste aus der BMP über in Form von abstrakten Methoden, die vom Programmierer nicht „mit Leben“ gefüllt werden können, und es findet eine unnatürliche Trennung von Variablen und der besitzenden Klasse statt.

Gerade die Erstellung solcher Persistenzmodelle war in der Vergangenheit und ist noch heute (EJB 3.0 ist bisher nur in Vorschauversionen erhältlich) mit Umwegen und einigen Kunstgriffen versehen. Die Erweiterung der Möglichkeiten der EJBs von Version 1 nach Version 2 brachte zwar eine größere Auswahl, damit mehr Komplexität und mehr Funktionsvielfalt, doch genau dieser Schritt sorgte auch für einen automatischen Wildwuchs in den Features, der den puren Fleiß und die Fähigkeit Copy & Paste zu verwenden regelrecht herausforderte.

Gründe gegen EJB 2.0 wären zum Beispiel¹:

- viele zum Teil ungenutzte Callback-Methoden
- viele Interfaces (remote, local), die immer nach dem selben Schema programmiert werden müssen
- alle Attribute einer EJB müssen als „public“ deklariert sein (Verstoß gegen das Prinzip der Kapselung)
- Vererbung und Polymorphie einer EJB sind nicht möglich
- EJB QL ist beschränkt in der Abfragesprache im Vergleich zu SQL
- XML-DD ist komplex mit teils redundanten Informationen
- Tests müssen im Normalfall in der Umgebung eines Container stattfinden, was mit hohem Aufwand verbunden ist (Start/Stop des Servers)
- der Java-Compiler findet nicht alle Fehler durch die hohe Anzahl von involvierten Java Interfaces, Klassen und dem außerhalb der Reichweite des Compilers verwalteten XML-DD
- viele Situationen in der Erstellung einer Applikation lassen sich mit Copy & Paste bewältigen, was zu Fehlern durch Routine führen kann

¹ Merkle, B.: EJB 3.0 Public Draft: Zurück zum Überschaubaren.
In: iX, 17. Jg., H. 10, 2005, S. 141

Insofern war es Zeit, dass wieder moderne Ansätze ihren Weg in die EJB-Spezifikation finden, welche diese Probleme der Programmierer beheben können.

5.2 Warum noch JDO?

Diese Frage haben sich in der Vergangenheit viele Menschen vor jeweils unterschiedlichem Hintergrund gestellt.

Die eigentliche Überlegung damals, ein Persistenzframework wie JDO zu erstellen war eine Interoperabilität für die unterliegenden Datenspeicher zu schaffen, wie Java es für die Systemarchitekturen geschafft hat. Der Fokus lag anfangs auf solchen Datenspeichern wie einfachen Dateien oder Objektdatenbanken, zumal die ODMG das Ergebnis ihrer langjährigen Arbeit zu diesem Vorhaben dazugesteuert hatte. Insofern wagte man sich zuerst in Gebiete, die abseits des Kernsegments des Marktes lagen und verfehlte damit das öffentliche Interesse: Warum sollte sich eine Firma mit einem Framework abgeben, das sich auf weniger als 10% der Datenbanksysteme am Markt spezialisiert und nicht Hibernate verwenden, deren Kerngeschäft exakt die anderen 90% andeckt?

Damit war JDO für lange Jahre in der gleichen Nische gelandet, wie die Datenspeicher, die es gut unterstützte. Die Hersteller mussten die Lücken in der Spezifikation mit eigenen Extensions ausfüllen und damit wurde ein Wechsel des Anbieters der Implementation unnötig erschwert; aber gerade mit dem Gegenteil warb JDO.

Das Framework jedoch für sich allein genommen verfolgte Ansätze, die vollkommen von denen EJB 2.0 abwichen, und nach den aktuellen Geschehnissen kann man diese Ansätze ruhig „visionär“ (oder „einen logischen Schritt nach vorne“, je nachdem ob man Anwender oder Entwickler der Frameworks ist) nennen.

Die Kernidee war anstatt Objekte zu benutzen, die ein kompliziertes Gebilde aus Schnittstellen und vielen abstrakten Methoden beinhalten, sich auf einfache, alte Java-Objekte (POJO) zu besinnen, die jeder Informatikschüler in seiner ersten Schulstunde zu Gesicht bekommt. Generell sollte der Programmierer die ihm vertrauten Techniken verwenden und sich auf das Erstellen der Applikation konzentrieren können, während der Datenbankadministrator sein spezialisiertes Wissen über die Datenspeicher beisteuert. Eventuelle Zusätze in den Features oder

Optimierungen in den Abfragen können beim Hersteller der Implementation zentral ins Produkt einfließen.

JDO 1.0 könnte also salopp als „Preview Version“ einer Technik, die sich im Reifeprozess befand, bezeichnet werden. Die Schnittstelle zur Erstellung war und ist bahnbrechend und lediglich die nicht vorhandene Standardisierung der Mappingfähigkeiten relationaler Datenbanken verhinderte eine breite Akzeptanz JDOs.

Doch an anderer Stelle ist die strategische Positionierung mit Hilfe von Persistenzframeworks in vollem Gange: Bspw. gibt es im Bereich der Applikationsserver Firmen, deren Kernkompetenzen in anderen Bereichen als der Container Managed Persistenz liegen und sich so eine erprobte Komponente gerne extern beschaffen. Während also manche Firmen sich für Hibernate entschieden und sich damit unweigerlich auf relationale Datenbanksysteme festlegten, entschieden sich andere Hersteller (z.B. SAP, Trifork) für JDO in Verbindung mit herstellerspezifischen Erweiterungen, um diese Funktionalität zu erreichen und mittlerweile auch zu übertreffen.

5.3 Der Streit um EJB 3.0 und JDO 2.0

Mit der Entwicklung von EJB 3.0 und dem damit verbundenen, neuen Persistenzmodell für diese Technik, wurde plötzlich eine Diskussion seitens Sun Microsystems mit einem öffentlichen Brief an die Entwicklergemeinde in Gang gesetzt, der – um es mit den Worten Craig Russells zu sagen – die Wirkung einer Bombe hatte¹: Die Ankündigung beschrieb eine Einstellung der Arbeiten an weiteren Features JDOs und eine Anpassung der Abfragesprache JDOQL an EJB 3.0, womit die beiden Lösungen faktisch zusammengelegt werden sollten², was das forcierte Ende für JDO gewesen wäre.

Doch wie kam es eigentlich zu diesem Brief?

Das Persistenzmodell für die älteren Versionen von EJB mit ihren Kritikpunkten wurden am Anfang dieses Kapitels beschrieben. Es musste also ein komplett neues Modell (zwar mit Abwärtskompatibilität, doch nicht

¹ Russell, C.: Re: inquiry about JDO / EJB for my dissertation
E-Mail: craig.russell@sun.com, 06.12.2005

² DeMichiel, L.; Russell, C.: A Letter to the Java Technology Community
<http://java.sun.com/j2ee/letter/persistence.html>, 30.09.2004

auf der Vorversion aufbauend) entworfen werden, um sich nicht noch weiter in Erweiterungen zu verstricken.

Aufgrund der vielen Erfahrungen zeigte sich in der Vergangenheit, dass das alte, umfangreiche Modell früher oder später in eine Sackgasse führen würde und eine Rückbesinnung auf alte, einfache Objekte (wieder einmal ist POJO das Schlagwort) die erhoffte Rettung versprechen könnte.

Als Neuerungen wären hier anzusprechen¹:

- die typischen EJB Component Interfaces (Remote-, Home- und die Local-Interfaces fallen ersatzlos weg)
- objektorientierte Standards wie Vererbung und Polymorphie werden möglich
- Java Annotations innerhalb des Quelltextes stellen einen optionalen (und empfohlenen) Weg dar, um Relationen, die Parameter für das Mapping, Transaktionen, Sicherheit und Umgebung in den sonst üblichen XML-Dateien zu ersetzen
- diese Annotations sollen über „Configuration by Exception“ erstellt werden, also nur bei einem Abweichen von einem vorher definierten Standardwert gesetzt werden
- eine trotzdem bestehender XML-DD setzt solche Annotations außer Kraft, womit ein Datenbankadministrator Änderungen vornehmen kann, ohne sich mit dem Quelltext zu beschäftigen
- die eigentlichen CRUD-Operationen werden über einen PersistenceManager abgewickelt, wie bei bereits erhältlichen Persistenzframeworks
- die Abfragesprache EJBQL wird um einige Sprachkonstrukte erweitert (z.B. group-by, explizite inner und outer Joins)
- direkte Anfragen an das RDBMS per SQL sind möglich
- EJBs können nun auch außerhalb des Containers getestet werden
- das Paket `javax.persistence` des EJB-Standards beschränkt sich nicht mehr nur auf die Java Enterprise Edition

Vergleicht man an dieser Stelle die Standards EJB 2.0 und 3.0 miteinander, sieht die inkrementierte Versionsnummer tatsächlich aus wie ein Kunstgriff aus dem Marketing. Die Standards EJB 3.0 und JDO 2.0 gleichen sich sehr

¹ Merkle, B.: EJB 3.0 Public Draft: Zurück zum Überschaubaren.
In: iX, 17. Jg., H. 10, 2005, S. 142 – 145

an, was eine Zusammenlegung der Experten beider Gruppen zu Ungunsten JDOs nahe legen würde.

In der Tat zeigte sich kurze Zeit später bei der ersten Abstimmungsrunde eine Tendenz mancher Firmen, die Spezifikation zu JDO 2.0 im Sande verlaufen zu lassen. Die erste Abstimmung wurde abgelehnt und erst durch den Einsatz der übrigen Firmen, die an einen natürliche Selektion durch den Markt in der Softwareindustrie glauben, konnte die zweite Abstimmungsrunde unter mehr oder weniger Protest zu grünem Licht für JDO führen.

6 Evaluation

6.1 Die aktuelle Situation um JDO 2.0

Mit Hibernate am Markt, jetzt EJB 3.0 und einem bisher mäßig bekannten JDO, dürfte die Frage nach JDOs Nutzen sicherlich berechtigt sein. In der Realität wird es u.U. auf die Bauchentscheidung eines Managers herauslaufen, oder auf die entsprechenden Vorlieben oder Kenntnisse derjenigen, die eine Persistenztechnik einsetzen müssen, um sich für ein Produkt zu entscheiden.

Leider hat in den letzten Monaten nach Erscheinen des offenen Briefes der beiden Leiter der Expertengruppen und der Fertigstellung der Spezifikation zu EJB 3.0 eine FUD-Kampagne stattgefunden. Zwei Beispiele, die im Zeitverlauf besonders deutlich hervorstechen und für viel Aufruhr sorgten, sind einmal Gavin Kings – ein Mitarbeiter Hibernates – Eintrag in seinem Weblog, in dem er Teile JDOs direkt angriff: Er bezeichnete JDOQL als Gegenstand des Abscheus, die Spezifikation als über-komplex wegen einiger optionaler Features und gab zum Schluss an, dass EJB 3.0 näher an Hibernate und TopLink angesiedelt wäre, als an JDO.¹ Sämtliche Punkte des Eintrags wurden später durch Abe White von Solarmetric widerlegt.² Andererseits sprachen die Kommentare zur Abstimmung des „Executive Committee for SE/EE“ Bände: Im ersten Durchgang waren die Firmen, die im Bereich der Softwareerstellung und im universitären Bereich tätig sind für eine direkte Verabschiedung der Spezifikation. Firmen in den Bereichen Hardware und Internet waren aufgrund der plötzlichen Veränderung in der Situation verunsichert und stimmten dagegen. Firmen, deren Kerngeschäft im Segment der Middleware und Datenbanken angesiedelt ist, stimmten ebenfalls dagegen mit der Begründung, dass JDO 2.0 im Verlauf der Zeit zu mächtig geworden sei.

Erst einen Monat und zahlreiche Gespräche später waren diese Verunsicherungen beseitigt. Es gab keine Gegenstimmen und drei Firmen enthielten sich unter stillem Protest: Oracle (Hersteller von TopLink),

¹ King, G.: EJB3

<http://blog.hibernate.org/cgi-bin/blosxom.cgi/2004/05/07#ejb3>, 07.05.2004

² White, A.: JDO FUD

http://www.theserverside.com/news/thread.tss?thread_id=25804, 08.05.2004

JBoss, Inc. (strategischer Partner von Hibernate) und IBM, die sich mit ihren Bedenken gegenüber dem eigentlichen Ziel von JDO 2.0 – einem Maintenance Release von JDO 1.0 und keiner neuen Versionsnummer – übergangen fühlten. Was diese Firmen gemein haben, dürfte offen liegen: Eine eigene Lösung für das Problem des objektrelationalen Mappings, weshalb sie an keiner weiteren Persistenzlösung interessiert sein sollten.

An dieser Stelle muss etwas mehr auf die Details eingegangen werden, die auf den ersten Blick kein Gewicht zu haben scheinen, doch am Ende den Unterschied zwischen einer willkürlichen Entscheidung und einem strategischen Vorteil für den Anwender machen:

Zuerst stellen wir EJB 3.0 und JDO 2.0 in einen direkten Vergleich. JDO ist nun zusammen mit EJB in eine neue Version gekommen. Eine Frage, welche Technik nun länger überleben wird, kann nicht beantwortet werden – selbst Mitglieder der entsprechenden Expertengruppen schreiben, dass eine weitere Versionsnummer bei beiden in den Sternen steht und der Markt entscheiden wird, was wie stark akzeptiert wird.

Weiterhin ist JDO 2.0 keine Neuentwicklung, sondern eine Fortentwicklung von JDO 1.0. Diese Technik hat einerseits einen längeren Weg hinter sich als das Paket `javax.persistence` aus der EJB 3.0 Spezifikation und hat andererseits schon seit drei Jahren bewiesen, was es in der Lage ist zu vollbringen. Dieses angesprochene Paket ist von einem Produktivstadium noch weit entfernt, muss seine Leistungsfähigkeit in der Praxis erst noch beweisen und ist einzig für RDBMS ausgelegt.

Wird JDO 2.0 in einen Vergleich mit Hibernate gesetzt, fällt das Ergebnis noch knapper aus. Zwar ist eine JDO Implementation in der Lage, die Zeit, die bei Hibernate durch Reflection verwendet wird, auf den Zeitpunkt des Kompilierens vorzulagern, doch sagt dies wenig über die eigentliche Performance der Routinen aus. Insofern ist es im höchsten Grade abhängig vom Hersteller, wie schnell oder langsam die Applikation abschneidet. Doch gerade dieses Feature bereitet manchen Entwicklern Kopfschmerzen, wenn so der Bytecode einer Klasse mit vermittelnden Methoden angereichert wird ohne deren Existenz im Quelltext nachvollziehen zu können; es wird als fremder Eingriff gewertet.

JDOQL ist an Java angelegt und HQL an SQL. Dem Programmierer ist es also unter JDOQL nicht in vollem Umfang möglich, eine Optimierung der Abfrage durchzuführen. Andererseits stellt sich auch die Frage, ob dies der Programmierer in einem Projekt will und muss, oder ob es in den Bereich der Optimierung des Frameworks fällt damit letztendlich dauerhaft von der Änderung profitiert werden kann. Für solche Fälle ist es in JDO 2.0 dem Programmierer möglich, seine Anfragen auch per SQL direkt zu stellen. Bei einem Punkt hat JDO hingegen klar einen Vorteil zu verzeichnen: bei der Unterstützung von unterschiedlichen Datenspeichern. Dieses Feature ist gerade im Bereich der langfristigen Investitionssicherheit von Belang. Hibernate orientiert sich am gegenwärtigen Markt und beschränkt das Framework auf relationale DBMS, während JDO überall speichern kann.

Insofern ist JDO als genau so ausgereift wie Hibernate (als bekanntestem Vertreter im Bereich Persistenzframeworks) zu bewerten, hat eine stärkere Rollenverteilung der Beteiligten in der Entwicklungsphase und eine breitere Basis an Datenspeichern auf die es zurückgreifen kann.

Eine sonderlich starke Verbreitung im Sinne von Medienpräsenz hat JDO bisher jedoch noch nicht erreicht: Literatur und Berichterstattung sind rar und auf Anfrage bei verschiedenen Herstellern gaben diese an, dass sie als Start-Up Unternehmen oder als Open-Source Projekt bisher Firmenkunden und professionelle Programmierer als ihre Zielgruppe definiert haben.

Zieht man dazu noch Mefferts Definition von Marktkommunikation hinzu:

„Die zentralen Merkmale der Kommunikation sind also: Übermittlung von Informationen und Bedeutungsinhalten zum Zweck der Steuerung von Meinungen, Einstellungen, Erwartungen und Verhaltensweisen gemäß spezifischen Zielsetzungen.“¹,

wird klar, dass hier eine sehr starke Fokussierung auf eben diese Gruppe stattfindet. Dies verbunden mit dem Fehlen eines klar erkennbaren „Champions“ hinter JDO in Form eines weltweit operierenden Unternehmens, erklärt die gesonderte Stellung JDOs unter den Persistenzframeworks.

¹ Meffert, H.: Marketing – Einführung in die Absatzpolitik, Wiesbaden, 1977, S. 412

6.2 Ergebnisse wirtschaftlich gesehen

6.2.1 Die Faktoren Personal, Kosten und Zeit

So unterschiedlich wie die Hersteller der Implementationen sind, so unterschiedlich sind auch deren Lizenzmodelle und rangieren von der Apache 2 Lizenz (JPOX) über Entwickler- und Runtime-Lizenzen (IntelliBO) hin zu reinen Entwicklerlizenzen ohne zusätzliche Kosten pro beim Kunden installierter Kopie (Kodo JDO).

Um zwei Beispiele aus dem kommerziellen Bereich herauszugreifen, werde ich an dieser Stelle die Produkte von Solarmetric und Signsoft mit ihren Listenpreisen vom 17.12.2005 näher beschreiben: Solarmetrics Kodo JDO teilt sich auf in die „Standard Edition“ (\$1.100,- pro Lizenz) und die „Enterprise Edition“ (\$4.000,- pro Lizenz). Wartungsverträge mit E-Mail Support und sämtlichen Neuerungen der Software (Bugfixes, Updates und Upgrades) schlagen mit \$500,- bzw. \$750,- pro Entwicklerlizenz zu Buche. Eine Schulung in der Länge von 2 – 3 Tagen plus einer Beratung in Länge von 2 – 3 Tagen kostet €11.000,- + Spesen.

Bei Signsofts IntelliBO kostet eine Entwicklerlizenz €2.450,- inkl. aller Updates und Upgrades für ein Jahr, während sich die Runtime-Lizenzen in „Server“ und „Local“ aufteilen. Server-Lizenzen sind an eine bestimmte Anzahl an Clients gebunden und reichen von €500,- (10 Clients) bis €1.500,- (unbegrenzte Anzahl an Clients). Local-Lizenzen rangieren zwischen €1.000,- (für 100 User) bis €5.000,- (1.000 User).

E-Mail Support für ein Jahr kostet €950,- pro Entwicklerlizenz.

Rechnet man diese Kosten gegen die reinen Wartungskosten eines eigenen Frameworks mit etwa zwei Tagen pro Monat, die ca. fünf Wochen pro Jahr ergeben, sind dies bei einem durchschnittlichen Jahresbruttoeinkommen für eine IT-Fachkraft von €46.700,-¹ ca. €5.425,- pro mit dem Framework betreuter Person, die der Arbeitgeber für diese Zeitspanne mindestens zu kalkulieren hat. Diese Summe setzt sich lediglich aus dem Bruttolohn plus den Lohnnebenkosten zusammen und berücksichtigt nicht, dass oft mehr als eine Person mit dieser Aufgabe betraut werden muss.

¹ Apfelbaum, D.; Becher, C.: Ergebnisse der c't-Gehaltsumfrage 2004
In: c't, 22. Jg., H. 6, 2005, S. 102ff.

Die indirekt betroffenen Sektoren in einem Unternehmen sind bei JDO zahlreich. Zwar ist es bereits möglich, die Zahlen direkt gegenüber zu stellen, aber man sollte die eher weichen Faktoren nicht vernachlässigen, die bei einer Wahl von JDO Bedeutung erlangen:

Z.B. ist zu beobachten, dass im Vergleich zu einer herkömmlichen Programmierung mit JDBC die Zeit der Kodierung der Applikation um durchschnittlich 25 – 30%¹ reduziert werden kann. Die gewonnene Zeit kann so für die Verbesserung der Qualität des Kundenprojekts oder für einen früheren Abgabetermin verwendet werden.

Durch die einfache Struktur JDOs und dadurch, dass das Personal produktiver wird, ist auch eine Neuallokation der Arbeitskräfte möglich: Es müssen sich nicht mehr die geübtesten und damit besser bezahlten Programmierer um die Wartung des Frameworks kümmern, da diese bei einer Neuentwicklung normalerweise auch die ausführenden Personen waren. Insofern wird an dieser Stelle in einem Unternehmen unnötig Know-how im Bereich der Wartung gebunden, welches in neuen Projekten besser aufgehoben wäre.

Im Bereich der Netzwerkinfrastruktur werden durch die verwendeten Caches und das Lazy Loading weniger Daten über die Leitungen gesendet, was auch zu einer Entlastung der Netzwerke führen kann (siehe Kapitel 3.3.2).

Eine Investition in JDO rechnet sich also schon nach höchstens einem Jahr – auch vor dem Hintergrund, dass zur Zeit eine Bewegung der Hersteller in Richtung Open-Source stattfindet: Die eigentlichen Implementationen werden durch Open-Source Gruppierungen entwickelt (so bereits geschehen bei bspw. TriActive JDO und Versant JDO) und die ehemaligen Hersteller konzentrieren sich auf die Vermarktung und den Support bei Firmenkunden.

¹ Von unterschiedlichen Unternehmensberatern der Hersteller als realistisch eingeschätzter Wert (Werbeversprechen gehen teilweise bis 70%, doch deren Wahrheitsgehalt darf für die Masse an möglichen Projekten als fragwürdig eingestuft werden)

6.2.2 Investitionssicherheit

Ist diese Investition jedoch sicher? Welche kurz- bis mittelfristigen Gefahren können auf das Unternehmen zukommen bei einer Wahl des Frameworks?

Zwei oft genannte Szenarien in diesem Zusammenhang, die die Benutzung eines Frameworks zum Scheitern verurteilen können, wären einmal die Einstellung der Entwicklung einer Implementation durch den betreffenden Hersteller und zum anderen Mal die Änderung des Datenspeichers durch eine Kundenentscheidung an einem späten Punkt des Projektes oder während der Wartungsphase.

Die Bytecodekompatibilität unter den Herstellern greift im ersten Szenario ein: Zwar werden die herstellerspezifischen Extensions nicht im weiteren Betrieb berücksichtigt, aber sofern diese nicht verwendet werden, kann ein direkter Umzug auf das Produkt eines anderen Herstellers stattfinden ohne Neukompilierung oder Änderung des Quelltextes. Weiterhin bieten zahlreiche Hersteller auch Tools an, um Metadaten zwischen JDO 1.0 und JDO 2.0 Standardtags, bzw. von verschiedenen Herstellern auf die eigenen Tags automatisch zu migrieren.

Gerade dieses Argument wird oft als negativ ausgelegt: Wenn JDO schon damit wirbt, dass dieser Fall berücksichtigt wird, wird damit nicht bereits der Teufel an die Wand gemalt? Drehen wir die Frage um: Was würde bei einer Aufspaltung Hibernates in zwei oder mehrere Projekte geschehen im Falle eines Streits? Würden die entstehenden Produkte untereinander im Bytecode kompatibel werden oder würden die Splittergruppen Insellösungen schaffen, um ihre Kunden möglichst eng an sich zu binden?

Das Prinzip „Write once, store anywhere.“ sorgt für die Kompatibilität zu den verschiedensten Datenspeichern im zweiten Szenario. Hat eine Datenbank eine JDBC-Anbindung, oder existiert ein Resource Adapter, kann der Wechsel vollzogen werden. Entscheidet sich ein Hersteller einer Datenbank diese Schnittstellen nicht bereitzustellen, ist eine Kompatibilität nicht gewährt. Dies sind dann Hersteller, die einen neuen Ansatz in der

Datenpersistenz verfolgen und kein Interesse an eine Anbindung an bestehende Standards haben.

Auf die aktuelle Situation mit JDBC als Standard unter der Anbindung an relationale Datenbanken dürfte dem Markt Genüge getan sein und die Anzahl an unterstützten Datenbanken kann nur zunehmen.

6.2.3 Risiken

Langfristige Risiken begründen sich in politischen Situationen oder in einer fehlenden Akzeptanz des Marktes.

Schon die angekündigte Zusammenlegung der Expertengruppen #220 (EJB 3.0) und #243 (JDO 2.0) und deren Abfragesprachen kündigt eine politisch motivierte Situation an, welche die Hersteller der JDO Persistenzframeworks aufgegriffen haben: Neben ihrer eigentlichen API für JDO 2.0 haben sie die Arbeiten an einer EJB 3.0 API für ihr Framework begonnen. Es ist also möglich, die API zu wechseln und beim gleichen Produkt zu bleiben, während JDOQL die verwendete Abfragesprache bleibt. Nur sollte sich jedem die Frage stellen, warum gerade Hersteller von Datenbanken und Middleware-Technologien versuchen JDO tot zu reden mit den Argumenten, dass das Framework zu leistungsfähig geworden sei und damit mit EJB 3.0 in Konkurrenz steht, was zu einer Verwirrung beim Kunden führen kann. Dies vor allem vor dem Hintergrund, dass EJB 3.0 von den Features mittlerweile eine Untermenge von JDO ist.

Letztendlich wird der Markt entscheiden, welches Framework welche Akzeptanz erfährt, doch noch sind nicht die Hersteller der Markt, sondern die Kunden.

6.2.4 Chancen

In einem Umfeld, in dem Kunden verstärkt mehr Qualität und schnellere Ergebnisse zu geringeren Preisen fordern, kommt einem jedes Hilfsmittel recht. Schafft dieses Hilfsmittel es dann auch noch, diverse Risiken innerhalb eines Projektes auszuschalten und Kosten zu senken, sollte eine Anschaffung in Betracht gezogen werden.

Ob diese so geschaffenen Zeitpuffer im gleichen Projekt, oder für andere Aufträge verwendet werden, liegt im Ermessen des Verwenders. Fakt jedoch ist, dass diese Puffer geschaffen wurden.

Mehr Zeit bedeutet mehr Raum für richtige Entscheidungen, ausführlichere Tests, oder einfach nur weniger Stress für das Team.

Auf jeden Fall sind mehr Zeit und weniger Kosten ein bedeutender Vorteil in der Anwendungsentwicklung.

7 Fazit

Das Feld der relationalen Datenbanken ist in der gegenwärtigen Geschäftswelt noch vorherrschend. Es gibt Schritte in Richtung Objektdatenbanken mit SQL3 und den objektrelationalen Erweiterungen, doch diese sind noch nicht die Regel geworden.

Was die Zukunft birgt, ist nicht mit Sicherheit vorauszusagen.

Wäre es also nicht vermessen zu prophezeien, dass die Zukunft aus der Technologie der Gegenwart bestehen wird? Oder einem Kunden einreden zu wollen, dass eine bereits bestehende Technik keine Marktakzeptanz erfahren wird, weil das eigene, neue Produkt besser sein wird (obwohl es noch nicht erprobt wurde und sich noch in der Alphaversion befindet)?

Solche Strategien erinnern stark an FUD-Kampagnen und Propaganda: Der Kunde wird durch das eigene Marketing so weit verunsichert, dass er mit seiner Investition zögert und auf die neue Technik wartet.

Dass bereits Firmen wie T-Systems, die Deutsche Bank, ThyssenKrupp Stahl oder Vodafone D2¹ das Framework Java Data Objects erfolgreich in ihren Projekten einsetzen, wird dabei gerne unter den Teppich gekehrt.

Sicherlich greifen die meisten der in dieser Diplomarbeit aufgezeigten Punkte für sämtliche Persistenzframeworks, doch einzig JDO hat einen Vorsprung in der Freiheit der Wahl der Datenspeicher kombiniert mit einer Java-ähnlichen Abfragesprache und der Unabhängigkeit vom Hersteller.

¹ Signsoft GmbH: Signsoft IntelliBO Referenzen
<http://www.intelliBO.com/de/intelliBO/references.jsp>, 17.12.2005

Anhang

A Lebenszyklen der Klassen innerhalb JDO¹

method / current state	Transient	P-new	P-clean	P-dirty	Hollow
makePersistent	P-new	unchanged	unchanged	unchanged	unchanged
deletePersistent	error	P-new-del	P-del	P-del	P-del
makeTransactional	T-clean	unchanged	unchanged	unchanged	P-clean
makeNontransactional	error	error	P-nontrans	error	unchanged
makeTransient	unchanged	error	Transient	error	Transient
commit retainValues = false	unchanged	Hollow	Hollow	Hollow	unchanged
commit retainValues = true	unchanged	P-nontrans	P-nontrans	P-nontrans	unchanged
rollback restoreValues = false	unchanged	Transient	Hollow	Hollow	unchanged
rollback restoreValues = true	unchanged	Transient	P-nontrans	P-nontrans	unchanged
refresh with active Datastore transaction	unchanged	unchanged	unchanged	P-clean	unchanged
refresh with active Optimistic transaction	unchanged	unchanged	unchanged	P-nontrans	unchanged
evict	n/a	unchanged	Hollow	unchanged	unchanged
read field outside transaction	unchanged	impossible	impossible	impossible	P-nontrans
read field with active Optimistic transaction	unchanged	unchanged	unchanged	unchanged	P-nontrans
read field with active Datastore transaction	unchanged	unchanged	unchanged	unchanged	P-clean
write field or makeDirty outside transaction	unchanged	impossible	impossible	impossible	P-nontrans
write field or makeDirty with active transaction	unchanged	unchanged	P-dirty	unchanged	P-dirty
retrieve outside or with active Optimistic transaction	unchanged	unchanged	unchanged	unchanged	P-nontrans
retrieve with active Datastore transaction	unchanged	unchanged	unchanged	unchanged	P-clean
close persistence manager with DetachOn-Close true	unchanged	impossible	impossible	impossible	detached

¹ Sun Microsystems, Inc.: Java Data Objects 2.0 Proposed Final Draft
<http://jcp.org/aboutJava/communityprocess/pfd/jsr243/index.html>, 10.08.2005, S. 58 – 60

method / current state	T-clean	T-dirty	P-new-del	P-del	P-nontrans
makePersistent	P-new	P-new	unchanged	unchanged	unchanged
deletePersistent	error	error	unchanged	unchanged	P-del
makeTransactional	unchanged	unchanged	unchanged	unchanged	P-clean
makeNontransactional	Transient	error	error	error	unchanged
makeTransient	unchanged	unchanged	error	error	Transient
commit retainValues = false	unchanged	T-clean	Transient	Transient	unchanged
commit retainValues = true	unchanged	T-clean	Transient	Transient	unchanged
rollback restoreValues = false	unchanged	T-clean	Transient	Hollow	unchanged
rollback restoreValues = true	unchanged	T-clean	Transient	P-nontrans	unchanged
refresh	unchanged	unchanged	unchanged	unchanged	unchanged
evict	unchanged	unchanged	unchanged	unchanged	Hollow
read field outside transaction	unchanged	impossible	impossible	impossible	unchanged
read field with Optimistic transaction	unchanged	unchanged	error	error	unchanged
read field with active Datastore transaction	unchanged	unchanged	error	error	P-clean
write field or makeDirty outside transaction	unchanged	impossible	impossible	impossible	unchanged
write field or makeDirty with active transaction	T-dirty	unchanged	error	error	P-dirty
retrieve outside or with active Optimistic transaction	unchanged	unchanged	unchanged	unchanged	unchanged
retrieve with active Datastore transaction	unchanged	unchanged	unchanged	unchanged	P-clean
close persistence manager with DetachOn-Close true	unchanged	unchanged	impossible	impossible	detached

B JDO 2.0 Metadaten DTD¹

```
<?xml version="1.0" encoding="UTF-8"?>

<!NOTATION JDO.2_0 PUBLIC "-//Sun Microsystems, Inc.//DTD Java
  Data Objects Metadata 2.0//EN">

<!-- The DOCTYPE should be as follows for metadata documents.
<!DOCTYPE jdo
  PUBLIC "-//Sun Microsystems, Inc.//DTD Java Data Objects
  Metadata 2.0//EN"
  "http://java.sun.com/dtd/jdo_2_0.dtd">
-->

<!ELEMENT jdo (extension*, (package|query)+, extension*)>
<!ATTLIST jdo catalog CDATA #IMPLIED>
<!ATTLIST jdo schema CDATA #IMPLIED>

<!ELEMENT package (extension*, (interface|class|sequence)+,
  extension*)>
<!ATTLIST package name CDATA ``>
<!ATTLIST package catalog CDATA #IMPLIED>
<!ATTLIST package schema CDATA #IMPLIED>

<!ELEMENT interface (extension*, datastore-identity?, primarykey?,
  inheritance?, version?, join*, foreign-key*,
  index*, unique*, property*, query*, fetch-
  group*, extension*)>
<!ATTLIST interface name CDATA #REQUIRED>
<!ATTLIST interface table CDATA #IMPLIED>
<!ATTLIST interface identity-type
  (datastore|application|nondurable) #IMPLIED>
<!ATTLIST interface objectid-class CDATA #IMPLIED>
<!ATTLIST interface requires-extent (true|false) 'true'>
<!ATTLIST interface detachable (true|false) 'false'>
<!ATTLIST interface catalog CDATA #IMPLIED>
<!ATTLIST interface schema CDATA #IMPLIED>

<!ELEMENT property (extension*, (array|collection|map)?, join?,
  embedded?, element?, key?, value?, order?,
  column*, foreign-key?, index?, unique?,
  extension)*>
<!ATTLIST property name CDATA #REQUIRED>
<!ATTLIST property default-fetch-group (true|false) #IMPLIED>
<!ATTLIST property load-fetch-group CDATA #IMPLIED>
<!ATTLIST property null-value (default|exception|none) 'none'>
<!ATTLIST property dependent (true|false) #IMPLIED>
<!ATTLIST property embedded (true|false) #IMPLIED>
<!ATTLIST property primary-key (true|false) 'false'>
<!ATTLIST property value-strategy CDATA #IMPLIED>
<!ATTLIST property sequence CDATA #IMPLIED>
<!ATTLIST property serialized (true|false) #IMPLIED>
<!ATTLIST property table CDATA #IMPLIED>
<!ATTLIST property column CDATA #IMPLIED>
<!ATTLIST property delete-action (restrict|cascade|null|default|
  none) #IMPLIED>
<!ATTLIST property indexed (true|false|unique) #IMPLIED>
<!ATTLIST property unique (true|false) #IMPLIED>
<!ATTLIST property mapped-by CDATA #IMPLIED>
```

¹ Sun Microsystems, Inc.: Java Data Objects 2.0 Proposed Final Draft
<http://jcp.org/aboutJava/communityprocess/pfd/jsr243/index.html>, 10.08.2005, S. 224 – 233

```

<!ATTLIST property fetch-group CDATA #IMPLIED>
<!ATTLIST property fetch-depth CDATA #IMPLIED>
<!ATTLIST property field-name CDATA #IMPLIED>

<!ELEMENT class (extension*, implements*, datastore-identity?,
  primary-key?, inheritance?, version?, join*,
  foreign-key*, index*, unique*, column*, field*,
  property*, query*, fetch-group*, extension*)>
<!ATTLIST class name CDATA #REQUIRED>
<!ATTLIST class identity-type (application|datastore|nondurable)
  #IMPLIED>
<!ATTLIST class objectid-class CDATA #IMPLIED>
<!ATTLIST class table CDATA #IMPLIED>
<!ATTLIST class requires-extent (true|false) 'true'>
<!ATTLIST class persistence-capable-superclass CDATA #IMPLIED>
<!ATTLIST class detachable (true|false) 'false'>
<!ATTLIST class embedded-only (true|false) #IMPLIED>
<!ATTLIST class persistence-modifier (persistence-
  capable|persistence-aware|non-persistent)
  #IMPLIED>
<!ATTLIST class catalog CDATA #IMPLIED>
<!ATTLIST class schema CDATA #IMPLIED>

<!ELEMENT primary-key (extension*, column*, extension*)>
<!ATTLIST primary-key name CDATA #IMPLIED>
<!ATTLIST primary-key column CDATA #IMPLIED>

<!ELEMENT join (extension*, primary-key?, column*, foreign-key?,
  index?, unique?, extension*)>
<!ATTLIST join table CDATA #IMPLIED>
<!ATTLIST join column CDATA #IMPLIED>
<!ATTLIST join outer (true|false) 'false'>
<!ATTLIST join delete-action (restrict|cascade|null|default|none)
  #IMPLIED>
<!ATTLIST join indexed (true|false|unique) #IMPLIED>
<!ATTLIST join unique (true|false) #IMPLIED>

<!ELEMENT version (extension*, column*, index?, extension*)>
<!ATTLIST version strategy CDATA #IMPLIED>
<!ATTLIST version column CDATA #IMPLIED>
<!ATTLIST version indexed (true|false|unique) #IMPLIED>

<!ELEMENT datastore-identity (extension*, column*, extension*)>
<!ATTLIST datastore-identity column CDATA #IMPLIED>
<!ATTLIST datastore-identity strategy CDATA 'native'>
<!ATTLIST datastore-identity sequence CDATA #IMPLIED>

<!ELEMENT implements (extension*, property*, extension*)>
<!ATTLIST implements name CDATA #REQUIRED>

<!ELEMENT inheritance (extension*, join?, discriminator?,
  extension*)>
<!ATTLIST inheritance strategy CDATA #IMPLIED>

<!ELEMENT discriminator (extension*, column*, index?, extension*)>
<!ATTLIST discriminator column CDATA #IMPLIED>
<!ATTLIST discriminator value CDATA #IMPLIED>
<!ATTLIST discriminator strategy CDATA #IMPLIED>
<!ATTLIST discriminator indexed (true|false|unique) #IMPLIED>

<!ELEMENT column (extension*)>
<!ATTLIST column name CDATA #IMPLIED>
<!ATTLIST column target CDATA #IMPLIED>
<!ATTLIST column target-field CDATA #IMPLIED>

```

```

<!ATTLIST column jdbc-type CDATA #IMPLIED>
<!ATTLIST column sql-type CDATA #IMPLIED>
<!ATTLIST column length CDATA #IMPLIED>
<!ATTLIST column scale CDATA #IMPLIED>
<!ATTLIST column allows-null CDATA #IMPLIED>
<!ATTLIST column default-value CDATA #IMPLIED>
<!ATTLIST column insert-value CDATA #IMPLIED>

<!ELEMENT field (extension*, (array|collection|map)?, join?,
embedded?, element?, key?, value?, order?,
column*, foreign-key?, index?, unique?,
extension*)>
<!ATTLIST field name CDATA #REQUIRED>
<!ATTLIST field persistence-modifier
(persistent|transactional|none) #IMPLIED>
<!ATTLIST field table CDATA #IMPLIED>
<!ATTLIST field column CDATA #IMPLIED>
<!ATTLIST field primary-key (true|false) `false`>
<!ATTLIST field null-value (exception|default|none) `none`>
<!ATTLIST field default-fetch-group (true|false) #IMPLIED>
<!ATTLIST field embedded (true|false) #IMPLIED>
<!ATTLIST field serialized (true|false) #IMPLIED>
<!ATTLIST field dependent (true|false) #IMPLIED>
<!ATTLIST field value-strategy CDATA #IMPLIED>
<!ATTLIST field delete-action (restrict|cascade|null|default|none)
#IMPLIED>
<!ATTLIST field indexed (true|false|unique) #IMPLIED>
<!ATTLIST field unique (true|false) #IMPLIED>
<!ATTLIST field sequence CDATA #IMPLIED>
<!ATTLIST field foreign-key CDATA #IMPLIED>
<!ATTLIST field load-fetch-group CDATA #IMPLIED>
<!ATTLIST field fetch-depth CDATA #IMPLIED>
<!ATTLIST field mapped-by CDATA #IMPLIED>

<!ELEMENT foreign-key (extension*, (column|field|property)*,
extension*)>
<!ATTLIST foreign-key table CDATA #IMPLIED>
<!ATTLIST foreign-key deferred (true|false) #IMPLIED>
<!ATTLIST foreign-key delete-action
(cascade|restrict|null|default) `restrict`>
<!ATTLIST foreign-key update-action
(cascade|restrict|null|default) `restrict`>
<!ATTLIST foreign-key unique (true|false) #IMPLIED>
<!ATTLIST foreign-key name CDATA #IMPLIED>

<!ELEMENT collection (extension*)>
<!ATTLIST collection element-type CDATA #IMPLIED>
<!ATTLIST collection embedded-element (true|false) #IMPLIED>
<!ATTLIST collection dependent-element (true|false) #IMPLIED>

<!ELEMENT map (extension)*>
<!ATTLIST map key-type CDATA #IMPLIED>
<!ATTLIST map embedded-key (true|false) #IMPLIED>
<!ATTLIST map dependent-key (true|false) #IMPLIED>
<!ATTLIST map value-type CDATA #IMPLIED>
<!ATTLIST map embedded-value (true|false) #IMPLIED>
<!ATTLIST map dependent-value (true|false) #IMPLIED>

<!ELEMENT key (extension*, embedded?, column*, foreign-key?,
index?, unique?, extension*)>
<!ATTLIST key column CDATA #IMPLIED>
<!ATTLIST key table CDATA #IMPLIED>
<!ATTLIST key serialized (true|false) #IMPLIED>

```

```

<!ATTLIST key delete-action (restrict|cascade|null|default|none)
                        #IMPLIED>
<!ATTLIST key indexed (true|false|unique) #IMPLIED>
<!ATTLIST key unique (true|false) #IMPLIED>
<!ATTLIST key mapped-by CDATA #IMPLIED>

<!ELEMENT value (extension*, embedded?, column*, foreign-key?,
                index?, unique?, extension*)>
<!ATTLIST value serialized (true|false) #IMPLIED>
<!ATTLIST value table CDATA #IMPLIED>
<!ATTLIST value column CDATA #IMPLIED>
<!ATTLIST value delete-action (restrict|cascade|null|default|none)
                        #IMPLIED>
<!ATTLIST value indexed (true|false|unique) #IMPLIED>
<!ATTLIST value unique (true|false) #IMPLIED>
<!ATTLIST value mapped-by CDATA #IMPLIED>

<!ELEMENT array (extension*)>
<!ATTLIST array embedded-element (true|false) #IMPLIED>
<!ATTLIST array dependent-element (true|false) #IMPLIED>

<!ELEMENT element (extension*, embedded?, column*, foreign-key?,
                  index?, unique?, extension*)>
<!ATTLIST element column CDATA #IMPLIED>
<!ATTLIST element table CDATA #IMPLIED>
<!ATTLIST element serialized (true|false) #IMPLIED>
<!ATTLIST element delete-action
        (restrict|cascade|null|default|none) #IMPLIED>
<!ATTLIST element update-action CDATA #IMPLIED>
<!ATTLIST element indexed (true|false|unique) #IMPLIED>
<!ATTLIST element unique (true|false) #IMPLIED>

<!ELEMENT order (extension*, column*, index?, extension*)>
<!ATTLIST order column CDATA #IMPLIED>
<!ATTLIST order indexed (true|false|unique) #IMPLIED>
<!ATTLIST order mapped-by CDATA #IMPLIED>

<!ELEMENT fetch-group (fetch-group|field)*>
<!ATTLIST fetch-group name CDATA #REQUIRED>
<!ATTLIST fetch-group post-load (true|false) #IMPLIED>

<!ELEMENT embedded (extension*, field*, extension*)>
<!ATTLIST embedded owner-field CDATA #IMPLIED>
<!ATTLIST embedded null-indicator-column CDATA #IMPLIED>
<!ATTLIST embedded null-indicator-value CDATA #IMPLIED>

<!ELEMENT sequence (extension*)>
<!ATTLIST sequence name CDATA #REQUIRED>
<!ATTLIST sequence datastore-sequence CDATA #IMPLIED>
<!ATTLIST sequence factory-class CDATA #IMPLIED>
<!ATTLIST sequence strategy
        (nontransactional|contiguous|noncontiguous) #REQUIRED>

<!ELEMENT index (extension*, (column|field|property)*,
                extension*)>
<!ATTLIST index name CDATA #IMPLIED>
<!ATTLIST index table CDATA #IMPLIED>
<!ATTLIST index unique (true|false) 'false'>

<!ELEMENT query (#PCDATA|extension)*>
<!ATTLIST query name CDATA #IMPLIED>
<!ATTLIST query language CDATA #IMPLIED>
<!ATTLIST query unmodifiable (true|false) 'false'>
<!ATTLIST query unique (true|false) 'false'>

```

```
<!ATTLIST query result-class CDATA #IMPLIED>

<!ELEMENT unique (extension*, (column|field|property)*,
                  extension*)>
<!ATTLIST unique name CDATA #IMPLIED>
<!ATTLIST unique table CDATA #IMPLIED>
<!ATTLIST unique deferred (true|false) 'false'>

<!ELEMENT extension ANY>
<!ATTLIST extension vendor-name CDATA #REQUIRED>
<!ATTLIST extension key CDATA #IMPLIED>
<!ATTLIST extension value CDATA #IMPLIED>
```

```

<?xml version="1.0" encoding="UTF-8"?>

<!NOTATION ORM.2_0 PUBLIC "-//Sun Microsystems, Inc.//DTD Java
  Data Objects Mapping Metadata 2.0//EN">

<!-- The DOCTYPE should be as follows for metadata documents.
<!DOCTYPE orm
  PUBLIC "-//Sun Microsystems, Inc.//DTD Java Data Objects
  Mapping Metadata 2.0//EN"
  "http://java.sun.com/dtd/orm_2_0.dtd">
-->

<!ELEMENT orm (extension*, (package|query)+, extension*)>
<!ATTLIST orm catalog CDATA #IMPLIED>
<!ATTLIST orm schema CDATA #IMPLIED>

<!ELEMENT package (extension*, (interface|class|sequence)+,
  extension*)>
<!ATTLIST package name CDATA ``>
<!ATTLIST package catalog CDATA #IMPLIED>
<!ATTLIST package schema CDATA #IMPLIED>

<!ELEMENT interface (extension*, datastore-identity?, primaryKey?,
  inheritance?, version?, join*, foreign-key*,
  index*, unique*, property*, query*,
  extension*)>
<!ATTLIST interface name CDATA #REQUIRED>
<!ATTLIST interface table #CDATA #IMPLIED>
<!ATTLIST interface catalog CDATA #IMPLIED>
<!ATTLIST interface schema CDATA #IMPLIED>

<!ELEMENT property (join?, element?, key?, value?, order?,
  column)?, extension*)>
<!ATTLIST property name CDATA #REQUIRED>
<!ATTLIST property column CDATA #IMPLIED>

<!ELEMENT class (extension*, datastore-identity?, primary-key?,
  inheritance?, version?, join*, foreign-key*,
  index*, unique*, column*, field*, query*,
  extension*)>
<!ATTLIST class name CDATA #REQUIRED>
<!ATTLIST class table CDATA #IMPLIED>
<!ATTLIST class catalog CDATA #IMPLIED>
<!ATTLIST class schema CDATA #IMPLIED>

<!ELEMENT primary-key (extension*, column*, extension*)>
<!ATTLIST primary-key name CDATA #IMPLIED>
<!ATTLIST primary-key column CDATA #IMPLIED>

<!ELEMENT join (extension*, primary-key?, column*, foreign-key?,
  index?, unique?, extension*)>
<!ATTLIST join table CDATA #IMPLIED>
<!ATTLIST join column CDATA #IMPLIED>
<!ATTLIST join outer (true|false) `false`>
<!ATTLIST join delete-action (restrict|cascade|null|default|none)
  #IMPLIED>
<!ATTLIST join indexed (true|false|unique) #IMPLIED>
<!ATTLIST join unique (true|false) #IMPLIED>

<!ELEMENT datastore-identity (extension*, column*, extension*)>
<!ATTLIST datastore-identity column CDATA #IMPLIED>
<!ATTLIST datastore-identity strategy CDATA `native`>
<!ATTLIST datastore-identity sequence CDATA #IMPLIED>

```

```

<!ELEMENT version (extension*, column*, index?, extension*)>
<!ATTLIST version strategy CDATA #REQUIRED>
<!ATTLIST version column CDATA #IMPLIED>
<!ATTLIST version indexed (true|false|unique) #IMPLIED>

<!ELEMENT implements ((property-field)+, (extension)*)>
<!ATTLIST implements name CDATA #REQUIRED>

<!ELEMENT inheritance (extension*, discriminator?, join?,
                        extension*)>
<!ATTLIST inheritance strategy CDATA #IMPLIED>

<!ELEMENT discriminator (extension*, column*, index?, extension*)>
<!ATTLIST discriminator column CDATA #IMPLIED>
<!ATTLIST discriminator value CDATA #IMPLIED>
<!ATTLIST discriminator strategy CDATA #IMPLIED>
<!ATTLIST discriminator indexed (true|false|unique) #IMPLIED>

<!ELEMENT column (extension*)>
<!ATTLIST column name CDATA #IMPLIED>
<!ATTLIST column target CDATA #IMPLIED>
<!ATTLIST column target-field CDATA #IMPLIED>
<!ATTLIST column jdbc-type CDATA #IMPLIED>
<!ATTLIST column sql-type CDATA #IMPLIED>
<!ATTLIST column length CDATA #IMPLIED>
<!ATTLIST column scale CDATA #IMPLIED>
<!ATTLIST column allows-null CDATA #IMPLIED>
<!ATTLIST column default-value CDATA #IMPLIED>
<!ATTLIST column insert-value CDATA #IMPLIED>

<!ELEMENT property (extension*, join?, embedded?, element?, key?,
                    value?, order?, column*, foreign-key?, index?,
                    unique?, extension*)>
<!ATTLIST property name CDATA #REQUIRED>
<!ATTLIST property value-strategy CDATA #IMPLIED>
<!ATTLIST property sequence CDATA #IMPLIED>
<!ATTLIST property serialized (true|false) #IMPLIED>
<!ATTLIST property table CDATA #IMPLIED>
<!ATTLIST property column CDATA #IMPLIED>
<!ATTLIST property delete-action
        (restrict|cascade|null|default|none) #IMPLIED>
<!ATTLIST property indexed (true|false|unique) #IMPLIED>
<!ATTLIST property unique (true|false) #IMPLIED>
<!ATTLIST property mapped-by CDATA #IMPLIED>

<!ELEMENT field (extension*, join?, embedded?, element?, key?,
                value?, order?, column*, foreign-key?, index?,
                unique?, extension*)>
<!ATTLIST field name CDATA #REQUIRED>
<!ATTLIST field column CDATA #IMPLIED>
<!ATTLIST field primary-key CDATA #IMPLIED>
<!ATTLIST field table CDATA #IMPLIED>
<!ATTLIST field delete-action (restrict|cascade|null|default|none)
        #IMPLIED>
<!ATTLIST field indexed (true|false|unique) #IMPLIED>
<!ATTLIST field unique (true|false) #IMPLIED>
<!ATTLIST field mapped-by CDATA #IMPLIED>
<!ATTLIST field value-strategy CDATA #IMPLIED>
<!ATTLIST field sequence CDATA #IMPLIED>

<!ELEMENT foreign-key (extension*, (column|field|property)*,
                       extension*)>
<!ATTLIST foreign-key table CDATA #IMPLIED>

```

```

<!ATTLIST foreign-key deferred (true|false) #IMPLIED>
<!ATTLIST foreign-key delete-action
    (restrict|cascade|null|default) 'restrict'>
<!ATTLIST foreign-key update-action
    (restrict|cascade|null|default) 'restrict'>
<!ATTLIST foreign-key unique (true|false) #IMPLIED>
<!ATTLIST foreign-key name CDATA #IMPLIED>

<!ELEMENT key (column*, index?, embedded?, foreign-key?,
    extension*)>
<!ATTLIST key column CDATA #IMPLIED>
<!ATTLIST key table CDATA #IMPLIED>
<!ATTLIST key serialized (true|false) #IMPLIED>
<!ATTLIST key delete-action (restrict|cascade|null|default|none)
    #IMPLIED>
<!ATTLIST key indexed (true|false|unique) #IMPLIED>
<!ATTLIST key unique (true|false) #IMPLIED>
<!ATTLIST key mapped-by CDATA #IMPLIED>

<!ELEMENT value (extension*, embedded?, column*, foreign-key?,
    index?, unique?, extension*)>
<!ATTLIST value column CDATA #IMPLIED>
<!ATTLIST value table CDATA #IMPLIED>
<!ATTLIST value serialized (true|false) #IMPLIED>
<!ATTLIST value delete-action (restrict|cascade|null|default|none)
    #IMPLIED>
<!ATTLIST value indexed (true|false|unique) #IMPLIED>
<!ATTLIST value unique (true|false) #IMPLIED>
<!ATTLIST value mapped-by CDATA #IMPLIED>

<!ELEMENT element (extension*, embedded?, column*, foreign-key?,
    index?, unique?, extension*)>
<!ATTLIST element column CDATA #IMPLIED>
<!ATTLIST element table CDATA #IMPLIED>
<!ATTLIST element serialized (true|false) #IMPLIED>
<!ATTLIST element delete-action
    (restrict|cascade|null|default|none) #IMPLIED>
<!ATTLIST element indexed (true|false|unique) #IMPLIED>
<!ATTLIST element unique (true|false) #IMPLIED>

<!ELEMENT order (extension*, column*, index?, extension*)>
<!ATTLIST order column CDATA #IMPLIED>
<!ATTLIST order indexed (true|false|unique) #IMPLIED>
<!ATTLIST order mapped-by CDATA #IMPLIED>

<!ELEMENT embedded (extension*, field*, extension*)>
<!ATTLIST embedded null-indicator-column CDATA #IMPLIED>
<!ATTLIST embedded null-indicator-value CDATA #IMPLIED>
<!ATTLIST embedded owner-field CDATA #IMPLIED>

<!ELEMENT sequence (extension*)>
<!ATTLIST sequence name CDATA #REQUIRED>
<!ATTLIST sequence datastore-sequence CDATA #IMPLIED>
<!ATTLIST sequence factory-class CDATA #IMPLIED>
<!ATTLIST sequence strategy
    (nontransactional|contiguous|noncontiguous) #REQUIRED>

<!ELEMENT index (extension*, (column|field|property)*,
    extension*)>
<!ATTLIST index name CDATA #IMPLIED>
<!ATTLIST index table CDATA #IMPLIED>
<!ATTLIST index unique (true|false) 'false'>

```

```
<!ELEMENT unique (extension*, (column|field|property)*,
                    extension*)>
<!ATTLIST unique name CDATA #IMPLIED>
<!ATTLIST unique table CDATA #IMPLIED>
<!ATTLIST unique deferred (true|false) 'false'>

<!ELEMENT query (#PCDATA|extension)*>
<!ATTLIST query name CDATA #IMPLIED>
<!ATTLIST query language CDATA #IMPLIED>
<!ATTLIST query unmodifiable (true|false) 'false'>
<!ATTLIST query unique (true|false) 'false'>
<!ATTLIST query result-class CDATA #IMPLIED>

<!ELEMENT extension ANY>
<!ATTLIST extension vendor-name CDATA #REQUIRED>
<!ATTLIST extension key CDATA #IMPLIED>
<!ATTLIST extension value CDATA #IMPLIED>
```

C JDOQL Backus-Naur-Form¹

Grammar Notation

The grammar notation is taken from the Java Language Specification, section 2.4 Grammar Notation.

- Terminal symbols are shown in bold fixed width font in the productions of the lexical and syntactic grammars, and throughout this specification whenever the text is directly referring to such a terminal symbol. These are to appear in a program exactly as written.
- Nonterminal symbols are shown in italic type. The definition of a nonterminal is introduced by the name of the nonterminal being defined followed by a colon. One or more alternative right-hand sides for the nonterminal then follow on succeeding lines.
- The subscripted suffix "opt", which may appear after a terminal or nonterminal, indicates an optional symbol. The alternative containing the optional symbol actually specifies two right-hand sides, one that omits the optional element and one that includes it.
- When the words "one of" follow the colon in a grammar definition, they signify that each of the terminal symbols on the following line or lines is an alternative definition.

Single-String JDOQL

This section describes the syntax of single-string JDOQL.

SingleStringJDOQL:

Select Fromopt Whereopt Decls Groupingopt Orderingopt Rangeopt

Select:

select unique*opt ResultClauseopt IntoClauseopt*

IntoClause:

into *ResultClassName*

From:

from *CandidateClassName ExcludeClauseopt*

ExcludeClause:

exclude subclasses

Where:

where *Expression*

Decls:

Variablesopt Parametersopt Importsopt

Variables:

variables *VariableList*

Parameters:

parameters *ParameterList*

Imports:

imports *ImportList*

¹ Sun Microsystems, Inc.: Java Data Objects 2.0 Proposed Final Draft
<http://jcp.org/aboutJava/communityprocess/pfd/jsr243/index.html>, 10.08.2005, S. 278 – 284

Grouping:

group by *GroupingClause*

Ordering:

order by *OrderingClause*

Range:

range *Expression to Expression*

Filter Specification

This section describes the syntax of the `setFilter` argument.

Basically, the query filter expression is a Java boolean expression, where some of the Java operators are not permitted. Specifically, pre- and post-increment and decrement (`++` and `--`), shift (`>>` and `<<`) and assignment expressions (`+=`, `-=`, etc.) are not permitted.

The Nonterminal *InfixOp* lists the valid operators for binary expressions in decreasing precedence. Operators on the same line have the same precedence. As in Java operators require operands of appropriate types. See the Java Language Specification for more information.

Please note, the grammar allows arbitrary method calls (see *MethodInvocation*), where JDO only permits the following methods:

Collection methods	<code>contains(Object)</code> , <code>isEmpty()</code>
Map methods	<code>containsKey(Object)</code> , <code>containsValue(Object)</code> , <code>isEmpty()</code> , <code>get(Object)</code>
String methods	<code>startsWith(String)</code> , <code>endsWith(String)</code> , <code>matches(String)</code> , <code>toLowerCase()</code> , <code>toUpperCase()</code> , <code>indexOf(String)</code> , <code>indexOf(String,</code> <code>int)</code> , <code>substring(int)</code> , <code>substring(int, int)</code>
Math methods	<code>Math.abs(numeric)</code> , <code>Math.sqrt(numeric)</code>
JDOHelper methods	<code>getObjectId(Object)</code>

Expression:

UnaryExpression

Expression InfixOp UnaryExpression

InfixOp: one of

*** / %**

+ -

> >= < <= instanceof

== !=

&

|

&&

||

UnaryExpression:

```

    PrefixOp UnaryExpression
    ( Type ) UnaryExpression
    Primary

```

PrefixOp: one of

```

    + - ~ !

```

Primary:

```

    Literal
    VariableName
    ParameterName
    this
    FieldAccess
    MethodInvocation
    ClassOrInterfaceName
    ( Expression )
    AggregateExpression1

```

FieldAccess:

```

    FieldName
    Primary . FieldName

```

MethodInvocation:

```

    Primary . MethodName ( ArgumentListopt )

```

ArgumentList:

```

    Expression
    ArgumentList , Expression

```

AggregateExpression:

```

    AggregateOp ( Expression )

```

AggregateOp: one of

```

    count sum min max avg

```

Parameter Declaration

This section describes the syntax of the `declareParameters` argument.

ParameterList:

```

    Parameters ,opt

```

ParameterDecls:

```

    ParameterDecl
    ParameterDecls , ParameterDecl

```

ParameterDecl:

```

    Type ParameterName

```

Please note, as a usability feature *ParameterList* supports an optional trailing comma (in addition to what the Java syntax allows in a parameter declaration).

¹ Please note, an *AggregateExpression* is only allowed as part of a result specification or a having specification.

Variable Declaration

This section describes the syntax of the `declareVariables` argument.

```
VariableList:  
    VariableDecls ; opt  
VariableDecls:  
    VariableDecl  
    VariableDecls ; VariableDecl  
VariableDecl:  
    Type ParameterName
```

Please note, as a usability feature *VariableList* defines the trailing semicolon as optional (in addition to what the Java syntax allows in a variable declaration).

Import Declaration

This section describes the syntax of the `declareImports` argument.

```
ImportList:  
    ImportDecls ; opt  
ImportDecls:  
    ImportDecl  
    ImportDecls ; ImportDecl  
ImportDecl:  
    import QualifiedIdentifier  
    import QualifiedIdentifier . *
```

Please note, as a usability feature *ImportList* defines the trailing semicolon as optional (in addition to what the Java syntax allows in an import statement).

Ordering Specification

This section describes the syntax of the `setOrdering` argument.

```
OrderingClause:  
    OrderingSpecifications , opt  
OrderingSpecs:  
    OrderingSpec  
    OrderingSpecs , OrderingSpec  
OrderingSpec:  
    Expression Ascending  
    Expression Descending  
Ascending: one of  
    asc ascending  
Descending: one of  
    desc descending
```

Please note, as a usability feature *OrderingClause* supports an optional trailing comma.

Result Specification

This section describes the syntax of the `setResult` argument.

ResultClause:

distinct_{opt} *ResultSpecifications* ,_{opt}

ResultSpecs:

ResultSpec

ResultSpecs , *ResultSpec*

ResultSpec:

Expression *ResultNaming*_{opt}

ResultNaming:

as *Identifier*

Please note, a result specification expression may be an aggregate expression. As a usability feature *ResultClause* supports an optional trailing comma.

Grouping Specification

This section describes the syntax of the `setGrouping` argument.

GroupingClause:

GroupingSpecs ,_{opt} *HavingSpec*_{opt}

GroupingSpecs:

Expression

GroupingSpecs , *Expression*

HavingSpec:

having *Expression*

Please note, a having specification expression may include an aggregate expression. As a usability feature *GroupingClause* supports an optional trailing comma.

Types

This section describes a type specification, used in a parameter or variable declaration or in a cast expression.

Type

PrimitiveType

ClassOrInterfaceName

PrimitiveType:

NumericType

boolean

NumericType:

IntegralType

FloatingPointType

IntegralType: one of

byte short int long char

FloatingPointType: one of

float double

Literals

A literal is the source code representation of a value of a primitive type, or the String type. Please refer to the Java

Language Specification for the lexical structure of Integer-, Floating Point-, and String-Literals.

JDOQL allows

String-Literals being enclosed in either single quotes or double quotes.

Literal:

IntegerLiteral

FloatingPointLiteral

BooleanLiteral

StringLiteral

NullLiteral

IntegerLiteral: ...

FloatingPointLiteral: ...

BooleanLiteral: one of

true false

StringLiteral: ...

NullLiteral:

null

Names

A name is a possibly qualified identifier. Please refer to the Java Language Specification for the lexical structure of

identifiers.

QualifiedIdentifier:

Identifier

QualifiedIdentifier . Identifier

CandidateClassName:

QualifiedIdentifier

ResultClassName:

QualifiedIdentifier

ClassOrInterfaceName:

QualifiedIdentifier

VariableName:

Identifier

ParameterName:

Identifier

ColonPrefixedIdentifier

FieldName:

Identifier

MethodName:

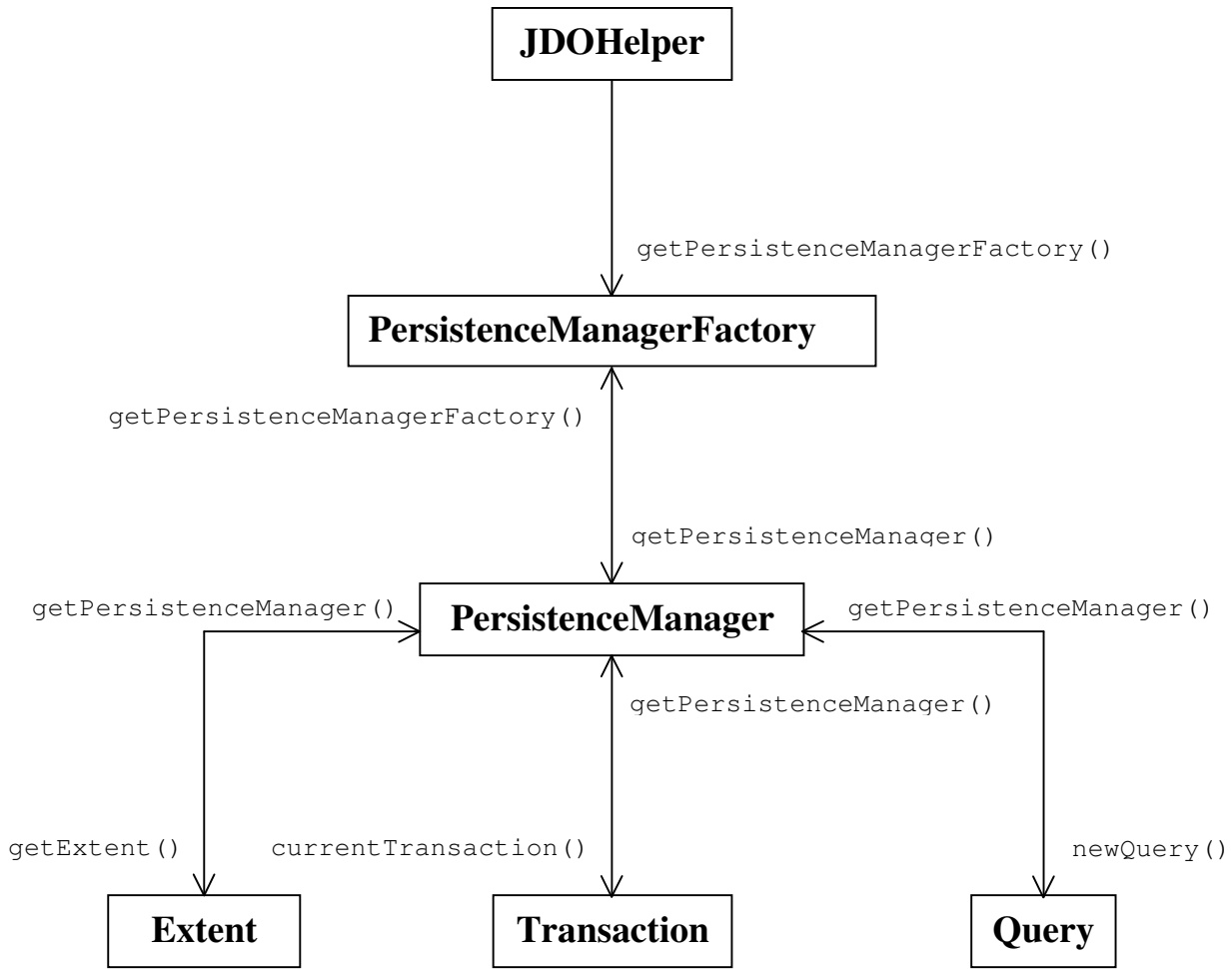
Identifier

Keywords

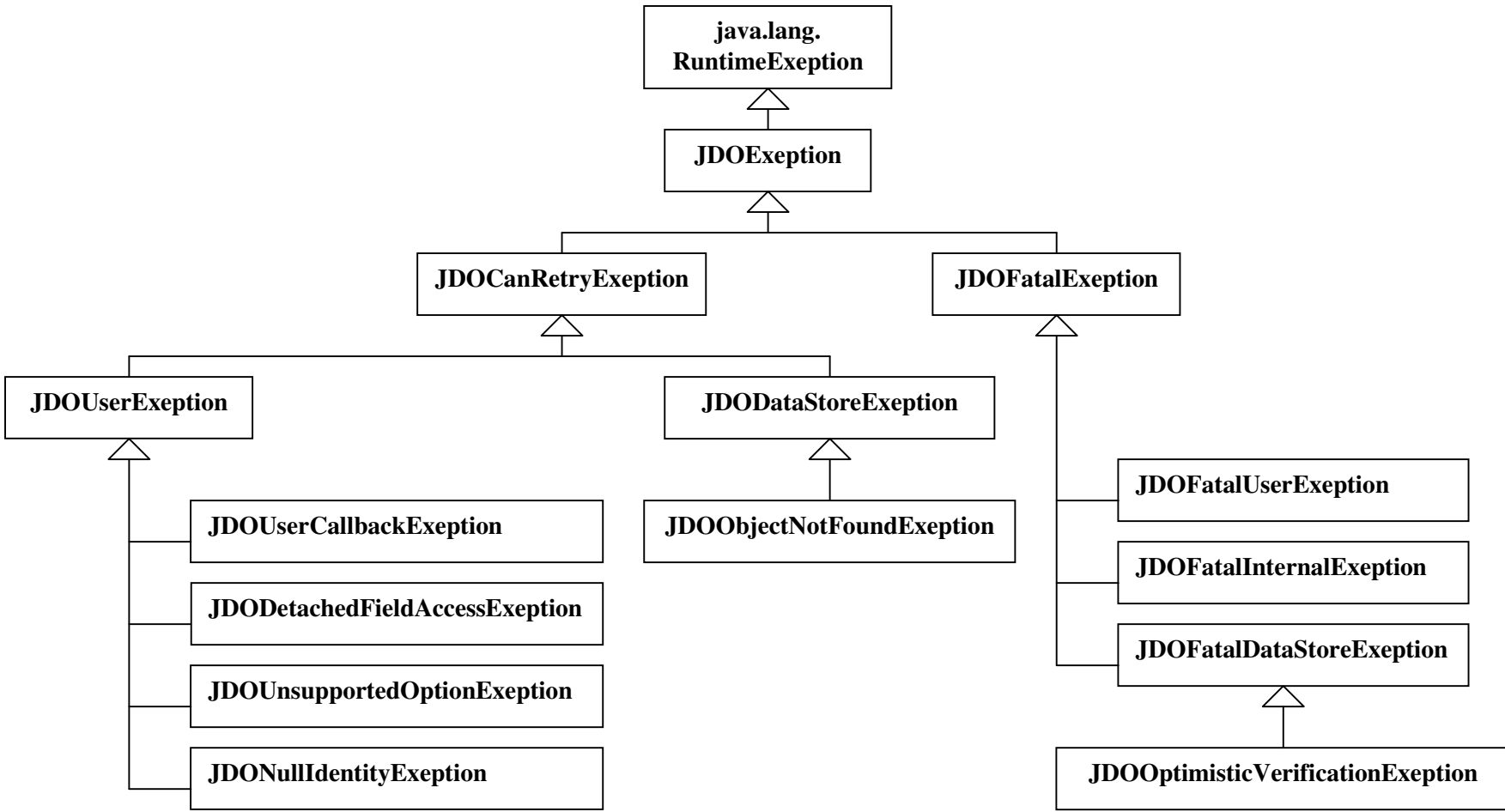
Keywords must not be used as package names, class names, parameter names, or variable names in queries. Keywords are permitted as field names only if they are on the right side of the “.” in field access expressions as defined in the Java Language Specification second edition, section 15.11. Keywords include the Java language keywords and the JDOQL keywords. Java keywords are as defined in the Java language specification section 3.9, plus the boolean literals true and false, and the null literal. JDOQL keywords maybe written in all lower case or all upper case.

JDOQLKeyword: one of

as	AS	asc	ASC
ascending	ASCENDING	avg	AVG
by	BY	count	COUNT
desc	DESC	descending	DESCENDING
distinct	DISTINCT	exclude	EXCLUDE
from	FROM	group	GROUP
having	HAVING	imports	IMPORTS
into	INTO	max	MAX
min	MIN	order	ORDER
parameters	PARAMETERS	range	RANGE
select	SELECT	subclasses	SUBCLASSES
sum	SUM	to	TO
unique	UNIQUE	variables	VARIABLES
where	WHERE		



¹ Jordan, D.; Russell, C.: Java Data Objects, Sebastopol 2003, S. 30



Literaturverzeichnis

I. BÜCHER

- Abts, D.; Mülder, W.: Grundkurs Wirtschaftsinformatik, 3. Aufl., Braunschweig, Wiesbaden 2001
- Jordan, D.; Russell, C.: Java Data Objects, Sebastopol 2003
- Meffert, H.: Marketing – Einführung in die Absatzpolitik, Wiesbaden, 1977

II. WISSENSCHAFTLICHE ABHANDLUNGEN

- Van Echelpoel, K.: Java Data Objects: A Revolution in Java Databanking?, Master Thesis, Antwerpen 2002

III. ZEITSCHRIFTEN

- Apfelbaum, D.; Becher, C.: Ergebnisse der c't-Gehaltsumfrage 2004.
In: c't, 22. Jg., H. 6, 2005, S. 102ff.
- Barry, D.; Stanienda, T.: Solving the Java Object Storage Problem.
In: Computer, 31. Jg., H. 11, 1998, S. 33 – 40
- Merkle, B.: EJB 3.0 Public Draft: Zurück zum Überschaubaren.
In: iX, 17. Jg., H. 10, 2005, S. 140 – 145

IV. DOKUMENTE AUS DEM INTERNET

- DeMichiel, L.; Russell, C.: A Letter to the Java Technology Community
<http://java.sun.com/j2ee/letter/persistence.html>, 30.09.2004
- Jefferson, A.; Bengtson E.: Java Persistent Objects JDO – JPOX History
<http://www.jpox.org/docs/history.html>, 15.12.2005
- King, G.: Announcing Hibernate 1.0 Open Source O/R Persistence Tool
http://www.theserverside.com/news/thread.tss?thread_id=14314, 06.07.2002
- King, G.: EJB3
<http://blog.hibernate.org/cgi-bin/blosxom.cgi/2004/05/07#ejb3>, 07.05.2004
- Object Data Management Group: ODMG Home Page
<http://www.odmg.org/>, 15.12.2005

- Ohne Verfasser:** PolePosition Benchmark Results
<http://polepos.sourceforge.net/results/PolePosition.pdf>, 20.12.2005
- Russell, C. et al.:** FrontPage – Jdo Wiki
<http://wiki.apache.org/jdo/>, 10.11.2005
- Signsoft GmbH:** Signsoft IntelliBO Referenzen
<http://www.intellibo.com/de/intellibo/references.jsp>, 17.12.2005
- Sun Microsystems, Inc.:** JavaBeans FAQ: General Questions
<http://java.sun.com/products/javabeans/faq/faq.general.html>, 15.12.2005
- Sun Microsystems, Inc.:** Java Data Objects 2.0 Proposed Final Draft
<http://jcp.org/aboutJava/communityprocess/pfd/jsr243/index.html>, 10.08.2005
- White, A.:** JDO FUD
http://www.theserverside.com/news/thread.tss?thread_id=25804, 08.05.2004
- Wikipedia:** Hierarchisches Datenbankmodell – Wikipedia
http://de.wikipedia.org/wiki/Hierarchische_Datenbank, 25.10.2005
- Wikipedia:** Netzwerkdatenbankmodell – Wikipedia
<http://de.wikipedia.org/wiki/Netzwerkdatenbankmodell>, 06.06.2005

V. E-MAILS

- Russell, C.:** Re: inquiry about JDO / EJB for my dissertation
E-Mail: craig.russell@sun.com, 06.12.2005

Versicherung

Ich versichere, dass ich die vorstehende Arbeit selbständig angefertigt und mich fremder Hilfe nicht bedient habe.

Alle Stellen, die wörtlich oder sinngemäß veröffentlichtem oder nicht veröffentlichtem Schrifttum entnommen sind, habe ich als solche kenntlich gemacht.

Datum, Ort

Unterschrift