Johannes Kepler Universität Linz

Objektrelationales Mapping mit Java 5 im Rahmen des Projekts „DigiPark“ des Nationalparks Gesäuse realisiert mit GIS-fähigen Datenbanken und XML als Austauschformat.

Diplomarbeit

zur Erlangung des akademischen Grades eines Magister
des individuellen Diplomstudiums Geoinformationstechnologie (GTEC)

angefertigt am
Universitätszentrum Rottenmann

eingereicht von
Bucher Martin

bei
Dr. Ingeborg Liedlbauer
am
Institut für Wirtschaftsinformatik - Data & Knowledge Engineering

Rottenmann, Juli 2007

Danksagung

Ich bedanke mich bei meiner Diplomarbeitsbetreuerin, Frau Dr. Ingeborg Liedlbauer, für die persönliche und fachlich kompetente Unterstützung und die Zeit, die sie mir während der Betreuung der Diplomarbeit entgegengebracht hat. Des Weiteren gilt der Dank meinem Diplomarbeitsbetreuer, Herrn DI Peter Hamader, für die technische Unterstützung.

Ein besonderer Dank ergeht an meine Freundin Jasmin, die immer ein offenes Ohr für meine Probleme hat und durch konstruktive Kritik, Rat und Hilfsbereitschaft diese vorliegende Arbeit mit ermöglicht hat.

Ein besonderer Dank ergeht an alle meine Freunde, die mich all die Jahre hinweg tatkräftig unterstützt haben.

Ein besonderer Dank ergeht an Herrn DI Christof Dallermassl für die technische und persönliche Unterstützung im Rahmen der Diplomarbeit.

Ein besonderer Dank ergeht an Frau Mag. Dr. Lisbeth Zechner, Mitarbeiterin des Nationalparks Gesäuse. Sie hat mir die aktuellen Daten für das Projekt geliefert und sich immer Zeit für meine Fragen und Anliegen genommen.

Ein besonderer Dank ergeht an alle meine Arbeitgeber während der Studiendauer in Rottenmann, da ohne Sie das Studium nicht möglich gewesen wäre. Besonders hervorheben möchte ich hier folgende Firmen und Institutionen: JoWood, Trenkwalder, Salzburger Landesregierung (Sagis) und AHT Cooling Systems.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die Diplomarbeit mit dem Titel *Objektrelationales Mapping mit Java 5 im Rahmen des Projekts „DigiPark“ des Nationalparks Gesäuse realisiert mit GIS-fähigen Datenbanken und XML als Austauschformat* selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen, als solche kenntlich gemacht habe.

Rottenmann, am 16.07.2007

_____	_____
Bucher Martin	Matrikelnummer

Informationen

Vor- und Zuname:	Martin Bucher
Institution:	Universitätszentrum Rottenmann
Studiengang:	Individuelles Diplomstudium *GTEC*
Titel der Diplomarbeit:	Objektrelationales Mapping mit Java 5 im Rahmen des Projekts „DigiPark“ des Nationalparks Gesäuse realisiert mit GIS-fähigen Datenbanken und *XML* als Austauschformat.
Betreuerin:	Dr. Ingeborg Liedlbauer

Schlagwörter

1. Schlagwort:	Objektrelationales Mapping (*ORM*)
2. Schlagwort:	Datenbank mit *GIS*-Erweiterung
3. Schlagwort:	*XML*
4. Schlagwort:	Java
5. Schlagwort:	*XML*-Schema
6. Schlagwort:	Hibernate
7. Schlagwort:	*PostgreSQL*

Kurzfassung

Die Verfügbarkeit von mobilen Endgeräten (*PDAs*) verbessert sich ständig. Die Speicherkapazitäten sowie die Geschwindigkeiten von Prozessoren nehmen sehr rasant zu. Das Resultat dieser Entwicklungen ist eine immer größer werdende Palette von Nutzungsmöglichkeiten.

Zielsetzung der Diplomarbeit ist es, diese Nutzungsmöglichkeiten bei der täglichen Arbeit im Nationalpark im Rahmen der *Biodiversitätserfassung* einzusetzen.

Gegenstand dieser Diplomarbeit war die theoretische und praktische Aufarbeitung des Themas Objektrelationales Mapping (*ORM*), um eine Brücke zwischen relationalen Datenbanken und der objektorientierten Programmiersprache Java 5 zu schaffen.

Im theoretischen Teil der Diplomarbeit wird ein Überblick über das Thema *ORM* gegeben. Es wird auf die Entwicklung von *ORM* eingegangen und eine Übersicht über verschiedene *ORM*-Ansätze mit Java gegeben. Zu diesen Ansätzen werden unterschiedliche *ORM*-Produkte vorgestellt, wobei die Eignung eines solchen Produkts mittels Vergleichsparametern bewertet wird. Abschließend wird die zukünftige Stellung und Entwicklung von *ORM* in der Softwareentwicklung behandelt.

Im praktischen Teil der Diplomarbeit soll im Rahmen des Projekts „DigiPark“ für den Nationalpark Gesäuse eine Anwendung geschaffen werden, um die alte Methode der Speicherung von *Biodiversitätsdaten* mit Excel-Tabellen zu ersetzen und einen durchgängigen Datenfluss zu schaffen. Die Datenbankanbindungen werden mittels *ORM* realisiert. Das Projekt soll für Mitarbeiter ein modernes und effizientes Werkzeug für die Erfassung von *Biodiversitätsdaten* sein. Durch die *GIS*-Datenanbindung ist es möglich, in der Web-Applikation und am *PDA* einen Zusammenhang zwischen interessanten Themengebieten und ihrer Lokation auf einer Karte herzustellen. Der durchgehende digitale Datenfluss beschleunigt die Verwertung von Beobachtungsdaten für statistische Auswertungen mittels der Standardsoftware „BioOffice“. Zusätzlich soll ein Nationalpark Guide entwickelt werden, um Besucher des Nationalparks mit Informationen vor Ort zu versorgen.

Abstract

The availability of mobile (end-) devices (*PDA*) is improving continuously. The storage capacities as well as the processor speed increase very fast.

The result of these developments means growing possibilities of applications. The objective of this thesis is to use a *PDA* for the National Park in the observation of species in order to gain knowledge about its biodiversity.

The theoretical part of this thesis contains an overview of the topic Object-Relational Mapping (*ORM*) and continues with the development of relational databases and the object-oriented programming language Java 5. In addition, various *ORM*-products are discussed and their applicability will be assessed by using parameters of comparison. As a last step, the future position and the development of *ORM* according to software engineering will be dealt with.

The practical part describes the project “DigiPark” which was specifically produced for the National Park “Gesäuse”. “DigiPark” is a tool for monitoring biodiversity in the National Park “Gesäuse” “DigiPark” should bring together the different functions of integrating data collected with the *PDA*, visualizing this data on a map on a web-page and utilisation of this collected observational data for statistical purposes by using the standard software “BioOffice”. This continuous digital data flow assists the members of the National Park in doing their work. A second goal was to provide a guide of the National Park with useful information for visitors. By using *GIS* data, it is possible to make a connection between interesting themes and their location on the map.

Inhaltsverzeichnis

1 Einleitung

Durch die zunehmende Bedeutung von objektorientierten Programmiersprachen, wie z.B. Java oder *.NET* im Bereich der Softwareentwicklung und der im Bereich Datenbanken stark vertretenen relationalen Datenbank-Management-Systeme, steigt der Bedarf nach *ORM*.

1.1 Ziele und Aufbau der Magisterarbeit

Das Ziel dieser Arbeit ist es, verschiedene Ansätze von *ORM* theoretisch und teilweise praktisch vorzustellen.

Im ersten Teil der Magisterarbeit wird die Bedeutung von *ORM* für die Softwareentwicklung erarbeitet, wie zum Beispiel im Bereich der objektorientierten Programmiersprachen. Als Grundlage für die Ausarbeitung wurde die Programmiersprache Java genommen, worauf sich alle praktischen Beispiele beziehen.

Der zweite Teil der Magisterarbeit beschäftigt sich mit dem unter Kapitel 1.2 beschriebenen Projekt „DigiPark“ für den Nationalpark Gesäuse. Das besondere Augenmerk wird dabei auf den bearbeiteten Teil „Datenbanken und Schnittstellen“ gelegt. Es wird eine Auswahl von *GIS*-fähigen Datenbanken vorgestellt. Das praktische Beispiel bezieht sich auf das *ORM* mit Hibernate und es wird auf den Datenaustausch mit *XML* eingegangen.

Alle kursiv geschriebenen Textteile werden im Glossar erklärt.

1.2 Projekt „DigiPark“

Das Projekt wurde im Oktober 2006 begonnen und im April 2007 vorübergehend eingestellt.

Projektteam:

- Nationalpark Gesäuse: Vertreten durch Mag. Dr. Lisbeth Zechner.
- BioGis Consulting GmbH: Vertreten durch Dr. Peter Schreilechner.
- UZR Rottenmann: Vertreten durch Dr. Ingeborg Liedlbauer und DI Peter Hamader.
- Diplomanden: Martin Bucher, Rene Kreuzbichler und Walter Schmiedhofer.

Zweck und Ziel

Vorbild für das Projekt „DigiPark“ war „WebPark“, ein EU-Projekt des Schweizer Nationalparks. „WebPark“ hat zum Ziel, eine Plattform für die Vermittlung von *Location Based Ser-*

vices (LBS) in Erholungs- und Naturschutzgebieten aufzubauen. Mitglieder dieses Konzepts sind sechs Unternehmen. Dazu zählen Geodan Mobile Solutions, European Aeronautic Defence and Space Company (EADS), City University, Universität Zürich, Laboratorio Nacional de Engenharia Civil und der Schweizer Nationalpark. [WebPark 2004]

Die Verfügbarkeit von *PDAs* verbessert sich ständig. Die Speicherkapazitäten sowie die Geschwindigkeiten von Prozessoren nehmen rasant zu. Das Resultat dieser Entwicklungen ist eine immer größer werdende Palette von Nutzungsmöglichkeiten. Zielsetzung des Projektes ist es, diese Nutzungsmöglichkeiten bei der täglichen Arbeit im Nationalpark im Rahmen der *Biodiversitätserfassung* einzusetzen.

Der Nationalpark Gesäuse will in Zukunft botanische und zoologische Daten vor Ort erheben. Motivation für das Projekt war, die veraltete Methode unter Verwendung von Excel-Tabellen zu ersetzen und einen durchgängigen Datenfluss zu erhalten.

Neben der Verortung der Beobachtungen soll es nun auch möglich sein, Besucher des Nationalparks mit Informationen vor Ort zu versorgen. Dem Interessenten soll auch die Möglichkeit geboten werden, sich in einer Web-Applikation interessante Themengebiete grafisch mittels des Produkts *UMN MapServer 4.0* darstellen zu lassen.

Ebenfalls soll es einem Besucher möglich sein, vor Ort Informationen über Wanderrouten in digitaler Form zu erhalten. Als Erweiterung des Angebots soll eine grafische Schnittstelle entwickelt werden, um die zu besichtigenden Informationen in einer Karte am *PDA* zu visualisieren.

Als zentrales Element soll eine Serverdatenbank entwickelt werden, von der aus die Web-Applikation und die *PDA*-Applikation mit den benötigten Daten versorgt werden. Bisher arbeitet der Nationalpark Gesäuse mit mündlichen Informationen und verfügt über keine ausreichende Abdeckung durch Datenbanken. Es soll eine 3-Schichten Architektur (Client-Server-Architektur) aufgebaut werden.

Weiters wird in Zukunft das Produkt „BioOffice“ von BioGis Consulting verwendet, um die erhobenen Daten statistisch auszuwerten. Es bietet sich die Möglichkeit mit einer Firma, die schon seit Jahren im *GIS*-Sektor tätig ist, ein Projekt abzuwickeln und eine Software, die in der Praxis angewendet wird, kennen zu lernen.

Die Anwendung soll in Zukunft von Studenten des Studiums *GTEC* gewartet werden können.

Das Gesamtprojekt wurde in drei getrennt zu bearbeitende Teilbereiche aufgegliedert:

- Web-Applikation (Bearbeiter: Walter Schmiedhofer)
- *PDA*-Applikation (Bearbeiter: Rene Kreuzbichler)
- Datenbanken und Schnittstellen (Bearbeiter: Martin Bucher)

In Abbildung 14 ist die Projektarchitektur des Projekts „DigiPark“ ersichtlich. Die vorliegende Diplomarbeit behandelt im praktischen Teil den Projektteil Datenbanken und Schnittstellen. Die Teilaufgaben des vorliegenden Projektteils sind im Kapitel 6.1 beschrieben.

2 Objektrelationales Mapping (*ORM*)

In diesem Kapitel wird erläutert, warum *ORM* als Brücke zwischen relationalen Datenbanken *(RDB)* z.B. *PostgreSQL* oder *MS SQL Server 2005* und objektorientierten Programmiersprachen mit allen dafür notwendigen Tätigkeiten im Rahmen einer Datenbankanbindung sinnvoll ist und welche Ansätze bestehen.

2.1 Definitionen

ORM bietet Programmierern eine objektorientierte Sicht auf Tabellen und Beziehungen in relationalen Datenbankmanagementsystemen (*RDBMS*). Statt mit *SQL*-Statements wird mit Objekten operiert. [Horn 2007]

ORM bezeichnet die Abbildung von objektorientierten Daten auf relationale Daten und umgekehrt. [Wikipedia 2007]

2.2 Ziele von *ORM*

Eine kurze Auflistung der Ziele von *ORM* [Wille 2007]:

- Ausnutzen der Vorteile von *SQL*-Datenbanken, ohne die Java-Welt von Klassen und Objekten zu verlassen.
- Weniger Zeit- und Arbeitsaufwand für Persistenz und Erreichung einer zufrieden stellenden Datenbankanbindung.
- Transparente Abbildung von Objekten auf relationale Strukturen, inklusive Objektreferenzen, Vererbung und *Polymorphie.*
- *Persistente Objekte* sollen sich selbst um ihre Persistierung kümmern.
- *Portabilität* zwischen Datenbankprodukten und -versionen erreichen.

- Etablierung einer objektorientierten Abfragesprache, wobei natives *SQL* und Datenbank spezifische Eigenschaften weiterhin nutzbar bleiben sollen.
- Sicherstellung der Trennung von Persistenz- und Geschäftslogik.

2.3 Gründe für *ORM*

Trotz der heute starken Verwendung einer objektorientierten Programmierung, werden die meisten Datenbestände in *RDB* gehalten. Somit ist *ORM* nötig, welches die Objekte, deren Attribute und Beziehungen auf die Tabellenstrukturen und Spalten und umgekehrt abbildet. [Meier 2003]

2.3.1 Relationale Datenbankmanagementsysteme

Ein relationales Datenbankmanagementsystem, abgekürzt *RDBMS* genannt, ist ein integriertes System zur einheitlichen Verwaltung relationaler Datenbanken. Jedes relationale Datenbanksystem besteht also aus einer Speicherungs- und einer Verwaltungskomponente (siehe Abbildung 1). [Saake und Sattler 2003]

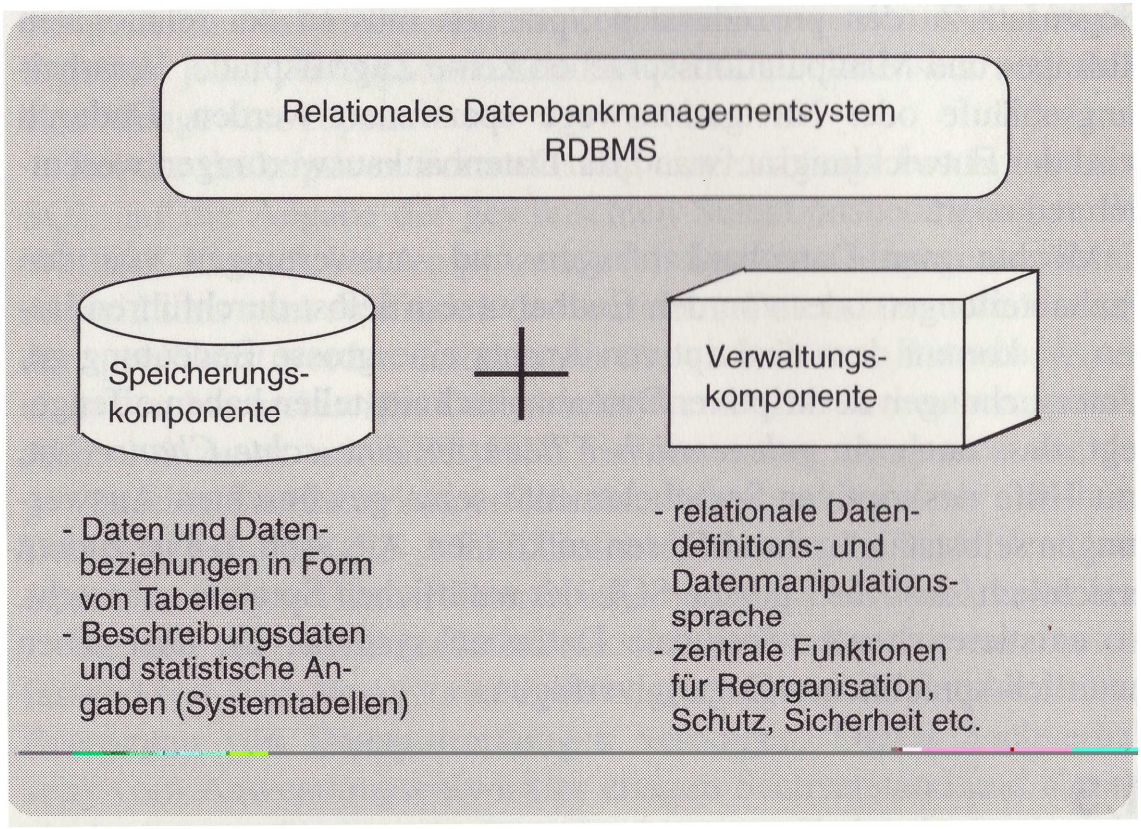

Abbildung 1: Die zwei Komponenten eines *RDBMS* [Meier 2003, S.8]

RDBMS sind seit Jahren für viele Anwendungen als Standardwerkzeug zur Verwaltung großer Datenbestände etabliert. Auch die Verbreitung objektorientierter Entwurfs- und Programmiertechniken hat daran nichts Grundsätzliches geändert. Einige Anwendungen speziell aus dem

technischen Bereich stellen jedoch Anforderungen, die durch *RDB* nur unzureichend erfüllt werden können. [Matthiessen und Unterstein 2003]

Beispiele für *RDBMS* sind:

- *PostgreSQL*
- Oracle
- DB2 (*IBM*™)
- Infomix
- Sybase
- MS-Access
- MySql
- Firebird
- MaxDB
- Microsoft *SQL* Server

2.3.2 Impedance Mismatch

Unter Impedance Mismatch wird der Unterschied zwischen relationalen Datenmodellen und objektorientierten Programmiersprachen verstanden. Objektorientierte Programmiersprachen sind satzorientiert und prozedural, im Gegensatz dazu sind relationale Datenbanken mengenorientiert und deklarativ. In objektorientierten Programmiersprachen kapseln Objekte Daten und Funktionalitäten, im relationalen Modell hingegen werden Daten in Relationen gespeichert. Die Manipulation der Daten erfolgt aber extern durch eine Datenbearbeitungssprache. [Edv-Buch 2007] In diesem Kapitel wird aufgrund der Komplexität dieses Themas nur ein Überblick über die wichtigsten Unterschiede im Zusammenhang mit *ORM* gegeben.

Durch Impedance Mismatch treten beim Mapping zwischen *RDB* und objektorientierten Programmiersprachen eine Reihe von Problemen auf. Die grundlegende Problematik des Impedance Mismatch entsteht durch die fehlende Abbildung von objektorientierten Strukturen auf das Datenmodell *RDBMS*. Dies betrifft unter anderem die Bindung von Klassenhierarchien an relationale Schemata, die Generierung von Primärschlüssel sowie die gleichzeitigen Zugriffe auf Daten.

Folgende Unterschiede gilt es zu überwinden [Edv-Buch 2007]:

- *Granularität*: Objekte können jegliche *Granularität* haben, Tabellen hingegen sind bezüglich der *Granularität* beschränkt. In einer *RDB* werden die Daten normalerweise in einer Tabelle gespeichert. [Hien und Kehle 2007]
- Vererbung: Vererbung ist in objektorientierten Programmiersprachen wie Java selbstverständlich. *RDB* kennen aber keine Vererbung. [Hien und Kehle 2007]
- Objektidentität: Objektidentität ist die Objekteigenschaft, die eine Instanz eines Objekts von allen anderen unterscheidet. In Java sind zwei Objekte identisch, wenn beide dasselbe Objekt sind. Wenn zwei Objekte den gleichen Inhalt haben, sind sie bezogen auf den Inhalt gleich, aber nicht identisch. In *RDB* wird ein Eintrag in einer Tabelle über die Daten, die er enthält, identifiziert. Es wird ein Primärschlüssel definiert, der die Identität künstlich ermöglicht. In den Objekten wird derselbe Primärschlüssel eingeführt und dadurch kann über diesen die Abbildung von einem Objekt auf einen Eintrag in einer *RDB* realisiert werden. [Hien und Kehle 2007]
- Beziehungen: Beziehungen gibt es auch in *RDB*. Mit einem Fremdschlüssel in der einen Tabelle wird ein Primärschlüssel in einer anderen Tabelle referenziert und somit die Tabellen in Beziehung gebracht. In der objektorientierten Welt gibt es mehrere Arten von Beziehungen: 1-zu-1, 1-zu-n, n-zu-1 und m-zu-n Beziehungen. Diese können alle mit Fremdschlüssel abgebildet werden. Etwas komplizierter ist die 1-zu-n Beziehung, die durch eine Beziehung von der n-Seite zur 1-Seite realisiert wird und die m-zu-n Beziehung, bei der eine Beziehungstabelle (Attributtabelle oder *Join*-Tabelle) eingeführt werden muss. [Hien und Kehle 2007]
- Navigation: Über Objekte mit Java zu navigieren ist leicht. Bei Datenbankzugriffen kann es sich jedoch bei 1-zu-n Beziehungen so verhalten, dass mehrere Datenbankzugriffe notwendig sind. Für dieses Problem müssen geeignete Konzepte bei den *ORM* entwickelt werden.
- Berücksichtigung von gleichzeitigen Zugriffen: Ein weiterer wichtiger Faktor bei der Datenintegrität auf Anwendungsebene ist der gleichzeitige Zugriff auf Daten. Es muss sichergestellt werden, dass die vom *RDBMS* bereitgestellten Sperrebenen in den Persistenzmethoden der Klasse widergespiegelt sind. [Schafer 2007]
- Berücksichtigung der Schemaentwicklung: Aufgrund der Zuordnung von objektorientiertem und relationalem Schema muss immer gewährleistet bleiben, dass die Schemata auch bei der Weiterentwicklung, gleich welcher Seite, aufeinander abgestimmt bleiben. Die mögliche Abweichung des objektorientierten Schemas vom ursprünglichen relationalen Schema verkompliziert dies. [Schafer 2007]

Ein Lösungsansatz für das Problem des Impedance Mismatch ist der Einsatz von *ORM*-Tools. Die darin enthaltenen Komponenten bedienen sich einer Zwischenspeicherung (*Caching*) der Operationen, um den Datendurchsatz zu erhöhen, die Integrität der Datenbankoperationen (Einfügungen, Aktualisierungen, Löschvorgänge, Auswahloperationen und die dazugehörigen Anforderungen für den gleichzeitigen Zugriff) zu gewährleisten, eindeutige Primärschlüssel zu generieren oder Bindings für den Einsatz der dazugehörigen Programmiersprache bereitzustellen.

2.3.3 3-Schichten Architektur

Die 3-Schichten Architektur ist eine Client-Server-Architektur. Im Gegensatz zur zweischichtigen Architektur, bei der die Rechenkapazität großteils auf die Client-Rechner ausgelagert wird, um den Server zu entlasten, existiert bei der dreischichtigen Architektur noch eine zusätzliche Schicht, die Logikschicht, welche die Datenverarbeitung vornimmt. Die Schichten dieser Architektur sind [Saake und Sattler 2003]:

- Schicht 1: Datenschicht
- Schicht 2: Logikschicht
- Schicht 3: Präsentationsschicht

ORM bietet den Vorteil, dass die *Implementierung* der Logikschicht und deren Abbildung auf Datenbankrelationen (Datenschicht) durch objektrelationale Mapping-Werkzeuge unterstützt werden. Dadurch wird die Erstellung einer 3-Schichten Architektur durch *ORM* vereinfacht. Abbildung 2 zeigt ein Beispiel für eine 3-Schichten Architektur.

2.3.4 Nachteile Objektorientierter Datenbankmanagementsysteme (*OODBMS*)

Herkömmliche Datenbanksysteme sind entweder satzorientiert (netzwerkartiges und hierarchisches Datenbankmodell) oder mengenorientiert (relationales Datenbankmodell). [Uni Karlsruhe 2007] *OODBMS* haben zum Ziel, Eigenschaften von objektorientierten Programmiersprachen (wie Vererbung, Objektidentität, Klassenhierarchie) in klassische Datenbankverwaltungssysteme zu integrieren. *OODBMS* ermöglichen die Repräsentation von komplexen Sachverhalten, wie zusammengesetzte Objekte, komplexe Strukturen oder neue Datentypen. Jedes *OODBMS* ist eine *Implementierung* eines objektorientierten Datenmodels. [Hansen 1998]

OODBMS weisen folgende Nachteile auf [Poeschek 2007]:

- Standards (*ODMG*) beginnen sich nur langsam durchzusetzen.

- Viele Dienste, die in *RDBMS* selbstverständlich vorhanden sind (Sichtenbildung, Schemaevolution, deklarative Abfragesprachen, Konsistenzbedingungen, Rechte und Tuning), sind nicht oder nur ansatzweise gegeben.
- Geringer Marktanteil von ca. 2% [Uni Karlsruhe 2007]

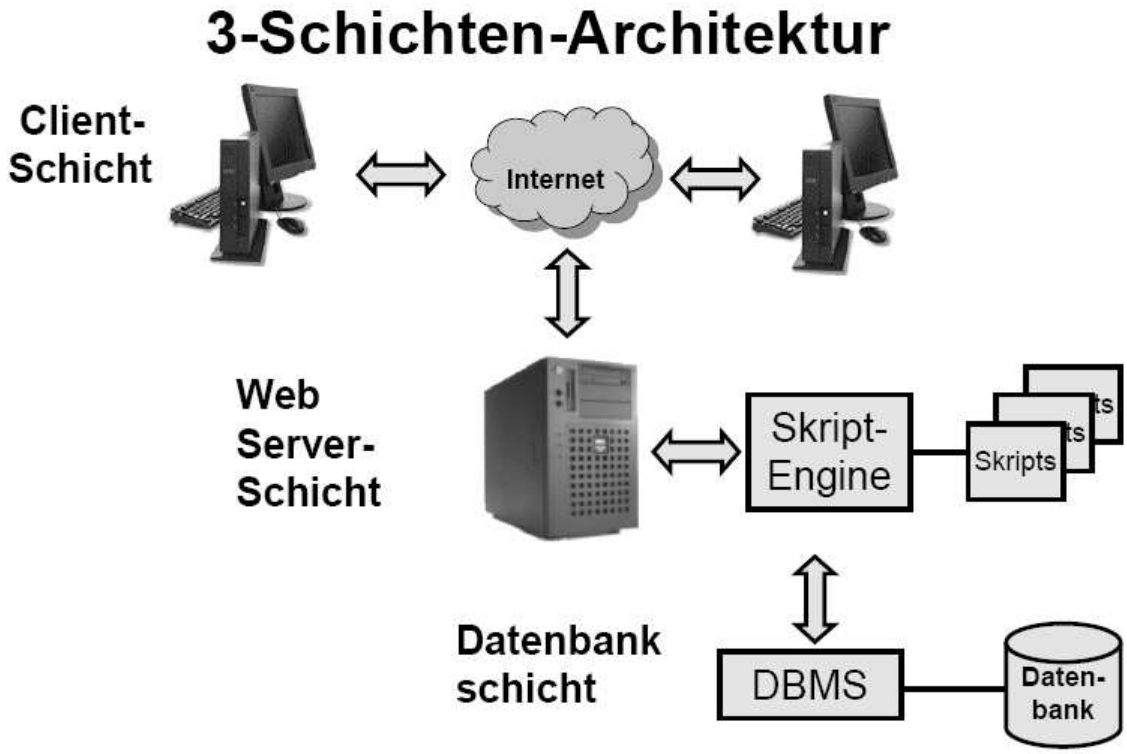

Abbildung 2: Beispiel für eine 3 Schichten Architektur [ITWissen 2007]

Prinzipielle Probleme von *OODBMS* [Uni Karlsruhe 2007]**:**

- Enge Verknüpfung von DB-Schema und Anwendung führt oft zu Inflexibilität.
- Mangels entsprechender polymorpher Operatoren ist keine flexible Neuverbindung der Daten (z.B. durch *Join*) möglich, sondern nur eine Verknüpfung entlang der im Schema festgelegten Beziehungen.

Begründet durch die angeführten Nachteile, entscheiden sich viele Softwareentwickler für *RDB*. Um nun die Vorteile der *RDB* und der objektorientierten Programmiersprachen nutzen zu können, empfiehlt sich der Einsatz von *ORM*.

2.3.5 Vorhandensein von Mapping-Werkzeugen und Tools

Die *Implementierung* der Objekte und deren Abbildung auf Datenbankrelationen werden durch objektrelationale Mapping-Werkzeuge unterstützt. Diese Werkzeuge realisieren zwei wesentliche Aspekte:

- den konzeptionellen Teil der Abbildung durch Festlegen der Zuordnung zwischen Klassen mit Attributen und Tabellen mit Spalten und
- die Laufzeitunterstützung durch Definition und Ausführung der entsprechenden *SQL*-Anweisungen, das Puffern von Objekten für ein schnelleres Verfolgen von Referenzen sowie die Realisierung von Transaktionen und Sperren. [Saake und Sattler 2003]

Eine Auflistung der Produkte wird im Kapitel 3.3 gegeben.

2.3.6 Weniger Code als bei Java Database Connectivity (*JDBC)*

Ein weiterer Vorteil von *ORM* ist, dass weniger Codezeilen für die Datenanbindung benötigt werden, jedoch ist diese Tatsache davon abhängig, wie im Vergleich dazu die *JDBC*-Datenanbindung aufgebaut wird.

2.3.7 Erweiterbarkeit und Änderbarkeit

Anwendungen, die mit *ORM* erstellt wurden, können einfach erweitert werden. Zusätzliche Tabellen oder Klassen lassen sich sehr einfach und ohne Eingriff in den Code erzeugen und in das bestehende Datenbankschema integrieren.

2.3.8 *Performance*-Optimierung möglich

Die Leistung einer mit *ORM* erstellten Anwendung kann optimiert werden. Es gibt zahlreiche Dokumentationen mit einer Anleitung, wie die Leistung eines mit *ORM* erstellten Programms verbessert werden kann. Die *Performance* kann z.B. verbessert werden, indem die Initialisierung der referenzierten Objekte gesteuert wird. [Wille 2007]

2.3.9 Portabilität

Anwendungen, die mit *ORM* erstellt wurden, weisen eine sehr gute *Portabilität* in Bezug auf den Grad der Plattformunabhängigkeit eines Computerprogramms auf. Sie wirken sich auch auf den eingesetzten Arbeitsaufwand, der benötigt wird, um das Programm in ein vollständig plattformunabhängiges umzuwandeln, sehr günstig aus. Der Vorgang der Umwandlung wird Portierung genannt und auch in diesem Bereich zeigen sich die Vorteile von *ORM* gegenüber einer herkömmlichen *JDBC*-Anbindung. [Herke 2007]

2.3.10 Verwendung von Plain Old Java Objects (*POJOs*) möglich

Anwendungen, die mit *ORM* erstellt wurden, können meistens *POJOs* nutzen. Dabei handelt es sich um einfach zu *implementierende* Java-Objekte im Gegensatz zu aufwendigen Enterprise JavaBeans (*EJBs*, siehe Kapitel 3.1.4*)* für *Java EE*-Container.

2.3.11 Forward Engineering und Reverse Engineering

Beim Forward Engineering bilden Java-Klassen den Ausgangspunkt. Diese sind auf ein Datenbankschema abzubilden, wobei die Art der Abbildung im Wesentlichen vom Typ des Datenverwaltungssystems abhängt. [Saake und Sattler 2003] Der Vorteil des Forward Engineering ist die Möglichkeit der automatischen Erstellung des Datenbank-Schemas durch ein spezielles Tool.

Beim Reverse Engineering bildet ein bereits vorhandenes Datenbankschema den Ausgangspunkt, von welchem Java-Klassen abzuleiten sind. [Hien und Kehle 2007]

2.3.12 Das Prinzip der Kapselung und Abstraktion

Zwei wichtige Grundprinzipien der Objektorientierung stellen Kapselung und *Abstraktion* dar. Das Prinzip der Kapselung ermöglicht die *Interaktion* mit einem Objekt und zwar stets so, dass Zustände eines Objektes nicht unerwartet verändert oder gelesen werden können. Viel mehr wird nur über ein bestimmtes Set von Methoden mit diesem Objekt kommuniziert (die Schnittstelle des Objektes). Das Prinzip der *Abstraktion* erlaubt es, bestimmte gemeinsame Eigenschaften von mehreren Klassen in einer Klasse zusammenzufassen. Die interne Realisierung bleibt dem Nutzer der Schnittstelle verborgen. [Software-Kompetenz 2007] Wenn eine relationale Datenbank mittels *ORM* und einer objektorientierten Programmiersprache angebunden wird, kann dieses Prinzip ebenfalls genutzt werden.

2.4 Nachteile von *ORM*

2.4.1 *ORM* im Vergleich zu Objektdatenbanken

Das Datenbankschema einer Objektdatenbank sind die persistierten Objekte. Die Objekte können so, wie sie in der Applikation existieren, persistiert werden. Bei Lösungen mit *ORM* erzeugt der Persistenzmanager von *ORM* das Datenbankschema über das gekoppelte Persistenzframework.

Bei *ORM* werden Datenbank-Abfragen meistens in einer eigenen Abfragesprache generiert. Hibernate verwendet die Hibernate Query Language. Bei Objektdatenbanken hingegen können die Abfragen direkt in der Programmiersprache der Applikation formuliert werden. Dies hat den Vorteil, dass der entstehende Code vom Compiler geprüft wird und der Entwickler seine bevorzugte Sprache verwenden kann. Ein weiterer Vorteil der Verwendung einer Objektdatenbank ist, dass kein zusätzlicher Code geschrieben werden muss. Die Lösung mittels *ORM* hingegen erzwingt meistens die *Implementierung* zusätzlicher *Interfaces* und Klassen z.B. eigene Klassen für zusammengesetzte Schlüssel. [Herke 2007] Ebenso ver-

wendet jede Objektdatenbank das *Prinzip der Kapselung*. [Poeschek 2007] Welche Lösung besser ist, sollte anhand von *Softwaremetriken* entschieden werden.

2.4.2 *ORM* im Vergleich zu *JDBC*

Hier muss der höhere Aufwand für die Konfiguration des *ORM* Frameworks angeführt werden. Die Komplexität der Konfiguration eines *ORM* Frameworks richtet sich nach dem verwendeten Datenbanktyp. Genügt für die Erstellung einer Schnittstelle zu einer *RDB* eine *JDBC*, ist diese sehr einfach zu *implementieren* und erfordert wenig technisches *Know-how*. Ein durchschnittliches Wissen über die Java-Programmierung ist dafür ausreichend. Für die Konfiguration eines *ORMs* mit Persistence Framework benötigt der Programmierer, je nach verwendeter *ORM*-Technologie, fundiertes Wissen in unterschiedlichen Technologien z.B. *Ant*. Für einfache Anwendungen mit wenigen Relationen empfiehlt sich aus Zeitgründen der Einsatz von *JDBC*.

2.4.3 Schnittstelle zu mehreren Datenbanken mit einer *ORM*-Technologie

Bei einer beabsichtigten Anbindung von unterschiedlichen Datenbanktypen z.B. *PostgreSQL* und *MS SQL Server* mittels einer *ORM*-Technologie ist eine Überprüfung in Hinsicht auf Eignung der eingesetzten Technologie dringend erforderlich. Nicht jede derzeit vorhandene Datenbanktechnologie wird von jeder *ORM*-Technologie optimal unterstützt.

2.5 Fazit

ORM verspricht einen einfachen Zugriff von objektorientierten Programmiersprachen auf *RDB*. Die technische Umsetzung des Mapping gestaltet sich in der Praxis jedoch sehr aufwendig. Deshalb wurden verschiedene Frameworks geschaffen, die dem Entwickler diese Arbeit erleichtern sollen. Als Alternative bietet sich der Einsatz einer Objektdatenbank an, bei der ein *ORM* überflüssig wird. [Poeschek 2007] Um die Vor- und Nachteile beider Strategien zu erheben, müssen die Alternativen für ein Projekt *empirisch* untersucht werden und dies sollte durch *Prototyping* geschehen.

Wichtige Entscheidungskriterien im Überblick sind:

- *Migration* eines bestehenden Datenbanksystems (Reverse Engineering möglich?) [Heinrich et al. 2004]
- Datenbanktyp (*RDB*, Objektdatenbank)
- geeignete *ORM*-Mapper und Tools
- Komplexität des Datenbankschemas

- *Know-how* der Programmierer
- verfügbare Zeit
- Entscheidung anhand geeigneter *Softwaremetriken*
- *Prototyping* und *Validierung*

3 Stand der Technik

Dieses Kapitel gibt einen Überblick über die verschiedenen Ansätze von *ORM*. Das Hauptaugenmerk liegt hier auf der Verwendung der Programmiersprache Java. Als Vergleichsmöglichkeit wird auch ein Ansatz ohne Java vorgestellt.

3.1 Ansätze von *ORM* mit Java als Programmiersprache

3.1.1 Java Data Objects (JDOs)

Java Data Objects (JDOs) sind eine offizielle *Sun-Spezifikation* für ein herstellerunabhängiges Framework zur persistenten Speicherung von Java Objekten in *RDB* [Jordan und Russel 2003]. Die folgenden Informationen beruhen auf [Saake und Sattler 2003] und [Jordan und Russel 2003].

Die erste Version von JDO wurde im Mai 2001 von *Sun*™, *IBM*™ und *Apple*™ verabschiedet. In der JDO-*Spezifikation* wird eine einheitliche Schnittstelle für den Zugriff auf persistente Daten definiert. Die Art und Weise der physikalischen Speicherung ist nicht festgelegt. JDO erfüllt alle Voraussetzungen für das *ORM*. [JDO-Architektur 2007] Für vertiefendes Wissen empfiehlt sich das Buch [Jordan und Russel 2003]. JDO verwendet als Abfragesprache JDOQL, welche in der derzeit vorhandenen Version ein mächtiges Werkzeug darstellt. Abbildung 3 zeigt einen Überblick über die JDO-Architektur.

Mit der Entwicklung von JDO wurde das Ziel einer transparenten Persistenz von Java-Objekten, unabhängig von konkreten Speichertechniken, verfolgt. Transparente Persistenz heißt dabei, dass Entwickler keine speziellen Vorkehrungen treffen müssen und Klassen per Definition persistenzfähig sein sollen. Es wird zwischen persistenten und transienten Objekten unterschieden. *Persistente Objekte* (JDO Instanzen) sind Instanzen persistenzfähiger Klassen in der Datenbank. [Jordan und Russel 2003]

Die *Implementierung* der erforderlichen Methoden wird nicht vom Entwickler bereitgestellt, sondern vom JDO-Enhancer. Der JDO-Enhancer ist ein Postprozessor zur Modifikation bzw. Erweiterung des *Byte-Codes* von Java-Klassen. Es lassen sich dadurch neben benutzerdefinierten Klassen auch bereits existierende Klassen nachträglich persistenzfähig machen.

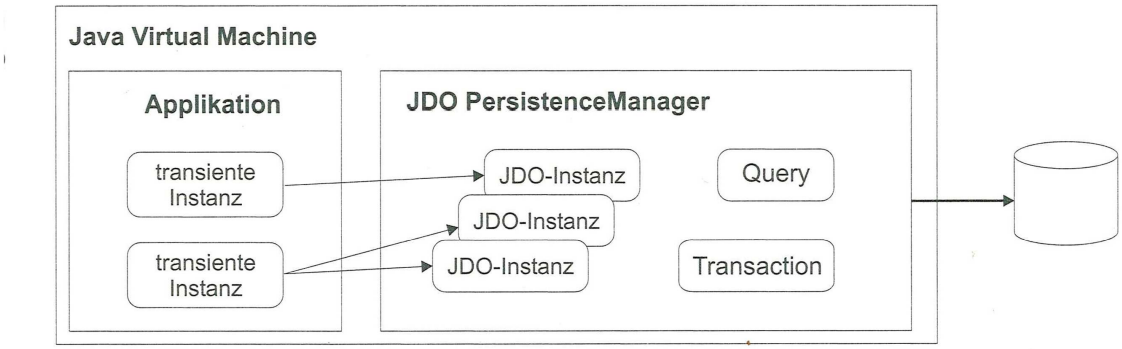

Abbildung 3: JDO-Architektur [Saake und Sattler 2003, S. 260]

Persistente Objekte sind jederzeit genau einem Persistence-Manager zugeordnet, welcher für Aktualisierung, Aktivierung, Speicherung und Gewährleistung der Transaktionssemantik verantwortlich ist. (siehe Abbildung 3)

In JDO wird zwischen First-Class-Objekten und Second-Class-Objekten unterschieden.

First-Class-Objekte:

- Instanzen von Klassen mit der Persistence Capable-Schnittstelle.
- In einem Datenspeicher unabhängig von anderen Objekten speicherbar.
- Besitzen eine eigene JDO-Identität (Objektidentifikator, z.B. ein Primärschlüssel).

Second-Class-Objekte:

- Durch das Fehlen einer eigenen JDO-Identität charakterisiert.
- Können nur als Komponenten eines anderen Objektes gespeichert werden, z.B. String Datentyp als ein Attribut einer Klasse.

Für First-Class-Objekte hat das Konzept der Objektidentität eine fundamentale Bedeutung.

Es werden folgende Formen der JDO-Identität unterschieden:

- Applikationsidentität: Hierbei werden die Identifikatoren durch die Anwendung vergeben und die Eindeutigkeit wird durch den Datenspeicher garantiert. Der Entwickler hat bei Nutzung dieser Identität durch *Implementierung* entsprechender ObjektId-Klassen selbst Sorge für die Abbildung auf den Primärschlüssel der Datenbanktabelle zu tragen.
- Datenspeicheridentität: Hier wird der Identifikator durch den Datenspeicher unabhängig von den Attributwerten der Objekte verwaltet. Dies entspricht der Verwendung von künstlichen Schlüsseln in relationalen Datenbanksystemen.

- Nichtdauerhafte Identität: Die Objektidentifikation erfolgt ausschließlich durch die *Java-VM*. Eine dauerhafte Identifikation ist somit nicht gewährleistet.
- Eine zentrale Rolle neben der Identität spielt die Frage der Zustandsrealisierung. Die Attributwerte der JDO-Instanzen sollten über die Laufzeit der Applikation hinaus gespeichert werden und der Transaktionssemantik unterliegen, d.h. bei einem Commit dauerhaft in der Datenbank erscheinen und bei einem Transaktionsabbruch auf den Ausgangszustand zurückgesetzt werden.

In JDO werden drei Arten von Attributen unterschieden:

- Persistent: Attribute erfüllen die Forderung nach Persistenz und Transaktionalität.
- Transaktional nicht persistent: Diese Attribute werden nicht in der Datenbank gespeichert, sind aber transaktional.
- Nicht transaktional – nicht persistent: Keine der oben genannten Forderungen treffen zu.

Basen für transaktionale und/oder persistente Attribute sind:

- alle primitiven Java-Datentypen (int, double etc.)
- Objektklassen (z.B. java.lang.Integer)
- benutzerdefinierte Objektklassen

Um eine Klasse persistenzfähig zu machen, werden die Klassen in Java *implementiert*, ohne dass dabei irgendwelche Besonderheiten (etwa die Nutzung spezieller Klassen oder die *Implementierung* vorgegebener Schnittstellen) berücksichtigt werden müssen. Ein parameterloser Konstruktor wird allerdings gefordert.

Damit der JDO-Enhancer die notwendigen Codeerweiterungen vornehmen kann, muss der Entwickler zusätzliche Informationen bereitstellen. Hierzu gehören u.a. persistenzfähige Klassen, die Aufzählung der Klassen, die persistiert werden sollen, die Art der Identität und die Eigenschaften der Attribute. Diese Informationen werden in einer Metadatendatei z.B. in *XML* (JDO-Datei) spezifiziert. Diese Datei trägt den Namen der Klasse bzw. den Namen eines Package und die Endung .jdo und enthält wichtige Mapping-Informationen, die vom JDO-Laufzeitsystem ausgewertet werden müssen. Daher muss die Datei zur Laufzeit im Klassenpfad verfügbar sein. Für jede persistenzfähige Klasse muss in dieser *XML*-Datei ein class-Element definiert sein. Dazu werden *XML*-Attribute verwendet. Das Beispiel Listing 1 zeigt eine JDO-Datei mit Mapping-Informationen.

```
<?xml version"1.0"! encoding="UTF-8"?>
<!DOCTYPE jdo SYSTEM "jdo.dtd">
<jdo>
<package name ="com.signsoft">
<class name ="Organisation"/>
</package>
</jdo>
```

Listing 1: JDO-Datei Beispiel

Die zukünftige Entwicklung des Ansatzes JDO ist schwer abzuschätzen, da *Sun*™ das JDO-Projekt abgegeben hat und diesen Persistenzansatz nicht mehr verfolgt. JDO wird aber im Rahmen des Apache DB Projects weitergeführt (siehe http://db.apache.org/jdo/).

3.1.2 Enterprise Objects Framework

Das Enterprise Objects Framework (EOF) ist ein *ORM*-Framework, welches von NEXT™, heute *Apple*™, 1995 entwickelt wurde, später nach Java portiert wurde und heute in der kommerziellen Lösung WebObjects [Apple EOF 2007] verkauft wird. Es gibt aber auch *Open Source Implementierungen* von EOF z.B. GNUstep [GNUstep Homepage 2007].

Folgende Informationen beruhen auf den Quellen [Seifert 2007] und [Apple EOF 2007].

Internet-/Intranetanwendungen, die auf Daten in Datenbanken zugreifen wollen, verwenden häufig diesen Ansatz, da die meisten Datenbanken, die in der Industrie gehalten werden, auf relationalen Datenmodellen beruhen. Es soll damit möglich werden, den *Web-Browser* als *Datenbank-Frontend* zu benutzen. Damit auf eine Datenbank mithilfe einer Internetanwendung zugegriffen werden kann, muss eine Technologie verwendet werden, die aus der laufenden Applikation einen Zugriff auf die Datenbank ermöglicht. Das kommerzielle WebObjects als Implementierungswerkzeug verwendet dazu EOF, um auf *RDB* zuzugreifen. Abbildung 4 zeigt einen Überblick über die EOF-Architektur.

Eine Entwicklung von objektorientierten Anwendungen, die den Zugriff auf *SQL*-Datenbanken ermöglichen, bedarf der Abbildung von Tabellen auf die Anwendungsobjekte. Die Umsetzung einer solchen Anwendung ist sehr aufwendig und damit fehlerträchtig.

EOF bietet folgenden Ansatz:

- Es werden Zwischenschichten oberhalb der Datenbank *implementiert*.
- Es werden Datenmodelle in so genannten Enterprise Objects abgebildet und die Kommunikation zwischen Datenbank und Anwendung wird über diese Objects abgewickelt.
- Das Datenmodell wird von EOF abstrahiert und damit verbessert sich die Wartbarkeit der Anwendung.

- Die Anwendung wird nun in Abhängigkeit vom Objektmodell entwickelt.
- Durch die Auswahl eines entsprechenden Adapters erfolgt der Zugriff auf die Datenbank.

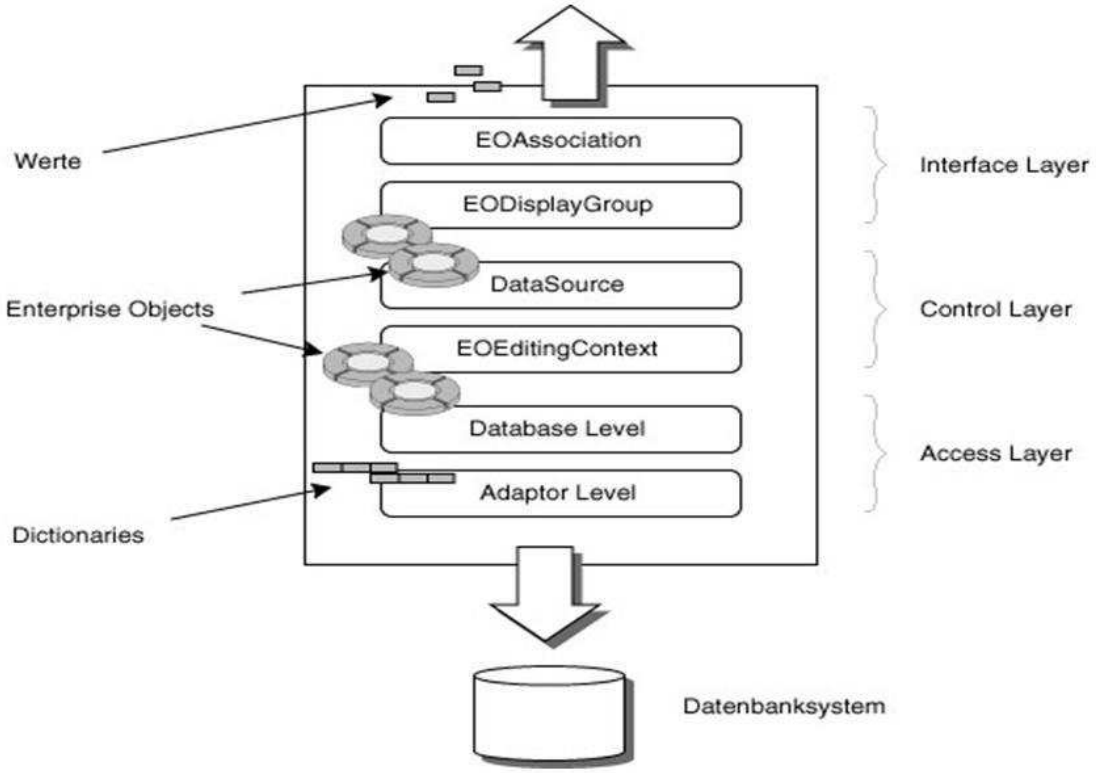

Abbildung 4: EOF-Architektur [Seifert 2007]

Die Architektur des EOF besteht aus drei Schichten (siehe Abbildung 4):

- *Interface* Layer: Dieser Layer beinhaltet Methoden für die Datendarstellung.
- Control Layer: Dieser Layer verwaltet den Graphen der Enterprise Objekte.
- Access Layer: Dieses Layer erzeugt Enterprise Objekte aus Datenbankzeilen.

Der Datenfluss in einer EOF-Anwendung geschieht wie folgt [Apple EOF 2007]:

- Im Access Layer werden die Daten in Form von Datenbankzeilen aus der Datenbank übernommen.
- Im Adaptor Level werden diese Daten als Wert/Name-Paar in NSDictionary-Objekte verpackt. Als Schlüssel dient der Name der Datenbankspalte und als Wert wird die Belegung dieser Spalte in der aktuellen Zeile eingesetzt.
- Im Database Level werden aus den NSDictionary-Objekten EOF-Objekte erzeugt. Das bedeutet, dass Methoden, die für die Erstellung der Anwendungslogik benötigt werden, hinzugefügt werden.

- Über den DataSource werden die Enterprise Objects in den *Interface* Layer gereicht. Die sogenannte DisplayGroup, die von einer Instanz der EODisplayGroup vertreten wird, verwaltet die Objekte für die Darstellung.
- Wenn sich Enterprise Objekte geändert haben, sorgt ein Benachrichtigungsmechanismus dafür, dass die EODisplayGroup-Instanz von der EOAssociation-Instanz informiert wird, die Änderungen nachzuziehen. Die Änderungen werden von den entsprechenden Instanzen entgegengenommen und aktualisieren das User-*Interface*.
- Da für die Formulierung des Objektmodels ein sehr gutes grafisches Tool, der EOModeler, zur Verfügung steht, kommt der Anwendungsentwickler mit dem Objektmodel selten weiter als bis zur EOF-Schnittstelle und ist nicht gezwungen, auf grundlegende Mechanismen zurückzugreifen.

EOF ist heute Teil der kommerziellen Lösung WebObjects und wird in einigen *Open Source Implementierungen* angeboten, die jedoch geringe Bedeutung haben. Im Bereich von WebObjects ist EOF eine sehr gute Technologie für die Erstellung von webbasierten Lösungen.

3.1.3 Hibernate Framework

Dieses Kapitel beschäftigt sich mit dem am häufigsten verwendeten Ansatz. Die Fülle der Informationen ist sehr groß und es ist notwendig, eine Einschränkung zu treffen.

Die folgenden Informationen beruhen auf den Quellen [Hien und Kehle 2007], [Bauer und King 2007], [Hibernate Homepage 2007], [JBoss 2007] und [Beeger et al. 2006].

Hibernate wurde von der Firma JBoss™ im Jahr 2003 ins Leben gerufen, ist Teil der *JBoss Enterprise Middleware Suite* und beschreibt sich selbst wie folgt:

- Hibernate ist ein *Open Source*-Persistenz-Framework für Java.
- Hibernate befreit den Programmierer von der Programmierung von *SQL*-Abfragen und hält die Applikation unabhängig vom *SQL*-Dialekt der verwendeten Datenbank.
- Es werden *POJOs* eingesetzt, welche gewöhnliche Java-Objekte mit Attributen und Methoden darstellen.
- Die Beziehungen zwischen den Objekten werden auf das Datenbankschema abgebildet. Hibernate ist mit fast allen Datenbanktypen kompatibel und verwendet eine an *SQL* angelehnte mächtige Abfragesprache HQL.
- Hibernate arbeitet mit Reflection.

- Hibernate bietet umfangreiche Verwendungsmöglichkeiten (Standalone, in Applikationsservern und in *EJBs*).
- Hibernate hat sich zum de facto Standard von *ORM* entwickelt und Gavin King (Gründer von Hibernate) war auch maßgeblich an der *Spezifikation* des im Kapitel 3.1.4 beschriebenen Ansatzes *EJB* beteiligt.

Abbildung 5 zeigt einen Überblick über die Hibernate-Architektur.

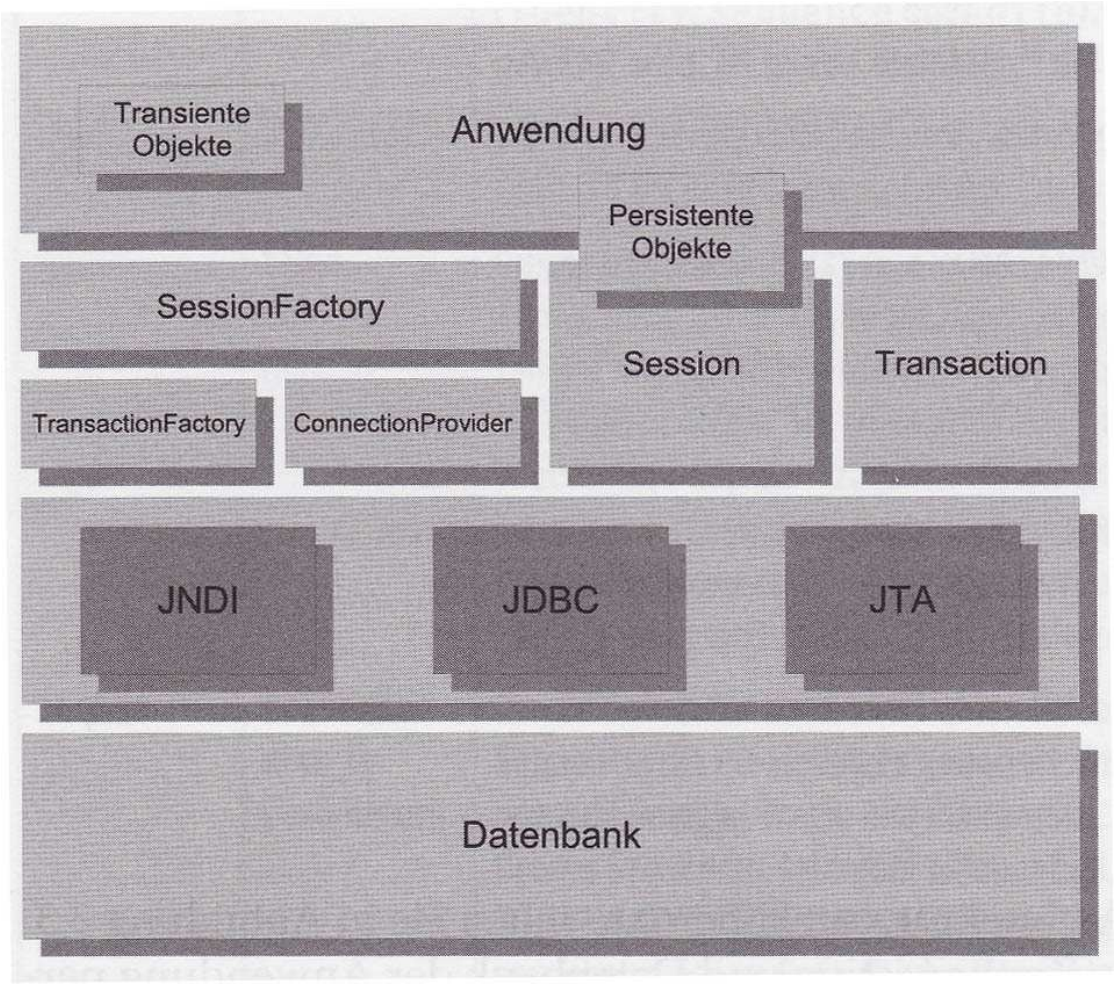

Abbildung 5: Hibernate-Architektur [Hien und Kehle 2007, S 62]

Hibernate verwendet verschiedene Konzepte. Abbildung 6 zeigt einen Überblick über das Kernstück von Hibernate, die Hibernate-*API*.

Persistente Klassen

Die persistenten Klassen spielen bei einem *ORM* eine zentrale Rolle, obwohl sie vom Aufbau her sehr einfach sind. Sie enthalten oftmals nur eine Sammlung von Eigenschaften und entsprechende Getter- und Setter-Methoden und folgen somit der Java Beans-*Spezifikation*. Allerdings müssen die Zugriffsmethoden im Gegensatz zu dieser *Spezifikation* nicht public sein, da Hibernate sie auch verwenden kann, wenn sie private sind.

Damit Hibernate die Klassen später auch instanziieren kann, müssen sie einen Default-Konstruktor besitzen. Diese Klassen enthalten die Daten, die in der Datenbank gespeichert werden sollen, und stellen somit das Domain-Modell dar. Bei den Klassen handelt es sich um einfache Java-Objekte, die von keiner Hibernate-Klasse abgeleitet werden müssen. [Langer et al. 2007] [Hien und Kehle 2007]

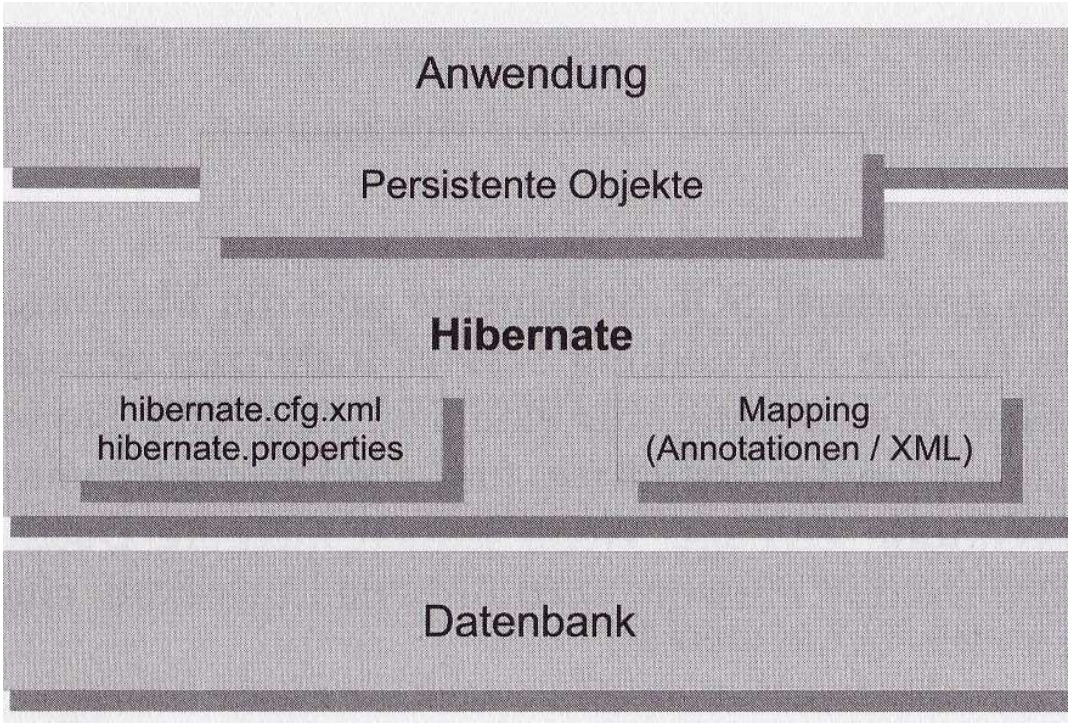

Abbildung 6: Überblick über das Hibernate *API* [Hien und Kehle 2007, S 61]

Hibernate Mapping

Das Mapping bildet die Brücke zwischen der Java- und der Datenbankwelt und bekommt dadurch die entscheidende Rolle innerhalb der Hibernate-Konfiguration. [Langer et al. 2007] [Hien und Kehle 2007]

Folgende Informationen werden im Mapping verwendet:

- Die Abbildung von Klassen auf Tabellen und umgekehrt.
- Die Abbildung von Attributen auf Spalten sowie Einschränkungen für Eigenschaften und Spalten, z.B. Not Null, Unique usw.
- Die Beziehungen zwischen den Klassen und wie diese mit Java repräsentiert werden sollen (als Objektreferenz, als Collection usw.).

Die Hibernate Tools, welche als Zusatzpaket verfügbar sind, verwenden diese Daten nicht nur für das eigentliche *ORM*, sondern auch zur Erstellung des Datenbankschemas bzw. zur Generierung von Java-Klassen.

Hibernate verwendet zwei Varianten für die Erstellung des Mappings. Die klassische Variante ist die Verwendung

- von *XML* Dateien (z.B. User.hbm.xml) oder
- die neuere Variante mit Verwendung der Java Persistence API (JPA),

welche im Kapitel 3.1.4 noch näher erläutert wird. Wenn das System gestartet wird, liest Hibernate die Mapping-Dateien ein und generiert zur Laufzeit *Byte-Code*, welcher für die transparente Persistenz sorgt.

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="objects">
  <class name="organisation" table="Organisation">
    <id name="id" column="orgid">
      <generator class="native"/>
    </id>
  </class>
</hibernate-mapping>
```

Listing 2: Beispiel für ein Mapping mittels einer *XML*-Datei

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@Table(name = "organisation")
@NamedQuery(name = "searchBOrganisationQuery",
    query = "from organisation where orgid like :orgid or orgname like :orgname ")
public class Organisation {
  Long orgid;

  public Organisation() { }
  @Id
  @GeneratedValue(generator = "hibSeq")
  @GenericGenerator(name = "hibSeq", strategy = "sequence")
  public Long getOrgId () { return id; }

  protected void setOrgId(Long id) { this.id = id; }
}
```

Listing 3: Beispiel für ein Mapping mittels JPA

Die Erstellung des Mapping mittels JPA stellt eine interessante Alternative dar.

Hibernate Session

Die Session ist das wichtigste *Interface* in einer Anwendung mit Hibernate. Sie verbraucht wenige Systemressourcen und ist einfach zu erzeugen. In Tabelle 1 [Langer et al. 2007] werden die wichtigsten Methoden der Session vorgestellt.

Neben der Funktion als Bindeglied zwischen der Datenbank und einer Anwendung, stellt die Session auch eine Factory für Transaktion-Instanzen bereit. Sie hält außerdem den *First-Level-Cache* und entdeckt automatisch Änderungen an Objekten. Eine konkrete Session-Instanz ist nicht *thread-safe* und darf nur von einem *Thread* verwendet werden.

Funktion	Methode	Beschreibung
SELECT	Session.load(Class, Object)	Liest ein Objekt anhand des Primärschlüssels aus der Datenbank.
SELECT	Session.createQuery(String)	Erstellt eine Abfrage.
INSERT	Session.save(Object)	Speichert ein neues Objekt in der Datenbank.
UPDATE	Session.merge(Object)	Aktualisiert ein Objekt.
DELETE	Session.delete(Object)	Löscht ein Objekt aus der Datenbank.
INSERT UPDATE DELETE	Session.flush()	Synchronisiert den Status der Objekte mit der Datenbank.

Tabelle 1: Methoden der Hibernate Session [Langer et al. 2007]

SessionFactory

Die SessionFactory hält in ihrem *Cache* generierte *SQL*-Statements und die Mapping-Daten. Mit der SessionFactory können für die Anwendung Session-Instanzen erzeugt werden. Typischerweise existiert nur eine SessionFactory, welche in der Erstellung sehr schwierig ist. [Hien und Kehle 2007] Sollten mehrere Datenbanken verwendet werden, ist die Erstellung einer zusätzlichen SessionFactory notwendig. Die SessionFactory kann als *thread-safe* und unveränderlich angesehen werden und wird beim Start der Anwendung einmalig instanziert und konfiguriert.

Für die Konfiguration verwendet die SessionFactory eine Datei, die sich im Klassenpfad befindet und entweder eine Property-Datei (z.B. hibernate.properties) oder eine *XML*-Datei (z.B. hibernate.cfg.xml) ist. [Langer et al. 2007] Die Initialisierung der SessionFactory erfolgt bei der Verwendung von JPA auf folgende Weise:

- Der erste Schritt ist das Laden der Konfiguration aus der Datei hibernate.cfg.xml:

```
AnnotationConfiguration configuration = new AnnotationConfiguration();
configuration.configure();
```

- Der Dateiname ist bei Hibernate auf hibernate.cfg.xml voreingestellt. Der nächste Schritt ist die Initialisierung der SessionFactory:

```
SessionFactory sessionFactory = configuration.buildSessionFactory();
```

Konfiguration

Die Konfiguration erlaubt der Anwendung die Verbindung zur Datenbank und das Mapping beim Erzeugen einer SessionFactory zu spezifizieren. Eine Konfiguration ist nur zur Initialisierungszeit von Hibernate relevant, nach der Erzeugung der SessionFactory kann keine Änderung mehr vorgenommen werden. Es wird auch keine Beziehung zur Konfiguration auf-

rechterhalten. [Hien und Kehle 2007] Wie im vorhergehenden Kapitel schon erwähnt werden dafür zwei Ansätze genutzt:

- Property-Datei: Ein nicht sehr oft verwendeter Ansatz zur Erstellung der Konfiguration mittels der Property-Datei hibernate.properties.
- *XML*-Datei: Der gebräuchliche Ansatz für die Erstellung einer Konfiguration mittels der *XML*-Datei hibernate.cfg.xml.

```
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
Einführung in Hibernate
<hibernate-configuration>
<session-factory>
<!-- Datenbank Connection Einstellungen -->
<property name="connection.driver_class">
org.hsqldb.jdbcDriver</property>
<property name="connection.url">jdbc:hsqldb:mem:pizza</property>
<property name="connection.username">martin</property>
<property name="connection.password">gtec</property>
<property name="hibernate.dialect">
org.hibernate.dialect.HSQLDialect</property>
<!-- Zusaetzliche Hibernate-Properties -->
<property name="hibernate.show_sql">true</property>
<property name="hibernate.format_sql">true</property>
<property name="hibernate.hbm2ddl.auto">create</property>

<!-- Auflistung der gemappten Klassen -->
<mapping class="objects.Organisation"/>

</session-factory>
</hibernate-configuration>
```

Listing 4: Beispiel für eine Konfiguration mittels *XML*-Datei

Transaction

Die Transaction abstrahiert die Anwendung von der *JDBC*-, *JTA*- oder Corba-Transaction. Eine Transaction ist ein Single-Threaded, kurzlebiges Objekt, das von der Anwendung benutzt wird, um atomare Abschnitte zu spezifizieren. [Hien und Kehle 2007]

ConnectionProvider

Der optionale ConnectionProvider ist eine Factory für Datenbankverbindungen. Er abstrahiert die Anwendungen von einer DataSource oder einem DriverManager. Der ConnectionProvider wird intern von Hibernate verwendet, um Datenverbindungen zu erhalten.

TransactionFactory

Eine ebenfalls optionale Factory für Instanzen von Transactions. Konkrete *Implementierungen* sind CMTTransactionFactory oder JDBCTransactionFactory. In der Konfiguration kann die gewünschte *Implementierung* festgelegt werden.

Lebenszyklus einer Hibernate Entity

Es ist wichtig den Lebenszyklus einer Hibernate Entity zu verstehen. Hibernate verwaltet diese Zustände transparent und der Entwickler hat dadurch nicht direkt damit zu tun. Für die Entwicklung einer Anwendung mit optimaler *Performance*, gilt es aber diesen Lebenszyklus zu beachten.

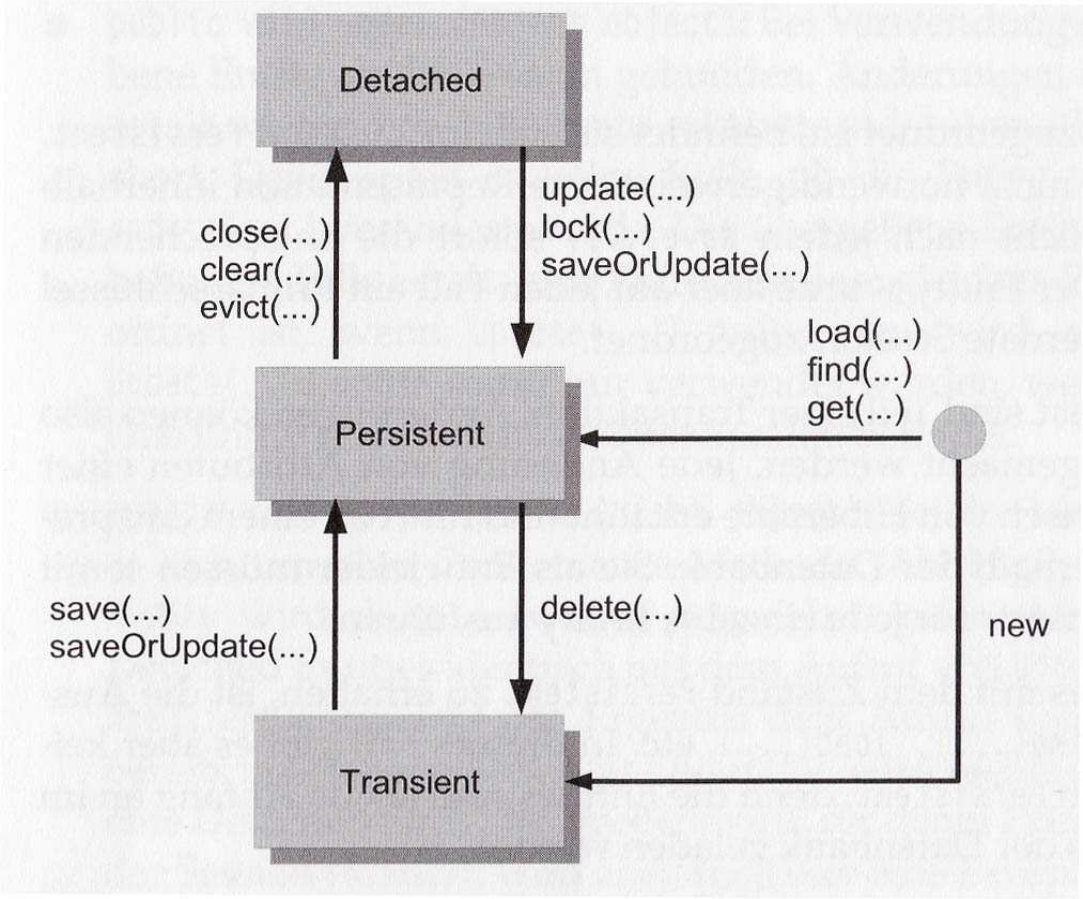

Abbildung 7: Zustände einer Hibernate Entity [Hien und Kehle 2007, S 65]

Eine Entity in Hibernate kann sich grundsätzlich in drei verschiedenen Zuständen befinden (siehe Abbildung 7):

- Die Objekte, die der Hibernate Session noch nicht bekannt sind, z.B. weil sie gerade mit dem new-Operator erzeugt worden sind, werden als transiente Objekte bezeichnet. Diese Objekte haben noch keine Datenbank-Repräsentation. Sobald diese Objekte nicht mehr von anderen Objekten referenziert werden, löscht der Garbage-Collector von Java diese Objekte und sie sind für immer verloren. Wird ein Objekt unter Zuhilfenahme der delete-Methode aus der Datenbank entfernt, verliert es seine Repräsentation in der Datenbank und wird wieder wie ein normales transientes Objekt behandelt. [Langer et al. 2007]

- Die Objekte, die Hibernate in der aktuellen Session bekannt sind, werden als *persistente Objekte* bezeichnet. Sie haben eine Repräsentation in der Datenbank und einen Wert für den Primärschlüssel. Der Zustand dieser Objekte wird von Hibernate verfolgt, das heißt, ein explizites Speichern von Änderungen ist nicht notwendig. Es entsteht ein *persistentes Objekt*, wenn z.B. ein neu angelegtes transientes Objekt mithilfe der save-Methode bei der Session persistiert wird. Alle Objekte, die als Ergebnis einer Abfrage von Hibernate geliefert werden, sind auch persistent. Am Ende einer Transaction bzw. spätestens beim Schließen einer Session wird ihr Zustand mit der Repräsentation in der Datenbank abgeglichen. [Langer et al. 2007] [Hien und Kehle 2007]
- Ein *persistentes Objekt* kann von der Hibernate Session wieder getrennt werden. In diesem Fall wird von einem detached Objekt gesprochen. Dabei kann zwischen dem expliziten Entfernen über die Session-Methode (evict) oder dem impliziten Trennen am Ende (close) der Session unterschieden werden. In beiden Fällen werden Objekte nicht mehr von Hibernate überwacht, das heißt, Änderungen an den Objekten führen nicht mehr automatisch zu einer Anpassung in der Datenbank. Diese Objekte können ohne Bedenken an ein Frontend übergeben werden. Das Besondere an detached Objekten ist die Tatsache, dass diese Objekte wieder mit einer anderen Hibernate Session verbunden werden können. Dafür steht in der Hibernate Session die Lock-Methode zur Verfügung, wodurch ein detached Objekt wieder unter die Kontrolle von Hibernate gelangt. Auch die Session-Methode update erzielt das gleiche Ergebnis, wobei hier das Aktualisieren des Objekts im Mittelpunkt steht. [Langer et al. 2007]

Hibernate-Abfragetechniken

Da es in Hibernate viele verschiedene Möglichkeiten gibt, Abfragen zu definieren, beinhaltet der folgende Abschnitt einen kurzen Überblick. Hibernate hat eine eigene Query Language, die Hibernate Query Language (kurz HQL).

- Ist angelehnt an *SQL* aber
- voll objektorientiert, versteht Vererbung, *Polymorphie* und Assoziationen und
- beinhaltet viele *Features* (from, select, where, having, order by, group by Aggregationsfunktionen etc.).

Query *Interface*

Das Query *Interface* stellt die zentrale Schnittstelle zur Ausführung von Abfragen dar. Sie kommt immer dann zum Einsatz, wenn die Primärschlüssel einer Entity nicht bekannt sind. [Hien und Kehle 2007]

Definition von Abfragen in den Metadaten

Es ist möglich, Abfragen in den Mapping-Metadaten, also entweder als Annotation (siehe Kapitel 6.5.1) oder in der *XML*-Mapping-Datei, zu definieren. [Hien und Kehle 2007]

CriteriaAPI

HQL ist nicht die einzige Möglichkeit, um Entities zu finden, deren Primärschlüssel nicht bekannt ist. Die CriteriaAPI ist eine sehr gute Alternative, welche ebenso einen objektorientierten Ansatz verfolgt. [Hien und Kehle 2007]

Natives *SQL*

Datenbankspezifische Funktionalitäten, die nicht von HQL oder CriteriaAPI unterstützt werden, benötigen natives *SQL*. Diese Abfragetechnik sollte allerdings so selten als möglich verwendet werden. [Hien und Kehle 2007]

Hibernate-Filter

Es besteht die Möglichkeit, die Ergebnismenge zu reduzieren, indem im Mapping ein Filter angelegt wird, ähnlich wie eine where-Bedingung in *SQL*. [Hien und Kehle 2007]

Zusammenfassung

Hibernate ist ein *Open Source* Ansatz. Es bietet eine gute Mapping Unterstützung und ein offenes *Caching-Interface*. Hibernate hat keine Basisklasse für zu persistierende Klassen, von der geerbt werden muss, und kein *Interface*, das erfüllt werden muss. HQL stellt eine sehr mächtige Abfragesprache dar. Durch die Fokussierung von *Sun*™ auf diesen Ansatz, ist Hibernate der Ansatz der Zukunft. Auch durch die Möglichkeit der Nutzung der JPA wird die Bedeutung von Hibernate zunehmen. Trotz der hervorragenden Dokumentation erweist sich die komplizierte Konfiguration für Neueinsteiger als schwierig.

3.1.4 Enterprise JavaBeans (*EJB*)

Enterprise JavaBeans (*EJB*) sind standardisierte Komponenten innerhalb eines *Java EE*-Servers (Java Enterprise Edition) und wurden ursprünglich von *IBM*™ im Jahre 1997 entwickelt. Sie vereinfachen die Entwicklung komplexer mehrschichtiger verteilter Softwaresysteme mittels Java. Mit *EJBs* können wichtige Konzepte für Unternehmensanwendungen (*EIS*), z.B. Transaktions-, Namens- oder Sicherheitsdienste umgesetzt werden, die für die Geschäftslogik einer Anwendung nötig sind. [Sun EJB 2007]

Folgende Informationen beruhen auf den Quellen [Sun EJB 2007], [JSR 220 2007], [Sun 2007] und [it-fws EJB 2007].

Abbildung 8 zeigt die Architektur von *EJB* 3.0 mit einem *Java EE* Application Server.

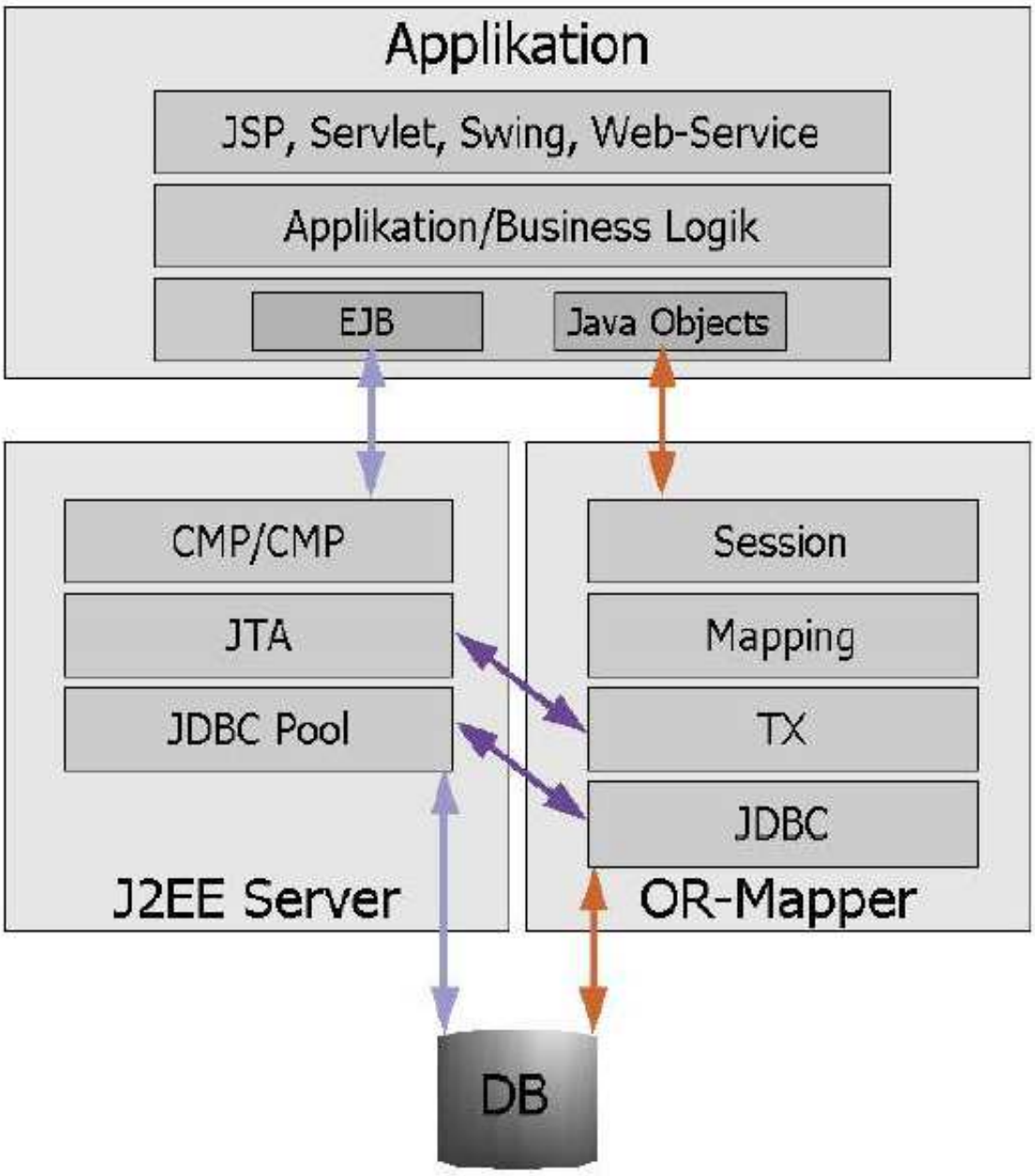

Abbildung 8: *EJB* 3.0 und *J2EE* Architektur [it-fws EJB 2007]

EJBs vor der Version 3.0 sind eine eher schwer zu erstellende Angelegenheit. Der Public Review Draft des JSR 220 beinhaltet tiefgreifende Änderungen. *EJB* 3.0 ist zwar abwärtskompatibel, von der alten Architektur bleibt jedoch nur noch ein Bruchteil bestehen. Um die Änderungen zu verstehen, werden im nächsten Abschnitt kurz die Komponenten und die Konfiguration vor der Version 3.0 vorgestellt.

Komponenten

Enterprise JavaBeans gibt es in mehreren unterschiedlichen Ausprägungen für verschiedene Klassen von Anwendungsfällen. Sie können entweder remote (entfernter Zugriff) oder lokal angesprochen werden.

Entity Bean

Entity Beans modellieren die persistenten Daten des Systems. Sie repräsentieren z.B. einen Datensatz aus einer Datenbank.

Die Persistenz kann entweder vom Bean-Entwickler selbst programmiert (*Bean Managed Persistence*, BMP) oder vom *EJB*-Container bereitgestellt werden (Container Managed Persistence, CMP).

Seit der Version 5 unterstützt *Java EE* ein Attachment, Detachment und Reattachment. Die Entity Bean ist nun ein *POJO*, dessen Persistenz mit Hilfe des Entity Managers gesteuert werden kann. Das bekannte *Java EE* Design Pattern Datentransferobjekt (englisch DataTransferObject, kurz: DTO) ist somit aus technischer Sicht nicht mehr erforderlich, da nun Geschäftsobjekte über verschiedene Schichten, beispielsweise zu einem Client, transportiert werden könnten. Datentransferobjekte dienen der *Abstraktion* von Geschäftsobjekten und der Entkopplung verschiedener Anwendungsschichten.

Session Bean

Session Beans bilden insbesondere Vorgänge ab, die der Nutzer mit dem System durchführt. Sie bedienen sich häufig mehrerer Entity Beans, um die Auswirkungen des Prozesses darzustellen.

Es wird zwischen zustandslosen und zustandsbehafteten Session Beans unterschieden.

Eine zustandsbehaftete Session Bean hat ein eigenes „Gedächtnis“. Sie kann Informationen aus einem Methodenaufruf speichern, damit diese bei einem späteren Aufruf einer anderen Methode wieder zur Verfügung stehen. Die Zustandsbehaftung wird durch die Vergabe einer eindeutigen ID umgesetzt, über diese können die zustandsbehafteten Session Beans unterschieden werden.

Im Gegensatz dazu müssen einer zustandslosen Session Bean bei jedem Aufruf alle Informationen als Parameter übergeben werden, die für die Abarbeitung dieses Aufrufs benötigt werden. Da eine zustandslose Session Bean keine Informationen speichern kann, ist sie nicht von anderen Session Beans der gleichen Klasse unterscheidbar. Sie hat also keine eigene Identität.

Message-Driven Bean

Message-Driven Beans sind diejenigen Komponenten, die *EJB*-Systeme für asynchrone Kommunikation zugänglich machen. Hierzu wird der Java Message Service (JMS) verwendet. Diese Sorte von Beans wird z.B. häufig für die Kommunikation mit *Legacy-Systemen* genutzt.

Der *EJB*-Standard schreibt neben den Enterprise JavaBeans auch die Verwendung von so genannten *Deployment Descriptoren* vor. Dieser *Deployment Descriptor* ist eine *XML*-Datei, in der die Eigenschaften von *EJBs* definiert werden, die nicht fix codiert werden. Folgende Eigenschaften werden in der *XML*-Datei definiert:

- Name, Klasse und Schnittstellen einer *EJB.*
- Informationen darüber, ob bestimmte Methoden unter bestimmten Arten von Transaktionen aufgerufen werden dürfen oder müssen.
- Referenzen auf Ressourcen, die der Bean vom Container bereitgestellt werden müssen, z.B. Datenquellen.
- Referenzen auf andere *EJBs* oder Webservices.
- Für Entity Beans mit Container Managed Persistence der Name ihres abstrakten Schemas sowie die Definition ihrer persistenten Felder und Beziehungen untereinander. Außerdem können Abfragen für bestimmte Suchmethoden (sogenannte Finders) definiert werden.

Die durch die Einführung der *EJB*-3.0-*Spezifikation* entstandenen Neuerungen werden im Folgenden kurz erläutert.

Die gesamte Konfiguration der Komponenten wird deutlich einfacher. Es müssen kaum noch *Deskriptoren* geschrieben werden, stattdessen werden direkt im Quelltext Annotationen verwendet, die dem Container die nötigen Informationen zur Verfügung stellen. Weiterhin ist die Standardeinstellung für die Kommunikation mit *EJBs* das Local- statt des Remoteinterfaces.

Auch das Testen ohne Container soll möglich werden. Die Verwendung von checked exceptions wurde deutlich reduziert. Damit entfallen die teils lästigen try/catch-Statements und der Code wird somit übersichtlicher.

Transparente *POJOs*

Bei den Entity Beans im Speziellen lässt sich ein durch Hibernate beeinflusster Trend hin zu *POJOs* erkennen. Der gesamte Persistenzmechanismus erfährt damit eine Verschiebung in Richtung *transparente Persistenz*. Es müssen keine bestimmten Voraussetzungen mehr ge-

geben sein, damit ein Objekt persistiert werden kann. Die Informationen, wie das *POJO* persistiert werden soll, werden dem Container über Annotationen innerhalb der JavaBeans mitgeteilt.

Default Konfiguration

EJB 3.0 verfolgt neben der Reduzierung der allgemein notwendigen Konfiguration des Weiteren den configuration by exception Ansatz. Darunter wird eine sinnvolle Vorbelegung aller relevanten Konfigurationsparameter für eine bestimmte Komponente verstanden. Es müssen dann nur noch Abweichungen explizit angegeben werden. Dies soll den Konfigurationsaufwand auf ein Minimum reduzieren.

Vererbung und Polymorphie

Vor *EJB* 3.0 war dies eines der Hauptkritikpunkte an Entity Beans, da ein mächtiger Hauptbestandteil der Objektorientierung nicht genutzt werden konnte.

Abfragesprache *EJB* QL

Die Abfragesprache *EJB* QL (Enterprise JavaBeans Query Language) wurde ebenfalls deutlich verbessert. So sollen neben batchupdates auch subqueries, outer joins und group-by Funktionalitäten angeboten werden. Dynamische Abfragen, sowie die Möglichkeit natives *SQL* abzusetzen, finden sich ebenfalls in der neuen *Spezifikation*.

Allgemein hat *EJB* QL eine Richtung ähnlich der Hibernate Query Language (HQL) eingeschlagen.

Detached Entities

Unter dem Begriff Detached Entities wird die zeitweilige Abkopplung eines persistenten Objektgraphen von der Datenbank verstanden. An dieser zunächst losgelösten Entität können dann Veränderungen vorgenommen werden. Später wird diese wieder an die Datenbank angekoppelt und synchronisiert.

Im Mai 2006 wurde der Final Release der *EJB*-3.0-*Spezifikation* (JSR-220) veröffentlicht. Vorrangiges Ziel der neuen *Spezifikation* war es, *Java EE* zu vereinfachen. Die Java Persistence *API (*JPA*)* wurde als eigenständiger Teil der *EJB*-3.0-*Spezifikation* entwickelt und löst die bekannten Entity Beans ab.

Wesentliche Neuerungen:

- Die Verwendung von Annotationen für das Mapping.
- Die Entities stellen einfache *POJOs* dar.

- Objektorientierte Klassenhierarchien mit Vererbung, Assoziationen, Polymorphismus usw. werden unterstützt.
- Die *API* ist nicht nur in den Java Enterprise-Umgebungen (Applikationsserver) einsetzbar, sondern auch in normalen Java-Standard Umgebungen lauffähig.

Mapping erfolgt in *EJB* 3.0 durch Meta-Annotationen. Meta-Annotationen sind dynamische Spracherweiterungen der Java-Programmiersprache (siehe Kapitel 6.5.1). Sie sind mit dem JDK 1.5 eingeführt worden. Meta-Annoationen ermöglichen es, direkt im Java Code Meta-Informationen unterzubringen, die sowohl zur Compile- als auch zur Laufzeit ausgewertet werden können. Diese Informationen können u.a. dafür genutzt werden, um Code zu generieren oder das Laufzeitverhalten eines Objektes bzw. einer Komponente zu steuern.

EJB 3.0 bewegt sich mit der Ausrichtung am *POJO*-Prinzip stark in die Richtung von Hibernate, die *Spezifikation* ist aber in einem noch frühen Stadium und birgt daher das Risiko einiger Änderungen. *EJB* lässt sich auch in Kombination mit Hibernate verwenden und dadurch gewinnt *EJB* zunehmend an Bedeutung. Dies liegt unter anderem sicher daran, dass Gavin King, der Begründer des Hibernate-Projekts, in die Spezifizierung von *EJB* 3.0 maßgeblich involviert ist.

3.1.5 JDBCPersistence

JDBCPersistence ist ein *ORM*-Framework. Es wurde für den Einsatz in Systemen, die hohe *Performance* benötigen, erfunden. Dieses Kapitel beruht auf [JDBCPersistence Homepage 2007]. In Abbildung 9 ist die gute *Performance* von JDBCPersistence für alle Operationen (lesen, schreiben, aktualisieren, suchen und speichern) dargestellt.
Für den Einsatz von JDBCPersistence benötigt der Entwickler nur Wissen über *SQL* und *JDBC API*s und er wird von zahlreichen Tools bei der Entwicklung unterstützt.
Programmierer können rasch Prototypen von Objekten erzeugen, indem sie die Objekte als Java *Interfaces implementieren* und das Framework dann während der Laufzeit *Implementierungen* dieser Objekte kreiert. Dadurch kann ein *Interface* genutzt werden, sobald es benötigt wird. Ein weiterer Vorteil liegt in der einfachen Erstellung der Beziehungen gegenüber anderen typischen *ORMs*.
JDBCPersistence verwendet Postprozessing, um *Byte-Code* für die Erstellung von Klassen, welche die Logik enthalten, sowie persistente Daten zu erhalten. Die Generierung des *Byte-Codes* geschieht zur Laufzeit. Für die *Byte-Code*-Generierung wird die neue Technik *ASM* [ASM 2007] verwendet, welche ab der Version Java 5 eingesetzt wird.

Die Mapping-Informationen werden in ausführbaren *Byte-Code* kompiliert, welcher wie ein *ORM*-Sprachen-*Compiler* funktioniert.

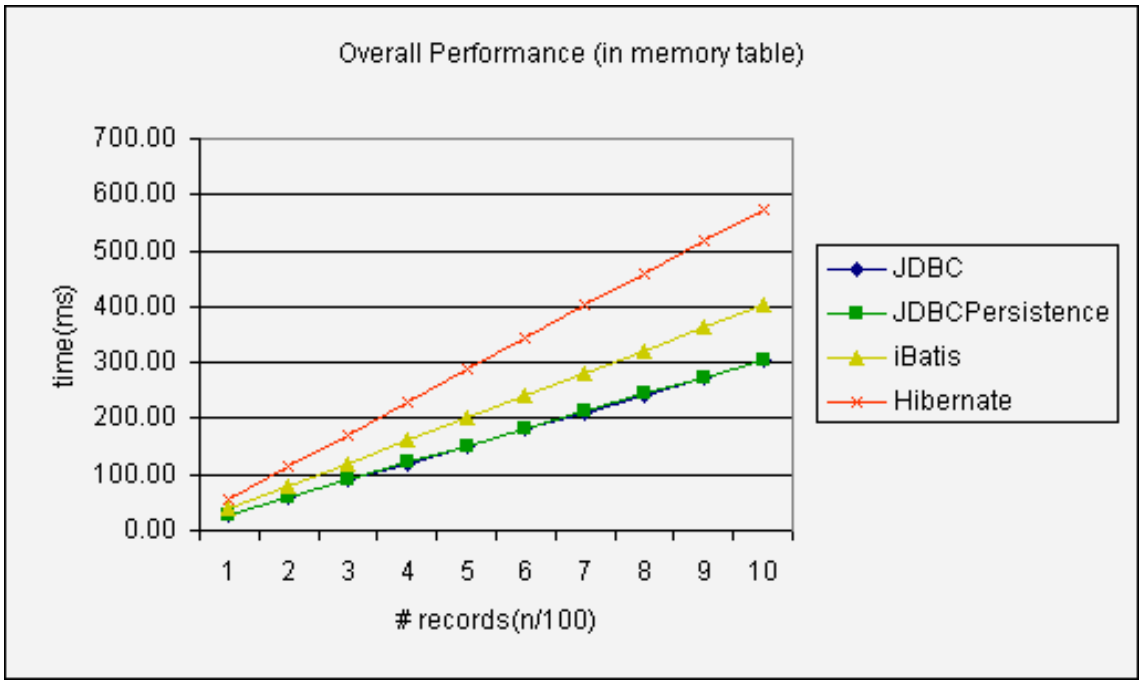

Abbildung 9: *Performance* der JDBCPersistence [JDBCPersistence Homepage 2007]

Der *Compiler* erstellt die Klassen während der Laufzeit, wenn eine *ORM* Mapping-Klasse benötigt wird. Jede Klasse, die durch JDBCPersistence generiert wird, ist eine *Implementierung* des JDBCPersistor-*Interface*, welches eine Bean zu einer Tabelle abbildet. Das *Interface* definiert Methoden für insert, update, delete, batch Insert, batch Update, batch Delete und findbyPK und läd die Daten in JavaBeans.
JDBCPersistence ist eine neue Technik für Anwendungen, die hohe Ansprüche an die *Performance* stellen. Der Vorteil liegt in der zielgerichteten, einfachen *Implementierung* und der erreichbaren *Performance*. Da dieser Ansatz noch sehr neu ist, muss die schlechte Dokumentation in Kauf genommen werden.

3.1.6 TopLink

TopLink ist ein objektrelationales Mapping Package für Java-Entwickler und wurde von Oracle™ im Jahre 1996 entwickelt. Es stellt ein umfangreiches und flexibles Framework für die Speicherung von Java-Objekten in *RDB* oder zur Konvertierung von Java-Objekten zu *XML*-Dateien zur Verfügung. [Oracle TopLink 2007]

In der aktuellen Version verwendet nun auch TopLink die *EJB*-3.0-*Spezifikation*, die schon in Kapitel 3.1.4 beschrieben wurde. TopLink ist neuerdings ein *Open Source* Produkt.

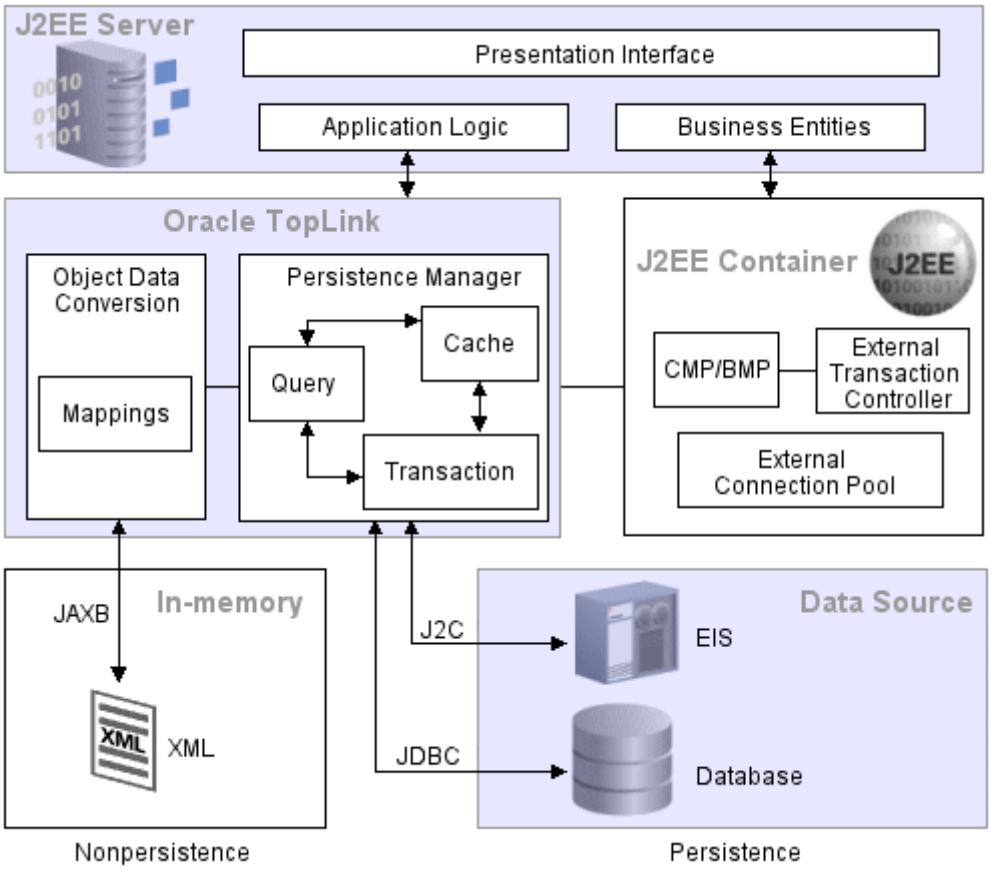

Abbildung 10: TopLink-Architektur [Oracle TopLink 2007]

Abbildung 10 zeigt, wie TopLink in eine typische *J2EE*-Architektur integriert ist. Diese besteht aus einer Datenbank, einem optionalen *J2EE*-Container, TopLink und einem *J2EE* Server. TopLink beruht auf einer Session-FrontEnd-Lösung und einer Datenzugriff-BackEnd-Lösung. Um auf TopLink mittels einer Session und Datenzugriffkomponenten zugreifen zu können, werden Mapping-Dateien, ein Abfrage Framework, ein *Cache*, Transaktionskomponenten und Client Anwendungen benötigt. Die Session bewerkstelligt den Zugriff auf das Abfrage-Framework und auf die Transaktionskomponenten. Beide Komponenten beziehen Nutzen aus dem *Cache*, um die Zugriffe auf die Datenbank zu minimieren. In einer *J2EE*-Container-Architektur können die Transaktionskomponenten durch *JTA* integriert werden. Die Datenzugriffkomponenten nutzen *JDBC* für den Zugriff auf die Datenbank. TopLink nutzt die *Bean-Managed-Persistence.*

Das Mapping erfolgt bei TopLink entweder durch *XML*-Dateien *(Deskriptoren)* oder durch *EJB* 3.0 (JPA). Ein *Deskriptor* enthält im *XML*-Format Informationen für die Generierung und Ausführung von Java-Komponenten zur Lauf- und Kompilierungszeit. Hierbei handelt es sich um *deklarative Programmierung*, da der Entwickler *Features* durch einfaches Angeben bestimmter Schlüsselwörter innerhalb eines *Deskriptors* erreichen kann, ohne programmieren zu müssen.

Die Metadatei session.xml beinhaltet das Login zur verwendeten Datenbank.

Eine TopLink Session beinhaltet eine Referenz zu einer speziellen Metadatei project.xml. Diese Datei wird von der Anwendung genutzt, um auf die Ausstattung der TopLink-Laufzeitumgebung zugreifen zu können.

Es kommen je nach Anwendungstyp (CMP Anwendung, Java-Anwendung) verschiedene *Deskriptoren* zum Einsatz.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
        xmlns="urn:digipark-example"
        targetNamespace="urn:digipark-example"
        elementFormDefault="unqualified">
        <xs:element name="digipark">
                <xs:complexType>
                        <xs:sequence>
                                <xs:element name="organisation">
                                        <xs:complexType>
                                                <xs:sequence>
                                                        <xs:element name="orgid" type="xs:int"/>
                                                        <xs:element name="orgname"
                                                        type="xs:string"></xs:element>
                                                </xs:sequence>
                                        </xs:complexType>
                                </xs:element>
                                <xs:element ref="contact"></xs:element>
                        </xs:sequence>
                </xs:complexType>
        </xs:element>
</xs:schema>
```

Listing 5: Beispiel für Mapping mittels *XML*-Datei

TopLink verwendet für alle Einstellungen das *Deskriptoren*-Konzept. Vertiefende Informationen können unter der Url: http://www.oracle.com/technology/-products/ias/toplink/doc bezogen werden.

TopLink verwendet folgende Abfragekonzepte:

- Session Queries: Eine Abfrage wird implizit erstellt und durch eine Session mit Eingabeparametern durchgeführt.
- Database Queries: Ein Fall einer Datenbankabfrage, welche kreiert und ausgeführt wird, um eine Datenbankaktion durchzuführen.
- Named Queries: Eine Datenbankabfrage, welche mit einem Namen in einer Session oder in einem *Deskriptor* erzeugt und gespeichert wird. Aufruf erfolgt über den Namen zur Laufzeit.
- Call Queries: Ein Fall eines Aufrufes, der entworfen wird und dann unter Verwendung einer speziellen Session *API* durchgeführt wird. TopLink unterstützt Aufrufe für gewöhnliche *SQL*-Abfragen, gespeicherte Prozeduren und *EIS*-Interaktionen.
- Historical Queries: Jede Abfrage wird im Kontext einer schon beendeten Session, unter Verwendung des time-aware features des TopLink-Frameworks, durchgeführt.
- *Interface* and Inheritance Queries: Jede Abfrage welche ein *Interface*, eine Superklasse oder Teilklasse von einer vererbten Hierarchie referenziert.

- Descriptor Query Manager Queries: Der Descriptor Query Manager definiert eine Standard Datenbankabfrage für jede grundsätzliche Datenbankoperation (erzeugen, lesen, aktualisieren und löschen).
- *EJB*-Finders: Eine Abfrage mit dem *Interface* von *EJB*, welche *EJBs* returniert. Solche Abfragen werden mit dem TopLink-Abfragetyp erstellt. Dazu wird die *EJB* QL verwendet.

TopLink stellt einen interessanten Ansatz dar. Das *Deskriptoren*-Konzept erscheint auf den ersten Blick sehr komplex, kann aber über Editoren einfach erstellt werden. TopLink verfügt auch über zahlreiche Abfragemöglichkeiten. Durch die zusätzliche Möglichkeit JPA zu verwenden ist TopLink ein interessanter Ansatz.

3.1.7 Java (SimpleORM)

Folgende Informationen beruhen auf den Quellen [Dr.Dobb's Portal 2007] und [SimpleORM Homepage 2007].

Dieser Ansatz bietet eine einfache Alternative zu den vorher genannten. Ein kurzer Überblick über die Schwerpunkte dieses Ansatzes verdeutlicht die Einfachheit:

- SimpleORM ist ein *Open Source* Java *ORM*, bietet eine einfache aber effektive *Implementierung* eines *ORM* aufgesetzt auf *JDBC* und es benötigt keine *XML*-Dateien für die Konfiguration. SimpleORM verwendet nur Java für die Umsetzung, minimale Reflektion, keine Vorprozessierung und kein *Byte-Code*-Postprocessing. Dieses einfache und simple Framework ist leicht anzuwenden.
- SimpleORM bezieht sich stark auf die Datenbanksemantik.
- Alle abgebildeten Objekte der Datenbank werden durch das SRecordInstance object *Interface* bearbeitet. Im Gegensatz zu Hibernate und JDO setzt es nicht auf *transparente Persistenz*.
- SimpleORM verwendet nur wenig Datenbank abhängigen Code, sondern verwendet eigene und erweiterbare Datenbanktreiber. Die meisten Datenbanken werden bereits unterstützt und die Verwendung neuer Datenbanken ist einfach.

Das Beispiel Listing 6 soll die Anbindung einer Datenbank verdeutlichen:

```
// organisation class mapped to a table organisation.
public class Organisation extends SRecordInstance {

  private static final SRecordMeta meta
    = new SRecordMeta(Organisation.class, "Organisation");
    // SRecordMeta objects describe SRecordInstances

  public static final SFieldString orgid
    = new SFieldString(meta, "orgid", 20, SFD_PRIMARY_KEY);

  public static final SFieldString orgname
    = new SFieldString(meta, "orgname", 40, SFD_DESCRIPTIVE);


  public SRecordMeta getMeta() { return meta };
    // (Abstract SRecordInstance method)

  public static Organisation findOrCreate(String orgid) { // Convenience
    return (Organisation)meta.findOrCreate(orgid);

  // Completely optional get/set methods.
  String getOrgName() {
     return getString(orgname);
  }
  String setOrgName(String value) {
     setString(orgname, value);  }
}
```

Listing 6: Beispiel für SimpleORM

Die Klasse Organisation repräsentiert die *SQL*-Tabelle organisation. Die Variablen orgid (Primärschlüssel) und orgname beziehen sich auf *SQL*-Spalten.

Daten können wie folgt erhalten werden:

```
String orgname = organisation.getString(Organisation.orgname);
```

Alternative, wenn eine Get-Methode definiert wurde:

```
String orgname = organisation.getOrgName();
```

Mit dem Teilprojekt „SimpleWeb“ von SimpleORM wurde eine einfache Möglichkeit geschaffen, Benutzeroberflächenkomponenten mit SimpleORM-Objekten zu mappen.

SimpleORM verwendet keine eigene Abfragesprache wie HQL sondern setzt auf *SQL*. Es gibt die Möglichkeit Daten über die Abfragetechnik findOrCreate(...) abzurufen. Diese Abfragetechnik bezieht sich auf die Gleichheit der Primärschlüssel, bietet aber auch die Möglichkeit auf Dateninhalte zuzugreifen. Ein kurzes Beispiel soll diese Abfragetechnik veranschaulichen, indem aus der Relation Organisation alle Organisationen abgefragt werden, deren Name mit „GTEC“ beginnt und die einen Kontakt beinhalten der „Bucher“ heißt.

```
SResultSet res = Organisation.meta.newQuery()
  .gt(Organisation.orgname, "'GTEC'")
  // .and() implied.  .or() etc. override.
  .eq(Contact.connachname, "'Bucher'"))
  .decending(Contact.conorgname)
  .execute();
```

Listing 7: SimpleORM findOrCreate(...)

Dieser Ansatz bietet die einfachste Art eines *ORM*. SimpleORM ist für geübte Java-Programmierer einfach zu *implementieren* und verlangt zusätzlich nur Kenntnisse in *SQL*. Es wird kein Persistenz Framework eingesetzt und keine *transparente Persistenz*. SimpleORM bietet überdies einfache Möglichkeiten für Forward Engineering und Reverse Engineering an. Die *Performance* von SimpleORM ist im Vergleich zur normalen *JDBC*-Anbindung sehr gut. Auch alle Abfragen weisen eine sehr gute *Performance* auf. Für vertiefende Informationen zu diesem Thema empfiehlt sich die Recherche von [SimpleORM Homepage 2007]. Durch „SimpleWeb“ (Teilprojekt von SimpleORM) ist es sehr einfach, Benutzeroberflächen zu erstellen. Der größte Vorteil von SimpleORM ist die einfache *Implementierung*. Es sind nur wenige zusätzliche Codezeilen notwendig, um diesen einfachen *ORM* zu erstellen.

3.2 Beispiel für einen Ansatz mit *PHP* 5 ohne Java

Als Beispiel für einen Ansatz ohne Java wird in diesem Kapitel auf *PHP* 5 mit objektorientierter Programmierung und weiters auf Propel und Creole eingegangen. BERGMANN definiert *PHP* wie folgt:

„PHP ist eine dynamische getypte Programmiersprache, die sowohl prozedurale als auch objektorientierte Programmierung unterstützt. Für Letztere bietet PHP ein klassenbasiertes Objektmodell an, das sich stark an das von Java anlehnt. Ein objektorientiertes PHP-Programm besteht aus einer Menge von PHP-Klassen, die den Regeln der PHP-Programmiersprache genügen und vom PHP-Interpreter übersetzt und ausgeführt werden.“ [Bergmann 2005, S 3]

Abbildung 11 zeigt einen Überblick über die Propel und Creole Architektur.

Zuerst wird die Datenbankabstraktionslösung Creole vorgestellt, welche die Grundlage für Propel bietet.

Creole ist eine an *JDBC* angelehnte Datenbankabstraktionslösung und ermöglicht die vollständig objektorientierte, von dem Datenbanksystem unabhängige Programmierung von Datenbankanwendungen mit *PHP*. Creole bietet eine an die Programmierschnittstelle von Hibernate angelehnte Möglichkeit für die *Introspektion* von Datenbanken an. [Bergmann 2005]

Mittels Creole können lesende und schreibende Datenbankabfragen unter Verwendung von *SQL* in verschiedenen Varianten durchgeführt werden:

- Verwendung der Methode executeQuery($SQL).
- Verwendung des ResultSet-Objektes mit einem foreach Operator.
- Verwendung eines Statement-Objektes.

- Verwendung der Creole Schnittstelle PreparedStatement.

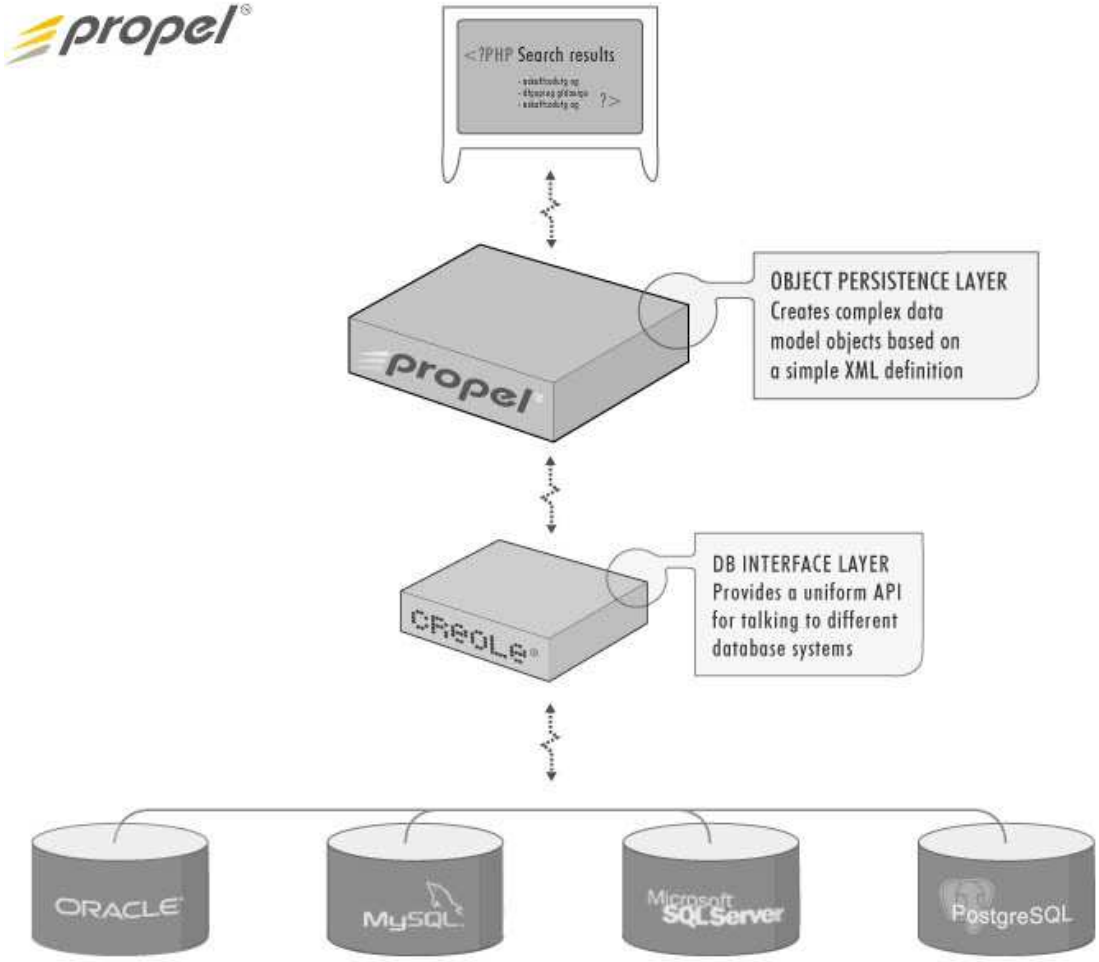

Abbildung 11: Propel- und Creole-Architektur [Propel Homepage 2007]

In Zukunft ist geplant, die Creole Datenbankabstraktionslösung durch eine auf JDO basierende Datenbankabstraktionslösung zu ersetzen. [Propel Homepage 2007]

Propel ist eine an Apache Torque [Torque 2007] angelehnte Lösung für die Abbildung von *PHP*-Klassen in einer *RDB*. Sie ermöglicht zum einen das Speichern von *PHP*-Objekten in einer Datenbank und umgekehrt die Kapselung von Datenbankinhalten durch *PHP*-Objekte. Propel dient hier als Persistence Layer (siehe Abbildung 11) zwischen der Datenbank und den *PHP*-Objekten. Der Programmierer der Anwendung muss keine *SQL*-Abfragen direkt an den Datenbankserver richten, da diese von Propel automatisch erzeugt werden. Somit ist bei einem Wechsel von einem *DBMS* auf ein anderes nur die Konfigurationsdatei anzupassen. [Bergmann 2005]

Propel besteht aus zwei Komponenten:

- Mit dem Propel-Generator werden aus einer *XML-Spezifikation* des Datenmodells die entsprechenden *PHP*-Klassen generiert. Es wird auch eine Datei erzeugt, welche *SQL*-

Anweisungen für die Erzeugung der Datenbank und der darin enthaltenen Tabellen enthält. [Propel Homepage 2007] [Bergmann 2005]

- Die Propel-Laufzeitumgebung bildet die Grundlage für die Verwendung der durch den Propel-Generator generierten *PHP*-Klassen.

Propel benötigt neben dem schon beschriebenen Creole auch noch die Pakete Pear::Log³ und Phing. Phing ist eine *PHP*-Portierung des in der Java-Welt beliebten Build-Werkzeugs Apache *Ant*, die für den Propel Generator benötigt wird. Pear::Log³ entspricht dem log4J (Apache™) von Java in Kombination mit Hibernate. [Propel Homepage 2007]

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<database name="digipark" defaultIdMethod="native">
  <table name="organistation" description="tabelle organisation">
    <column name="orgid" required="true" primaryKey="true"
            type="INTEGER" description="Organisation-ID"/>
    <column name="orgname" required="true"
            type="VARCHAR" size="100" description="Name der Organisation"/>
  </table>
</database>
```

Listing 8: Propel Spezifikation des Datenmodells in *XML*

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<config>
  <log>
    <ident>propel-organisation</ident>
    <level>7</level>
  </log>
  <propel>
    <datasources default="organisation">
      <datasource id="organisation">
      <adapter>mysql</adapter>
      <connection>
        <phptype>mysql</phptype>
        <hostspec>localhost</hostspec>
        <database>organisation</database>
        <username>martin</username>
        <password>gtec</password>
      </connection>
      </datasource>
    </datasources>
  </propel>
</config>
```

Listing 9: Propel Konfiguration des Objektspeichers in *XML*

Mittels Propel-Generator können nun die für die Erzeugung der entsprechenden Datenbank und Tabellen benötigten *SQL*-Anweisungen, sowie die entsprechenden *PHP*-Klassen erzeugt werden. Der Propel-Generator erzeugt für jede Tabelle eine abstrakte Basisklasse, die sich von der abstrakten Klasse BaseObject der Propel-Laufzeitumgebung ableitet, sowie eine leere Instanz der Subklasse hiervon. [Bergmann 2005]

Propel stellt auch verschiedene Möglichkeiten der Abfrage von Daten zur Verfügung:

- Zugriff über die Methode retrieveByPK().
- Zugriff über die Klasse Criteria sowie die Methode doSelect().

Propel und *PHP* 5 stellen eine sehr gute Alternative dar und es empfiehlt sich der Einsatz in allen Bereichen mit webbasierten Lösungen, da in diesen Bereichen *PHP* und MySql-Datenbanken oft in Kombination verwendet werden. Dieser Ansatz ist dadurch als *ORM* sehr interessant.

3.3 Produkte

Dieses Kapitel beinhaltet einen Überblick über die *ORM*-Produkte. Die Quellen für den Download der Produkte werden aufgrund der besseren Übersichtlichkeit direkt im Text angegeben.

3.3.1 Hibernate

Hibernate basiert auf dem Ansatz Hibernate Framework (siehe Kapitel 3.1.3) und wurde von Gavin King entwickelt. Hibernate ist heute Teil der *JBoss Enterprise Middleware Suite* und ein *Open Source* Produkt. Unter dem *Link* http://www.hibernate.org/ kann die jeweils passende Version bezogen werden.

Hibernate bietet sehr viele Tools für Reverse Engineering und Forward Engineering an, wobei hier nur die wichtigsten Tools vorgestellt werden.

Hibernate Tools

Es wird ein *Toolset* zur Verfügung gestellt, welches einem Anwender die automatisierte Erstellung von Hibernate-Anwendungen ermöglicht (siehe http://www.hibernate.org/255.html). Das *Toolset* besteht aus *Ant*-Tasks und einer Integration in die Entwicklungsumgebung Eclipse. Die Tools können auch über den Java-QuellCode genutzt werden [Hien und Kehle 2007] und können über den *Link* http://tools.hibernate.org unter Verwendung des Eclipse-Update-Managers oder manuell installiert werden.

Um die Funktionsweise der *Exporter* zu verstehen, wird nun der Entwicklungsprozess einer Hibernate-Anwendung in Zusammenhang mit der Verwendung der Tools (*Exporter*) anhand der 4 Strategien vorgestellt:

- Top-Down: Es wird mit der Erstellung des Objektmodells begonnen, also der *Implementierung* der *POJOs.* Sollten *EJBs* nicht verwendet werden, werden im nächsten Schritt Mapping-Dateien erstellt. Aus den *POJOs* oder Mapping-Dateien wird dann anschließend ein Datenbankschema erzeugt. Um aus den Mapping-Dateien oder aus den *POJOs* mit *EJB* das Datenbankschema zu generieren, kann der *Exporter* <hbm2ddl> verwendet werden. Diese Strategie ist für die Entwicklung von neuen

Anwendungen sehr gut geeignet, wenn noch kein Datenbankschema vorliegt. [Hien und Kehle 2007]

- Bottom-Up: Im ersten Schritt wird das Datenbankschema, sollte noch keines bestehen, erstellt und im nächsten Schritt werden die Mapping-Dateien erzeugt. Aus diesen Mapping-Dateien können mittels des Einsatzes des *Exporters* <hbm2.java> die *POJOs*, also die Java-Quelldateien, generiert werden. Diese Strategie empfiehlt sich, wenn Hibernate in Zusammenhang mit einem schon vorhandenen Datenbankschema genutzt werden soll. [Hien und Kehle 2007]
- Middle-Out: Im ersten Schritt werden die Mapping-Dateien erstellt, dann unter Einsatz des *Exporters* <hbm2ddl> das Datenbankschema generiert und anschließend mittels des *Exporters* <hbm2.java> die Java-Quelldateien generiert. Diese Strategie wird selten verwendet, da die Erstellung der Mapping-Dateien bei der Entwicklung einer Hibernate-Anwendung nicht im Vordergrund steht. [Hien und Kehle 2007]
- Meet-in-the-Middle: Im ersten Schritt werden aus den Java-Quelldateien und aus dem Datenbankschema die Mapping-Dateien generiert. Diese Strategie eignet sich sehr gut für den Umbau einer bereits bestehenden Datenbankanbindung z.B. mittels *JDBC* zu einer Hibernate-Anwendung. Bei dieser Strategie ist unklar, ob der Java-Quellcode oder das Datenbankschema angepasst wird. Diese Strategie wird allerdings sehr selten verwendet. [Hien und Kehle 2007]

Welche Strategie verfolgt wird, hängt sehr stark von den Voraussetzungen eines Projekts ab. Die *Exporter* werden in zahlreichen Tools für die Entwicklung von Hibernate-Anwendungen eingesetzt. Die zwei wichtigsten Vertreter werden im folgenden Absatz vorgestellt.

MyEclipse

MyEclipse in der Version 5.0 GA der Firma Genuitec™ [MyEclipse 2007] ist eine kommerzielle Erweiterung der *Open Source* Entwicklungsumgebung Eclipse, beinhaltet zahlreiche nützliche *Features* und fasst diese im Hibernate Capabilities Wizard zusammen. Folgende zusätzliche Funktionalitäten im Zusammenhang mit Hibernate werden angeboten [MyEclipse 2007]:

- Anlegen von Hibernate-Projekten.
- Erzeugt myhibernatedata-Datei im Projekt Wurzelverzeichnis.
- Editor für die Erstellung der Konfiguration (hibernate.cfg.xml).
- Kopiert Datenbanktreiber zur Projektbibliothek.
- Die Hibernate-Code Generierung.
- Installation der für Hibernate benötigten Klassenbibliotheken.

- Generiert *EJBs* und Data-Access-Objects (*DAO*).

Die Hibernate-Code Generierung ist sehr hilfreich, da sie die Generierung von Java-Klassen mit JPA aus dem Datenbankschema ermöglicht und dabei automatisch die in Hibernate integrierten *Exporter* einsetzt. Durch den Einsatz von *MyEclipse* wird die Konfiguration eines Hibernate-Projekts sehr erleichtert.

Middlegen for Hibernate

Middlegen in der Version 2.1 der Firma boss™ (*Link*: http://boss.bekk.no/boss/middlegen/) ist ein *Open Source* Framework für die Generierung von Code. Es verwendet dafür die schon bekannte *Ant*-Task Lösung und bietet folgende Funktionalität an [Middlegen Homepage 2007]:

- Generiert Mapping-Dateien und Java-Quellcode.
- Bietet einen nahezu vollständigen Hibernate Support.
- Generiert die Konfiguration (hibernate.cfg.xml).
- Ermöglich den Einsatz aller *Exporter*.
- Middlegen generiert automatisch Annotationen *(EJB).*

Im praktischen Einsatz hat sich das Tool von *MyEclipse* sehr bewährt. Die generierten Quelldateien weisen wenige Fehler im Vergleich zu Middlegen auf und konnten mit wenig Aufwand ausgebessert werden. Die zusätzliche Generierung von Annotationen (*EJB*) kann bei Middlegen und *MyEclipse* gut genutzt werden. Ein weiterer Vorteil von *MyEclipse* ist der einfachere Einsatz, es müssen keine *Ant*-Tasks geschrieben bzw. modifiziert werden. *MyEclipse* bietet sehr gute Funktionalität im Bereich der Bottom-Up Strategie an. Der Einsatz der *Exporter* ohne Middlegen und *MyEclipse* hat sich sehr bewährt.

3.3.2 WebObjects

WebObjects basiert auf dem Ansatz EOF (siehe Kapitel 3.1.2) und wurde von NEXT™, heute Teil von *Apple*™, entwickelt. WebObjects ist ein kommerzielles Produkt und kann unter dem *Link* http://www.apple.com/support/webobjects/ bezogen werden. Das Herzstück von WebObjects ist der EOModeler, mit dem EOF effizient *implementiert* werden kann. [Apple WebObjects 2007]

Abbildung 12 zeigt eine Übersicht über das Produkt WebObjects.

Abbildung 12: WebObjects-Produktübersicht [GNUstep Homepage 2007]

WebObjects Tools

Der EOModeler stellt laut Apache™ [GNUstep Homepage 2007] folgende Funktionen zur Verfügung:

- Erzeugt Datenmodelle von einem schon existierenden Datenbankschema (Reverse Engineering).
- Ermöglicht das Festlegen von Beziehungen zwischen Relationen.
- Ermöglicht das Anlegen von Tabellen und Tabelleneinträgen.
- Generiert *SQL* von einem Datenmodell, um ein Datenbankschema zu erzeugen oder zu aktualisieren.
- Generiert Java-Klassen aus den erzeugten Datenmodellen.
- Synchronisiert Änderungen zwischen Datenbankschema und Daten-Modell.

Durch diese Möglichkeiten wird der Entwickler beim Erzeugen einer Anwendung von WebObjects optimal unterstützt.

Durch das Eclipse-Plugin WOLips kann die Erstellung einer WebObjects-Anwendung sehr erleichtert werden. Das Plugin kann unter dem *Link* http://wiki.objectstyle.org/confluence/display/WOL/WOLips bezogen werden. Da WebObjects ein bereits sehr ausgereiftes Produkt ist, ist es für Firmen, die professionell Web-Anwendungen oder Java-Anwendungen bauen, sehr interessant.

3.3.3 JPOX

Auf dem Ansatz JDO basieren zahlreiche Produkte. Stellvertretend für die gesamte Palette an Produkten wird JPOX von Apache™ beschrieben. Dies ist vor allem deswegen von Bedeutung, da Apache™ den Ansatz JDO nach dem Ausstieg von *Sun*™ weiterführt. JPOX ist ein *Open Source* Produkt. Unter dem *Link* http://www.jpox.org/docs/download.html kann die jeweils passende Version bezogen werden.

JPOX unterstützt die JDO-*Spezifikation* (JDO1, JDO2) und wird ab Version 1.2 auch Annotationen (*EJB*) unterstützen.

JPOX bietet die in Tabelle 2 ersichtlichen Abfragesprachen an:

Abfragesprache	JDO-*Spezifikation*	JPOX 1.0	JPOX 1.1	JPOX 1.2
JDO Query Language	JDO1, JDO2	✓	✓	✓
JDOQL Single-String Query	JDO2		✓	✓
SQL Query Language	JDO2, JPA1	✓	✓	✓
JPA Query Language	JPA1			✓
Named Queries	JDO2, JPA1		✓	✓
Extensible JDOQL Query Language			✓	✓

Tabelle 2: JBOX-Abfragesprachen [JPOX Homepage 2007]

Der Entwicklungsprozess und die Funktionsweise einer JPOX-Anwendung werden in der Abbildung 13 verdeutlicht.

Der Prozess der Erzeugung von persistenten Daten ist bei JPOX ein transparenter Prozess. Ab Version 1.2 werden *POJOs* verwendet, die mit Hilfe von Metadateien oder Annotationen (*EJB*) persistiert werden.

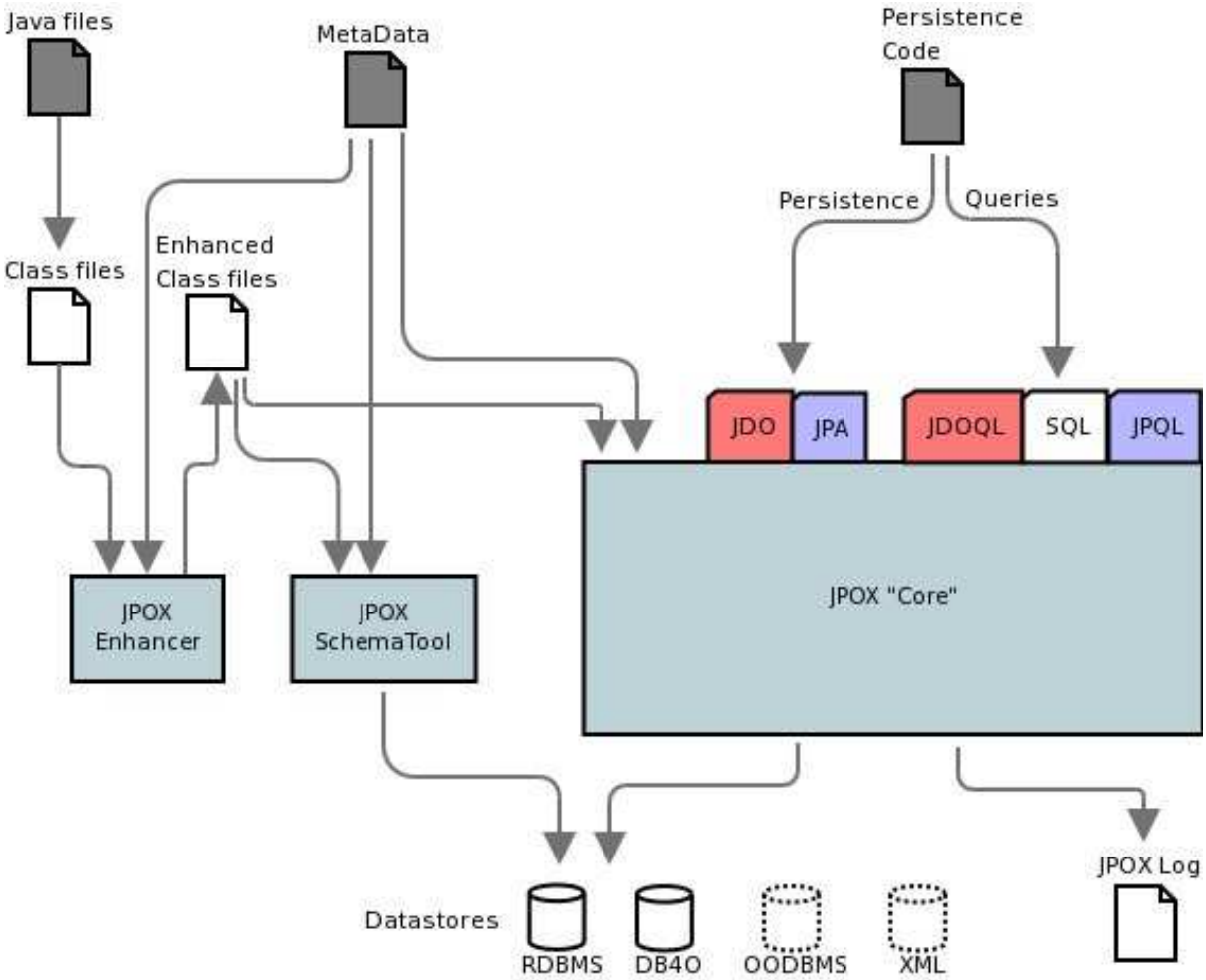

Abbildung 13: Funktionsweise von JPOX [JPOX Homepage 2007]

JPOX Tools

JPOX bietet auch Tools für Forward Engineering und Reverse Engineering an, dadurch wird es für den Programmierer leichter Anwendungen zu erstellen oder zu konfigurieren. Tools werden im Entwicklungsprozess benötigt, wenn das Datenbankschema:

- schon existiert,
- manuell vom User erstellt wird,
- automatisch während der Laufzeit durch org.jpox.autoCreateSchema generiert wird oder,
- bevor die Anwendung gestartet wird, durch Verwendung eines JPOX-SchemaTools erzeugt wird.

Die Verwendung des JPOX-SchemaTools ist sehr wichtig für die Erstellung von JPOX-Anwendungen. Es ist in der Java-Klassenbibliothek JPOXCore.jar enthalten.

Das JPOX-SchemaTool bietet laut [JPOX Homepage 2007] folgende Funktionalitäten:

- Create: Das Datenbankschema wird aufgrund der JDO-Metadateien erzeugt. Alternativ kann auch ein Schema-*DDL* erzeugt werden, um anschließend das Datenbankschema zu erzeugen.
- Delete: Alle Datenbanktabellen, die durch JDO-Metadateien referenziert werden, können gelöscht werden.
- Validate: Alle Datenbanktabellen und JDO-Metadateien werden, seit der letzten Verwendung von JPOX, auf korrekte Struktur überprüft.
- Dbinfo: Liefert detaillierte Informationen über die Datenbank z.B. Einschränkungen oder Datentypen.
- Schemainfo: Liefert detaillierte Informationen über das Datenbankschema.

Das Schema Tool kann laut [JPOX Homepage 2007] auf verschiedene Weise eingesetzt werden:

- Durch Aufruf über die Kommandozeile.
- Durch Verwendung des JPOX Maven1 Plugins.
- Durch Verwendung des JPOX Maven2 Plugins.
- Durch Verwendung einer *Ant*-Task.
- Durch Verwendung des JPOX Eclipse Plugins.
- Durch Verwendung von Java-Code in einer Anwendung.

JPOX eignet sich sehr gut für den Ansatz JDO. Durch die Möglichkeit des Einsatzes des JPOX-SchemaTools lassen sich Anwendungen automatisiert erstellen. Die Dokumentation auf der Homepage von JPOX ist sehr ausführlich und durch den Einsatz von Annotationen (*EJB*) gewinnt dieses Produkt zusätzlich an Bedeutung.

3.3.4 JDBCPersistence

Das Produkt JDBCPersistence basiert auf dem Ansatz JDBCPersistence (siehe Kapitel 3.1.5) und wurde von *Linux*® entwickelt. JDBCPersistence ist ein *Open Source* Produkt und kann unter dem *Link* http://www.jdbcpersistence.org/c/performance.html bezogen werden. JDBCPersistence bietet jedoch keine Tools für Reverse Engineering und Forward Engineering an. Dies liegt daran, dass der Code für das Mapping erst während der Laufzeit aus dem Java-Code generiert wird.

JDBCPersistence ist ein neues Produkt und wurde für Anwendungen konzipiert, die hohe Ansprüche an die *Performance* stellen. Der Vorteil liegt in der zielgerichteten, einfachen *Implementierung* und der vom Hersteller angepriesenen, guten *Performance*. Der Hersteller bietet leider keine umfassende Dokumentation und daher sollte das Produkt, bevor es verwendet wird, auf Eignung für die Erstellung einer Anwendung genau geprüft werden.

3.3.5 OpenJpa

OpenJpa basiert auf dem Ansatz *EJB* (siehe Kapitel 3.1.4) und wurde von Apache™ entwickelt. OpenJpa ist ein *Open Source* Produkt und kann unter dem *Link* http://incubator.apache.org/openjpa/downloads.html erworben werden.

OpenJpa Tools

Durch den Einsatz des OpenJpa Maven Plugins (http://mojo.codehaus.org/openjpa-maven-plugin/usage.html) kann die Erstellung einer Anwendung automatisiert werden. Es besteht dadurch die Möglichkeit eines Reverse Engineerings und Forward Engineerings.

OpenJpa-Anwendungen können durch ein Plugin auch mittels der Entwicklungsumgebung Eclipse entwickelt werden.

Dieses Produkt stellt eine weitere Möglichkeit dar, unter Verwendung von *EJB,* ein Persistence Framework aufzubauen.

3.3.6 SimpleORM

SimpleORM basiert auf dem Ansatz SimpleORM (siehe Kapitel 3.1.7) und wurde von Apache™ entwickelt. SimpleORM ist ein *Open Source* Produkt und kann unter dem *Link* http://www.simpleorm.org/download.html bezogen werden. SimpleORM bietet Tools für Reverse Engineering und Forward Engineering an. [SimpleORM Homepage 2007]

SimpleORM Tools

SimpleORM Anwendungen können mit *Ant* erzeugt werden, jedoch geschieht dies sehr selten.

Die Generierung von SimpleORM Definitionen (Reverse Engineering) kann durch ein eigenes SimpleORM-Generate-Package bewerkstelligt werden. SimpleORM spezifische Annotationen können noch nicht erstellt werden. [SimpleORM Homepage 2007**]**

Mit dem Aufruf SRecordMeta.createTableSQL kann das *SQL* Create Table Statement ausgelesen werden. Ein Datenbankschema kann durch Verwenden einer bestehenden *DDL* neu erzeugt werden (Forward Engineering). [SimpleORM Homepage 2007**]**

Dieses Produkt stellt ein sehr einfach zu *implementierendes ORM* zur Verfügung. Da nur Java-Code verwendet wird, ist es vor allem für Programmierer sehr interessant, welche sich nicht mit anderen Techniken, wie z.B. *XML,* beschäftigen wollen.

3.3.7 GNUstep und JIGS

GNUstep basiert auf dem Ansatz EOF (siehe Kapitel 3.1.2) und wurde von Paul Kunz entwickelt. GNUstep ist ein *Open Source* Produkt und kann unter dem *Link* http://www.gnustep.it/jigs/Download.html erworben werden. GNUstep hat den gleichen Ausgangspunkt wie das Produkt WebObjects (siehe Kapitel 3.3.2) und ist im Gegensatz zu WebObjects frei erhältlich. [GNUstep Homepage 2007] Als Programmiersprache verwendet GNUstep *Objective-C,* bietet aber über das Projekt JIGS (Java *Interface* for GnuStep) ein Java Framework an, welches die Integration zwischen Java und *Objective-C* ermöglicht. JIGS ermöglicht Java-Programmierern die GNUstep Libraries (*Objective-C*) zu nutzen. [JIGS Homepage 2007]

Dieses Produkt ist eine Übergangslösung, für einen Ansatz mit EOF, wenn kein kommerzielles Produkt erworben werden soll.

3.3.8 TopLink

TopLink verwendet den Ansatz TopLink (siehe Kapitel 3.1.6) und wurde von Oracle™ entwickelt. TopLink ist ein *Open Source* Produkt und kann unter folgendem *Link* erworben werden: http://www.oracle.com/technology/products/ias/toplink/jpa/download.html. Es verwendet JavaServer Faces (JSF), ein Java Framework für die Erzeugung einer Web-Applikation. Tools für die Erstellung eines *ORM* sind im JDeveloper integriert und sind auch in der TopLink Workbench vorhanden. [Oracle TopLink 2007]

Die Erstellung einer *ORM*-Anwendung wird durch die grafische Benutzeroberfläche erleichtert. TopLink ist für den Aufbau einer neuen *ORM*-Anwendung sehr gut geeignet.

3.3.9 Cayenne

Cayenne basiert auf dem Ansatz EOF (siehe Kapitel 3.1.2) und wurde von Apache™ entwickelt. Cayenne ist ein *Open Source* Produkt und kann unter dem *Link* http://cayenne.apache.org/download.html bezogen werden. Cayenne bietet über den CayenneModeler eine grafische Benutzeroberfläche für die Erzeugung von *ORM* an [Cayenne Homepage 2007] und ein Plugin für die Entwicklungsumgebung Eclipse.

3.3.10 BEA Kodo

BEA Kodo basiert auf dem Ansatz JDO (siehe Kapitel 3.1.1) und *EJB* (siehe Kapitel 3.1.4) und wurde von BEA Systems™ entwickelt. BEA Kodo ist kein *Open Source* Produkt, jedoch kann eine drei monatige Testversion unter dem *Link* http://bea.com/framework.jsp?CNT=index.htm&FP=/content/products/weblogic/kodo/ bezogen werden. BEA Kodo bietet über die BEA Kodo Management Console ein GUI für die Erzeugung von *ORM*-Anwendungen an. [BEA Kodo 2007]

3.3.11 CocoBase

CocoBase basiert auf dem Ansatz *EJB* (siehe Kapitel 3.1.4) und wurde von THOUGHT™ entwickelt. CocoBase ist ein kommerzielles Produkt, jedoch kann eine Testversion unter dem *Link* http://www.thoughtinc.com/cber_info.html bezogen werden. CocoBase bietet mit dem Magic Mapper eine grafische Benutzeroberfläche für die Erstellung von *ORM*-Anwendungen an und kann in Eclipse integriert werden. [CocoBase Homepage 2007]

3.3.12 EasyBeans

EasyBeans basiert auf dem Ansatz *EJB* (siehe Kapitel 3.1.4) und wurde von ObjectWeb™ entwickelt. EasyBeans ist ein *Open Source* Produkt und kann unter dem *Link* http://easybeans.objectweb.org/download.html bezogen werden. [EasyBeans Homepage 2007]

3.3.13 Hibernator

Hibernator, entwickelt von Jozsa Kristof, basiert auf dem Ansatz Hibernate Framework (siehe Kapitel 3.1.3). Hibernator ist ein *Open Source* Produkt und kann unter dem *Link* http://hibernator.sourceforge.net/ bezogen werden. Es ist ein Eclipse Plugin und wird in Kombination mit Hibernate verwendet. [Hibernator Homepage 2007]

4 Produktvergleich

In diesem Kapitel wird die Eignung eines Produkts für die Realisierung einer *ORM*-Anwendung ermittelt. Im ersten Schritt werden allgemeine Vergleiche angestellt, um dann auf die technischen Eigenschaften eines Produkts einzugehen.

4.1 Allgemeine Vergleichsparameter für Produkte

Für den Vergleich der Produkte wurden allgemeine Vergleichsparameter definiert, um den Vergleich der Produkte zu standardisieren. Für die Bewertung einzelner Kriterien wurde teilweise das Schulnotensystem von 1 bis 5 eingesetzt. In Tabelle 3 werden die Ergebnisse der Bewertung zusammengefasst. Als Grundlage für die Erstellung der Vergleichsparameter diente [Madgeek 2007].

Produktbezogene Parameter

1. Preis: Wie teuer ist das Produkt in der Anschaffung?
2. Stabilität des Produkts: Bleibt das Produkt über mehrere Jahre am Markt verfügbar?
3. Dokumentation: Verfügt das Produkt über ausreichende Dokumentationen? Handelt es sich um eine neue Technik?
4. Support: Wie sieht es mit Unterstützung (Support) seitens des Herstellers aus?
5. Frequenz von Aktualisierungen und Neuerungen: Wie oft kommt es zu neuen Versionen oder Verbesserungen an den Produkten?

Projektbezogene Parameter

6. Komplexität: Wie schwierig ist es dieses *ORM*-Produkt zu *implementieren*? Wie viele verschiedene Aufgaben fallen an?
7. Zeitlicher Aufwand: Wie hoch wird der zeitliche Aufwand für die Erstellung einer *ORM*-Anwendung unter Verwendung eines dafür geeigneten Produktes geschätzt?
8. Fachwissen: Wie viel Fachwissen wird von den Entwicklern verlangt (1 = sehr wenig, 5 = sehr viel)?

Allgemeine technische Parameter

9. *Performance*: Die *Performance* ist ein wichtiger Faktor für die Beurteilung eines Produkts.
10. Java Code: Verwendet das Produkt nur Java Code?
11. *Portabilität*: Wie leicht kann das Produkt auf verschiedenen Plattformen verwendet werden?
12. Erweiterbarkeit: Kann das Produkt mit geringem Zeitaufwand erweitert werden?
13. Wartbarkeit: Kann das Produkt mit geringem Zeitaufwand gewartet werden?
14. Änderbarkeit: Kann das Produkt mit geringem Zeitaufwand an neue Anforderungen angepasst werden?

4.1.1 Allgemeine Ergebnisse

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Hibernate	0	ja	1	1	2	sehr komplex	hoch	4	3	nein	1	1	2	1
WebObjects	500	ja	3	2	2	komplex	hoch	3	3	nein	1	1	2	1
JPOX	0	ja	2	2	2	komplex	hoch	3	3	nein	2	2	2	1
JDBCPersistence	0	jein	4	4	3	weniger komplex	mitel	1	1	ja	3	3	4	3
OpenJpa	0	ja	3	3	3	komplex	mitel	3	2	nein	2	2	3	2
SimpleORM	0	jein	4	4	4	weniger komplex	wenig	1	1	ja	3	3	4	3
GNUstep u. JIGS	0	ja	3	3	3	komplex	mitel	4	4	nein	?	2	2	2
TopLink	0	ja	2	2	2	sehr komplex	mitel	2	3	nein	1	1	2	1
Cayenne	0	ja	3	2	3	komplex	mitel	3	3	nein	2	2	2	2
BEA Kodo	Testversion	ja	2	3	3	komplex	mitel	3	3	nein	2	2	2	1
CocoBase	Testversion	ja	3	2	3	komplex	mitel	3	2	nein	2	2	3	1
EasyBeans	0	ja	3	2	2	weniger komplex	wenig	3	2	ja	2	3	2	1
Hibernator	0	jein	3	3	2	sehr komplex	hoch	4	3	nein	1	1	2	1

Tabelle 3: Ergebnis des allgemeinen Produktvergleichs

4.2 Technische Vergleichsparameter für Produkte

Für die Bewertung der technischen Eignung eines Produktes wurde eine Auswahl von Vergleichsparametern vorgenommen. In Tabelle 4 werden die Ergebnisse der Bewertung zusammengefasst. Als Grundlage für die Erstellung der Vergleichsparameter diente [Madgeek 2007].

Tools

1. Verfügt das Produkt über ein Mapping GUI oder Eclipse Plugin?
2. Bewertung der Tools für Reverse und Forward Engineering.

Datenmodell

3. Unterstützt das Produkt Beziehungen zwischen Objekten?
4. Unterstützt das Produkt zusammengesetzte Schlüssel?
5. Unterstützt das Produkt m-zu-n und 1-zu-n Beziehungen?
6. Unterstützt das Produkt 1-zu-1 Beziehungen?
7. Unterstützt das Produkt ternäre Beziehungen?
8. Unterstützt das Produkt Abfragen über Vererbungen?

Kompatibilität mit anderen Technologien

9. Bietet das Produkt *EJB*-Unterstützung an?
10. Benötigt das Produkt manuell erzeugtes *SQL*?
11. Benötigt das Produkt Code-Erzeugung bzw. *Byte-Code*?
12. Bietet das Produkt *RDBMS*-Unterstützung an?
13. Unterstützt das Produkt Datenhaltung für String, Integer, Date, etc.?
14. Unterstützt das Produkt das Persistieren frei wählbarer Klassen (keine Superklasse oder *Interface*)?
15. Benötigt das Produkt Laufzeitreflexion?
16. Unterstützt das Produkt die Persistenz von Klassen durch private Felder?
17. Unterstützt das Produkt die Persistenz von Klassen durch Getter- und Setter-Methoden?

4.2.1 Technische Ergebnisse

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Hibernate	ja	1	ja	ja	ja	ja	ja	ja	ja	nein	ja	ja	ja	ja	nein	ja	ja
WebObjects	ja	3	ja	ja	ja	ja	ja	ja	ja	nein	nein	ja	ja	ja	ja	nein	ja
JPOX	ja	3	ja	ja	ja	ja	ja	ja	ja	nein	ja	ja	ja	ja	nein	ja	nein
JDBCPersistence	ja	4	ja	ja	ja	ja	ja	ja	nein	nein	nein	ja	ja	ja	nein	?	?
OpenJpa	ja	3	ja	ja	ja	ja	ja	ja	ja	nein	nein	ja	ja	ja	nein	?	?
SimpleORM	nein	2	ja	ja	ja	ja	ja	ja	nein	nein	nein	ja	ja	ja	nein	?	?
GNUstep u. JIGS	ja	2	ja	ja	ja	ja	ja	ja	nein	nein	nein	ja	ja	ja	ja	nein	ja
TopLink	ja	3	ja	ja	ja	ja	ja	ja	ja	nein	ja	ja	ja	ja	nein	?	?
Cayenne	ja	?	ja	ja	ja	ja	ja	ja	ja	nein	nein	ja	ja	ja	nein	nein	ja
BEA Kodo	ja	3	ja	ja	ja	ja	ja	ja	ja	nein	ja	ja	ja	ja	nein	ja	nein
CocoBase	ja	?	ja	ja	ja	ja	ja	ja	ja	nein	nein	ja	ja	ja	ja	?	?
EasyBeans	nein	4	ja	ja	ja	ja	ja	ja	ja	nein	nein	ja	ja	ja	nein	?	?
Hibernator	ja	1	ja	ja	ja	ja	ja	ja	ja	nein	ja	ja	ja	ja	nein	ja	?

Tabelle 4: Ergebnis des technischen Produktvergleichs

4.3 Entscheidung

Die Entscheidung für ein Produkt wurde aufgrund von Literaturrecherchen (siehe Kapitel 3) und durch die Produktvergleiche (siehe Kapitel 4) getroffen. Die Entscheidung für Hibernate und JPA als *ORM*-Produkt lässt sich wie folgt begründen:

- Gute Bewertung laut technischem und allgemeinem Produktvergleich.
- Hibernate bietet sehr gute Unterstützung für Reverse Engineering und Forward Engineering an und verwendet *POJOs*.
- Durch die Verwendung von Hibernate und JPA lassen sich zusätzliche Tabellen oder Klassen sehr einfach erzeugen und in das bestehende Datenbankschema integrieren.
- Die Vorteile von *RDBMS* bleiben durch die Verwendung von Hibernate bestehen.
- Es wird wenig Zeit und Arbeit für Persistenz aufgewand und die Datenbankanbindung ist sehr zufriedenstellend.
- Hibernate unterstützt eine transparente Abbildung von Objekten auf relationale Strukturen, inklusive Objektreferenzen, Vererbung und *Polymorphie*.
- Hibernate gewährleistet, dass sich *persistente Objekte* selbst um ihre Persistierung kümmern.
- Hibernate gewährleistet eine gute *Portabilität* zwischen unterschiedlichen Plattformen.
- Hibernate etabliert zwei objektorientierte Abfragesprachen (HQL und CriteriaAPI), wobei natives *SQL* und datenbankspezifische Eigenschaften weiterhin nutzbar bleiben.
- Die Trennung von Persistenz- und Geschäftslogik wird durch die Verwendung von Hibernate mit JPA sichergestellt.
- Durch die Verwendung der Hibernate Fetching-Strategien ist eine *Performance*-Optimierung für transaktionale Mehrbenutzerapplikationen möglich.
- Durch die Verwendung von Hibernate und JPA wird der Impedance Mismatch zwischen *RDBMS* und der objektorientierten Programmiersprache Java überwunden.

5 Ausblick

In diesem Kapitel werden Prognosen über die zukünftige Entwicklung von *ORM* aufgestellt.

5.1 Zukünftige Bedeutung von *ORM* für das Softwareengineering

Da die Bedeutung der objektorientierten Programmiersprachen (z.B. Java) zunehmen wird und im Bereich der Datenbanken *RDBMS* weiterhin starke Bedeutung haben, wird es vermehrt notwendig sein, zwischen diesen eine Brücke mit *ORM* zu bauen. Da immer mehr Altsysteme (*Legacy-Systeme*) migriert werden müssen, wird auch im Bereich des Softwareengineerings die Bedeutung von *ORM* zusätzlich steigen. Für Anwendungen, die hohe Anforderungen an die *Performance* stellen, empfiehlt sich der Einsatz von JDBCPersistence oder SimpleORM. Die steigende Anzahl an Tools für Reverse Engineering wird zu verstärkter Nutzung von *ORM* führen, da Anwendungen einfacher zu erstellen sind.

6 Praxisbeispiel „DigiPark“

Die Rahmenbedingungen für dieses Projekt sind im Kapitel 1.2 bereits vorgestellt worden. Abbildung 14 zeigt einen Überblick über die Architektur des Projekts „DigiPark“. Das Praxisbeispiel bezieht sich auf den Teilbereich „Datenbanken und Schnittstellen“. Folgende Anforderungen wurden an diesen Teilbereich gestellt:

- Datenbank am Server: Im Rahmen des Projekts ist es notwendig eine eigene Datenbank (siehe Kapitel 6.3), die hinter der Web-Applikation steht, zu entwickeln. Die Datenbank wird benötigt, um sich die vom Benutzer gewünschten Informationen, wie zum Beispiel eine Übersichtskarte aller Wanderwege mit einer kurzen Beschreibung auf der Web-Oberfläche anzeigen zu lassen. Aus diesem Grund müssen alle Daten für die wählbaren Themenbereiche, alle eingetragenen Beobachtungen sowie die Kartengrundlagen in der Datenbank gespeichert werden. Weiters müssen die einzelnen Benutzer, welche berechtigt sind Daten in der Datenbank zu manipulieren, verwaltet werden. Die Datenbank muss weiters Grunddaten, wie zum Beispiel Artenlisten oder Sammlungen verwalten, damit das Eintragen neuer Beobachtungen für den Benutzer erleichtert und vereinheitlicht wird.
- Schnittstelle zwischen Serverdatenbank und *PDA*-Datenbank: Um die Daten, welche mit einem *PDA* erfasst werden, auf der Web-Applikation in einer Übersichtskarte darstellen zu können, muss eine Schnittstelle zwischen der *PDA*-Datenbank und der Serverdatenbank entwickelt werden. Über diese Schnittstelle sollen auch alle Daten übertragen werden, die für die Erfassung von Beobachtungsdaten notwendig sind, wie z.B. die Taxaliste. Es muss auch der umgekehrte Weg möglich sein, das heißt, die am *PDA* erfassten Daten sollen über die zu entwickelnde Schnittstelle in die Serverdatenbank übertragen werden können.

- *Replikation* zwischen Serverdatenbank und „BioOffice“-Datenbank: Es soll den Mitarbeitern des Nationalparks möglich sein, Daten aus den Beobachtungen direkt in die „BioOffice“-Datenbank einzuspielen. Dadurch haben die Mitarbeiter des Nationalparks anschließend die Möglichkeit, in „BioOffice“ diverse Analysen mit den digital erfassten Daten durchzuführen.
- Datenaustauschformat: Als Datenaustauschformat zwischen den einzelnen Projektteilen soll *XML* verwendet werden (siehe Kapitel 6.11).
- Die Datenbankanbindung soll mittels eines *ORM*-Produkts realisiert werden.
- Mehrsprachigkeit: Die Interaktion mit dem Benutzer soll in mehreren Sprachen möglich sein (siehe Kapitel 6.4.7).
- Ein Mehrbenutzerbetrieb ist nicht vorgesehen, da die „BioOffice“-Anwendung eine Einzelplatzversion ist und der Nationalpark Gesäuse die Daten zentral erfasst.
- Die Datengrundlage für die beabsichtigte Routenplanung soll geschaffen werden.
- Mitarbeiter des Nationalparks sollen zusätzlich zu den Standardattributen („BioOffice“-Datenmodell) noch zusätzliche Beobachtungsattribute (Brutstatus, Verhalten etc.) erfassen können.

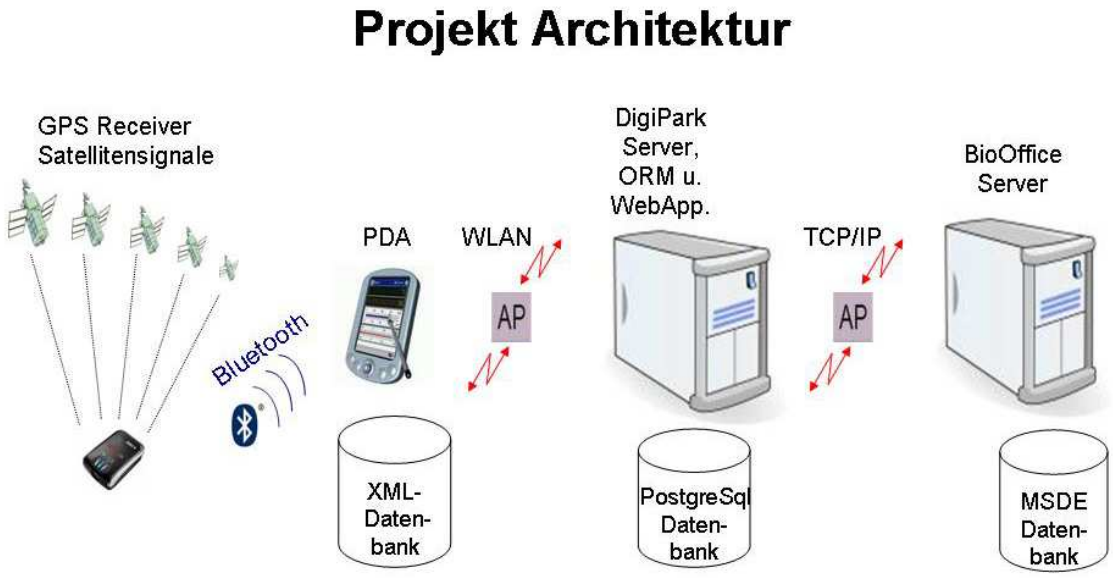

Abbildung 14: Architektur des Projekts "DigiPark"

6.1 Projektplanung

Zu Beginn des Projekts fanden umfassende Planungen und *empirische* Erhebungen der Anforderungen für den Teilbereich „Datenbanken und Schnittstellen“ des Projekts „DigiPark“ statt.

6.1.1 Spezifikation

Um die Anforderungen an das Projekt zu definieren und zu fixieren, wurde gemeinsam mit den Projektbeteiligten eine *Spezifikation* erstellt. Für die Erstellung der *Spezifikation* wurde der international anerkannte Standard *IEEE-STD-830-1998* verwendet.

6.1.2 Meilensteinplanung

Im Teilbereich „Datenbanken und Schnittstellen“ wurde mit einer Meilensteinplanung begonnen, um den zeitlichen Aufwand und die Fortschritte zu fixieren. Die Planung basierte auf folgenden Grundlagen:

- *Spezifikation*
- 4 Monate Zeit für die *Implementierung*
- 2 Monate Zeit für den schriftlichen Teil
- Berücksichtigung der zeitlichen Ressourcen der Diplomanden

Aufgrund der Entwicklungen im Projekt wurden zwei Meilensteinplanungen vorgenommen.

Abbildung 15 veranschaulicht den geplanten zeitlichen Ablauf der Diplomarbeit.

6.1.3 Konzeptionelles Modell

Das Modell für das Datenbankschema wurde von allen Projektbeteiligten gemeinsam erstellt. Folgende wichtige Einflüsse waren zu berücksichtigen:

- Anlehnung an das Datenmodell von „BioOffice“.
- Modellierung der Informationen für den Nationalpark Guide.
- Modellierung der Attribute für den Nationalpark Guide.
- Modellierung der Einbindung von *GIS*-Daten für den *UMN-MapServer*.
- Modellierung der zusätzlichen Beobachtungsattribute.

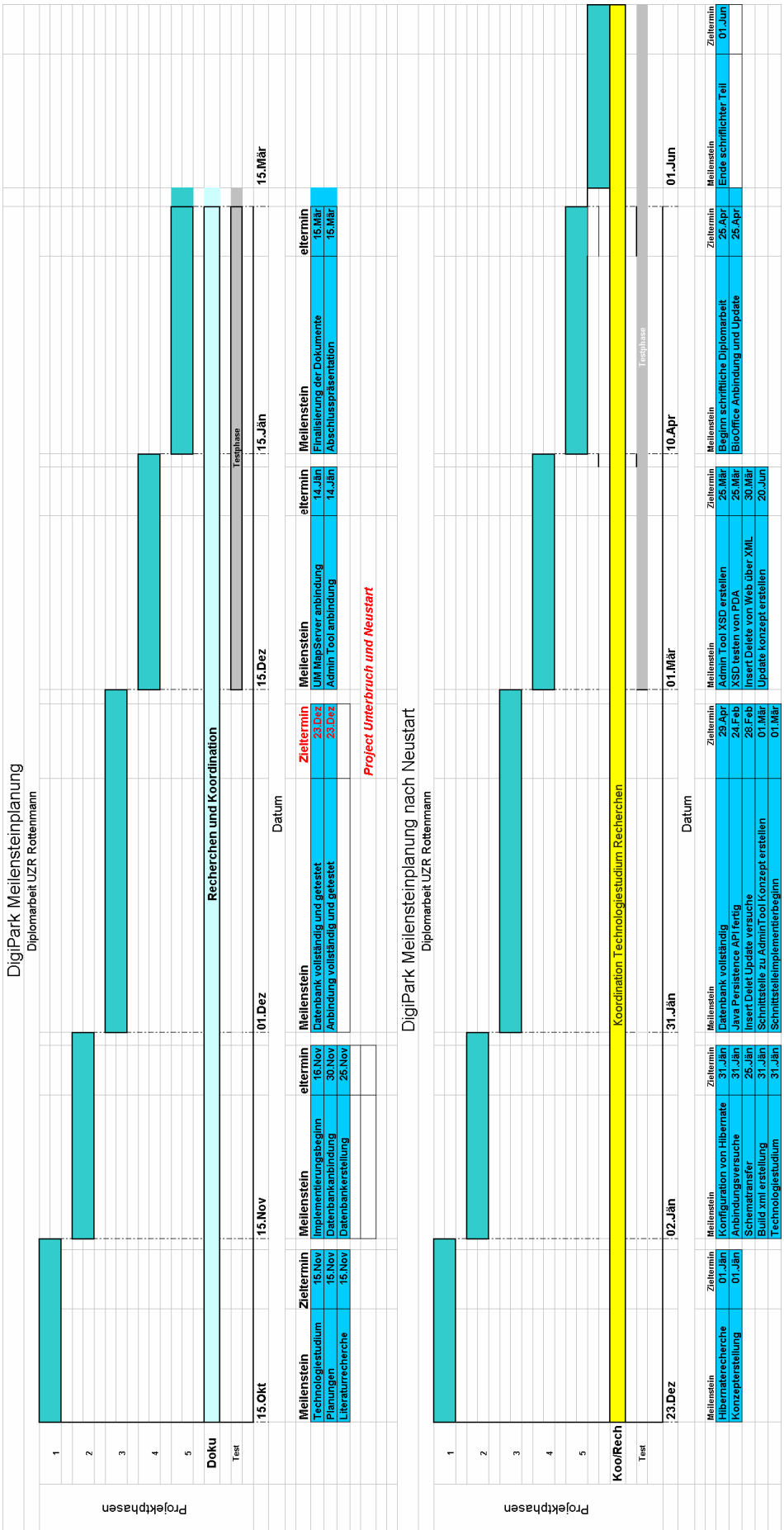

Abbildung 15: Meilensteinplanung

In der Projektgruppe wurde ein Modell entworfen, um das zu entwickelnde System besser verstehen zu können. Alle modellierten Aggregationen stellen Löschweitergaben dar. Die Grundlage für die Erstellung dieses *UML*-Diagramms bildeten die Quellen [Matthiessen und Unterstein 2003], [Fischer 2003], [Booch et al. 1999] und [Pilone 2004].

Namenskonvention

Das Modell für das „DigiPark“-Datenbankschema verwendet die Namenskonvention des „BioOffice“-Modells, um konsistente Klassen- und Attributnamen sicherzustellen.

Alle „Name1Name2Attribut“-Klassen beziehen sich auf aufgelöste m-zu-n Beziehungen zwischen Klassen, wobei „Name1“ und „Name2“ hier beispielhaft für die Namen der Klassen stehen.

Es wurden alle Attributsnamen vom „BioOffice“-Modell übernommen. Neu modellierte Attribute wurden angelehnt an die Namenskonvention des „BioOffice“-Modells erstellt.

Abbildung 16 zeigt das vom Projektteam entwickelte *UML*-Diagramm.

6.1.4 Projektneustart

Ende Dezember 2006 kam es aufgrund weitreichender Veränderungen im Konzept des Projekts zu einem Neustart. Bis zu diesem Zeitpunkt wurden einfache *Implementierungen* des Teilbereiches „Datenbanken und Schnittstellen“ mittels *JDBC* und Java 5 forciert. Ab Jänner 2007 wurde an der Umsetzung einer *ORM*-Anwendung gearbeitet. Die *Spezifikation* und die Meilensteinplanung wurden aufgrund von neuerlich durchgeführten Literaturrecherchen überarbeitet bzw. neu geplant. Alle nachfolgenden Kapitel beziehen sich auf die Arbeiten ab diesem Zeitpunkt.

6.2 *ORM* für „DigiPark“

In Kapitel 4 wurde auf die allgemeinen und technischen Eigenschaften von *ORM*-Produkten eingegangen. Das Projekt „DigiPark“ stellt aber noch zusätzliche Anforderungen an ein *ORM*-Produkt. Folgende Anforderungen, welche in die Produktentscheidung miteinfließen sollen, gilt es zu berücksichtigen:

1. *Open Source*: Das Projekt „DigiPark“ für den Nationalpark Gesäuse verfügt über keine finanziellen Mittel für den Ankauf von Software, dadurch ist es wichtig, ein *Open Source* Produkt zu verwenden.
2. Erfahrungen: Sind in der Projektgruppe Erfahrungen mit diesem Produkt vorhanden?
3. Projektleitung: Werden die Vorgaben der Projektleitung erfüllt?

4. Integration ins Gesamtprojekt: Lässt sich eine *Implementierung* unter Verwendung dieses Produkts in das Gesamtprojekt integrieren?
5. Eignung für verwendete *RDBMS*: Annahme der Verwendung einer *PostgreSQL*-Datenbank und *MS SQL 2005 Express Version*.
6. Zeitfaktor: Kann die *Implementierung* in der gewünschten Zeit (4 Monate für *Implementierung*) erfolgreich abgeschlossen werden?
7. Wartbarkeit: Kann die *implementierte* Anwendung von Studenten (GTEC) weiterhin betreut werden?

Die Bewertung der Auswahlkriterien ist in Tabelle 5 ersichtlich.

6.2.1 Entscheidung

Die Entscheidung für Hibernate und JPA als *ORM*-Produkt für die *Implementierung* des Projekts „DigiPark“ lässt sich wie folgt begründen:

- Entscheidung für Hibernate mit JPA nach dem Produktvergleich (siehe Kapitel 4.3).
- Hibernate ist ein *Open Source* Produkt.
- Hibernate wurde von der Projektleitung favorisiert.
- Hibernate lässt sich sehr gut ins Gesamtprojekt integrieren (Hibernate und Spring Web-Applikation).
- Hibernate eignet sich sehr gut für die verwendeten *RDBMS* (*PostgreSQL* und die angenommene *MS SQL 2005 Express Version*).
- Die Erstellung einer Anwendung mit Hibernate und JPA sollte sich in der gewünschten Zeit realisieren lassen.
- Die erstellte Anwendung sollte von Studenten (*GTEC*) nach einer Einarbeitungsphase gewartet werden können.

6.3 Auswahl einer *GIS*-fähigen Datenbank

Die Daten in diesem Kapitel beziehen sich auf den Stand Oktober 2006.

Hibernate und JPA eignen sich für sehr viele *RDBMS*. Die Auswahl einer *GIS*-fähigen Datenbank wurde durch den Projektteil „Web-Applikation“ bestimmt, da sich die Datenbank vor allem für den Einsatz mit dem *UMN-MapServer 4.0* eignen sollte. Es wird in diesem Kapitel ein kurzer Überblick über die Entscheidungsfindung gegeben. Ob der Einsatz einer *Open Source* Datenbank möglich und sinnvoll ist, hängt von den Anforderungen an den Funktionsumfang des Systems ab.

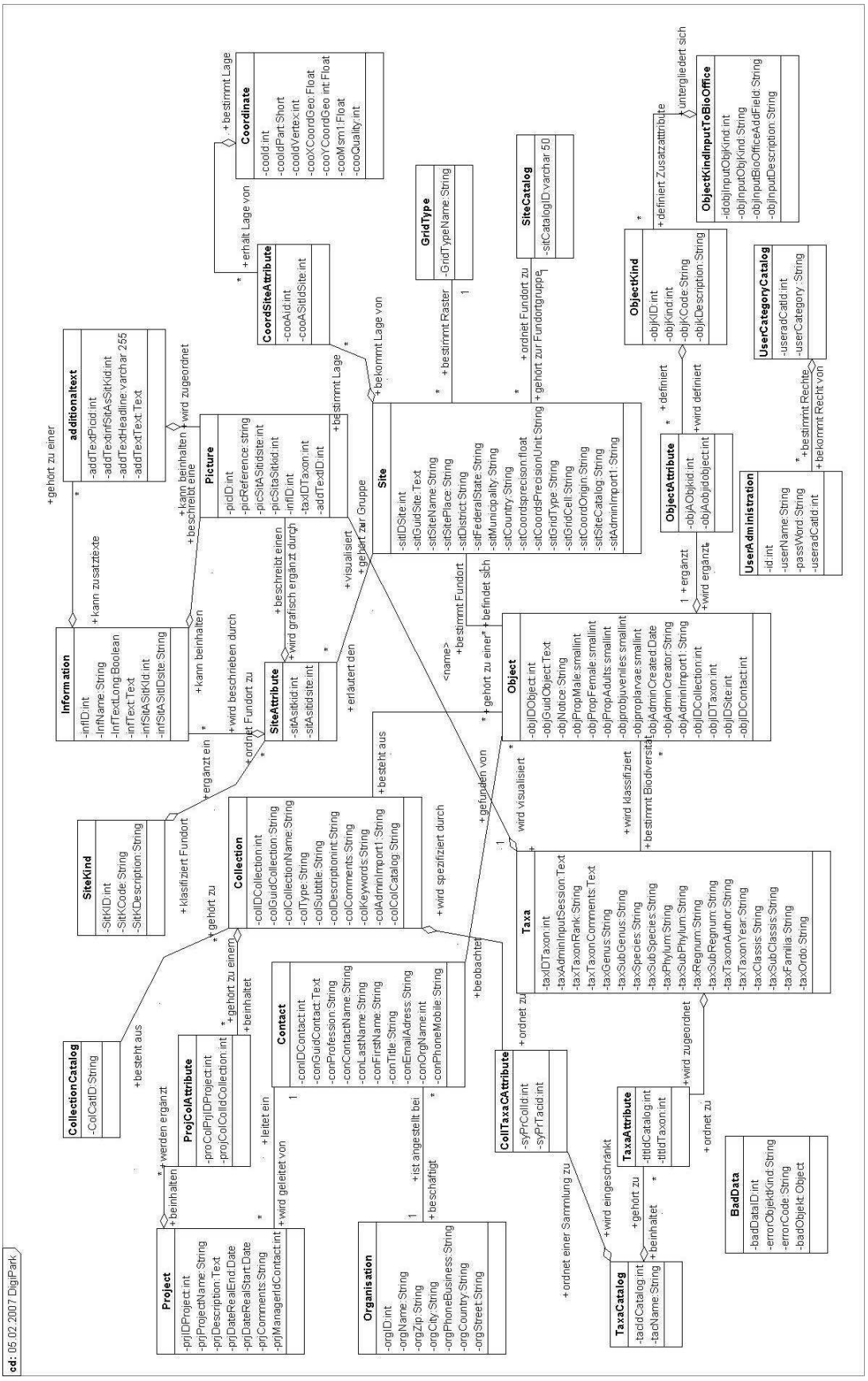

Abbildung 16: *UML*-Diagramm

Tabelle 5: Bewertung der *ORM*-Produkte für „DigiPark“

	1	2	3	4	5	6	7
Hibernate	ja	4	1	3	1	3	4
WebObjects	nein	4	1	4	1	2	3
JPOX	ja	4	1	2	1	3	4
JDBCPersistence	ja	2	3	2	2	1	1
OpenJpa	ja	4	1	3	1	2	4
SimpleORM	ja	1	3	1	2	1	1
GNUstep u. JIGS	ja	5	2	4	1	4	4
TopLink	ja	4	1	3	1	3	4
Cayenne	ja	4	1	3	1	2	3
BEA Kodo	ja	4	1	3	1	2	3
CocoBase	ja	4	1	3	1	2	3
EasyBeans	ja	3	3	2	1	3	3
Hibernator	ja	4	1	3	1	3	4

In den folgenden Bereichen wurden vom Diplomanden des Projektteils „Web-Applikation“ empirische Untersuchungen über die Eignung eines *Open Source*-Produkts für die Datenbank des Projekts „DigiPark“ durchgeführt [Heise 2007]:

- Datenintegrität
- Datenorganisation
- Datenzugriff
- Umfangreiche *SQL-Features*
- Möglichkeiten der Integration von Geschäftslogik in die Datenbank
- Betrieb
- *Performance* und Skalierbarkeit
- Verfügbarkeit
- Sicherheit
- *GIS*-Erweiterung

Tabelle 6 zeigt einen Überblick über die Bewertung der Kriterien der aufgelisteten Bereiche.

6.3.1 Entscheidung

Die Entscheidung für das Datenbankprodukt wurde vom Diplomanden des Projektteils „Web-Applikation“ selbst getroffen, da sich die Datenbank vor allem für den Einsatz mit dem *UMN-MapServer 4.0* eignen sollte. Die Entscheidung für das Produkt *PostgreSQL* wurde wie folgt begründet:

- Die technischen Eigenschaften des Produkts waren überzeugend (siehe Tabelle 6).
- Das Produkt beinhaltet eine *GIS*-Erweiterung, die vom *UMN-MapServer 4.0* verwendet werden kann.

6.3.2 Installation der Datenbank

Es wurde die Version 8.1.5 und die in diesem Paket enthaltene Entwicklungsumgebung *pgAdmin* III, sowie die PostGis-Erweiterung installiert. Da der Typ des Dateisystems NTFS sein muss, wird dieser von FAT32 auf NTFS konvertiert. Die Konvertierung erfolgt über den Kommandozeileneintrag „convert c: /FS:NTFS“. Die Installation ist einfach und die gewünschte Funktionalität wurde schnell erreicht. Im November 2006 (vor dem Projektneustart) wurde das Datenbankschema mit Hilfe von *pgAdmin* III (Stand Oktober 2006) mittels *SQL*-Statements *implementiert*.

	Firebird	Ingres	MaxDB	MySQL	PostgreSQL
Version	01.Mai	3.0	07.Jun	5.0	08.Jän
Datenintegrität					
ACID-Transaktionen	Ja	Ja	Ja	Ja*	Ja
2-phasiges Commit	Ja	Ja	Nein	Ja*	Ja
Fremdschlüssel	Ja	Ja	Ja	Ja*	Ja
CHECK-Bedingung	Ja	Nein	Ja	Nein	Ja
Savepoints	Ja	Ja	Ja	Ja*	Ja
Locking	MVCC	zeilenweise	zeilenweise, MVCC ab 7.7	MVCC und zeilenweise*	MVCC und zeilenweise
Datenbankobjekte					
Schema	Nein	Ja	Ja	Ja	Ja
Temporäre Tabellen	Nein	Ja	Ja	Ja	Ja
Stored Procedures	Ja	Ja	Ja	Ja	Ja
Trigger	Before/After	After	After	Before/After	Before/After
View	Ja	Ja	Ja	Ja	Ja
Materialized View	Nein	Nein	Nein	Nein	Nein
Updatable View	Ja	Nein	Ja	Ja	Ja
Expression Index	Nein	Nein	Nein	Nein	Ja
Partial Index	Nein	Nein	Nein	Nein	Ja
Bitmap Index	Nein	Nein	Nein	Nein	Ja
Volltext-Index	Nein	Nein	Nein	Ja (MyISAM)	Ja (Tsearch2)
SQL, Datentypen					
SQL-Standard	92, 99	92, 99	92	92, 99	92, 99, 03
Nutzerdef. Typen	Nein	Ja	Nein	Nein	Ja
Nutzerdef. Funktionen	Ja	Ja	Ja	Ja	Ja
GIS	Nein	Ja	Nein	Ja	Ja
Boolean	Nein	Nein	Ja	Nein	Ja
Sub-Select	Ja	Ja	Ja	Ja	Ja
Full Outer Join	Ja	Ja	Ja	Nein	Ja
Betrieb					
Multi-Threading	Ja (Super)	Ja (BS)	Ja (DB)	Ja (BS)	Nein
Multi-Processing	Ja (Classic)	Ja	Ja	Ja	Ja
Abfrage-Parallelisierung	Nein	Ja	Ja	Ja (Cluster)	Nein
Replikation	Ja	Ja	Ja	Ja	Ja (Slony)
Multimaster Repl.	Ja	Ja	Nein	Nein	Nein
Clustering	Nein	Ja	Ja	Ja	Nein
Load Balancing	Nein	Ja	Nein	Ja	Nein
Tablespaces	Nein	Nein	Nein	Ja	Ja
Partitionierung	Nein	Ja	Nein	Nein (V. 5.1)	Ja (CE)
Point-In-Time Recovery	Nein	Ja	Ja	Ja*	Ja

Tabelle 6: Vergleich von *Open Source* Datenbanken [Heise 2007]

6.4 Konfiguration von Hibernate

Die Konfiguration einer Hibernate-Anwendung ist eine komplexe Tätigkeit. Die folgenden Kapitel sollen einen Überblick über jene Tätigkeiten geben, die notwendig sind, damit eine Hibernate-Anwendung lauffähig wird.

6.4.1 Eingesetzte Hibernate-Verfahren und Tools

Die praktische Arbeit kann in zwei Teilbereiche unterteilt werden:

- „DigiPark“: *PostgreSQL*-Datenbank
- „BioOffice“: MSDE 2000 (nicht wie geplant *MS SQL Server 2005 Express Edition*)

Die Grundlage im Bereich „DigiPark“ bildete ein noch nicht vollständiges Datenbankschema, welches vor dem Neustart (Stand Dezember 2006) des Projekts erzeugt wurde. Aufbauend auf diesem wurde das Verfahren Top-Down (siehe Kapitel 3.3.1) verwendet. Für das Verfahren

Top-Down sollten *Ant*-Tasks genutzt werden, um damit das Datenbankschema generieren zu lassen. Es wurde *MyEclipse* in der Version 5.1 GA von Genuitec™ [MyEclipse 2007] verwendet, da es besonders viele unterstützende Elemente enthält und sich sehr gut in die Entwicklungsumgebung Eclipse integrieren lässt.

Im Bereich „BioOffice“ wurde das Verfahren Bottom-Up verwendet, wodurch die Java-Quelldateien automatisch generiert wurden. Für dieses Verfahren wurden Reverse Engineering Tools eingesetzt, um die Arbeit beim Erstellen der zahlreichen Klassen zu erleichtern.

6.4.2 Konfigurationstätigkeiten

Bevor mit der Konfiguration begonnen wird, empfiehlt es sich, eine normale *JDBC*-Verbindung zur Datenbank aufzubauen, um den verwendeten Datenbanktreiber zu testen und um den richtigen Connection String zu kreieren. Mittels *MyEclipse* kann ein Java-Projekt angelegt und anschließend als Hibernate-Projekt klassifiziert werden, indem die Hibernate Capatilities eingefügt werden. Automatisch wird die richtige Struktur des Projekts angelegt. Mittels Editor erfolgt die Aufforderung, die Konfigurationsdatei (hibernate.cfg.xml) und die Hibernate SessionFactory zu erstellen. Die Erstellung der Konfigurationsdatei ist sehr einfach, wenn dies unter Zuhilfenahme des Editors geschieht.

6.4.3 Klassenbibliotheken

Folgende Übersicht zeigt die wichtigsten Klassenbibliotheken, die zur Projektbibliothek des Projekts hinzugefügt wurden [Beeger et al. 2006]:

- für die Datenbankverbindung (z.B. pg74.216.jdbc3.jar für *PostgreSQL* Version 8.1.5 und sqljdbc.jar für MSDE 2000)
- Bibliotheken mit allen Hibernate-Klassen (hibernate-3.2.0.cr2.jar, hibernate-annotations-3.2.0.cr1.jar, hibernate-entitymanager-3.2.0.cr2.jar, hibernate3.jar)
- für die Erzeugung von dynamischen Proxies (cglib-X.jar)
- Bibliothek zum Verändern des *Byte-Codes* (asm.jar, asm-attrs.jar)
- Caching Bibliothek (ehcache-X.jar)
- Wrapper für log4J (commons-logging-X.jar)
- Standard Java Transaktions *API* (jta.jar)
- Bibliothek mit zusätzlichen Collection-Klassen (commons-collections.jar)
- *Parser* für HQL-Ausdrücke (antlr-X.jar)
- *XML*-Bibliotheken zum Lesen der Konfiguration (z.B. dom4j-X.jar)

- Erweiterungen zu *JDBC2* (jdbc2-0-sdtext.jar)

Es empfiehlt sich der Einsatz eines Beispielprojekts oder einer Recherche unter [Hibernate Homepage 2007], um die richtigen Klassenbibliotheken je nach verwendetem *RDBMS* zu erhalten. Die Klassenbibliotheken für die Utilities werden in den jeweiligen Kapiteln behandelt.

6.4.4 Datenbanken konfigurieren

Im Bereich „DigiPark“ musste die Konfigurationsdatei unter Zuhilfenahme des *MyEclipse* Hibernate Config Editor angepasst werden. Um die Konfigurationsdatei richtig zu adaptieren, wurde in Foren und unter [Hibernate Homepage 2007] recherchiert. Es empfiehlt sich auch der Einsatz folgenden Eintrages:

```
<property name="hbm2ddl.auto">update</property>
```

Mit diesem Eintrag wird bei jedem Aufruf einer Session eine Aktualisierung des Datenbankschemas vorgenommen.

Um die *PostgreSQL*-Datenbank mit Hibernate verwenden zu können, musste das in Listing 10 beschriebene *SQL*-Statement beim Anlegen des Datenbankschemas einmalig durchgeführt werden:

```
CREATE OR REPLACE FUNCTION information_schema._pg_keypositions()
  RETURNS SETOF int4 AS
$BODY$select g.s
        from generate_series(1,current_setting('max_index_keys')::int,1)
        as g(s)$BODY$
  LANGUAGE 'sql' IMMUTABLE;
ALTER FUNCTION information_schema._pg_keypositions() OWNER TO postgres;
```

Listing 10: Statement für *PostgreSQL*-Konfiguration

Für die Anbindung der MSDE 2000 Datenbank der Standardanwendung „BioOffice“ musste eine eigene Konfigurationsdatei erstellt werden. Am Anfang wurde eine *JDBC*-Anbindung *implementiert*, um den verwendeten Treiber zu testen. Die *JDBC*-Verbindung funktionierte aufgrund des nicht aktivierten TCP/IP-Protokolls der MSDE 2000-Datenbank nicht. Mittels des *SQL* Server Management Studio konnten diese Eigenschaften nicht eingestellt werden. So wurde diese Datenbank manuell konfiguriert. Nach einer Recherche im Internet wurde eine geeignete Anleitung gefunden, mit welcher es möglich war, das TCP/IP-Protokoll zu aktivieren. Nach der manuellen Konfiguration funktionierte die *JDBC*-Verbindung und die zweite Konfigurationsdatei konnte erstellt werden.

6.4.5 Hibernate SessionFactory

Beim Start der Anwendung wird die SessionFactory aus der Konfiguration einmal erzeugt und muss während der gesamten Laufzeit der Anwendung erreichbar sein. [Hien und Kehle

2007] Für die Anbindung der *PostgreSQL*-Datenbank musste keine Hibernate SessionFactory angelegt werden (siehe Kapitel 6.4.6).
Für die Anbindung der MSDE 2000 Datenbank wurde die automatisch erstellte und bis jetzt nicht genutzte Hibernate SessionFactory verwendet. Folgende Änderung führte zu dem gewünschten Ergebnis:

```
private static String CONFIG_FILE_LOCATION = "/biooffice.cfg.xml";
```

Der Pfad der Konfigurationsdatei verwies nun auf die MSDE 2000 Datenbank und damit wurde die Hibernate SessionFactory richtig umgebaut.

6.4.6 Hibernate Util und Hibernate Session

Um mit Hibernate ein Objekt zu sichern, zu laden oder zu aktualisieren, wird eine Session benötigt. Eine Session wird von einer SessionFactory erzeugt. [Hien und Kehle 2007] Eine Session ermöglicht den Zugriff auf Datenbankverbindungen und Transaktionen und bildet die *CRUD-Operationen*. Zur Laufzeit sollen Zugriffe auf unterschiedliche Sessions möglich sein und aus diesem Grund müssen zwei Klassen (HibernateUtil) erstellt werden, die eine Session mit einer statischen Methode zurückgeben.

Im Bereich „DigiPark“ wurde dafür die Klasse HibernateUtil.java verwendet, welche automatisch generiert wird. Folgende Codezeile bewirkt das automatische Aufrufen der Standard Konfigurationsdatei (hibernate.cfg.xml):

```
sessionFactory = new AnnotationConfiguration().configure().buildSessionFactory();
```

Wichtig ist hier die Anweisung new AnnotationConfiguration(), welche die Konfigurationsdatei (hibernate.cfg.xml) automatisch aufruft und aufgrund der Annotationen in den *POJOs* das Mapping und eine Session erzeugt.

Im Bereich „BioOffice“ wird über die Java-Klasse HibernateUtilBioOffice die SessionFactory aufgerufen und damit eine Session erzeugt. Für die zweite Session muss dieser Aufruf konfiguriert werden, da hier die Konfigurationsdatei biooffice.cfg.xml genutzt werden soll.

Listing 11 zeigt ein Hibernate Util Beispiel, welches die gewünschte Konfigurationsdatei aufruft:

```
HibernateSessionFactory.setConfigFile("/biooffice.cfg.xml");
HibernateSessionFactory. rebuildSessionFactory();
Configuration cfg = new AnnotationConfiguration().configure("/biooffice.cfg.xml");
sessionFactory = HibernateSessionFactory.getSessionFactory();
```

Listing 11: Beispiel Hibernate Util

Durch diese Vorgehensweise können mehrere Datenbanken innerhalb eines Hibernate-Projekts angebunden und dadurch unterschiedliche Sessions für beide Datenbanken erzeugt werden.

Im Bereich „DigiPark“:

```
Session session = HibernateUtil.getSessionFactory().openSession();
```

Im Bereich „BioOffice“:

```
Session session = HibernateUtilBioOffice.getSessionFactory().openSession();
```

6.4.7 Utilities

Dieses Kapitel beschäftigt sich mit zusätzlichen *Features.* Es wird nur ein kurzer Überblick gegeben, da eine genauere Betrachtung sehr aufwendig wäre. Für vertiefende Informationen wird im Text auf Quellen verwiesen.

Ant-Task

Durch die Verwendung des Apache-*Ant*-Tasks können Java Class Dateien erzeugt werden und bei Bedarf in die richtigen Verzeichnisse kopiert werden. Es können alle *Exporter* von Hibernate verwendet werden (siehe Kapitel 3.1.3). Wichtig ist dabei vor allem das Aktualisieren und Erzeugen des Datenbankschemas. Im Wurzelverzeichnis des Projekts wird die Datei build.xml angelegt und die enthaltenen *Ant*-Skripts übersetzen die Sourcen. Durch die Verwendung des Compile-Task werden die Klassen z.B. in das gewünschte Verzeichnis kompiliert. Für die Verwendung des *Ant*-Tasks wurden die Klassenbibliotheken freemarker.jar und hibernate-tools.jar zur Projektbibliothek hinzugefügt. Alle Informationen für die Erstellung des *Ant*-Skripts können unter dem *Link* http://www.hibernate.org/hib/docs/tools/-reference/en/html/ant.html nachgelesen werden.

Aus praktischer Erfahrung wird angeführt, dass das Aktualisieren und Erzeugen des Schemas mittels *Ant*-Tasks in der vorliegenden Version von Hibernate und JPA noch nicht funktioniert. Eine Anfrage bei JBoss™ bestätigte diesen Fehler, brachte jedoch keine Lösung. Im Beispiel Listing 12 ist ersichtlich, wie diese Aufgaben durch *Implementierung* in einer Java-Methode gelöst werden können.

```
Configuration cfg = new AnnotationConfiguration().configure(configFile);
SchemaExport schemaExport = new SchemaExport(cfg);
schemaExport.create(false, true);
```

Listing 12: Java Schema-Export

log4j

Um Fehler zu entdecken, wurde das Logging aktiviert. Dafür wurde das *Open Source* Logging Framework log4j verwendet. Für die Konfiguration wurde die Datei log4j.properties angelegt, welche sich im Klassenpfad befinden muss. Dadurch werden alle Logausgaben im Debug-Level von Hibernate in der Datei MeineLogDatei.log ausgegeben. Für die Verwendung von log4j muss die Klassenbibliothek log4j.jar zur Projektbibliothek hinzugefügt

werden. Exakte Informationen für die Verwendung des Frameworks können unter [log4j Homepage 2007] bezogen werden.

Mehrsprachigkeit

Eine Anforderung an das Projekt war die Realisierung einer mehrsprachigen Ausgabe der Benutzerinteraktion. Dafür wurde das Package java.util.Properties verwendet. Es wurde eine englische und deutsche *XML*-Datei erzeugt, wodurch die Benutzerinteraktion während der Laufzeit in deutscher oder englischer Sprache geführt werden kann. Genaue Informationen können unter [Ullenboom 2007] erhalten werden.

6.4.8 Top-Down-Verfahren im Bereich „DigiPark“

Das Top-Down-Verfahren bietet die Möglichkeit aus *POJOs* mittels Verwendung der JPA (Annotationen) das Datenbankschema zu erzeugen. Es müssen durch die Verwendung von Annotationen keine Metadateien angelegt oder mittels *Exporter* generiert werden.

6.4.9 Bottom-Up-Verfahren im Bereich „BioOffice“

Das Bottom-Up-Verfahren bietet die Möglichkeit aus einem existierenden Datenbankschema mittels *Exporter* (siehe Kapitel 3.3.1) *POJOs* zu generieren. Annotationen können ebenfalls generiert werden. Aus dem Datenbankschema der „BioOffice“-Datenbank wurden mittels des Tools von *MyEclipse POJOs* und Annotationen automatisch generiert. Da nicht alle Annotationen generiert werden oder fehlerhaft sind, müssen die generierten *POJOs* bzw. Annotationen verbessert oder ersetzt werden. Die dafür eingesetzten Techniken werden im Kapitel 6.6 erläutert.

6.5 Java 5

In diesen Java-Standard wurden, mit Erscheinen der neuen Java Release 5 (Codename Tiger), eine Reihe von hilfreichen und wichtigen Neuerungen aufgenommen. [Hien und Kehle 2007]

6.5.1 Annotationen

Annotationen bieten die Möglichkeit, Metadaten direkt in den Sourcecode zu hinterlegen. Zusätzliche Dateien zur Speicherung der Metadaten (*XML*- oder Property-Dateien) sind nicht mehr notwendig. Die per Annotation hinterlegten Metadaten können von *Exportern* für die Generierung des Datenbankschemas genutzt werden. Der Vorteil ist darin zu sehen, dass alle Informationen zentral im Sourcecode abgelegt sind. In Hibernate ist es somit möglich, Mapping-Informationen in Annotationen anzugeben. Eine ähnliche Technik bot vor der Verwendung von Annotationen Xdoclet. Xdoclet ist ein Werkzeug zum attributorientierten Arbei-

ten in Java und wurde nicht vom *Compiler* zur Laufzeit übersetzt. Eine Verwendung von Xdoclet in Javadoc ist somit auch eine veraltete Technologie. [Hien und Kehle 2007]

Folgende Annotationen laut HIEN wurden verwendet [Hien und Kehle 2007]:

- @Deprecated: Mittels dieser Annotation können Methoden und Klassen gekennzeichnet werden, welche nicht mehr verwendet werden.
- @Override: Wird verwendet um eine Methode zu markieren und um sicherzustellen, dass der *Compiler* eine Methode einer Basisklasse überschreibt.
- @SuppressWarnings: Dient zur Unterdrückung von *Compiler*-Warnungen, die durch das markierte Element (z.B. Methode) eingeschlossen werden.
- @Target: Mit Target wird definiert, welchen Elementen (Klasse, Methode, Parameter, usw.) eine Annotation zugeordnet werden kann.

```
@Override
public String toString() {
  return "Organsiation ID"+ this.getOrgId();
}
```

Listing 13: Verwendung einer Annotation

Alle anderen standardmäßig vorhandenen Annotationen wurden nicht genutzt. Es besteht auch die Möglichkeit, sich eine eigens definierte Annotation zu erzeugen.

6.5.2 Generics

Die in der Java Standard Edition 5.0 (Java 5) eingeführten Generics erlauben die *Abstraktion* von Typen. Es können damit Klassen und Methoden definiert werden, die generisch, also unabhängig von einem konkreten Typ, sind. In der verwendeten Version unterstützt Hibernate neuerdings dieses Konzept. Generics wurden in der vorliegenden Arbeit durchgehend verwendet.

6.6 *EJB* 3.0 - JPA

Die unter Kapitel 3.1.4 vorgestellte *EJB*-3.0-*Spezifikation* führte zur aktuellen JPA. Neu ist, dass diese *Spezifikation* auch unter Java SE, also in normalen Java-Anwendungen, eingesetzt werden kann. Hibernate und *EJB*-3.0 (JPA) verwenden den gleichen Ansatz, nämlich die Darstellung der Entitäten durch *POJOs*. [Hien und Kehle 2007]

6.6.1 Entities

Entities sind *persistente Objekte* und der zentrale Bestandteil der JPA. Es können abstrakte und konkrete Java-Klassen verwendet werden und auch Vererbung, polymorphe Assoziationen und polymorphe Abfragen werden unterstützt.

Der Einsatz von Entities ist laut HIEN an folgende Bedingungen gebunden [Hien und Kehle 2007]:

- Die Entity-Klasse muss mit der Annotation @Entity markiert sein.
- Die Entity-Klasse benötigt einen parameterlosen Konstruktor, der public oder protected ist.
- Die Entity-Klasse und die Methoden dürfen nicht als final deklariert sein.
- Die Entity-Klasse muss einen Primärschlüssel haben.

Die Art der Markierung persistenter Attribute muss innerhalb einer Klasse einheitlich sein, entweder die Klassenattribute oder die Getter-Methoden. In der vorliegenden *Implementierung* wurden die Klassenattribute markiert, um die Lesbarkeit zu verbessern. Folgende Typen sind als persistente Attribute erlaubt und wurden im Projekt eingesetzt [Hien und Kehle 2007]:

- alle primitiven Javatypen
- Java.lang.String
- serialisierbare Java-Klassen (z.B. java.util.Date)
- Byte[]
- Enums
- Java.util. Collection, Set, List und Map
- embeddable Klassen für zusammengesetzte Primärschlüssel
- andere Entity-Klassen und Collections von Entity-Klassen

6.6.2 Arbeiten mit Hibernate Entities.

Im Kapitel 3.3.1 wurde der Lebenszyklus einer Hibernate Entity theoretisch vorgestellt und nun wird auf die praktische Verwendung eingegangen. Im Wesentlichen geht es dabei um die Überführung einer Entity in die gewollten Zustände.

Zu Beginn ist jedes Entity-Objekt, nach der Erzeugung mit dem new Operator, im Zustand transient, das heißt, es besteht keine Verbindung zwischen der Hibernate Session und einer

Entity. In diesem Zustand verhält sich das Entity-Objekt wie ein normales Java-Objekt. Wird dieser Zustand beibehalten und verweist keine Referenz auf das Objekt, würde es vom Garbage-Collector gelöscht werden. Um nun das Objekt vom transienten Zustand in den persistenten Zustand überzuführen, können die Session-Methoden save(Objekt), saveOrUpdate(Objekt) oder persist(Objekt) genutzt werden. Es genügt auch wenn die neu erzeugte Entity von einer schon persistenten Entity referenziert wird. [Hien und Kehle 2007]

Die Entity befindet sich jetzt in einem persistenten Zustand, das heißt sie besitzt einen Primärschlüssel und sie ist einer Hibernate Session zugeordnet, jedoch muss sie noch keine Repräsentation in der Datenbank haben. Änderungen können also durch ein Rollback rückgängig gemacht werden. Objekte, die aus der Datenbank geladen werden, befinden sich automatisch im persistenten Zustand. Soll nun ein persistentes Objekt wieder in den transienten Zustand überführt werden, genügt die Session-Methode delete(Objekt). Die Daten werden dadurch innerhalb der Datenbank gelöscht. Wird das Objekt nicht mehr referenziert, löscht der Garbage-Collector dieses automatisch. [Hien und Kehle 2007]

Wird eine Session mittels der Anweisung session.close(Object) geschlossen, ist die Zuordnung der Entities zu einer Session beendet. Änderungen werden nicht mehr mit der Datenbank synchronisiert. Sollten jetzt aber Änderungen in der Datenbank durchgeführt werden, enthalten die Java-Objekte alte Daten. Diese Entities befinden sich nun im Zustand detached und können für den Datenaustausch genutzt werden. Werden diese Entities wieder einer Session zugeordnet, müssen die Session-Methoden update(Objekt), saveOrUpdate(Objekt) merge(Objekt) oder lock(Objekt) aufgerufen werden. [Hien und Kehle 2007]

Durch diese Zustandsänderungen einer Entity lassen sich auch alle *CRUD-Operationen* auf einfache Weise bewerkstelligen. Listing 14 zeigt ein Beispiel für eine Einfügemethode.

```
  public void insert(DigiObjects digio) {
       Session session = HibernateUtil.getSessionFactory().openSession();
              Organisation organisation =(Organisation) digio;
              Transaction tx = session.beginTransaction();
                     try {
                            session.persist(organisation);
                     } catch (PersistentObjectException e1) {
                            tx.rollback();
                            session.close();
                            e1.printStackTrace();
                            throw e1;
                     }catch (HibernateException e2) {
                            tx.rollback();
                            session.close();
                     }
              tx.commit();
       session.close();
}
```

Listing 14: Beispiel für eine Einfügemethode

6.6.3 Primärschlüssel

Der Primärschlüssel einer Entity muss mittels der Annotation @Id (siehe Listing 16) gekennzeichnet werden. Es besteht die Möglichkeit, über die Definition einer Primärschlüsselklasse, auch zusammengesetzte Schlüssel zu verwenden. In der vorliegenden Diplomarbeit wurde dieser Ansatz verwendet. Diese Embeddable Klassen werden mit der Annotation @Embeddable markiert und enthalten logischerweise keinen Primärschlüssel (siehe Listing 15). [Hien und Kehle 2007]

```
@Embeddable
public class CoordinateattributeId implements java.io.Serializable, DigiObjects {

        /** Attribut cooaid – Schlüssel der Relation Coordinate*/
        @Column(name = "cooaid", nullable = false, columnDefinition = "int4")
        @Type(type = "int")
        private int cooaid;

        /** Attribut cooasitidsite - Schlüssel der Relation Site /
        @Column(name = "cooasitidsite", nullable = false, columnDefinition = "int4")
        @Type(type = "int")
        private int cooasitidsite;

        /** Standard Konstruktor */
        public CoordinateattributeId() {
        }
}
```

Listing 15: Beispiel für einen zusammengesetzten Schlüssel (Ausschnitt der Klasse)

6.6.4 Generatorstrategien

Mit der Annotation @GeneratedValue der JPA kann ein ID-Generator gewählt werden. Diese Generatorstrategien sind besonders wichtig, da verschiedene *RDBMS* verschiedene Strategien benötigen. In diesem Kapitel wird auf die verwendeten Generatorstrategien eingegangen.

Im Bereich „DigiPark“ wurde der Generator Identity verwendet, welcher dem *PostgreSQL* serial Datentyp einer Spalte entspricht. Listing 16 zeigt ein Beispiel für die praktische Verwendung eines Generators.

```
  @Id
  @Column(name = "orgid")
  @GeneratedValue(strategy=GenerationType.IDENTITY)@GenericGenerator(name = "object_serial",
```

Listing 16: Beispiel für Generatorstrategie (Generator Identity)

Für Relationen, in denen kein Primärschlüssel definiert wird, ist die geeignete Strategie assigned. Diese Strategie ist die standardmäßige Einstellung und muss deshalb nicht gesetzt werden.

Im Bereich „BioOffice“ sollte die Strategie native verwendet werden, welche im Bereich der MS *SQL* Datenbanken Standard ist. Leider funktionierte keine Hibernate-Generatorstrategie

für die MSDE 2000 Datenbank. Die Strategie native funktioniert jedoch in allen anderen MS *SQL* Datenbanken. Auf dieses Problem wird im Kapitel 6.10 noch gesondert eingegangen.

6.6.5 Beziehungen

Die JPA unterstützt Beziehungen zwischen Entities und folgende Beziehungen wurden verwendet:

- 1-zu-1 (Annotation: @OneToOne)
- 1-zu-n (Annotation: @OneToMany)
- n-zu-1 (Annotation: @ManyToOne)

Die Beziehung m-zu-n wurde nicht verwendet, da diese bereits aufgelöst wurde. Folgende grundlegende Regeln, gilt es bei bidirektionalen Beziehungen zu beachten:

- Die referenzierte Seite verweist durch Angabe des Feldes mappedBy der Annotation (OneToMany oder ManyToOne) auf den Besitzer.
- Die „n“ Seite einer Beziehung ist die Besitzerseite.
- In einer 1-zu-1-Beziehung ist die Besitzerseite jene mit dem Fremdschlüssel in der Datenbank.
- Mittels @JoinColumn wird das referenzierte Attribut festgelegt.

Beispiel für eine 1-zu-n Beziehung und eine n-zu-1 Beziehung (Organisation zu Kontakt) siehe Listing 17 und Listing 18.

6.6.6 Transitive Persistenz

Im vorhergehenden Kapitel wurde die Erstellung von Beziehungen zwischen Entity-Klassen beschrieben. Im Kapitel 6.6.2 wurde das Arbeiten mit den Hibernate Entities behandelt. Mittels der transitiven Persistenz können Entitiy-Operationen wie z.B. persist(), merge() oder delete() an in Beziehung stehende Entities weitergegeben werden. Diese Technik ist mit der Lösch- und Aktualisierungsweitergabe in *RDBMS* am ehesten vergleichbar. Wird die Session-Methode persist(Objekt) für eine Instanz eines Objekts aufgerufen, erfolgt kein automatischer persist(Objekt) Aufruf für alle in Beziehung stehenden Objekte. Alle Beziehungsannotationen erlauben das Durchreichen von Entitiy-Operationen. Listing 17 und Listing 18 veranschaulichen die praktische Umsetzung der transitiven Persistenz.

Wird nun eine Instanz des Objekts Organisation durch eine Session-Methode persist(Objekt) persistent in die Datenbank geschrieben, kann (wenn so gewünscht) der gleiche Aufruf für alle in Beziehung stehende Kontaktobjekte geschehen (siehe Listing 17 und Listing 18). Unter

Verwendung der Annotation @org.hibernate.annotations.Cascade geschieht dies durch folgende Anweisung:

```
@org.hibernate.annotations.Cascade({org.hibernate.annotations.CascadeType.PERSIST})
```

```
@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "conorgid")
private Organisation organisation;
```

Listing 17: Kontakt („n“ Seite)

```
@OneToMany(fetch = FetchType.LAZY, mappedBy = "organisation")
@org.hibernate.annotations.Cascade({org.hibernate.annotations.CascadeType.PERSIST})
private Set<Contact> contacts = new HashSet<Contact>();
```

Listing 18: Klasse Organisation („1“ Seite)

Es gibt laut HIEN folgende CascadeTypes der JPA [Hien und Kehle 2007]:

- Persist
- Merge
- Remove
- Refresh
- ALL (entspricht Persist, Merge, Remove und Refresh)

Hibernate bietet aber noch die Möglichkeit, die CascadeTypes zu überschreiben.

6.6.7 Types

In den Hibernate Mappings kann über die Annotation @Type ein Hibernate Typ für ein Attribut einer Entity oder für eine Komponente vergeben werden. In der JPA sind bereits Mapping Types für eine Vielzahl von Javatypen (primitive Typen wie auch Klassen) enthalten. Das Beispiel zeigt die Verwendung der Annotation @Type:

```
@Column(name = "orgname", columnDefinition = "varchar(100)")
@Type(type = "java.lang.String")
String orgname;
```

Listing 19: Verwendung der Annotation @Type

Durch die Annotation @Column werden die Eigenschaften für ein Tupel in der Datenbank festgelegt und durch die Eigenschaft columnDefinition der Datentyp in der Datenbank. Viele Felder vom Datentyp varchar() weisen unterschiedliche Längen auf. In der vorliegenden Arbeit entstand das Problem, die Länge der String-Attribute festzulegen. Es gab zwei Lösungsansätze:

- Verwendung der Annotation @Length().

- Verwendung des StringBuffer-Objekts.

Die Verwendung von StringBuffer-Objekten wurde ausgewählt. Dadurch wurde aber die Erzeugung eines benutzerdefinierten Mapping Types notwendig, da ein StringBuffer-Objekt kein Standard JPA-Typ ist. Die Erstellung des benutzerdefinierten Mapping Types war, nach einigen Recherchen in Foren, nicht schwierig. Es wurde eine eigene Java-Klasse StringBufferToStringUserType *implementiert* und es konnte nun über den Namen dieser Klasse die Typzuweisung, wie im Beispiel Listing 20 ersichtlich, festgelegt werden.

```
@Column(name = "orgname", unique = true, nullable = false, columnDefinition="varchar(100)")
@Type (type = "usertypespostgresql.StringBufferToStringUserType")
private StringBuffer orgname = new StringBuffer(100);
```

Listing 20: Typzuweisung eines benutzerdefinierten Mapping Types

6.6.8 Collections in Hibernate

Beziehungen zwischen Entities werden mittels Collections umgesetzt. Um Collections als Attribute verwenden zu können, ist es notwendig, deren Typ als *Interface* zu deklarieren. [Hien und Kehle 2007]

Folgende *Interfaces* stehen zur Verfügung:

- java.util.Collection
- java.util.List
- java.util.Map
- java.util.Set
- java.util.SortedSet
- java.util.SortedMap

Es können *Interfaces* auch selbst definiert werden, wofür aber die *Implementierung* von org.hibernate.usertype.UserCollectionType notwendig ist. Es wurde fast ausschließlich java.util.Set verwendet. Ein Beispiel für die Erstellung einer Beziehung, unter Einbindung einer Collection, ist in den Beispielen Listing 17 und Listing 18 ersichtlich.

6.6.9 Transaktionen

Transaktionen in Hibernate basieren auf der Verwendung von *JDBC* sowie der Java Transaction *API* (*JTA*). Das Verhalten einer Hibernate-Anwendung unterscheidet sich bzgl. des transaktionalen Verhaltens nicht von einer herkömmlichen *JDBC*-Persistenzlösung. Jede Dankenbankoperation von Hibernate muss innerhalb einer Transaktion ausgeführt werden.

Der Auto-Commit-Modus wird durch Hibernate deaktiviert. Listing 14 zeigt ein Beispiel für eine Insert-Methode unter Verwendung einer Transaktion.

6.7 Fetching-Strategien und *Caches* (*Performance*-Optimierung)

In diesem Kapitel soll ein Überblick über die Möglichkeiten der *Performance*-Optimierung von einer mit Hibernate und der JPA erstellten Anwendung gegeben werden. Folgende Informationen beruhen auf [Hien und Kehle 2007], [Hibernator Homepage 2007] und [Beeger et al. 2006].

6.7.1 Fetching-Strategien

Die meisten *Performance*-Probleme entstehen durch Datenbankzugriffe. Dafür bietet Hibernate das Konzept der Fetching-Strategien wie folgt an [Hien und Kehle 2007]:

- Lazy Load: Die Daten werden erst geladen, wenn ein Zugriff auf das Attribut oder das Objekt erfolgt.
- Eager Load: Die Daten werden sofort und vollständig geladen.

Diese Fetching-Strategien können bei Attributen und Beziehungen angegeben werden, wurden jedoch nur bei Beziehungen eingesetzt. In der vorliegenden Arbeit wurde fast ausschließlich die Fetching-Strategie Lazy Load verwendet. Wird nun eine Beziehung als Lazy Load definiert, dann wird bei einem Zugriff auf das in Beziehung stehende Objekt eine Exception geworfen. Um dieses Problem zu vermeiden, muss vor einem Zugriff das in Beziehung stehende Objekt initialisiert werden.

```
Hibernate.initialize(dobject.getContact());
contact = (Contact)session.get(Contact.class,dobject.getContact().getConidcontact());
contact.adddObject(this);
```

Listing 21: Initialisierung bei Fetching-Strategie Lazy Load

Im Beispiel Listing 21 wird zu einem schon vorhandenen Contact-Objekt ein neues dObjekt-Objekt hinzugefügt (Kontakt, welcher eine Beobachtung gemacht hat). Die Beziehung zwischen der Klasse dObject zu Contact ist mittels der Fetching-Strategie Lazy Load realisiert. Um nun ein Contact-Objekt zu aktivieren, wird diese Initialisierung (siehe Listing 21) benötigt und kann dann auf das persistente Contact-Objekt zugreifen.

Die Strategie Eager Load erspart eine vorhergehende Initialisierung, verursacht aber *Performance*-Probleme. Dadurch empfiehlt sich die Verwendung von Lazy Load.

6.7.2 Batch-Fetching

Die Batchgröße kann bei einem Zugriff auf eine Relation angegeben werden und erhöht dadurch die *Performance*. Bei jedem Zugriff werden entsprechend der Batchgröße die Objekte im Voraus geladen. Mit HQL sieht die Abfrage wie folgt aus [Hien und Kehle 2007]:

```
from organisation org batch fetch size= 5 org.contact
```

Das Batch-Fetching kann für eine Beziehung auch global mittels Annotation definiert werden [Hien und Kehle 2007]:

```
@org.hibernate.annotations.BatchSize(size = 5)
```

6.7.3 *Join*-Fetching

Mittels *Join*-Fetching lässt sich die Anzahl der Datenbankzugriffe auf einen einzigen reduzieren, indem mittels eines *Joins* die abhängigen Datensätze geladen werden. Mit HQL sieht die Abfrage wie folgt aus:

```
from organisation org join fetch org.contact
```

Diese Abfrageoptimierung entspricht der schon vorgestellten Fetching-Strategie Eager Load bei Beziehungen.

6.7.4 Hibernate Query *Cache*

Für sich wiederholende Abfragen macht es Sinn, den Hibernate Query *Cache* zu aktivieren. Der Query *Cache* hinterlegt nur die Primärschlüssel in der Ergebnisliste einer Abfrage. Um eine bessere *Performance* zu erreichen, muss ein Second Level *Cache* vorhanden sein, der die Entities für die Primärschlüssel im Query *Cache* bereitstellen kann. Die Aktivierung des Query *Caches* erfolgt in der Konfiguration (hibernate.cfg.xml) mit folgender Anweisung:

```
hibernate.cache.user_query_cache
```

6.7.5 Second Level *Cache*

Caches werden generell genutzt, um Datenbankanwendungen zu optimieren. Der *Cache* hält bereits geladene Daten. Datenbankzugriffe sind nur nötig, wenn die gesuchten Daten im *Cache* nicht enthalten sind. Für vertiefende Informationen zu diesem Kapitel empfiehlt sich eine Recherche in [Hien und Kehle 2007], da nicht alle Strategien in diesem Rahmen vorgestellt werden. Hibernate nutzt einen First Level *Cache* (enthalten in der Hibernate Session) welcher auf die Lebensdauer der Session beschränkt ist. Der Second Level *Cache* ist hierarchisch oberhalb der Hibernate Session bzw. des EntityManagers angeordnet, das bedeutet, dass die Entities der gesamten Anwendung zur Verfügung stehen. Die *Caching*-Strategie kann auf Entity-Ebene und für Collections konfiguriert werden. Beispiel für die Annotation auf Ebene einer Entity:

```
@Cache(usage = CacheConcurrencyStrategy.Read_Write)
```

Auf Ebene einer Beziehung:

```
@Cache(usage = CacheConcurrencyStrategy.NONSTRICT_Read_Write)
```

Der *Cache* Provider wird in der Konfiguration (hibernate.cfg.xml) mit folgender Anweisung definiert:

```
<property name="hibernate.cache.provider_class">
org.hibernate.cache.OSCacheProvider </property>
```

Es gibt keine für alle Anwendungsfälle perfekt passende Fetching-Strategie oder *Caching*-Strategie. Die Beziehungen sollen grundsätzlich als lazy definiert werden. Second Level *Caches* sollten nach *empirischen Performance*-Tests dort gesetzt werden, wo die beste *Performance*-Optimierung erreicht werden soll.

6.8 Locking mit Hibernate (Mehrbenutzerbetrieb)

In diesem Kapitel wird auf die Möglichkeiten eines Mehrbenutzerbetriebes mit Hibernate eingegangen. Die „DigiPark"-Anwendung ist eine Einzelplatzversion. Die nachfolgenden, im Überblick beschriebenen, Techniken wurden nicht in die praktische Umsetzung des Projekts einbezogen. Die „DigiPark"-Anwendung könnte durch die erläuterten Techniken relativ einfach in eine Mehrbenutzeranwendung umgebaut werden. Folgende Informationen beruhen auf [Hien und Kehle 2007] und [Beeger et al. 2006].

6.8.1 Optimistisches Locking mit Hibernate

Um der Forderung nach möglichst hoher *Performance* gerecht zu werden, wird optimistisches Locking verwendet. Es wird dabei prinzipiell angenommen, dass es zu keinen Überschneidungen von Änderungen durch parallele Transaktionen kommt. Optimistisches Locking bedeutet in diesem Fall, dass es erst nach dem Auftreten einer Überschneidung eine aufwendige manuelle Korrektur gibt. Jede Entity bekommt ein Versionsattribut, welches bei einem Update mit dem Wert in der Datenbank verglichen wird und sich bei jedem Update erhöht. Stimmt dieses Attribut nicht überein, wurde eine Entity zwischenzeitlich durch einen parallelen Zugriff verändert. Es besteht nun die Möglichkeit, die Versionen manuell zu überprüfen. Durch folgende Anweisung innerhalb einer Session und Trankaktion wird die Versionsnummer zurückgegeben:

```
int versionsNummer = organisation.getVersion();
```

Der aktuelle Zustand der Entity wird dann aus der Datenbank geladen und die Versionsnummern verglichen. Bei Ungleichheit ist die Entity zwischenzeitlich verändert worden. Die manuelle Überprüfung ist sehr aufwendig und deshalb wird von Hibernate eine automatische Versionsprüfung angeboten. Diese automatische Versionsprüfung kommt bei der Ver-

wendung von erweiterten Sessions und von Detached Objects (siehe Kapitel 6.6.2) zum Einsatz. Eine erweiterte Session wird am Ende einer Transaktion nicht geschlossen und somit bleiben alle Entities erhalten, die mit dieser Session geladen wurden. In diesen Fällen überprüft Hibernate vor der Ausführung von Updates automatisch die Version der Entities. Stimmen die Einträge nicht überein wird eine StaleObjectStateException geworfen, um den aufgetretenen Konflikt anzuzeigen. Mit dem Aufruf lock(Object object, LockMode lockmode) der Session kann die Versionsprüfung auch für Entities durchgeführt werden, die nicht innerhalb einer Transaktion verändert wurden. Für alle zusätzlichen Möglichkeiten wird auf [Hien und Kehle 2007] und [Hibernate Homepage 2007] verwiesen.

6.8.2 Pessimistisches Locking mit Hibernate

In diesem Fall wird von häufigen Kollisionen zwischen konkurrierenden Transaktionen ausgegangen. Es wird ein sehr restriktives Vorgehen verwendet und Tabellenzeilen werden explizit gesperrt, noch bevor eine Überlappung stattfinden kann. Dadurch kann es zu keinen nachträglichen Konflikten kommen. Sollte eine konkurrierende Transaktion versuchen, die gesperrten Zeilen zu verwenden, wird umgehend eine Fehlermeldung ausgegeben oder die Transaktion muss warten, bis die Zeilen wieder freigegeben werden. Nachteil des pessimistischen Lockings ist die sich verschlechternde *Performance* durch die wartenden Transaktionen und daher ist der Einsatz des pessimistischen Lockings gut zu planen. Hibernate nutzt für dieses Locking ausschließlich die Funktionalität der Datenbank, es werden keine Locks auf Entity-Objekte im Speicher gesetzt. Die Klasse LockMode definiert die verschiedenen Lock-Möglichkeiten, die innerhalb von Hibernate verwendet werden können. Für zusätzliche Informationen wird auf [Hien und Kehle 2007] und [Hibernate Homepage 2007] verwiesen.

6.9 Abfrage Techniken

Im Kapitel 3.1.3 wurden die Abfragemöglichkeiten von Hibernate schon theoretisch vorgestellt. Nun wird kurz auf die im Projekt „DigiPark“ verwendeten Abfragetechniken eingegangen.

6.9.1 Abfragen mittels Query *Interface*

Das Query *Interface* ist die zentrale Schnittstelle für Abfragen. Sie wird dann eingesetzt, wenn die Primärschlüssel einer Entity nicht bekannt sind. Erzeugt wird eine Query-Instanz mit der aktuellen Session. Listing 22 zeigt ein Beispiel für eine Abfrage unter Verwendung des Query *Interfaces*. [Hien und Kehle 2007]

```
Session session = HibernateUtil.getSessionFactory().openSession();
Transaction tx = session.beginTransaction();
Query q = session.createQuery(hql.toString()) ;
List<DigiObjects> result = q.list();
tx.commit();
session.close();
```

Listing 22: Abfrage mittels Query *Interface*

6.9.2 Abfragen in den Metadaten

Der folgende Abfragetyp wird bei Verwendung der JPA in den Annotationen definiert. Diese Abfrage wurde nur zu Versuchszwecken *implementiert*. Beispiel für eine Abfrage in den Metadaten mit @NamedQuery:

```
@NamedQuery(name="objects.Contact.alldatas", query ="from Contact")
```

Der Aufruf erfolgt durch Verwendung des Query *Inferfaces*:

```
Query q = session.getNamedQuery(„objects.Contact.alldatas”)
```

6.9.3 HQL (Hibernate Query Language)

Dieser Abfragetyp wurde am häufigsten verwendet. Mittels HQL kann objektorientiert gearbeitet werden und es werden Abfragen von Objekten über Vererbungsbeziehungen unterstützt. [Hien und Kehle 2007] Der Aufbau erinnert prinzipiell an *SQL*, es bestehen aber von Fall zu Fall leichte Unterschiede. Die Verwendung von HQL gestaltet sich einfach, jedoch kann es zu Problemen bei Abfragen kommen, wenn Beziehungen mit der Fetching-Strategie Lazy Load definiert wurden. Die Lösung für dieses Problem wurde im Kapitel 6.7.1 beschrieben.

```
public static List<DigiObjects> getSelection(String objectName,
       Map<String, String> queryContent) {
       StringBuffer hlq = new StringBuffer("from objects." + objectName + " "+ "as" + " " +
"digi ");
       if (queryContent != null) {
              String query = createWhereStatement(queryContent);
              hlq.append(query);
       }
       Session session = HibernateUtil.getSessionFactory().openSession();
       Query q = session.createQuery(hlq.toString());
       List<DigiObjects> result = q.list();
       session.close();
       return result;
}
```

Listing 23: Abfrage mit HQL

Listing 23 zeigt eine Methode, welche, je nach Eingangsparameter, unterschiedliche HQL Select-Abfragen automatisch generiert.

6.9.4 CriteriaAPI

Die Verwendung des Abfragetyps CriteriaAPI ist eine Alternative zu HQL und ebenfalls vollständig objektorientiert. Ergebnismengen können über Restrictions eingeschränkt werden. Dieser Abfragetyp wurde nur für Testzwecke verwendet. Listing 24 zeigt ein Beispiel für die Verwendung der CriteriaAPI anhand einer Select-Abfrage (Abfrage eines Contact-Objekts unter Verwendung des Primärschlüssels).

```
@SuppressWarnings({ "unchecked", "hiding" })
public  <DigiObjects>List getDataFromRelationByIdByCriteriaApi(DigiObjects digio){
        Session session = HibernateUtil.getSessionFactory().openSession();
        Collection collection = (Collection) digio;
        Criteria criteria = session.createCriteria(Collection.class);
        criteria.add(Restrictions.eq("colidcollection",collection.getColidcollection()));
         List <DigiObjects>result = (List<DigiObjects>) criteria.list();
        session.close();
        return result;
}
```

Listing 24: CriteriaAPI-Verwendung

6.10 *Replikation* „BioOffice“ zu „DigiPark“

Die *Replikation* von der „DigiPark“-Datenbank zur „BioOffice“-Datenbank und umgekehrt befindet sich durch den Abbruch des Projekts im Zustand eines fortgeschrittenen Prototypen.

Das Konzept für die *Implementierung* der *Replikatio*n war aus folgenden Gründen sehr schwierig zu erstellen:

- „BioOffice“ und „DigiPark“ beruhen auf unterschiedlichen Datenmodellen.
- Unklarheiten, welche Relationen repliziert werden sollen.
- Rückstand des Admin-Tools der Web-Applikation.
- MSDE 2000 Datenbank ist mit Hibernate eingeschränkt nutzbar (Key-Generator).
- Komplexes Datenmodell der „BioOffice“-Datenbank.

Die *Implementierung* deckt folgende Bereiche ab:

- Einfügen und Aktualisieren der Daten von „BioOffice“ zu „DigiPark“.
- Einfügen und Aktualisieren der Daten von „DigiPark“ zu „BioOffice“.

Das Löschen von Daten wurde aufgrund des Projektabbruches nicht mehr *implementiert*. Die größte Schwierigkeit war, dass es keine geeignete Hibernate-Generatorstrategie für eine MSDE Datenbank gibt. Dadurch musste das Einfügen von Daten in der Form gelöst werden, dass alle Einfügemethoden eine Referenz zwischen einer schon persistenten Entity zu einer neuen Entity erzeugen.

6.11 Austauschformat *XML*

In der Projektplanung wurde *XML* als Austauschformat festgelegt. In der Projektgruppe herrschten unterschiedliche Meinungen, wie die Dokumentstrukturen beschrieben werden sollten. Meine Projektkollegen favorisierten den Einsatz einer Document Type Definition (DTD). Es sollte im Projektteil *PDA*-Applikation eine *XML*-Datenbank verwendet werden. Da es zu Verzögerungen bei der *Implementierung* der *XML*-Datenbank kam, wurde vom Diplomanden des vorliegenden Projektteils die *XML*-Schnittstelle erstellt, die dem Schema der *XML*-Datenbank entspricht.

Anmerkung: Normalerweise kann ein *XML*-Schema mittels Software z.B. xml spy generiert werden aber xml spy unterstützt keine *PostgreSQL*-Datenbank und so musste das *XML*-Schema manuell *implementiert* werden.

6.11.1 Anforderungen

Folgende Anforderungen wurden durch die Projektplanung festgelegt:

- Die Dateninhalte sollen überprüfbar sein.
- Die Beziehungen sollen überprüfbar sein.
- Die *XML*-Datei soll als Grundlage für die Datenbank am *PDA* dienen.
- Die Primarschlüssel und Unique-Elemente sollen überprüft werden.
- Die Länge des Datentyps String soll überprüfbar sein.
- Nullable eines Attributes oder Elements soll festlegbar sein.

6.11.2 DTD versus *XML*-Schema

SEEMANN beschreibt die Vor- und Nachteile des *W3C XML*-Schema (WXS) gegenüber einer DTD wie folgt [Seemann 2003]:

- Ein Nachteil des WXSs ist die zu erlernende komplexe Sprache im Vergleich zu DTD.
- Mittels WXS können die Inhalte von Elementen und Attributen dahingehend überprüft werden, ob sie einem bestimmten Datentyp entsprechen. Eine DTD kennt nur Text. Die Kardinalitäten des konzeptuellen Entwurfs können von einer WXS genau übernommen werden, bei einer DTD besteht nur die Möglichkeit des minimalen Auftretens 0 oder 1 und des maximalen Auftretens 1 oder n.
- Der objektorientierte Ansatz des WXSs setzt sich auch in den *XML*-Dokumenten fort. Dadurch ist der Informationsverlust, der durch die Umwandlung des konzeptuellen in

ein logisches Schema entstehen würde, geringer als bei der Verwendung von DTD. Jedes WXS ist ein *XML*-Dokument und dies ist einer der wichtigsten Vorteile gegenüber einer DTD.

Diese Vorteile und die Anforderungen des Projekts führten zur Verwendung eines WXSs.

6.11.3 *W3C XML*-Schema als Datenbank

Die Entwicklung des *XML*-Schemas wird nun praktisch vorgestellt. Der erste Schritt ist, das schon vorhandene *PostgreSQL*-Datenbankschema als Ausgangspunkt für die Entwicklung des *W3C XML*-Schemas zu verwenden.

Struktur

Der nächste Schritt war die Festlegung der Struktur. Laut SEEMANN gibt es drei Strukturen für den Aufbau eines WXSs [Seemann 2003]:

- Matroschka: Nacheinander werden die Strukturelemente mit weiteren Strukturen gefüllt.
- Salami-Taktik: Definition der Elemente und Attribute auf oberster Ebene unter dem Schemaelement. Die globalen Struktureinheiten werden dann wieder referenziert. Das Problem dabei ist, dass alle Strukturelemente zu einem Namensraum gehören.
- Jalousie: Die Strukturelemente werden als benannte Typen definiert, dadurch wird eine Typhierarchie aufgebaut. Strukturelemente können zu unterschiedlichen Namensräumen gehören.

Nach einigen Versuchen fiel die Entscheidung auf die Variante Jalousie, da sie für die weitere Verwendung des WXSs am besten geeignet erschien.

Datentypen

Im WXS werden zwei Datentypen unterschieden, einfache Datentypen und komplexe Datentypen. [Seemann 2003]

Einfache Datentypen werden durch Einschränkung schon vorhandener Datentypen (z.B. String, int oder short) definiert.

```
<xsd:simpleType name="PLZ">
        <xsd:restriction base="xsd:string">
                <xsd:minLength value="4" />
                <xsd:maxLength value="13" ></xsd:maxLength>
                <xsd:whiteSpace value="collapse" />
                <xsd:pattern value="[0-9,a-z,A-Z,\-]{1,10}" />
        </xsd:restriction>
</xsd:simpleType>
```

Listing 25: globalgültiger Datentyp im *XML*-Schema

Im Beispiel Listing 25 wird der Inhalt des einfachen globalen Datentyps Postleitzahl (enthalten in Organisation und Kontakt) festgelegt und überprüft.

```
<xsd:simpleType name="varchar_100">
	<xsd:restriction base="xsd:string">
		<xsd:minLength value="0" />
		<xsd:maxLength value="100" />
	</xsd:restriction>
</xsd:simpleType>
```

Listing 26: Beispiel simple Type im *XML*-Schema

Im Beispiel Listing 26 wird die Länge eines String-Elements festgelegt und überprüft. Im Beispiel Listing 27 wird die Verwendung des einfachen Datentyps „varchar_100“ demonstriert.

Komplexe Datentypen werden gebildet, indem ihr Inhalt definiert wird. Wenn dem komplexen Datentypen ein Name zugewiesen wird, kann er von jeder Stelle des Schemas aus referenziert werden. Listing 27 zeigt ein Beispiel für einen komplexen Datentyp.

```
<!—The rows of the relation organisation -->
<xsd:complexType name="organisation">
	<xsd:sequence>
		<xsd:element name="orgid" type="INTEGER"
			nillable="false" minOccurs="1" maxOccurs="1">
		</xsd:element>
		<xsd:element name="orgname" type="varchar_100"
			nillable="false" minOccurs="1" maxOccurs="1" >
		</xsd:element>
	</xsd:sequence>
</xsd:complexType>
```

Listing 27: komplexer Datentyp im *XML*-Schema

Das Beispiel beschreibt die Erzeugung des komplexen Datentyps Organisation mit einem Auszug der dafür benötigten Elemente und entspricht auszugsweise der Relation „Organisation“ der „DigiPark“-Datenbank. Durch die Angabe der Eigenschaft nillable kann festgelegt werden, ob ein Element null sein darf.

Primary key, Foreign key und Unique

In einem relationalen Datenbankschema können Primary Keys, Foreign Keys und Unique Elemente festgelegt werden. In einem WXS sind diese Einstellungen auch möglich. Beispiel Listing 28 zeigt die Implementierung dieser Einstellungen.

```
<!—key, foreign key unique for relation contact-->
<xsd:key name="contactPrimaryKey">
	<xsd:selector xpath="contacts/*" />
	<xsd:field xpath="conidcontact" />
</xsd:key>
<xsd:unique name="contactUniqueKey">
	<xsd:selector xpath="contacts/*" />
	<xsd:field xpath="concontactname" />
</xsd:unique>
<xsd:keyref name="contactForeignKey" refer="organisationKey">
	<xsd:selector xpath="contacts/*" />
	<xsd:field xpath="conorgid" />
</xsd:keyref>
```

Listing 28: key, foreign key und unique im *XML*-Schema

6.11.4 Erfahrungsbericht

Die Erstellung eines WXS ist für einen WXS-Neuling am Anfang sehr schwierig, jedoch wird die in der Projektplanung festgelegte Funktionalität schnell erreicht und alle Anforderungen (siehe Kapitel 6.11.1) werden erfüllt. Es müssen sehr viele Codezeilen geschrieben werden, wenn ein großes Datenbankschema mit WXS verwirklicht werden soll. Sollte es möglich sein, sich ein WXS mittels Software z.B. xml spy vom Datenbankschema erstellen zu lassen, müssen nur mehr Verbesserungen gemacht werden. Alle Dateninhalte können wie gefordert überprüft werden. Dadurch können alle Daten, die in die Datenbank des Projektes eingespielt werden, durch eine *Validierung* mittels *Parser* auf Richtigkeit überprüft werden. Im praktischen Einsatz hat sich das WXS sehr bewährt.

7 Ausblick

In diesem Kapitel werden ein Überblick über das Gesamtprojekt, Anregungen für die Weiterführung des Projekts und ein persönliches Resümee gegeben.

7.1 Stand des Projekts

Das Projekt „DigiPark“ wurde im April 2007 unterbrochen. In Abbildung 14 ist eine Übersicht über die Projektarchitektur ersichtlich. Die drei Teile des Projekts weisen einen unterschiedlichen Grad der Fertigstellung auf.

7.1.1 Web-Applikation

In diesem Teilbereich des Projekts sollten Themengebiete für Interessenten mittels *UMN MapServer 4.0* visualisiert werden. Es sollte auch ein Admin Tool entwickelt werden, um die Anwendung zu verwalten und zu steuern. Der Fertigstellungsgrad dieses Teilbereiches ermöglicht noch keine praktische Verwendung.

7.1.2 *PDA*-Applikation

In diesem Teilbereich sollten *Biodiversitäts*-Daten vor Ort mittels einer *PDA*-Applikation erhoben werden. Als Erweiterung des Angebots sollte auch eine grafische Schnittstelle für den Nationalpark Guide entwickelt werden, um die zu besichtigenden Informationen in einer Karte am *PDA* zu visualisieren. Mit der Entwicklung des Nationalpark Guides wurde noch nicht begonnen. Der Fertigstellungsgrad dieses Teilbereiches ermöglicht noch keine praktische Verwendung.

7.1.3 Datenbanken und Schnittstellen

Als zentrales Element sollte eine Serverdatenbank entwickelt werden, von der aus die Web-Applikation und die *PDA*-Applikation mit den benötigten Daten versorgt werden. Durch die *Replikation* mit der „BioOffice“-Datenbank sollte der durchgehende digitale Datenfluss beschleunigt werden. Die Verwertung von Beobachtungsdaten für statistische Auswertungen mittels der Standardsoftware „BioOffice“ sollte ermöglicht werden.

Folgende Anforderungen wurden erfolgreich umgesetzt:

- „DigiPark“-Datenbank
- Schnittstelle zwischen Serverdatenbank und *PDA*-Datenbank
- Exportschnittstelle der *RDB* zur Web-Oberfläche
- *Replikation* zur „BioOffice“-Datenbank (fortgeschrittener Prototyp)
- Mehrsprachigkeit der Benutzerinteraktion
- *XML*-Austauschformat
- Datengrundlage für die beabsichtigte Routenplanung
- Zusätzliche Beobachtungsattribute

Zusätzlich wurde das *XML*-Schema für die *XML*-Datenbank des Projektteils „*PDA*-Applikation“ entwickelt.

Das generelle Ziel des Projekts, einen durchgehenden digitalen Datenfluss zu realisieren, wurde nicht erreicht. Eine Weiterführung des Projekts ist derzeit nicht geplant. Teile des Projekts sollen in neue Lösungen integriert werden.

7.2 Empfehlungen für das Projekt „DigiPark“

Für einen eventuell geplanten Wiederanlauf empfiehlt sich die Beachtung folgender Punkte:

- Die Anbindung der „BioOffice“-Datenbank sollte neu geplant werden.
- Da die „DigiPark“-Anwendung hohe Ansprüche an die *Performance* stellt, sollte die *Performance* der *ORM*-Produkte *empirisch* erhoben werden.
- Der *Performance*-Optimierung von Hibernate und der *PostgreSQL* Datenbank sollten mehr Beachtung geschenkt werden.
- Eine Überarbeitung des Datenmodells im Bereich „BioOffice“ wäre wünschenswert (durch BioGis Consulting).

- Ein eigenes Projekt für die Entwicklung des Nationalpark Guides.
- Der Einsatz von größeren personellen Ressourcen.
- Einreichung eines Bundesprojekts oder EU-Projekts (Vorbild „WebPark“ des Schweizer Nationalparks).
- Die Sicherstellung der technischen Realisierbarkeit (GPS-Empfang) beachten.
- Projektteile sollten nicht gleichzeitig, sondern in einer geplanten Reihenfolge begonnen werden (Vermeidung von Wechselwirkungen zwischen den Projektteilen).
- Die Zusammenarbeit und die Kommunikation zwischen den Projektbeteiligten sollten genau festgelegt werden.
- Es sollten Fortschrittskontrollen durchgeführt werden.

8 Persönliches Resümee

Die Erstellung einer *ORM*-Anwendung sollte im Hinblick auf die zunehmende Bedeutung vermehrt Einzug in Lehrveranstaltungen finden. Welcher *ORM*-Ansatz verwendet wird, hängt sehr stark von den Gegebenheiten des Projekts ab. Für einen *ORM*-Neuling empfiehlt es sich, zuerst an kleinen Beispielprojekten die unterschiedlichen *ORM*-Ansätze zu erlernen, um dann die Auswahl für ein Projekt treffen zu können.

Für mich stellte die theoretische und praktische Umsetzung einer *ORM*-Anwendung eine Bereicherung meiner Ausbildung dar. Ich konnte mein theoretisches Wissen über verschiedene Techniken (z.B. Java, *XML* und *SQL*-Datenbanken) praktisch einsetzen und neue Techniken (z.B. *XML*-Schema, *ORM*) erlernen.

Literaturverzeichnis

[Apple EOF 2007] http://developer.apple.com/documentation/WebObjects/Enterprise_Objects/, 02.05.2007

[Apple WebObjects 2007] http://www.apple.com/webobjects, 01.05.2007

[ASM 2007] http://asm.objectweb.org/, 10.05.2007

[Bauer und King 2007] BAUER Christian, Gavin KING (2007): Java Persistence mit Hibernate. München: Hanser Fachbuchverlag, 880 S.

[BEA Kodo 2007] http://bea.com/framework.jsp?CNT=index.htm&FP=/content/products/weblogic/kodo/, 20.05.2007

[Beeger et al. 2006] BEEGER Robert, Arno HAASE und Stefan ROOCK et al. (2006): Hibernate. Persistenz in Java-Systemen mit Hibernate 3. Heidelberg: dpunkt.verlag GmbH, 348 S.

[Bergmann 2005] BERGMANN Sebastian (2005): Professionelle Softwareentwicklung mit PHP 5, Objektorientierung, Entwurfsmuster, Modellierung und fortgeschrittene Datenbankprogrammierung. Heidelberg: dpunkt.verlag GmbH, 201 S.

[Booch et al. 1999] BOOCH Grady, Jim RUMBAUGH und Ivar JACOPSON (1999): Das UML-Benutzerhandbuch. Bonn: Addison Wesley Verlag, 561 S.

[Cayenne Homepage 2007] http://cayenne.apache.org/why-cayenne.html, 19.05.2007

[CocoBase Homepage 2007] http://www.thoughtinc.com/cber_index.html, 20.05.2007

[Dr.Dobb's Portal 2007] http://www.ddj.com/184406344#l5, 13.05.2007

[Duden 2007] http://www.duden.de/, 20.05.2007

[EasyBeans Homepage 2007] http://easybeans.objectweb.org, 07.05.2007

[Edv-Buch 2007] http://www.edv-buchversand.de/suse/chapter.php?cnt=getchapter&id=ep-42965_0.pdf, 15.05.2007

[Eisentraut 2003] EISENTRAUT Peter (2003): PostgreSQL. Das Offizielle Handbuch. Bonn: mitp-Verlag, 814 S.

[Fischer 2003] FISCHER Thorsten (2003): UMN MapServer 4.0. Handbuch und Referenz. Berlin: MapMedia GmbH, 286 S.

[Flanagan 2003] FLANAGAN David (2003): Java in a Nutshell. Köln: O'Reilly Verlag GmbH & Co. KG, 561 S.

[GNUstep Homepage 2007] http://www.gnustep.org/, 18.05.2007

[Hansen 1998] HANSEN Hans Robert (1998): Wirtschaftsinformatik I. 7 Auflage. Stuttgart: Lucius & Lucius, 1186 S.

[Heinrich et al. 2004] HEINRICH Lutz j, Armin HEINZL und Friedrich ROITHMAYR (2004): Wirtschaftsinformatik Lexikon. München: R.Oldenbourg Verlag München Wien, 935 S

[Heise 2007] http://www.heise.de/open/artikel/70100/1, 15.10.2006

[Herke 2007] http://209.85.135.104/search?q=cache:xAdwJ1Lc9uEJ:se2007.de/downloads/WG0/150-herke.pdf+herke+nachteile+objektrelationales&hl=dec &t=clnk&cd=1&gl=at, 08.05.2007

[Hibernate Homepage 2007] http://www.hibernate.org/, 25.04.2007

[Hibernator Homepage 2007] http://hibernator.sourceforge.net/, 20.05.2007

[Hien und Kehle 2007]	HIEN Robert, Markus KEHLE (2007): Hibernate und die Java Persistence API. Paderborn: entwickler.press, 269 S.
[Hitz und Kappel 1999]	HITZ Martin, Gerti KAPPEL (1999): UML@WORK. Von der Analyse zur Realisierung. Heidelberg: dpunkt.verlag GmbH, 358 S.
[Horn 2007]	www.torsten-horn.de/techdocs/java-hibernate.htm2, 07.05.2007
[it-fws EJB 2007]	http://www.it-fws.de/publics/orm022005.pdf, 12.05.2007
[ITWissen 2007]	http://www.itwissen.info/definition/lexikon, 12.06.2007
[JBoss 2007]	http://www.jboss.com/pdf/HibernateBrochure-Jun2005.pdf, 14.05.2007
[JDBCPersistence Homepage 2007]	http://www.jdbcpersistence.org/c/jdbcpersistence.html, 11.05.2007
[JDO-Architektur 2007]	http://www.oio.de/m/*EJB*-persistenz-jdo/jdo-architecture-none-managed-1.htm, 04.05.2007
[JIGS Homepage 2007]	http://www.gnustep.it/jigs/, 18.05.2007
[Jordan und Russel 2003]	JORDAN David, Craig RUSSELL (2003): Java Data Objects. Cambridge: O'Reilly Media, 380 S.
[JPOX Homepage 2007]	http://www.jpox.org/, 17.05.2005
[JSR 220 2007]	http://www.jcp.org/en/jsr/detail?id=220, 12.05.2007
[Langer et al. 2007]	http://files.hanser.de/hanser/docs/20061026_2612694133-47_Oates_Leseprobe.pdf, 14.05.2007
[Lexikon 2007]	http://networks.siemens.de/communications/lexikon_en/5/f005425.htm, 07.06.2007

[log4j Homepage 2007]	http://logging.apache.org/log4j/docs/, 02.05.2007
[Madgeek 2007]	http://madgeek.com/Articles/ORMapping/EN/mapping.htm, 17.05.2007
[Matthiessen und Unterstein 2003]	MATTHIESSEN Günter, Michael UNTERSTEIN (2003): Relationale Datenbanken und SQL. München: Addison-Wesley Verlag, 384 S.
[Meier 2003]	MEIER Andreas (2003): Relationale Datenbanken. Leitfaden für die Praxis. Berlin Heidelberg: Springer-Verlag, 239 S.
[Meyers 2007]	http://lexikon.meyers.de, 05.06.2007
[Middlegen Homepage 2007]	http://boss.bekk.no/boss/middlegen/, 17.05.2007
[MyEclipse 2007]	http://www.genuitec.com, 14.05.2007
[Oracle TopLink 2007]	http://www.oracle.com/technology/products/ias/toplink, 03.05.2005
[Pilone 2004]	PILONE Dan (2004): UML kurz & gut. Köln: O'Reilly Verlag GmbH & Co. KG, 92 S.
[Poeschek 2007]	www.poeschek.at/files/publications/objektorientierte_datenbanken.pdf, 04.05.2007
[Propel Homepage 2007]	http://propel.phpdb.org, 08.05.2007
[Rechenberg und Pomberger 2002]	RECHENBERG Peter, Gustav POMBERGER (2002): Informatik-Handbuch 3. Aktualisierte und erweiterte Auflage. München: Carl Hanser Verlag, 1189 S.
[Saake und Sattler 2003]	SAAKE Gunter, Kai-Uwe SATTLER (2003):Datenbanken & Java. Heidelberg: dpunkt.verlag GmbH, 352 S.
[Schafer 2007]	http://www.sigs.de/publications/os/2003/03/schafer_OS_03_03.pdf, 01.06.2007

[Seemann 2003] SEEMANN Michael (2003): Native XML-Datenbanken im Praxiseinsatz Frankfurt: Software & Support Verlag GmbH, 313 S.

[Seifert 2007] http://www.mit-h.de/01_IntelligenterKnoten/03_ProzessLeitstellenKnoten/3.1_FeldKnoten/3.1.1_EIB_Bus/EIB_Seifert/gesamt-3.7.9.html, 01.05.2007

[SimpleORM Homepage 2007] http://www.simpleorm.org/, 13.05.2007

[Software-Kompetenz 2007] http://www.software-kompetenz.de/?28860, 01.06.2007

[Sun EJB 2007] http://java.sun.com/products/*EJB*/, 12.05.2007

[Sun 2007] http://java.sun.com, 12.05.2007

[Torque 2007] http://db.apache.org/torque/, 30.06.2007

[Ullenboom 2007] http://www.galileocomputing.de/openbook/javainsel6/javainsel_11_009.htm#mj5678b74e921de4d08c4d57c46963bfa0, 12.02.2007

[Uni Karlsruhe 2007] www.aifb.uni-karlsruhe.de/Lehrangebot/Winter2000-01/Iwm/folien-001122.pdf, 07.05.2007

[WebPark 2004] http://www.webparkservices.info/Assets/WebPark_GIUZ_form.pdf, 29.06.2007

[Wikipedia 2007] http://de.wikipedia.org, 07.05.2007

[Wille 2007] http://209.85.135.104/search?q=cache:AR23VAFflKYJ:www.stefan-wille.com/lehmanns/Hibernate%2520-%2520Stefan%2520Wille.pdf+ORM+warum&hl=de&ct=clnk&cd=6&gl=at, 07.05.2007

Glossar

.NET	.NET ist eine Plattform für Programme, die mit unterstützenden Programmiersprachen entwickelt wurden. .NET ist eine Technologie, die verschiedene Betriebssystemfunktionen zusammenfasst und diese an einem zentralen Punkt anbietet. Es soll veraltete Technologien und Vorgehensweisen der Programmierer, wie z.B. COM oder API-Aufrufe im Programmcode, ersetzen. [Hansen 1998]
Abstraktion	Ab|s|trak|ti|on die; -, -en: 1.a) Begriffsbildung; b) Verallgemeinerung; c) Begriff. 2. (Stilk.) auf zufällige Einzelheiten verzichtende, begrifflich zusammengefasste Darstellung. [Duden 2007]
Ant	Ant ist ein Produkt der Apache Software Foundation und stellt ein Build-Tool hauptsächlich für die automatisierte Java-Entwicklung bereit. [Cayenne Homepage 2007]
API	API (application programming interface) ist eine Bibliothek, um Systemabhängigkeiten durch direkten Aufruf von Betriebssystemfunktionen zu vermeiden. [Rechenberg und Pomberger 2002]
Apple™	Apple ist ein Unternehmen mit Sitz in Cupertino, Kalifornien (USA), welches Computer und Unterhaltungselektronik sowie Betriebssysteme und Anwendungssoftware herstellt. [Wikipedia 2007]
ASM	ASM ist ein Framework zur Manipulation von Java Byte-Code. [ASM 2007]
Bean Managed Persistence	Bedeutet, dass bei EJB die Persistenz vom Bean-Entwickler selbst programmiert wird. [Sun EJB 2007]
Biodiversität	Biodiversität, biologische Vielfalt; Eigenschaft biologischer Systeme, voneinander verschieden zu sein. Sie umfasst die genet. Variabilität innerhalb einer Art, die Mannigfaltigkeit der Arten und die Vielfalt von Ökosystemen. Biodiversität gestattet den Arten und Lebensgemeinschaften, sich wandelnden abiotischen (Luft, Wasser, Boden) und biotischen (Mikroorganismen, Flora, Fauna) Umweltbedingungen anzupassen und damit ihr Fortbestehen zu sichern. [Meyers 2007]

Byte-Code	Ein Java-Kompiler erzeugt aus Java-Quelltext den Java Byte-Code, der in einer Objektdatei abgelegt wird. Die Byte-Code-Instruktionen stellen die Maschinensprache der virtuellen Java-Maschine dar. [Rechenberg und Pomberger 2002]
Cache	Cache (Pufferspeicherverwaltung) ist in der Zentraleinheit eines Prozessors ein sehr schneller RAM-Speicher, der auf die Prozessorgeschwindigkeit abgestimmt ist. [Hansen 1998]
Caching	Vorgang, bei dem Daten für einen schnelleren Zugriff auf ein schnelleres Medium zwischengespeichert werden. [Rechenberg und Pomberger 2002]
Compiler	EDV-Programm zur Übersetzung einer Programmiersprache in die Maschinensprache eines Computers. [Duden 2007]
CRUD-Operationen	Kurzform für alle create, read, update und delete SQL-Operationen. [Hibernate Homepage 2007]
DAO	DAO (Data Access Object, deutsch „Datenbankzugriffsobjekt“). Bei den Datenbankzugriffsobjekten handelt es sich um eine Programmierschnittstelle, mit der Programmierer auf Datenbanken zugreifen können. Über diese API werden die Objekte für die Datenbasis erfasst. In Verbindung mit einer Datenbankmaschine können diese Daten dann dargestellt und manipuliert werden. [ITWissen 2007]
Datenbank-Frontend	Wird in Verbindung mit einer Schichteneinteilung verwendet, wobei das Front-End näher am Benutzer ist. [Heinrich et al. 2004]
DDL	DDL (data definition language) ist eine Datendefinitionssprache und ist der sprachliche Ausdruck eines Datenmodells. [Rechenberg und Pomberger 2002]
deklarative Programmierung	Die deklarative Programmierung ist ein Programmierparadigma, welches auf mathematischer rechnerunabhängiger Theorie beruht. [Heinrich et al. 2004]

Deployment Descriptor	Der Deployment Descriptor enthält im XML-Format Informationen für die Generierung und Ausführung von EJB-Komponenten zur Lauf- und Kompilierungszeit. Hierbei handelt es sich um deklarative Programmierung, da der Entwickler Features/Funktionalitäten der EJB-Komponenten durch "einfaches" Angeben bestimmter Schlüsselwörter innerhalb eines Deployment Descriptors erreichen kann, ohne programmieren zu müssen. [Oracle TopLink 2007]
Deskriptor	Ein Deskriptor enthält im XML-Format Informationen für die Generierung und Ausführung von Komponenten zur Lauf- und Kompilierungszeit. Hierbei handelt es sich um deklarative Programmierung, da der Entwickler Features/Funktionalitäten der Komponenten durch "einfaches" Angeben bestimmter Schlüsselwörter innerhalb eines Deskriptors erreichen kann, ohne programmieren zu müssen. [Oracle TopLink 2007]
EIS	Enterprise Information Systems (EIS) sind Unternehmensinformationssysteme. [Hansen 1998]
EJB	Enterprise JavaBeans (EJB) sind Komponenten mit einem Standard innerhalb eines J2EE-Servers. [Sun EJB 2007]
empirisch	Erhebungsmethode; die auf möglichst exaktem Messen beruhende Ermittlung von Daten mit dem Ziel den Wahrheitsgehalt von Aussagen an der Wirklichkeit zu überprüfen. [Heinrich et al. 2004]
Exporter	Hibernate bietet die Klasse org.hibernate.tool.ant.HibernateTool-Task an. Diese Klasse stellt eine Implementierung nach der Ant-Richtlinie zum Erstellen von eigenen Tasks dar. Hibernate spricht von Exportern, die innerhalb der Ant-Tasks die Aufgaben durchführen. [Hien und Kehle 2007]
Features	Funktionen [Duden 2007]
First-Level-Cache	First-Level-Cache ist ein besonders schneller Pufferspeicher im Prozesser. [Rechenberg und Pomberger 2002]
Getypte Programmiersprache	Ist eine Programmiersprache, bei der die Zuteilung des Typs einer Variablen zur Laufzeit eines Programmes erfolgt. [Bergmann 2005]

GIS	geographisches Informationssystem [Duden 2007]
Granularität	Anzahl von Untergliederungen eines Elements. [Duden 2007]
GTEC	Individuelles Diplomstudium am Universitäts Zentrum Rottenmann
IBM™	Die International Business Machines Corporation (IBM) ist eines der ältesten US-amerikanischen IT-Unternehmen mit Firmensitz in Armonk im US-Bundesstaat New York. [Wikipedia 2007]
IEEE-STD-830-1998	IEEE Recommended Practice for Software Requirements Specifications beschreibt die Qualitätskriterien für das Pflichtenheft. [Heinrich et al. 2004]
Implementierung	Implementieren ist abgeleitet von dem lateinischen Wort »implere«, was für erfüllen und ergänzen steht. Es erfüllt dem Sinn nach vollständig das, was in Computernetzen, Systemen oder Programmen damit gemeint ist, nämlich das Einfügen eines neu entwickelten Systems oder Softwareteils in ein bestehendes, das dadurch ergänzt wird. Mit einbezogen sind auch alle zusätzlichen Arbeitsgänge, die das ordnungsgemäße Funktionieren des zusätzlich Eingebrachten mit dem bisher Vorhandenen garantieren. [Lexikon 2007]
Interaktion	Aufeinander bezogenes Handeln zweier od. mehrerer Personen; Wechselbeziehung zwischen Handlungspartnern. [Duden 2007]
Interface	Interface [engl. für: Schnittstelle] Die Übergangsstelle zwischen zwei Ebenen, zwischen denen ein Datenaustausch stattfinden soll, ist das Interface, das sich auf Hardware (Stecker, Leitungen, Rechner) oder Software (Programme, Spiele) beziehen kann. Als Schnittstelle zwischen Mensch und Computer ist auch die Tastatur oder die Darstellung eines Programms auf dem Monitor gemeint. [Duden 2007]
Introspektion	Introspektion bedeutet, dass ein Programm Erkenntnisse über seine eigene Struktur gewinnen kann. [Heinrich et al. 2004]
Java EE (J2EE),	Die Java 2 Enterprise Edition (J2EE), eine Erweiterung der Java 2 Standard Edition (J2SE), ist eine Java-Plattform für Unternehmensanwendungen. Der J2EE-Standard arbeitet mit Basiskomponenten, mit denen die Anwendungen realisiert werden. Ist eine Anwendung J2EE-konform, kann sie auf andere J2EE-Application-Server portiert werden. [ITWissen 2007]

Java-VM	Technisch gesehen steht im Zentrum der Java-Technologie ein plattformunabhängiges Format für Objektdateien. Dieses Format wird bei der Übertragung von Programmen von einem Server zu einem Client verwendet und spezifiziert eine virtuelle Maschine, die zur Ausführung solcher Programme geeignet ist. [Flanagan 2003]
JBoss Enterprise Middleware Suite	JBoss Enterprise Middleware System (JEMS) ist eine hochgradig erweiterbare und skalierbare Produktsuite als Grundlage für die Entwicklung und Bereitstellung von E-Business-Applikationen. [JBoss 2007]
JDBC	Java Database Connectivity (JDBC) ist eine Datenbankschnittstelle der Java-Plattform, die eine einheitliche Schnittstelle zu Datenbanken verschiedener Hersteller bietet und speziell auf relationale Datenbanken ausgerichtet ist. [Flanagan 2003]
Join	Ein Join (zu deutsch Verbund) ist eine Operation der relationalen Algebra der Datenbanksprache SQL und bezeichnet die beiden hintereinander ausgeführten Operationen kartesisches Produkt und Selektion. [Rechenberg und Pomberger 2002]
JTA	Java Transaction API [Sun 2007]
Know-how	Das Wissen, wie eine Sache praktisch verwirklicht werden kann. [Duden 2007]
Legacy-System	Der Begriff Legacy-System (Altsystem) bezeichnet in der Wirtschaftsinformatik eine etablierte, historisch gewachsene Anwendung im Bereich Unternehmenssoftware. [Heinrich et al. 2004]
Link	Verweis auf ein Web-Dokument [Hansen 1998]
Linux	(EDV): frei verfügbare Variante des Betriebssystems UNIX [Duden 2007]
Location Based Services (LBS)	Standortbezogene Dienste (engl. Location Based Services (LBS)), sind über ein Telekommunikationsnetz erbrachte mobile Dienste, die unter Zuhilfenahme von positions-, zeit- und personenabhängigen Daten dem Endbenutzer selektive Informationen bereitstellen oder Dienste anderer Art erbringen. [Fischer 2003]

Migration	Der koordinierte Übergang von einer Ausgangsplattform auf eine Zielplattform bei Weiterverwendung einzelner Komponenten der Informationsinfrastruktur. [Heinrich et al. 2004]
MS SQL Server 2005 Express Edition	Seit der Einführung von SQL Server 2005 heißt die kostenlose Variante des SQL-Servers nicht mehr MSDE, sondern, angelehnt an die kostenlosen Varianten von Visual Studio, SQL Server 2005 Express Edition. [Wikipedia 2007]
MyEclipse	MyEclipse Enterprise Workbench ist eine kommerzielle in Eclipse integrierte Entwicklungsumgebung für Java und wird von der Firma Genuitec™ vertrieben. [MyEclipse 2007]
Objective-C	Objective-C ist eine Erweiterung der Programmiersprache C, um Sprachmittel zur objektorientierten Programmierung.
ODMG	Für objektorientierte Datenbanksysteme hat das Firmenkonsortium ODMG (object data management group) einen Industrie-Standard vorgelegt, der Object Query Language als eine Abfragesprache enthält, ebenso ODL (object definition language). [Rechenberg und Pomberger 2002]
OODBMS	Die seit Ende der achtziger Jahre angebotenen objektorientierten Datenbanksysteme fußen auf einem objektorientierten Datenmodell, welches die bekannten Konzepte der Objektorientierung mehr oder minder enthält. [Rechenberg und Pomberger 2002]
Open Source	engl. open-source = frei verfügbar (EDV) [Duden 2007]
ORM	ORM bietet Programmierern eine objektorientierte Sicht auf Tabellen und Beziehungen in relationalen Datenbankmanagementsystemen (RDBMS). Statt mit SQL-Statements wird mit Objekten operiert. [Horn 2007]
Parser	Parser werden für die Analyse von XML-Dokumenten verwendet und stellen die darin enthaltenen Informationen für die weitere Verarbeitung zur Verfügung. [Seemann 2003]
PDA	Ein persönlicher digitaler Assistent (engl.: personal digital assistant; abgekürzt: PDA) ist ein batteriebetriebener „Jackentaschen-PC“. [Hansen 1998]

Performance	Die Performance (Leistung) bezeichnet in der Informatik den Ressourcenverbrauch und die Qualität der Ausgabe von Programmen und Hardware. Meistens ist mit Performance die Datenrate gemeint, also die Menge von Daten, die innerhalb einer bestimmten Zeitspanne verarbeitet werden kann. [Rechenberg und Pomberger 2002]
persistente Objekte	Objekte, die auch vor Beginn oder nach dem Ende der Ausführung eines Programmlaufs existieren, z.B. weil sie zu einer Datenbank gehören. [Rechenberg und Pomberger 2002]
pgAdmin	Programm zur Administration einer PostgreSQL-Datenbank. [Eisentraut 2003]
PHP	PHP (Hypertext Preprocessor) ist eine Skriptsprache mit einer an C bzw. C++ angelehnten Syntax, die hauptsächlich zur Erstellung von dynamischen Web-Seiten oder Web-Snwendungen verwendet wird. [Bergmann 2005]
POJO	POJO steht für Plain Old Java Object und bedeutet, dass es sich um ein normales Objekt in der Programmiersprache Java handelt. [Hien und Kehle 2007]
Polymorphie	Eigenschaft von Java und bedeutet, dass Variablen verschiedenartige Objekte bezeichnen können. Allgemein geschriebene Programmteile sind damit auf verschiedene Arten von Objekten anwendbar. [Rechenberg und Pomberger 2002]
Portabilität	Portabilität bedeutet die Gewährleistung der Lauffähgikeit von Anwendungssoftware auf unterschiedlichen Systemen. [Rechenberg und Pomberger 2002]
PostgreSQL	PostgreSQL ist ein objektrelationales Datenbankverwaltungssystem auf Basis von POSTGRES, Version 4.2, welches in der Informatikfakultät der Universität von Kalifornien in Berkley entwickelt wurde. PostgreSQL ist ein Open Source-Nachfolger dieses Codes von Berkley. [Eisentraut 2003]
Prinzip der Kapselung (Geheimnisprinzip)	Ein Prinzip, nach dem die Zerlegung eines Systems in Module so erfolgt, dass deren interne Konstruktion, einschließlich der verwendeten Datenstruktur der Umgebung verborgen bleibt. [Heinrich et al. 2004]

Prototyping	Konstruktionsmethodik; bedeutet einen Ansatz zur Konstruktion von Informationssystem oder eines Teils davon mit ausgeprägter Benutzerbeteiligung unter Verwendung spezieller Werkzeuge. [Heinrich et al. 2004]
RDB	relationale Datenbanken [Meier 2003]
RDBMS	Relationales Datenbank Management System [Matthiessen und Unterstein 2003]
Replikation	Abgleich von Daten zwischen verschiedenen Datenbeständen. [Hansen 1998]
Softwaremetriken	Die Eigenschaft eines Informationssystems, dessen Ausprägung (Messwert) mit einer geeigneten Messmethode ermittelt werden kann. [Heinrich et al. 2004]
Spezifikation	Darstellungsmethode, um die Anforderungen in ein formales Modell zu bringen bzw. das Ergebnis dieses Vorgangs als Dokument. [Heinrich et al. 2004]
SQL	SQL ist eine genormte Datenbanksprache (structured query language) zur Definition, Abfrage und Manipulation von Daten für relationale Datenbank Management Systeme. [Rechenberg und Pomberger 2002]
Sun™	Sun Microsystems (Kurzform: Sun) ist ein in Silicon Valley ansässiger Hersteller von Computern und Software. [Sun 2007]
Thread	Ein Thread (auch Aktivitätsträger), in der deutschen Literatur vereinzelt auch als Faden bezeichnet, ist in der Informatik ein Ausführungsstrang beziehungsweise eine Ausführungsreihenfolge der Abarbeitung der Software. Folge von Nachrichten zu einem Thema in einer Newsgroup (EDV). [Duden 2007]
thread-safe	Threadsicherheit (engl. thread safety) ist eine Eigenschaft von Softwarekomponenten und hat eine wichtige Bedeutung in der Softwareentwicklung. Sie besagt, dass eine Komponente gleichzeitig von verschiedenen Programmbereichen mehrfach ausgeführt werden kann, ohne dass diese sich gegenseitig behindern. [Wikipedia 2007] [Rechenberg und Pomberger 2002]
Toolset	Sammlung von Anwendungen. [Duden 2007]

transparente Persistenz	(Persistence [engl. für: Persistenz]) Das erkennbare und durchschaubare Bestehenbleiben eines Zustands über längere Zeit. [Duden 2007]
UML	Die Unified Modeling Language (UML) ist ein objektorientierter Modellierungsansatz. [Booch et al. 1999]
UMN MapServer	Die UMN MapServer Software ist eine *Open Source* Entwicklungsumgebung für die Erstellung von Internet Anwendungen mit dynamischen Karteninhalten. [Fischer 2003]
Validierung	Qualitätsmanagement; die Überprüfung von Entwurfs- und Entwicklungsergebnissen für Produkte und Dienstleistungen auf externe Korrektheit mittels systematischer Vorgehensweise, zumeist experimentellen Verfahren. [Heinrich et al. 2004]
W3C	World Wide Web Consortium (kurz: W3C) ist das Gremium zur Standardisierung der das World Wide Web betreffenden Techniken. [Hansen 1998]
Web-Browser	Ein Internet-Programm zum Auffinden von Informationen insbesondere im WWW. [Heinrich et al. 2004]
XML	Extensible Markup Language - zu deutsch: erweiterbare Auszeichnungssprache. [Seemann 2003]

Abbildungsverzeichnis

Tabellenverzeichnis

Codefragmentverzeichnis