HOCHSCHULE
MITTWEIDA
UNIVERSITY OF
APPLIED SCIENCES

BACHELORARBEIT

Herr
Daniel Storm

Analyse und Einsatzmöglichkeiten des ORM-Frameworks Hibernate im DELECO® -Umfeld und prototypische Umsetzung eines Anwendungsszenarios.

2010

Fakultät **Mathematik/Naturwissenschaften/Informatik**

BACHELORARBEIT

Analyse und Einsatzmöglichkeiten des ORM-Frameworks Hibernate im DELECO® -Umfeld und prototypische Umsetzung eines Anwendungsszenarios.

Autor:
Daniel Storm

Studiengang:
Informatik

Seminargruppe:
IF07w1-B

Erstprüfer:
Prof. Dr. Rudolf Stübner

Zweitprüfer:
Dipl.-Wirtschaftsinf. (BA) Frank Otto

Mittweida, 09 2010

Bibliografische Angaben

Storm, Daniel: Analyse und Einsatzmöglichkeiten des ORM-Frameworks Hibernate im DELECO® -Umfeld und prototypische Umsetzung eines Anwendungsszenarios., 51 Seiten, 16 Abbildungen, Hochschule Mittweida (FH), Fakultät Mathematik/Naturwissenschaften/Informatik

Bachelorarbeit, 2010

Daniel Storm

Referat

In dieser Bachelor-Arbeit geht es um Objekt relationales Mapping durch das Open-Source ORM-Framework Hibernate. Es werden die Vor- und Nachteile der theoretischen Implementation aufgezeigt und diese an einem prototypischen Beispiel näher betrachtet und verifiziert. Ziel ist es, eine datenbankunabhängige Lösung für den Einsatz im DELECO®-Umfeld zu finden.

I. Inhaltsverzeichnis

II. Abbildungs- und Tabellenverzeichnis

Abbildungen

Tabellen

III. Abkürzungsverzeichnis

4GL ..... Fourth Generation Language, Seite 6

API ..... Application Programming Interface, Seite 9

ASP.NET ..... Active Server Pages .NET, Seite 28

CAD ..... Computer Aided Design, Seite 4

CLR ..... Common Language Runtime, Seite 7

CORBA ..... Common Object Request Broker Architecture, Seite 15

CRUD ..... Create, Read, Update, Delete, Seite 9

EDV ..... Elektronische Datenverarbeitung, Seite 1

EJB ..... Enterprise Java Bean, Seite 13

ERP ..... Enterprise Ressource Planing, Seite 2

HQL ..... Hibernate Query Language, Seite 24

JDBC ..... Java Database Connection, Seite 13, 14

JIT ..... Just In Time, Seite 6

JNDI ..... Java Naming and Directory API, Seite 13

JTA ..... Java Transaction API, Seite 13

JVM ..... Java Virtual Machine, Seite 6

ORM ..... Objektrelationales Mapping, Seite 2

POCO ..... Plain Old Common Language Runtime Objects, Seite 28

POJO ..... Plain Old Java Objects, Seite 14

SQL ..... Structured Query Language, Seite 10

Listings

1 Einleitung

1.1 Vorwort

Relationale Datenbanksysteme sind seit Jahrzehnten ein fester Bestandteil der EDV (Elektronische Datenverarbeitung) -Branche und nicht mehr wegzudenken. Sie beruhen auf dem relationale Datenbankmodell, dass von Edgar F. Codd 1970 vorgeschlagen wurde. Dieses Modell baut auf dem Konzept der mathematische Beschreibung einer Tabelle auf. Hierbei werden Daten in zweidimensionalen Tabellen abgebildet und mittels eindeutigem Primärschlüssel identifiziert.

Durch diesen Erfolg sind viele Firmen auf den Zug der relationalen Datenbanken aufgesprungen. Die Folgen war, dass diese Datenbanksysteme mit unterschiedlichen Aufbauten und Eigenschaften wie z.B. unterschiedliche Datentypen, unterschiedliche Abfrage-Syntax, usw. ausgestattet sind.

Im Bereich der Programmiersprachen ist durch den Erfolg der objektorientierten Sprachen das objektorientierte Modell in den Vordergrund gerückt. Dieses zeichnet sich aus in dem man Daten und dazugehörige Funktionen in Objekte zusammenfasst. Um Objekte dauerhaft zu speichern, entwickelten sich objektorientierte Datenbanken. Diese besitzen die Fähigkeit Objekte in der Datenbank zu speichern und zu laden. Durch die im Vergleich zu relationalen Datenbanken hohen Kosten in der Anschaffung sind aber nur wenig verbreitet. Daher sind heutzutage relationale Datenbanken das Maß der Dinge in Sachen Kosten, Verbreitung und Verfügbarkeit.

Der Unterschied der tabellarischen Repräsentation der Daten und der Darstellung von Objekten in objektorientierten Programmiersprachen führte zu der objekt-relationalen Unvereinbarkeit. Um diese Problem zu lösen wurde objekt-relationales Mapping eingeführt um die bestehenden relationalen Datenbanken und die neuen objektorientierten Programmiersprachen zu verbinden.

1.2 Motivation und Aufgabenstellung

Mit DELECO® hat das DELTA BARTH Systemhaus GmbH eine ERP(Enterprise Resource Planing) -Anwendung für den Mittelstand entwickelt. Diese Software kann mit unterschiedlichen Datenbanken zusammenarbeiten und daher soll das ORM (Objektrelationales Mapping) -Tool *Hibernate* analysiert und Möglichkeiten des Einsatzes im DELECO® -Umfeld aufgezeigt werden.

Dieses Tool arbeitet als Schicht zwischen Anwendung und Datenbank. Somit abstrahiert sie die Arbeit an der Datenbank und erlaubt eine datenbankunabhängige Entwicklung. Außerdem ermöglicht Hibernate durch verschiedene Caching- und Locking-Strategien die Performance zu verbessern und die Auslastung der Datenbank zu verringern.

Nach der Analyse wird eine prototypische Umsetzung, einer Anwendung aus dem DELECO® -Umfeld, entwickelt und die theoretischen Ansätze in der Praxis geprüft.

2 DELECO®

2.1 Allgemein

Mit dem DELECO®ERP hat das DELTA BARTH Systemhaus ein Produkt entwickelt, dass Strukturen und Abläufe eines mittelständigen Unternehmens in einer Softwarelösung bündelt. Um ein möglichst großes Spektrum an Kunden zu erreichen ist die Software modular aufgebaut, dass heißt das für die jeweilige Unternehmensart eine individuelle Lösung zusammengestellt werden kann.

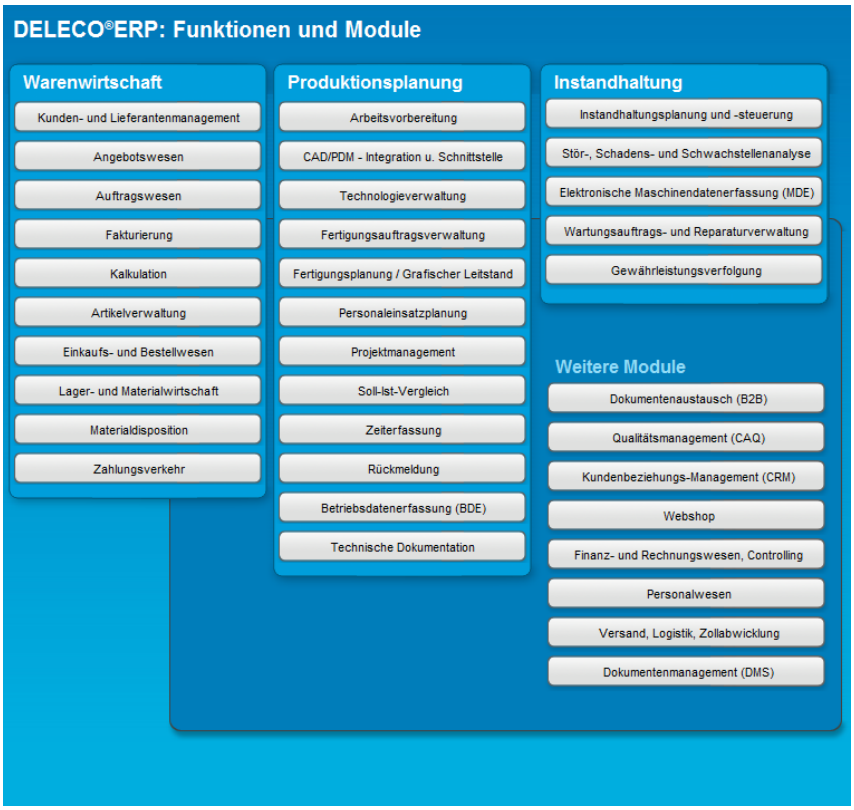

Abbildung 2.1: Übersicht der Funktionen und Module des DELECO® ERP laut [DELECO]

2.2 Module

2.2.1 Leitstand

Der DELECO®Leitstand ist ein Modul zur Übersicht der Produktionsprozesse. Durch interaktive Visualisierung können Terminplanung, Auftragsdurchlauf und Ressourcenbelegung verdeutlicht werden. Hinzu kommt die Möglichkeit Fertigungsaufträge, Res-

sourcen und Personal zu planen. Dies wird mittels hinterlegter Qualifikation, Schicht- und Urlaubspläne ermöglicht. Der Leitstand besteht aus einem Server und dem Client. Beide sind komplett in C# geschrieben.

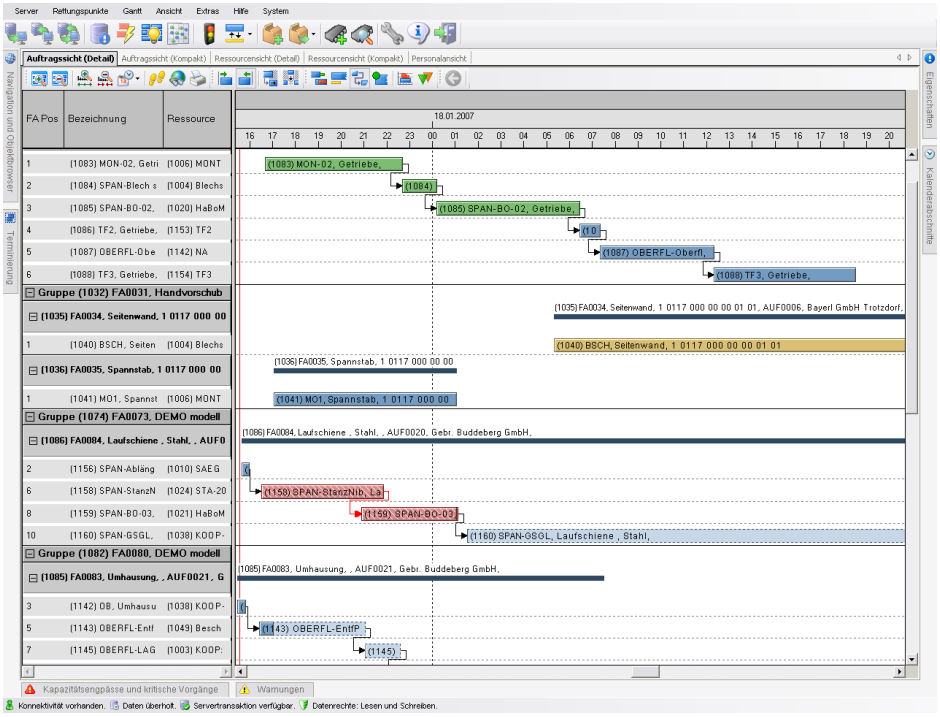

Abbildung 2.2: Übersicht der Funktionen und Module des DELECO® Leitstand

2.2.2 CAD-Addin

Das Cad(Computer Aided Design)-Addin arbeitet mit SolidWorks®zusammen und ist ebenfalls durch die Programmiersprache C# entwickelt. Das Addin erlaubt Teile-/Baugruppen und Stücklisten an DELECO®zu übergeben und andersherum die Anzeige von Stammdaten und Artikelnummern in SolidWorks®.

2.2.3 Cockpit

Das Cockpit ist eine auf der Portalsoftware *Jetspeed* basierende Anwendung die mit Java realisiert wurde. Hiermit ist es möglich, auf einer im Browser oder im DELECO® angezeigten Webseite, Portlets anzuzeigen. Unter den Portlets befinden sich z.B.: Kundenumsatz, Geschäftsentwicklung oder auch Ressourcenauslastung. Durch das Hinzufügen und Entfernen kann sich jeder Nutzer individuelle Informationsseiten zusammenstellen und anzeigen lassen. Bei den Portlets wird nur aus der Datenbank gelesen und nicht geschrieben. Somit kann durch Caching eine hohe Ersparnis an Zeit und Ressourcen erlangt werden.

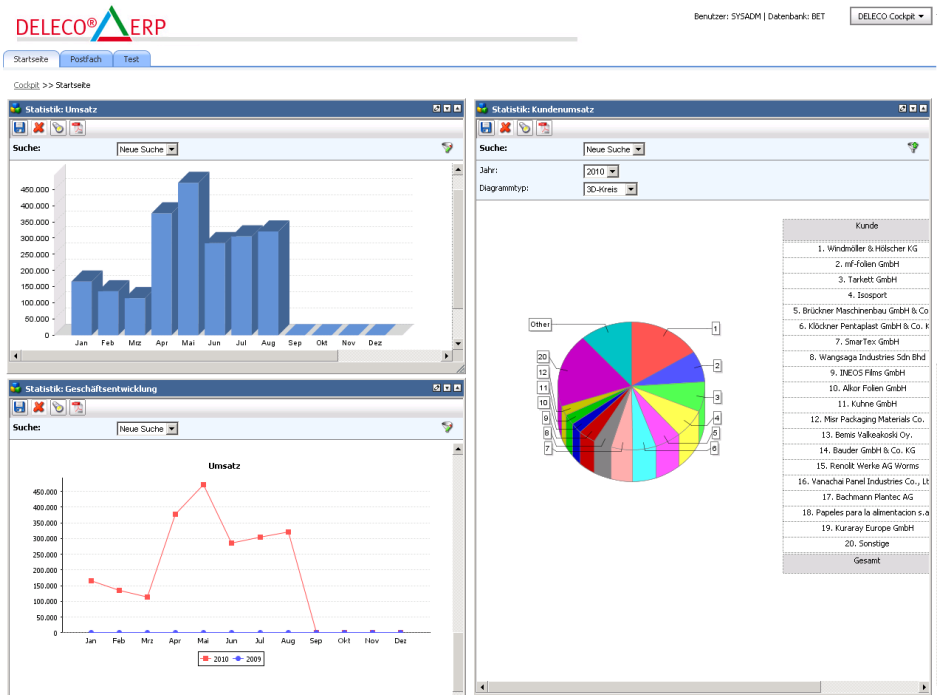

Abbildung 2.3: DELECO® Cockpit

2.2.4 Ereignismanagment

Das Ereignismanagement ist ein, durch selbst erstellte Ereignisse, gesteuertes System, dass im Webbrowser oder direkt im DELECO® ausgeführt werden kann. Mit dieser Anwendung können Ereignisse definiert werden, bei denen Daten zu unterschiedlichen Bedingungen von der Datenbank abgefragt werden. Dadurch können spezialisierte Abfragen einfach für den Anwender erstellt werden. Die Chance auf mehrere gleichartige Abfragen ist relativ groß, daher kann man mittels aktiviertem Cache einen Performance-Gewinn erwarten. Außerdem würde die Datenbankunabhängigkeit ebenfalls die Implementation von *Hibernate* rechtfertigen.

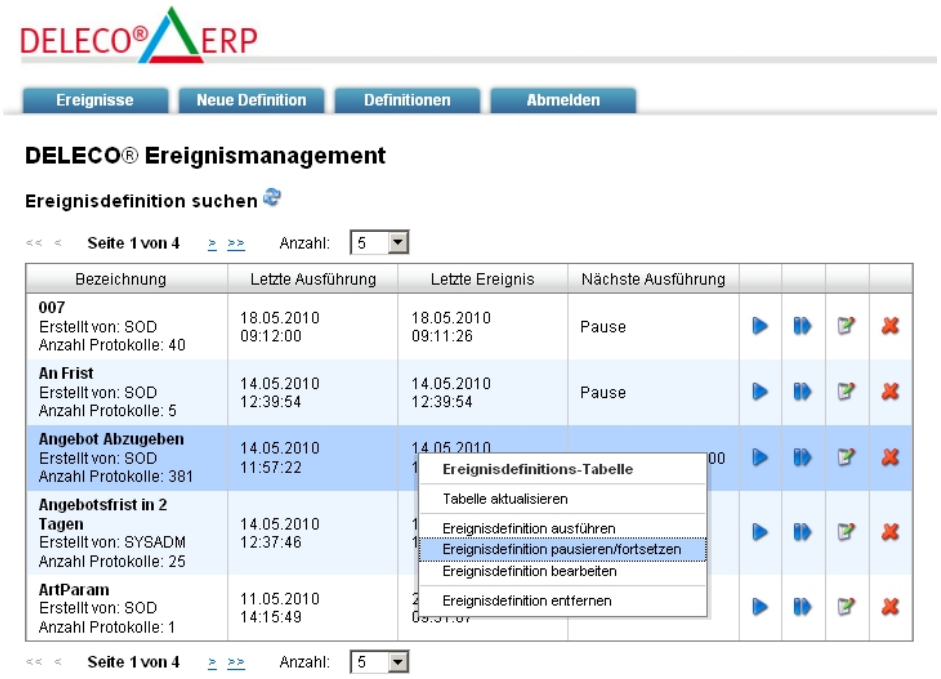

Abbildung 2.4: Benutzeroberfläche des DELECO® Ereignismanagement

2.3 Programmiersprachen

Der größte Teil von DELECO® ist in *SQLWindows*, einer 4GL(Fourth Generation Language) -Programmiersprache, entwickelt. Sie erlaubt es per Graphischer Programmierung Windows-Applikation zu erstellen. Für die Entwicklung wird das Entwicklungswerkzeug *Team Developer* der Firma Unify benötigt.

Weitere Module wie z.B. die Webanwendungen DELECO® Cockpit, DELECO® Ereignismanagement und weitere werden in Java entwickelt.

Java ist eine Programmiersprache die Mitte der Neunziger auf den Markt kam und unter anderem von James Gosling im Auftrag von *Sun Microsystems* entwickelt wurde. Ziel war es eine Programmiersprache auf Basis von C++ zu entwickeln die das objektorientierte Paradigma umsetzt. Eine Haupteigenschaft von *Java* ist die Plattformunabhängigkeit die dadurch erreicht wird, dass der Programmcode in sogenannten *Byte-Code* übersetzt wird. Dieser wird dann in einer virtuellen Maschine, der *JVM*(Java Virtual Machine), ausgeführt. Um den Geschwindigkeitsnachteil, der Interpretation des Bytecodes in der JVM, abzumildern wurde ein JIT(Just in Time) -Compiler eingeführt. Dieser übersetzt zur Laufzeit den virtuellen Code in Maschinencode der jeweiligen Plattform. Ein weiterer Vorteil der JVM ist das die Programme in einem abgetrennten Bereich laufen. Außerdem bietet *Java* weitere Eigenschaften wie die Speicherverwaltung, die durch *Garbage Collection* selbst die Entfernung nicht mehr benötigter Objekte aus dem Speicher vornimmt oder die *Ausnahmebehandlung*, die es ermöglicht Laufzeitfehler ab-

zufangen und darauf zu reagieren. [1]

Für den DELECO® Leitstand und das CAD-Addin wird die Programmiersprache C# verwendet. Diese Sprache ist 2002 für die ,ebenfalls 2002 veröffentlichte, Entwicklungsplattform *.NET* von *Mircrosoft* entwickelt wurden. Sie bietet ähnlich wie *Java* Objektorientierung, Plattformunabhängigkeit durch die CLR(Commons Language Runtime), automatische Speicherverwaltung durch *Garbage Collection* und Außnahmebahndlung. [2]

2.4 Unterstützte Datenbanken

DELECO® arbeitet mit einer Vielzahl von Datenbanken zusammen, darunter *Microsoft SQL Server*, *IBM Informix*, *PostgreSQL*, *IBM DB2*, *Oracle* und *Unify SQLBase*.

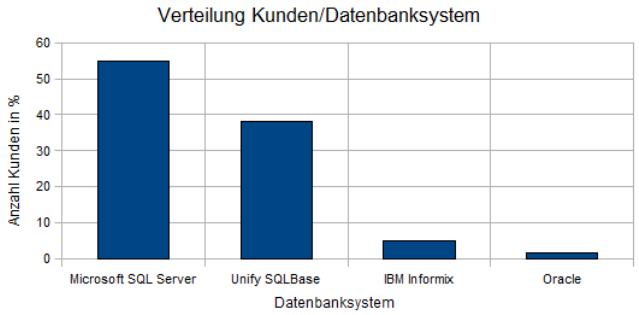

Abbildung 2.5: Verteilung Kunden/Datenbanksystem

Die aktuell am meisten genutzte Datenbank, ist die *Microsoft SQL Server*. Diese wird an Kunden, ohne bestehendes Datenbanksystem, mit DELECO® ausgeliefert. Anfänglich wurde *SQLBase* mitgeliefert, aber durch gesunkene Kosten und bessere Handhabung wurde später auf das System von *Microsoft* gewechselt. Die Systeme *Informix* und *Oracle* werden unterstützt, da sie als bestehende Datenbanklösung, bei verschiedene Kunden, vorhanden waren. Außerdem wird *Informix* als Datenbank bei Kunden mit *Linux*-Betriebssystem eingesetzt. *PostgreSQL* wird nur für externe Module wie den *Webshop* oder den *B2B(Business to Business)*-Dokumentenaustausch verwendet.

[1] Vergleiche [Ullenboom09]
[2] Vergleiche [Kühnel08]

3 Theoretische Grundlagen

3.1 Allgemein

In diesem Kapitel wird auf die Grundlagen der Persistenz und besonders dessen Technologie des objektrelationalen Mappings eingegangen. Es werden die Bestandteile, die Formen und die Unverträglichkeit des Objekt-Relationalen Mappings aufgezeigt.

3.2 Persistenz

Unter Persistenz versteht man das Speichern von Daten auf Speichermedien, wie Festplatten etc., die Daten auch nach Beendigung des Programmes behalten. Dadurch können diese zu einem späteren Zeitpunkt wiederverwendet werden. Im heutigen Programmier-Umfeld werden dafür hauptsächlich die folgenden zwei Technologien eingesetzt. [3]

- **Serialisierung**
  Bei der Serialisierung wird der komplette Zustand eines Objektes in einen Byte-Strom geschrieben, der dann in eine Datei oder einer Datenbank persistiert werden kann. Nachteil ist, dass man nur als Ganzes auf den Byte-Strom zugreifen kann. Das heißt, man muss immer den ganzen Byte-Strom deserialisieren um an die Daten zu gelangen. Damit ist die Serialisierung ungeeignet für Suchläufe oder die Gruppierung von großen Datensätzen.
- **Objekt-Relationales Mapping**
  Unter ORM versteht man das Speichern von Objekten in Tabellen von relationalen Datenbanken. Hierbei wird das Mapping zwischen Objekten und der Datenbank mit Metadaten beschrieben.

3.3 ORM - Bestandteile

Eine ORM-Lösung besteht laut [King07] aus folgenden vier Bestandteilen:

- Eine API(Application Programming Interface) zur Durchführung einfacher CRUD (Create, Read, Update, Delete) -Operationen mit Objekten persistenter Klassen
- Eine Sprache oder API(Application Programming Interface) für die Formulierung von Abfragen, die sich auf Klassen und Klasseneigenschaften beziehen
- Eine Einrichtung für die Spezifizierung des Mappings von Metadaten

[3] Vergleiche [King07]

- Eine Technik um mit transaktionalen Objekten zu interagieren, wie Dirty Checking, Lazy Association Fetching.

3.4 ORM - Formen

In dem folgendem Abschnitt werden die verschiedenen Formen des Objekt-Relationalen Mapping aufgezählt. [4]

- **Rein relational**
  Die gesamte Applikation inklusive Benutzerschnittstelle ist mit dem relationalen Modell und mit auf *SQL*(Structured Query Language) basierten relationalen Operationen entworfen worden. Diese Form ist performant, führt aber bei großen Projekten zu Unübersichtlichkeit und Mängeln bei der Wartbarkeit.
- **Light Object Mapping**
  Die Entitäten werden als Klassen repräsentiert, die manuell zu den relationalen Tabellen gemappt werden. Hand codiertes *SQL* wird vor der Business-Logik über bekannte Entwurfsmuster verborgen.
- **Medium Object Mapping**
  Die Applikation ist anhand eines Objektmodells entworfen wurden. *SQL* wird beim kompilieren über ein Tool zur Codegenerierung oder zur Laufzeit durch Framework-Code erstellt. Die Verknüpfung zwischen den Objekten werden über den Persistenzmechanismus unterstützt und Abfragen können über eine objektorientierte Sprache spezifiziert werden. Objekte werden durch die Persistenzschicht gecachet.
- **Full Object Mapping**
  Bei Full Object Mapping unterstützt die Objektmodellierung Komposition, Vererbung, Polymorphismus und Persistenz durch Erreichbarkeit. Die Persistenzschicht implementiert eine transparente Persistenz. Persistente Klassen erben nicht von speziellen Basisklassen oder müssen besondere Interface implementieren. Effiziente Fetching- und Caching-Strategien werden transparent in der Applikation implementiert. Zu dieser Form gehört auch *Hibernate*.

[4] Vergleiche [King07]

3.5 Objektrelationale Unverträglichkeit

Die objektrelationale Unverträglichkeit tritt auf wenn Objekte, von objektorientierten Programmiersprachen, in relationale Datenbanken gespeichert werden. Die vier Hauptprobleme werden hier näher erläutert.

- **Problem der Granularität**
  Granularität bezieht sich auf die relative Größe der Datentypen, mit denen man arbeitet.[5]

  Das Problem bezieht sich auf die Feingranularität der Objekte der objektorientierten Programmierung und der Grobgranularität der Tabellen aus relationalen Datenbanken.
- **Problem der Vererbung**
  Dieses Problem bezieht sich auf die Vererbung die in objektorientierten Programmiersprachen vorhanden ist, aber nicht in relationalen Datenbanken.
- **Problem der Identität**
  Hier geht es um den Unterschied zwischen den zwei Konzepten aus Java

  – Identität - Ist Objekt a auf dem gleichen Speicherort wie Objekt b

    ```
    a == b
    ```

  – Gleichheit - Hat Objekt a den gleichen Zustand wie Objekt b

    ```
    a.equals(b)
    ```

  und den zwei Konzepten von relationalen Datenbanken. In RDBS(Relationale Datenbank-Systeme) werden gleiche Daten über den Vergleich der Einträge gefunden. Identische Daten können allerdings nur mittels eindeutigem Primärschlüssel garantiert werden.
- **Problem der Assoziation**
  Dieses Problem handelt von den verschiedenen Arten der Assoziation in objektorientierten Programmiersprachen wie

  – 1 zu 1
  – 1 zu n
  – n zu 1
  – n zu m

  und über die Assoziationen in relationallen Datenbanken, die nur über Fremdschlüssel dargestellt werden.

[5] Vergleiche [King07]

4 Hibernate

4.1 Allgemein

In diesem Kapitel geht es um *Hibernate*, ein Open-Source ORM-Framework für Java, das seit 2001 entwickelt wird und sich aktuell in der Version 3.5.3 befindet. Es sollte als Alternative zu den EJB(Enterprise Java Bean) 2.1 dienen, das unter der Entwickler-Community, im Bereich Entity Beans und Persistenz, kritisch betrachtet wurde. Durch den Erfolg von Hibernate im Bereich Persistenz wurde die Version 3 von EJB mithilfe der Hibernate-Entwickler gestaltet.

4.2 Architektur

Bei *Hibernate* handelt es sich um eine Schicht, die sich zwischen der Anwendung und der Datenbank befindet. Es abstrahiert die Datenbank von der Anwendung. Siehe Abbildung 4.1.

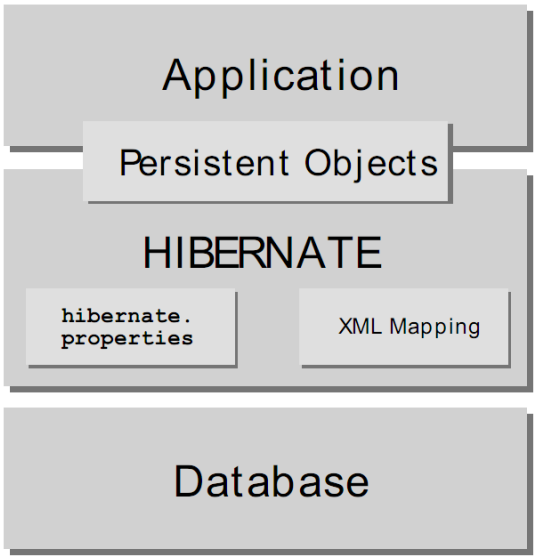

Abbildung 4.1: Hibernate Architektur [King10]

Es gibt verschiedene Einsatzmöglichkeiten wie z.B. das die Anwendung ihre JDBC(Java Database Connection), JTA(Java Transaction API) oder JNDI(Java Naming and Directory API) selbst verwaltet. JTA ist das standardisierte Service-Interface für Transaktionen in *Java Enterprise Applications*. Sie erlaubt z.B. das Arbeiten mit zwei Datenbanken in einer Transaktion. JNDI dagegen erlaubt es Objekte in einer hierarchischen Struktur zu speichern und wieder zu holen. Dies bietet sich bei den Connections und der SessionFactory an, da diese wieder leicht über JNDI geholt werden können. Allerdings benötigt

man dafür einen *Java EE Aplication Server*, der diese zusätzlichen Features bereitstellt. Weiterhin besteht die Möglichkeit diese Erweiterungen kapseln zu lassen und dies damit *Hibernate* zu überlassen. [6] Siehe dazu Abbildung 4.2

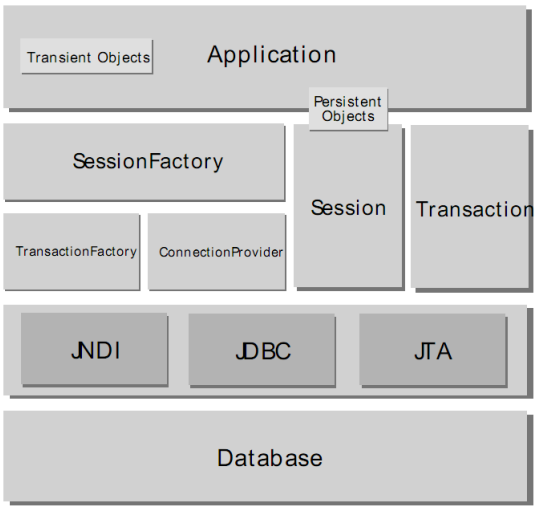

Abbildung 4.2: Hibernate - erweiterte Architektur [King10]

Die wichtigsten Elemente in *Hibernate* sind laut [King10]:

- **SessionFactory(org.hibernate.SessionFactory)**

  Ein threadsicherer, unveränderlicher Cache mit generierten SQL-Statements und den Mapping-Dateien. Die SessionFactory ist aufwendig in der Erstellung und Instantiiert die Sessions. Unterstützt die Anwendung mehrere Datenbanken, bekommt jede ihre eigene SessionFactory.

- **Session(org.hibernate.Session)** Ein nicht thread-sicheres, kurz lebendes Objekt das eine Konversation zwischen Anwendung und dem persistierten Daten repräsentiert. Es umhüllt die JDBC und stellt eine Transaktion bereit. Session hält den First-level Chache für persistente Objekte, die in Benutzung sind, während über den Objekt-Graph oder per Identifikator navigiert wird.

- **persistente Objekte und Collections** Kurzlebige und single-threaded, Objekte die den persistenzstatus und business Funktionen beinhalten. Diese können einfache JavaBeans/POJO(Plain Old Java Objects) sein. Sie sind mit genau einer Session verbunden. Wenn diese Session geschlossen wird, sind die Objekte und Collections freistehend. Es kann nicht garantiert werden, dass diese Daten noch aktuell sind. Sie können in allen Anwendungsschichten benutzt werden.

[6] Vergleiche [Röder10]

- **transiente und freistehende Objekte und Collections**

  Instanzen von persistenten Klassen die derzeit nicht mit einer Session verbunden sind. Sie können schon von einer Session instantiiert sein, aber sie sind noch nicht persistent oder sie wurden von einer geschlossenen Session instantiiert.

- **Transaction(org.hibernate.Transaction) - Optional** Single-threaded, kurzlebige Objekte die von der Annwendung genutzt werden, um spezielle atomare Arbeitsschritte zu erledigen. Es abstrahiert die Anwendung von den darunterliegenden JDBC, JTA/CORBA(Common Object Request Broker Architecture) Transaktionen. In bestimmten Fällen öffnet eine Session mehrere Transaktionen.

- **ConnectionProvider(org.hibernate.connection.ConnectionProvider)**

  Stellt einen Pool von JDBC-Connections bereit. Es abstrahiert die Anwendung von den darunterliegenden Datasource und DriverManager.

- **TransactionFactory(org.hibernate.TransactionFactory) - Optional**

  Sie stellt Transaction-Instanzen bereit.

- **Erweiterungsschnittstellen**

  Hibernate erlaubt es weitere Schnittstellen zu implementieren. um so das Verhalten der Persistenzschicht zu verändern.

4.3 Persistenzkontext

Als Persistenzkontext bezeichnet man die Menge von Entity-Objekten, in der für jede Instanz einer Entität innerhalb der Datenbank höchstens ein Java-Objekt im Kontext existiert.[7] Jede Session hat einen Persistenzkontext in dem alle persistenten Objekte, dieser Session, verwaltet werden. Darüber hinaus bietet er Vorteile, wie das **transactional write-behind**. Es werden dadurch die Änderungen so spät wie möglich in die Datenbank geschrieben. Damit werden die Zeiten, in den keine andere Transaktion auf den Datensatz zugreifen kann, so gering wie möglich gehalten. Des weiteren werden Objekte mittels **Dirty Checking** geprüft, ob Veränderungen mit der Datenbank synchronisiert wurden. Nicht synchronisierte Daten werden als *dirty* markiert und synchronisiert. Dadurch werden unnötige Belastungen der Datenbank reduziert und die Performance der Anwendung gesteigert.

[7] Vergleiche [Röder10]

4.4 Lebenszyklus der Objekte

Im Laufe einer Anwendung können Objekte verschieden Zustände annehmen, diese werden in der Abbildung 4.3 aufgezeigt und anschließend näher erläutert.[8]

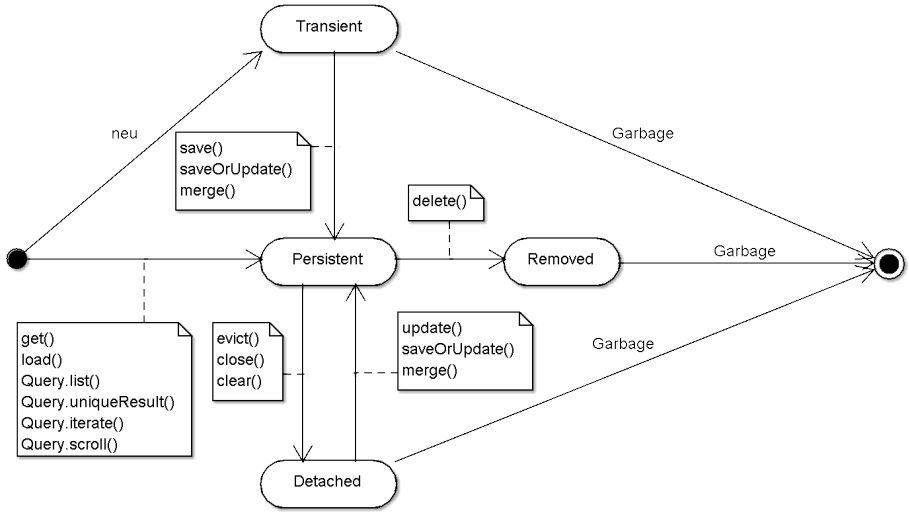

Abbildung 4.3: Lebenszyklus - Objektzustände und Übergänge

- **transient**

  Wird ein neues Objekt erzeugt, ist es in dem Zustand *Transient*. Das heißt, es ist nicht mit einem persistenzkontext verbunden und hat keine Identität(Primary Key). Das Objekt kann mittels Funktionen wie z.B. *save()* persisitert werden. Andernfalls wird das Objekt vom Garbage Collector entfernt, wenn keine Referenz darauf existiert.

- **persistent**

  Die Instanz ist aktuell mit einem Persistenzkontext verbunden. Sie hat eine persistente Identität(Primary Key) und hat eine korrespondierene Reihe in der Datenbank. Per *delete()* können persistente Objekte in den Zustand *removed* gesetzt werden. Andernfalls können Sie z.B. durch *clear()* in den *detached*-Zustand versetzt werden.

- **detached**

[8] Vergleiche [King07]

Die Instanz war einmal mit dem Persistenzkontext verbunden, dieser wurde geschlossen und dadurch kann nicht garantiert werden kann, dass der Zustand des Objektes aktuell ist. Änderungen an diesem Objekt werden nicht mehr mit der Datenbank synchronisiert. Mittels *merge()*, *update()* oder *saveOrUpdate()* wird das Objekt wieder an den Persistenzkontext gebunden und erhält den Zustand *persistent*.

- **removed**

  Das Objekt wurde per *delete()* für das Löschen vorgesehen und wird nach Beendigung der Unit of Work für den Garbage Collector freigegeben.

4.5 Vererbung

In objektorientierten Programmiersprachen, wie Java, ist die Vererbung ein Thema der ersten Stunde. Mit ihr können aus existierenden Klassen neue Klassen mit den selben Eigenschaften und Funktionen erstellt werden. Da relationale Datenbanken Vererbung nicht unterstützen, bietet *Hibernate* vier Arten, die Vererbung auf Datenbanken abzubilden.[9]

- **Tabelle pro konkreter Klasse mit implizitem Polymorphismus**
  Es wird für jede nicht abstrakte Klasse eine Tabelle benutzt, in der sich auch die geerbten Eigenschaften als Spalten befinden. Es gibt aber das Problem der schlechten Unterstützung für polymorphe Assoziation. Außerdem muss bei polymorphen Abfragen jede Subklasse abgefragt werden müssen. Dafür ist die Abfrage einer speziellen Subklasse mit einer Anweisung erledigt.
- **Tabelle pro konkreter Klasse mit Unions**
  Es wird für jede nicht abstrakte Klasse eine Tabelle benutzt. Mit der Änderung, dass mittels *UNION*-Operators, dass heißt Ergebnisse von mindestens zwei Abfragen zu kombinieren, die meisten Probleme der polymorphen Abfragen und der Assoziationen entfernt werden.
- **Tabelle pro Klassenhierarchie**
  Hierbei wird die gesamte Klassenhierarchie auf eine Tabelle abgebildet. In dieser befinden sich alle Eigenschaften der jeweiligen Klassen der Hierarchie. Es wird ein Discriminatorvalue, zur Unterscheidung von welcher Klasse der Eintrag in der Tabelle stammt, benötigt. Diese Strategie bietet die beste Performance, da alle Abfragen nur auf einer Tabelle stattfinden. Die Nachteile sind, dass es unnötige Spalten gibt, da die abgeleiteten Klassen nicht die selben Eigenschaften besitzen. Außerdem müssen alle Spalten nullable sein, dass die Datenintegrität betrifft.
- **Tabelle pro Subklasse**
  Bei dieser Vorgehensweise wird jeder Klasse, einschließlich abstrakter Klassen,

[9] Vergleiche [King07]

eine Tabelle zugeordnet und die Vererbung wird mittels Fremdschlüsselbeziehungen dargestellt. Die Performance ist durch nötige *JOIN*-Operationen beeinträchtigt.

4.6 Assoziationen

Eine Assoziation ist die Verbindung mindestens zweier Entities und erlaubt das Navigieren von der einen zur anderen Entity.[10]

Der Grad der Beziehung wird mittels der *Kardinalität* angegeben. Dabei wird zwischen *1 zu 1* , *1 zu n* / *n zu 1* und *n zu m* unterschieden. Des weiteren wird noch zwischen *unidirektional*, dass bedeutet nur in eine Richtung, und *bidirektional*, in beide Richtungen, differenziert.

Die Möglichkeiten der Assoziation sind:

- **1 zu 1**
  Bei einer *1 zu 1* -Beziehung gibt es zwei verschieden Arten für die Umsetzung. Bei der *Primärschlüsselassoziation* werden die Zeilen in zwei Tabellen, über Primärschlüssel verbunden. Außerdem besteht die Möglichkeit über *Fremdschlüsselassoziation*, bei der eine Tabelle eine Fremdschlüsselspalte mit dem Primärschlüssel der assoziierten Tabelle hat. Beide *1 zu 1* -Beziehungen können *uni* - oder *bidirektional* ausfallen.
- **1 zu n** / **n zu 1**
  Bei *1 zu n* - und *n zu 1* -Beziehungen gibt es die Möglichkeit sie über Fremdschlüssel abzubilden, dies ist laut [King10] nicht empfohlen. Desweiteren existiert die Möglichkeit über eine sogenannte *Join* -Tabelle die Beziehung abzubilden. Diese Tabelle hat zwei Fremdschlüsselspalten, die auf die Tabellen der jeweiligen Beziehungspartner referenzieren. Ebenfalls können*1 zu n* / *n zu 1* -Beziehungen *uni* - oder *bidirektional* ausfallen.
- **n zu m**
  Bei einer *n zu m* Beziehung wird immer die Join-Tabelle benutzt, diese ist analog der *1 zu n* / *n zu 1 Join* -Tabelle aufgebaut und besitzt auch die Möglichkeit *uni* - oder *bidirektionale* Beziehungen zu erstellen.

4.7 Caching

Da jegliche Art von Abfrage eine gewisse Ausführungszeit in Anspruch nimmt, ist Caching im Bereich von Datenbanken ein wichtiges Thema. Bei tiefgreifenden *JOIN* und

[10] Vergleiche [Röder10]

UNION -Konstrukten können aus wenigen Millisekunden gar mehrere Minuten entstehen. Diese Zeit muss der Nutzer der Anwendung warten bis er seine Arbeit fortführen kann. Um dies zu vermeiden gibt es das Caching. Hier werden bereits abgefragt Objekte in einen Zwischenspeicher gelegt, um später aus diesem darauf zuzugreifen, anstatt erneut eine aufwendige Abfrage zu starten. Dies spart nicht nur Zeit, sondern verringert auch die Belastung der Datenbank. Ein Problem entsteht, wenn die Einstellungen falsch gewählt wurden, z.B. Lebensdauer der Objekte im Cache. Dann dauert die Abfrage womöglich länger als eine ungecachte Abfrage. Ein weiteres Problem beim Caching ist die Datenkonsistenz. Wenn man Daten/Objekte zwischenspeichert, wie kann man sich sicher sein das diese Daten aktuell sind? Denn schlimmer als kurzfristiges Warten des Nutzers auf das Programm, ist das der Nutzer mit falschen Daten arbeitet. Für diesen Zweck stellt Hibernate mehrere Caching-Möglichkeiten zur Verfügung die im Folgenden näher beschrieben werden.

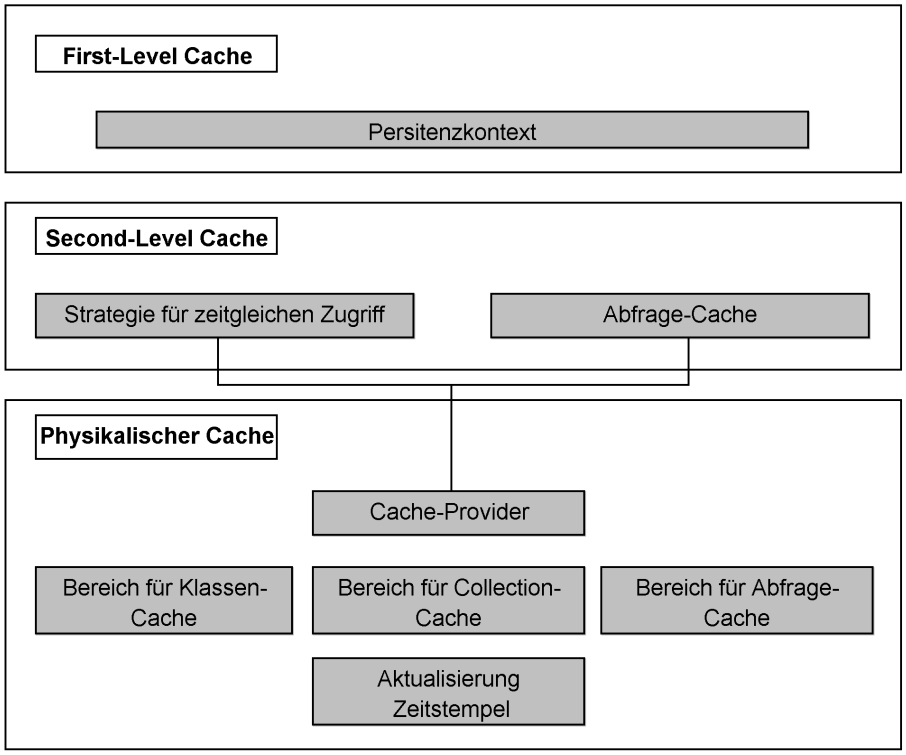

Abbildung 4.4: Caching-Architektur von Hibernate

4.7.1 First-level Cache

Der First-Level Cache ist der Persistenzkontext-Cache. Dieser wird von der Hibernate Session selbst implementiert und lebt genau so lange wie die Session.

Anwendung:
Eine existierende *SessionFactory* öffnet eine Session, daraufhin aktiviert sich der First-

Level Cache. Bei der ersten Abfrage wird geprüft, ob die gesuchten Objekte sich bereits im Cache befinden. Ist dies nicht der Fall wird der *SELECT* an die Datenbank übermittelt und die zurückgelieferten Daten in Objekte gespeichert. Außerdem werden diese Objekte dann in den dazugehörigen Cache-Bereich geladen. Befindet sich dagegen das gesuchte Objekt im Cache, werden sie ohne Datenbankzugriff aus dem Cache-Bereich geladen. Dadurch wird die Datenbanklast reduziert und durch das schnellere Speichermedium können Antwortzeiten deutlich reduziert werden.

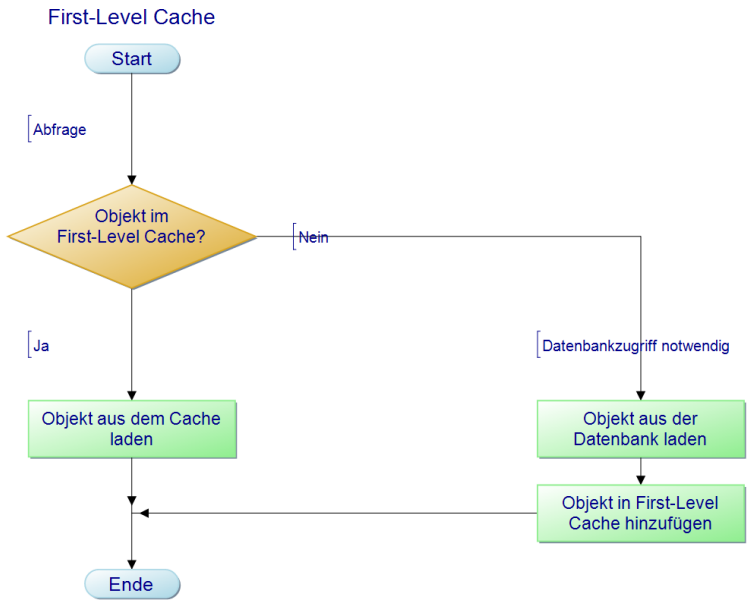

Abbildung 4.5: Ablaufdiagramm First-Level Cache

4.7.2 Second-level Cache

Der Second-Level Cache ist optional zuschaltbar und nur einmal pro Applikation vorhanden. Er lebt solange wie die Session-Factory, was im Idealfall bedeutet, solange wie die gesamte Applikation.
Dieser kann optional zugeschaltet werden und ist ein Cache des Zustandes und nicht der Instanzen der Persistenz. Für den zeitgleichen Zugriff definiert eine Caching-Strategie die Details der Transaktionsisolation. Die physikalische CacheImplementierung repräsentiert der Cache-Provider. Dieser Cache sollte bloß dann aktiviert werden, wenn die Daten selten aktualisiert werden. Da die *Aktualisierung* des Caches die Performancevorteile der schnelleren Lesevorgänge zunichte machen könnte. Im schlimmsten Fall verschlechtert sich die Performance.

Anwendung:
Ein Thread baut mithilfe einer Session-Factory eine Session auf. Bei der ersten Abfrage werden die gefunden Daten in Objekte gespeichert und in den First- und Second-Level

Cache geladen. Bei einer erneuten Abfrage werden die Objekte aus dem First-Level Cache geholt. Wird ein neuer Thread, mit neuer Session, aufgebaut, kommt es bei der ersten Abfrage dazu, dass im First-Level Cache nachgeschaut wird. Da dieser aber leer ist, da seine Lebensdauer auf die Session begrenzt ist, wird im Second-Level Cache nachgeschaut. Dieser besitzt die Lebenszeit der Session-Factory und so befinden sich hier die Objekte des ersten Threads. Dadurch verringert sich auch die Zugriffszeit der ersten Abfrage des zweiten Threads.

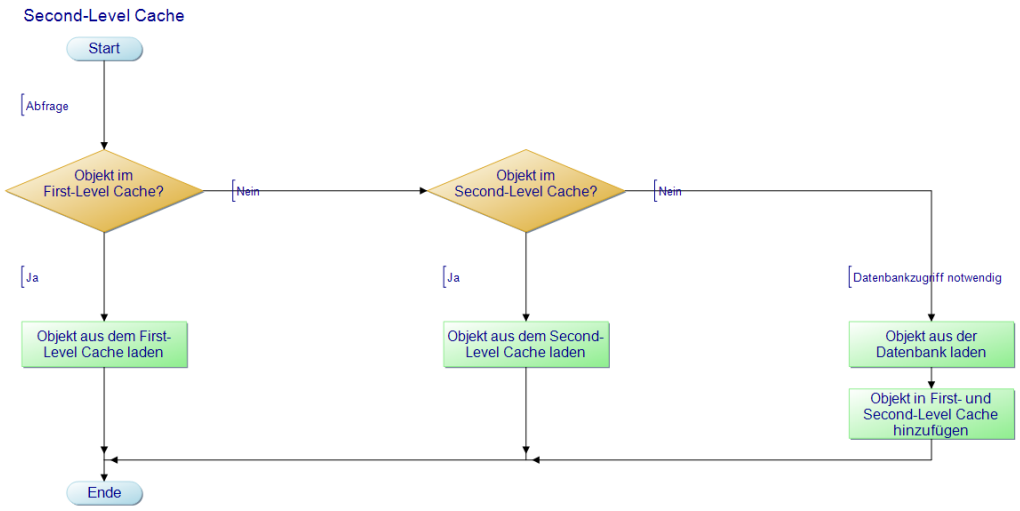

Abbildung 4.6: Ablaufdiagramm Second-Level Cache

4.7.3 Abfrage-Cache

Dieser Cache ist ebenfalls optional zuschaltbar und ist dann sinnvoll wenn sich Abfragen oft wiederholen. Er benötigt einen aktivierten Second-Level Cache. Zusätzlich erfordert er zwei zusätzliche physikalische Cache-Bereiche. Einer für die gecachten Abfrage-resultate und einer für die Zeitstempel mit der Zeit, wann die Tabelle das letzte mal aktualisiert wurde.

Anwendung:
Bei einer Abfrage werden alle Cachebereiche gefüllt, First- und Second-Level Cache speichern die Objekte und der Abfrage-Cache speichert die *SQL-Statements* und die Identifikatoren der Ereignismenge. Bei einer zweiten Abfrage wird im Abfrage-Cache geprüft, ob das Statement vorhanden ist. Wenn ja dann wird mittels dem Identifikator die Objekte aus dem Second-Level Cache geladen.

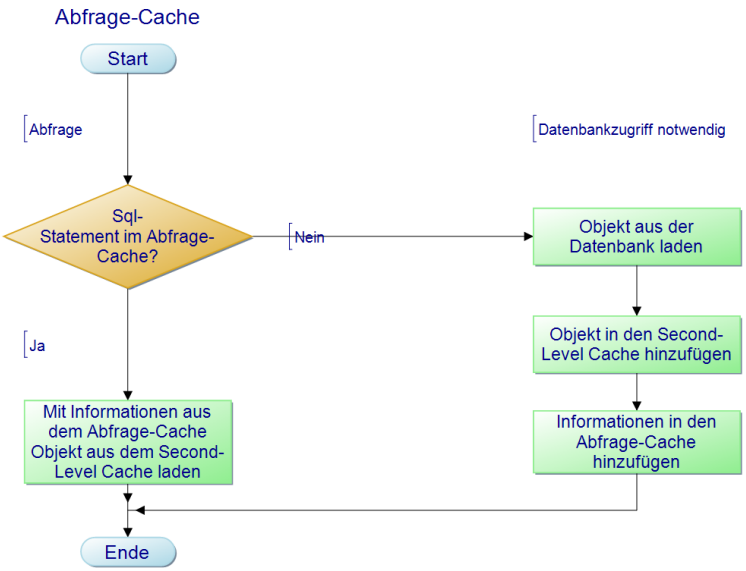

Abbildung 4.7: Ablaufdiagramm Abfrage Cache

4.7.4 Caching-Strategien

Der Second-Level Cache verfügt über vier Strategien um mit bereits geladenen Daten umzugehen. [11]

- **read-only**
  Die performanteste Strategie, da Objekte nur gelesen werden. Es findet keine Transaktionisolation statt und wird ein Objekt geändert, so wird eine Exception geworfen. Dies ist die passende Strategie wenn die Daten nur gelesen werden.
- **nonstrict read/write**
  Bei dieser Strategie können Daten verändert werden. Es gibt aber keine Garantie, dass die Daten aus dem Cache aktuell sind. Wenn sich Daten über einen längeren Zeitraum nicht ändern oder das Risiko verfallener Daten vernachlässigbar ist, ist diese Startegie einsetzbar.
- **read/write**
  Hier wird mit einer read-commited-Isolation gearbeitet, die einen Zeitstempelmechanismus verwendet. Es werden nur Daten gesehen, die commited sind. Daten aus nicht beendeten Transaktionen werden nicht betrachtet. Diese Strategie kann eingesetzt werden, wenn Daten vor allem gelesen werden. Außerdem wenn es wichtig ist, dass nicht mehr aktuelle Daten nicht in zeitgleich ablaufenden Transaktionen vorkommen.
- **transactional**
  Bei *transactional* wird eine vollständige Transaktionsisolation garantiert. Hierbei können laufende Transaktionen keine Daten ändern, die von nicht beendeten

[11] Vergleiche [King07]

Transaktionen gelesen werden. Diese Caching-Strategie passt dann, wenn Daten vor allem gelesen werden und strikt darauf zu achten ist, dass nicht mehr aktuelle Daten in zeitgleich ablaufenden Transaktionen vorkommen. Diese Strategie kann nur in einer JTA-Umgebung verwendet werden.

Für den Second-Level und den Abfrage-Cache benötigt man einen Cache-Provider. Hibernate liefert standardmässig EHCache mit. Der Funktionsumfang im Bezug auf die Cache-Strategien ist hier unterschiedlich und wenn nötig ist auf einen anderen Provider zu wechseln. In der folgenden Tabelle sind die populärsten Cache-Provider und deren Funktionsumfang aufgelistet.

Cache-Provider	**read-only**	**nonstrict read/write**	**read/write**	**transactional**
EHCache	x	x	x	
OSCache	x	x	x	
SwarmCache	x	x		
JBossCache	x			x

Tabelle 4.1: Übersicht der von Hibernate unterstützten Cache-Provider

4.8 Loading

Um Daten aus der Datenbank zu laden kann man in Hibernate zwei verschiedene Strategien verfolgen. Die erste mit dem Namen *Lazy Loading* generiert beim Laden einen Proxy(Platzhalter) des Objektes. Dies hat zum Vorteil, dass dabei noch kein Zufgriff auf die Datenbank erfolgt. Andere Objekte können auf diesen Proxy referenzieren ohne je einmal die Datenbank abgefragt zu haben. Wird jedoch auf die Attribute des Objektes zugegriffen werden die Daten geladen. Andersherum verhält es sich mit *Eager Loading*, bei dem das Objekt mit den dazugehörigen Assoziationen und Collections sofort aus der Datenbank geladen werden.

4.9 Locking

Unter *Locking* versteht man das Blockieren eines Datensatzes in der Datenbank, um diesen gegenüber Veränderung anderer Transaktionen zu schützen. *Hibernate* bietet zwei Arten des *Lockings*. Bei *Pessimistic Locking* werden alle Datensätze gesperrt, die durch die laufende Transaktion geladen wurden. Dadurch wird sichergestellt, dass keine andere Transaktion die geladenenen Objekte bearbeitet oder entfernt. Dies ist sicher, kann sich aber Negativ auf die Performance der Anwendung auswirken. Es kann vorkommen, dass benötigte Objekte nicht geladen werden können, weil sie von einer anderen Transaktion blockiert sind. Dagegen wird bei *Optimistic-Locking* während der Transktion nichts geblockt. Allerdings wird vor dem Schreibversuch überprüft, ob der Da-

tensatz noch aktuell ist. *Optimistic-Locking* ist Grundeinstellung bei *Hibernate*. Desweiteren erlaubt *Hibernate* das Verändern des Isolationslevels der jeweiligen Datenbank. Dieses Level bestimmt, wie Änderungen an Daten, in gleichzeitigen Transaktionen, gehandhabt werden.

- **Read-Uncommited-Isolation**
- **Read-Commited-Isolation**
- **Serializable-Isolation**
- **Repeatable-Read**

4.10 Abfrageverfahren

4.10.1 Hibernate Query Language

Um Anfragen an die Datenbank zu stellen, kann man 3 unterschiedliche Methoden verwenden. Die *HQL*(Hibernate Query Language) bietet die Möglichkeit, Anfragen zu stellen, die mit der Syntax nahe an der Sprache *SQL* ist. Dadurch ist sie leicht zu Erlernen und des weiteren können Parameter den *SELECTS* übergeben werden. Durch diese Parameter werden die Abfragen zusätzlich vor *SQL-Injections* geschützt, in dem sie vor dem einfügen Sonderzeichen usw. entfernt. Dies ist vor allem bei Web-Anwendungen sinnvoll, da viele Nutzer die Möglichkeit besitzen Datenbankbefehle mittels Eingabefeld einer Webseite einzugeben.

```
1 Query q = session.createQuery(
2     "from User as u where u.firstname = :fname"
3 );
4 q.setString("fname", "John");
5 List result = q.list();
```

Listing 4.1: Hibernate Query Language

4.10.2 Query by Criteria

Eine weitere Art Anfragen zu stellen, ist mittels *Query by Criteria* möglich. Diese Variante zeichnet sich durch ihre objektorientierte Erstellung der Abfragen aus und bedeutet, dass die Einschränkungen per Methodenverkettung zusammengesetzt werden. Dadurch können dynamische Query leicht erstellt werden. Da diese Anfragen ohne String-Manipulationen auskommen, können die Abfragen beim Kompilieren geparst und damit

die Fehler beim Erstellen verringert werden. Da keine Ähnlichkeit zu *SQL* besteht, sind die *Criteria*-Abfragen am Anfang schwer zu lesen und zu verstehen.

```
Criteria criteria = session.createCriteria(User.class);
criteria.add(Restrictions.like("firstname", "John");

//Beispiel der dynamische erweiterung
if(nachname != null && !nachname.equals("")) {
    criteria.add(Restrictions.eq("lastname",nachname));
}

List result = criteria.list();
```

Listing 4.2: Query by Criteria

4.10.3 Natives SQL

Eine weiter Methode besteht darin, *natives SQL* des jeweiligen Datenbanksystemes zu verwenden. Dies ist die einfachste Variante bestehende Anwendungen auf Hibernate umzustellen, da man sich datenbankspezifische Funktionen ausführen lassen kann. Es erlischt allerdings dabei der Vorteil der Datenbankunabhängigkeit. Außerdem besitzt *natives SQL* auch die Möglichkeit Parameter zu benutzen.

```
List list = session.createSQLQuery("SELECT * FROM user WHERE firstname = John").list();
```

Listing 4.3: Natives SQL

4.10.4 Named Querys

Hibernate bietet die Möglichkeit Abfragen aus dem Programmcode zu entfernen und in die Mapping-Dateien als sogenannte *Named-Querys* auszulagern. In Hibernate besteht die Möglichkeit Abfragen aus dem Code zu entfernen und als *Named-Query* in die zu der jeweiligen Datenbanktabelle gehörenden Mapping-Datei zu schreiben. Dies bietet den Vorteil häufig benutzte Abfragen zentral in der Mapping-Datei zu speichern und evtl. zu bearbeiten.

```
<query name="TestQuery">
    <![CDATA[
        from auftrag where auftrag.id = ?
    ] ]>
</query>
```

Listing 4.4: Named Query in Mapping-Datei

Diese Abfragen können dann im Javacode verwendet werden.

```
1 Query query = session.getNamedQuery("TestQuery");
2 query.setInt(0,auftragsId);
3 Auftrag auftrag = query.uniqueResult();
```

Listing 4.5: Named Query in Java

4.11 Fetching

Im folgenden werden die Strategien, die Hibernate einsetzt, um in Beziehung stehende Objekte zu erhalten. Diese werden in den Meta-Daten deklariert, können aber teilweise von HQL oder Criteria überschrieben werden. [12]

- **Select Fetching**
  Um Assoziationsentitäten oder Collections zu erhalten wird bei dieser Strategie ein zweiter Select benutzt. Auch wenn Lazy Loading deaktiviert wurde, wird die zweite Abfrage nur ausgeführt, wenn man auf die Beziehung zugreift. Bei vielen Beziehungen kann diese Strategie zu einem enormen Performanceverlust führen, da hier das *N+1* -Problem auftreten kann. Dieses Problem tritt auf, wenn ein Objekt geladen wird (1) und durch die Beziehungen zu dem Objekt müssen weitere Ladevorgänge gestartet werden (n).
- **Join Fetching**
  Hier werden die Assoziationsinstanzen oder Collections mittels *Outer-Join* in dem ersten Select mit abgefragt. Wenn im vorhinein bekannt ist, dass die Objekte der Assoziationen benötigt werden, kann die Strategie hilfreich sein. Allerdings kann die Performance bei Beziehungen mit großen Ergebnismengen durch den *Join* stark einbrechen. Diese Strategie wird von Hibernate empfohlen.
- **Subselect Fetching**
  Bei dieser Strategie werden alle assoziativen Collections mittels *Subselect* in einem zusätzlichen Statement geladen.
- **Batch Fetching**
  Dies ist eine Verbesserung des Select-Fetching, in dem durch die Angabe einer Batchgröße bestimmt wird, wie viele Referenzen gleichzeitig in einem Select geladen werden. Dadurch wird aus dem *N+1* -Problem ein *N/Batchgröße+1* -Problem.

4.12 unterstützte Datenbanksysteme

Hibernate unterstützt von Haus aus eine große Menge Datenbanken. Darüber hinaus besteht die Möglichkeit nicht unterstützte Datenbanksysteme durch Schreiben eines passenden Datenbank-Dialektes in Hibernate hinzuzufügen. Dies wird der Fall sein bei dem SQLBase Dialekt, da SQLBase nicht zu den unterstützten Datenbanken gehört.

[12] Vergleiche [King07]

Datenbank	unterstützt von Hibernate	unterstützt von DELECO®
Corel Paradox	Ja	Nein
DB2	Ja	Ja
Firebird	Ja	Nein
HypersonicSQL	Ja	Nein
HP NonStop SQL/MX	Ja	Nein
Informix	Ja	Ja
Ingres	Ja	Nein
Interbase	Ja	Nein
InterSystems Cache'	Ja	Nein
Microsoft Access	Ja	Nein
Microsoft Excel	Ja	Nein
Micrososft SQL Server	Ja	Ja
MySQL	Ja	Nein
Oracle	Ja	Ja
Pointbase	Ja	Nein
PostgreSQL	Ja	Ja
SAP DB	Ja	Nein
SQLBase	Nein	Ja
Sybase	Ja	Nein
Xbase	Ja	Nein

Tabelle 4.2: Übersicht der von Hibernate und von DELECO® unterstützten Datenbanksysteme

4.13 Vor- und Nachteile

Ein Vorteil von Hibernate ist es, dass es durch die Abstraktion der verwendeten Datenbank möglich ist, Anwendungen zu entwickeln, ohne Bezug auf die Datenbank zu nehmen. Dadurch kann man mit jedem unterstützten Datenbanksystem auf die gleiche Weise kommunizieren, ohne auf spezielle Eigenschaften der Datenbank, Rücksicht zu nehmen. Außerdem erleichtert dies die Entwicklung mit mehreren Datenbanken, da der Code nur einmal geschrieben werden muss, um auf den verschiedenen Datenbanken zu funktionieren. Ein weiterer Vorteil ist die objektorientierte Programmierweise die durch das ORM entsteht. Damit lassen sich alle Tabelleneinträge als Objekte, mit Eigenschaften, ansprechen. Des weiteren ist es möglich, mittels *Query by Criteria*, Abfragen dynamisch, per Methodenverknüpfung, zu erstellen. In Sachen Performance lassen sich durch Caching und den verschiedene Cachingstrategien einzelne Objekte bis hin zu ganzen Abfragen zwischenspeichern. Dies erhöht, mit den richtigen Einstellungen, erheblich die Performance, da der Datenbankzugriff vermindert wird. Um aus vorhandene Datenbankschemas oder Javaklassen nicht die ganzen Einstellungen von Hand durchzuführen, erlauben die *Hibernate-Tools Reverse Engineering*. Daraus folgt das man sich aus einem vorhandenen Datenbankschema die erforderlichen POJO-Klassen und die dazugehörigen Konfigurationsdateien generieren lassen kann. Andersherum besteht auch die Möglichkeit, sich aus selbst erstellten POJOs und Konfigurationsdateien, ein neues Datenbankschema zu erstellen.

Ein Nachteil ist das die Ausführung von Statements, ohne aktivierten Cache, minimal langsamer ist als ohne Hibernate. Außerdem benötigt der Aufbau der SessionFactory, abhängig von der Anzahl und der Größe der Tabellen, Zeit im zweistelligen Sekundenbereich. Daher eignet sich *Hibernate* vorzüglich für Web-Anwendungen, da diese im Idealfall *endlos* laufen und so die Zeit für die Erstellung der SessionFactory, nach dem ersten Start, wegfällt. Ein weiterer Nachteil tritt auf, wenn eine bereits existierende Anwendung auf *Hibernate* umgestellt wird, da der Aufwand relativ groß ist die vorhandenen Datenbankzugriffe, mit Hibernate-Zugriffen, zu ersetzten. Mit *HQL* und *nativen SQL* ist er relativ gering, aber bei dem Wechsel auf *Criteria* wird der Aufwand extrem hoch, da nichts, von dem vorher geschriebenen SQL-Befehlen, bestehen bleibt.

4.14 NHibernate

Durch den Erfolg von *Hibernate* wurde mit Nhibernate ist eine Portierung C# vorgenommen. Die aktuelle Version ist NHibernate 2.1.2. Bei dem Thema Reverse-Engineering geht bei NHibernate von Haus aus nur die Schema-Generation. Für das Erstellen von Klassen- und Mappingdateien benötigt man das Paket *NHibernateContrib*. Die Konfigurationsdatei hat den gleichen Aufbau und ähnliche Eigenschaften wie in Hibernate. Die POCOs(Plain Old Common Language Runtime Object) sind wie die POJOs in Java, einfache Klassen mit default Konstruktor sowie getter- und setter Methoden. Die Klassen dürfen nicht mit dem *sealed* -Modifizierer ausgestattet sein, dass heißt von der Klasse darf nicht geerbt werden. Ebenfalls müssen die Methoden *virtual* sein, dass heißt diese Methoden können in abgeleiteten Klassen überschrieben werden. Die Mapping-Dateien sind ähnlich den von *Hibernate* für Java. Bei den Abfragen sind ebenfalls HQL, Criteria und natives SQL möglich. Für den Second-Level Cache stehen Hashtable, ASP.NET(Active Server Pages .NET) Cache und PrevalenceCache zur Verfügung. Bis auf *transactional* sind alle Caching-Strategien aus Hibernate übernommen worden und sie werden von allen Cache-Providern implementiert.

5 Istzustand

Auf dem aktuellen Stand werden im DELECO® die benötigten *Selects* immer in *SQL-Base* -Syntax geschrieben. Diese werden mittels selbstgeschriebener **clib33.dll**, einer dynamischen Bibliothek, in die jeweilige Datenbanksyntax übersetzt. Dies hat den Vorteil Abfragen nur einmal zu schreiben und trotzdem mehrere Datenbanken damit anzusprechen. Nachteil ist, dass die Zahl der Datenbankdialekte begrenzt ist und nicht alle Datenbankspezifischen Funktionen implementiert sind.

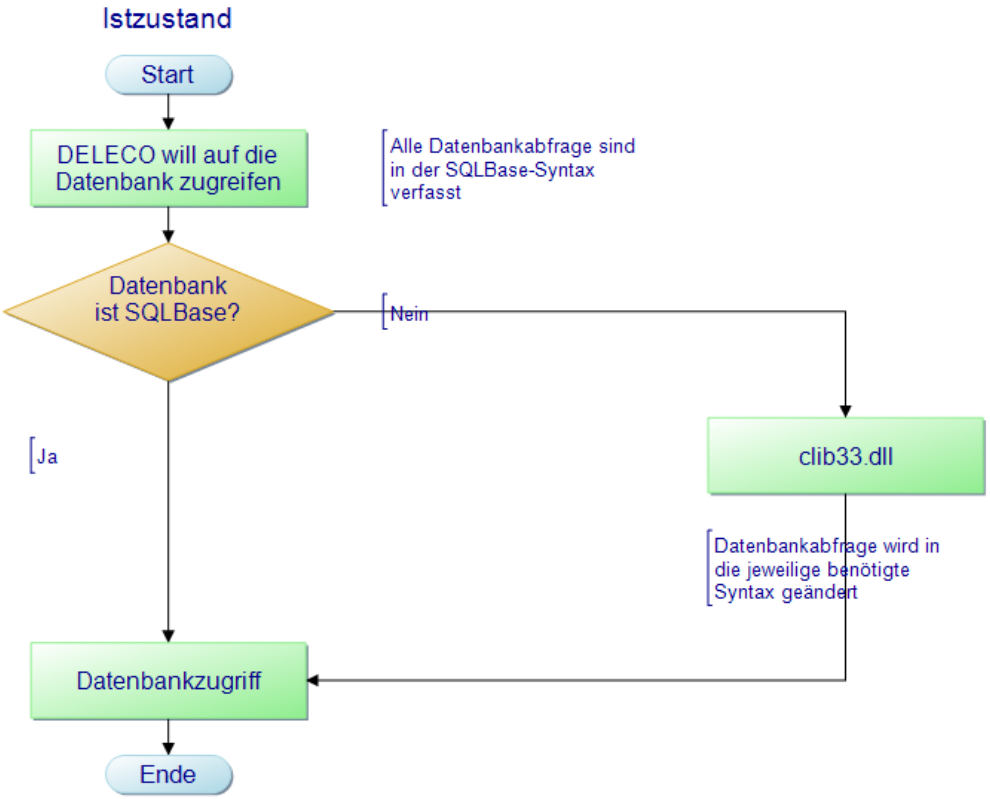

Abbildung 5.1: Istzustand DELECO®

Bei den Java-Anwendungen wird per Java-Klasse *SQLStringParser* mittels *switch-case* Blöcken die jeweilige Syntax aus *Strings* zusammengesetzt.

6 Sollkonzept

Nach erfolgreicher Integration von Hibernate muß es das Ziel, sein alle Datenbankzugriffe über *Hibernate* laufen zu lassen. Dies beginnt bei der einheitlichen Abfrage-Spache *HQL* als einzige verbleibende Abfrage-Sprache. Dadurch kann nach kurzer Eingwöhnung jede Abfrage mit *HQL* definiert werden und *Hibernate* übernimmt im Hintergrund die Anpassungen an die vorhandenen und zukünftigen Dialekte. Des weiteren wird durch den objektorientierten Zugriff der Daten, die Benutzbarkeit steigen. In Sachen Performance sollte bei einfachen Abfragen mindestens die gleiche Zeit benötigt werden wie ihne Zwischenschicht. Bei Anwendungen, die den Einsatz von Caching einfach machen, durch wiederholte Abfragen, usw., wie z.B. das Cockpit oder das Ereignismanagement, sollte der Gewinn in Sachen performance deutlich bemerkbar machen.

Abbildung 6.1: Sollzustand DELECO®

7 Implementierung des Prototyps

7.1 Allgemein

Bei dem Prototyp handelt es sich um ein Portlet, mit dem Namen *Ressourcenauslastung* aus dem DELECO®Cockpit. Es veranschaulicht die Auslastung jeweiliger Ressourcen in einem bestimmten Zeitraum. Um die Ergebnisse zu Individualisieren, kann man den Zeitraum von einem Tag bis zu einem Jahr begrenzen. Ebenso ob man alle Ressourcen oder nur bestimmte auswerten will. Die Ergebnisse können nicht nur anhand von Zahlenwerten sondern auch übersichtlich als Diagramm angezeigt werden.

Da im DELECO®-Umfeld nicht nur mit Java sondern auch mit C# gearbeitet wird, wäre eine möglichst einheitliche Herangehensweise und Implementation von Vorteil. Dass heißt, Besonderheiten der jeweiligen Version außen vorlassen und auf eine gemeinsame Basis aufbauen.

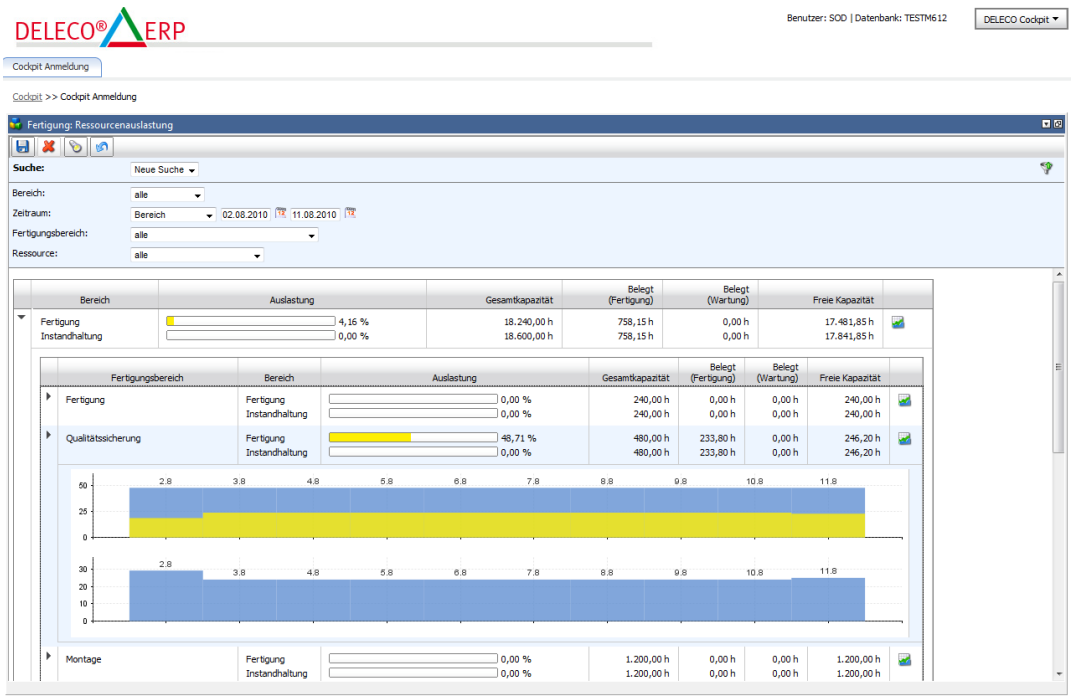

Abbildung 7.1: Portlet Ressourcenauslastung

7.2 Prototypische Umsetzung

Da es sich bei dem Prototypen um eine Web-Anwendung handelt, die in Java und mit Servlets realisiert wurde, kann man den *ServletContext* benutzen. Dieser wird in jeder Servlet-Anwendung genau einmal gestartet und einmal beendet. Da man die *Session-*

Factory auch einmal starten muss, um davon die einzelne *Session* abzurufen, erstellt man einen Listener. Der Listener startet die *SessionFactory* dann, wenn der *Servlet-Context* gestartet wird und beendet diese wieder.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5">
    ...
    <listener>
        <listener-class>com.deltabarth.utils.log4j.Log4JLifecycleListener</listener-class>
    </listener>
    <listener>
        <listener-class>com.deltabarth.deleco.portlet.HibernateListener</listener-class>
    </listener>
    ...
</web-app>
```

Listing 7.1: HibernateListener in web.xml

7.3 Hibernate-Einstellungen

Zuerst wird eine Hibernate-Konfigurationsdatei namens *hibernate.cfg.xml* angelegt. Diese muss root-Verzeichnis des Java-Quell-Codes liegen. In diesen Einstellungen müssen auf jeden Fall die Verbindungsdaten für die Datenbank enthalten sein.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <!-- Datenbank-Dialekt -->
    <property name="hibernate.dialect">org.hibernate.dialect.HSQLDialect</property>
    <!-- JDBC-Treiber -->
    <property name="hibernate.connection.driver_class">com.mircosoft.sqlserver.jdbc.
        SQLServerDriver</property>
    <!-- JDBC-Connection URL -->
    <property name="hibernate.connection.url">jdbc:sqlserver://DELSQL\MSSQL2005;Database=testm612
        </property>
    <!-- DB-Benuttzername -->
    <property name="hibernate.connection.username">SYSADM</property>
    <!-- DB-Passwort -->
    <property name="hibernate.connection.password">SYSADM</property>
    <!-- Session-Kontext -->
    <property name="current_session_context_class">thread</property>
    <!-- Second-Level-Cache Provider -->
    <property name="hibernate.cache.provider_class">org.hibernate.cache.EhCacheProvider</
        property>
    <!-- Aktivierung des Query-Caches -->
    <property name="hibernate.cache.use_query_cache">true</property>
    <!-- c3p0 Connectionpool -->
    <!-- minimale Connections -->
    <property name="hibernate.c3p0.min_size">5</property>
    <!-- maximale Connections -->
    <property name="hibernate.c3p0.max_size">20</property>
    <!-- timeout -->
    <property name="hibernate.c3p0.timeout">300</property>
    <!-- maximale Statements -->
```

```
30      <property name="hibernate.c3p0.max_statements">50</property>
31      <!-- Lebenszeit im Idle -->
32      <property name="hibernate.c3p0.idle_test_period">3000</property>
33      <!-- Generierte SQL-Befehle ausgeben -->
34      <property name="hibernate.show_sql">true</property>
35      <!-- Mapping-Datein -->
36      <mapping resource="com/deltabarth/deleco/portlet/orm/domain/Allinone.hbm.xml"/>
37      <mapping resource="com/deltabarth/deleco/portlet/orm/domain/WwsKapa.hbm.xml"/>
38      <mapping resource="com/deltabarth/deleco/portlet/orm/domain/WwsRes.hbm.xml"/>
39      . . .
40    </session-factory>
41  </hibernate-configuration>
```

Listing 7.2: Hibernate-Konfiguration für Microsoft SQL-Server 2005

7.4 Datenbank-Dialekte

Hibernate benötigt für die Ausführung von Anweisungen zu einer bestimmten Datenbank einen sogenannten Dialekt. Dieser enthält die speziellen Datentypen, Syntax und wenn vorhanden, eigene Funktionen des Datenbanksystems.

- **SQLBase**
  Da es für SQLBase keinen vordefinierte Dialekt gibt, muss er selbst erstellt werden. Dazu leitet man eine neuen Dialekt aus der Oberklasse *Dialect.java* aus dem Paket *org.hibernate.dialect* ab. In dieser werden dann die speziellen Datentypen und die speziellen Funktionen der Datenbank gesetzt. Die Besonderheit bei den Funktionen ist im Grunde nur, dass der Präfix @ davor steht.

```
1
2  public class SQLBaseDialect extends Dialect {
3
4      public SQLBaseDialect() {
5          registerColumnType(Types.BOOLEAN, "SMALLINT");
6          registerColumnType(Types.INTEGER, "INTEGER");
7          ...
8
9          registerFunction("substring",
10             new SQLFunctionTemplate(Hibernate.STRING,"@mid(?1,?2,?3)"));
11         registerFunction("length",
12         new SQLFunctionTemplater(Hibernate.INTEGER,"@length(?1)"));
13         ...
14     }
15 }
```

Listing 7.3: SQLBase Dialekt

- **Informix**
  Bei Informix sind keinerlei Einstellungen zu tätigen.
- **Microsoft SQL-Server**
  Bei Microsofts SQL-Server sind keinerlei Einstellungen zu tätigen.
- **Oracle**
  Bei Oracle 9 sind keinerlei Einstellungen zu tätigen.

7.5 Mapping-Datein

Beim automatischen Erstellen der Mapping-Dateien, wird durch einen Fehler in Hibernate, jede Spalte die den Namen *version* trägt nicht als nötige *<property>*-Eigenschaft sondern als *<version>* eingetragen.

```
<hibernate-mapping>
    <class name="model.Au" table="AU" schema="SYSADM">

        ...

        <version name="version" type="big_decimal">
            <column name="VERSION" precision="22" scale="0" />
        </version>

        ...

    </class>
</hibernate-mapping>
```

Listing 7.4: Fehlerhafte Mapping-Datei nach automatischer Generierung

Dies muss manuell geändert werden, da es sonst schon beim Aufbau der SessonFactory zu Fehlern kommt und sich das Programm beendet.

```
<hibernate-mapping>
    <class name="model.Au" table="AU" schema="SYSADM">

        ...

        <property name="version" type="big_decimal">
            <column name="VERSION" precision="22" scale="0" />
        </property>

        ...

    </class>
</hibernate-mapping>
```

Listing 7.5: Überarbeitete Mapping-Datei nach automatischer Generierung

- **Informix**
  Beim automatischen Erstellen der Mapping-Dateien von Hibernate muss darauf geachtet werden, dass man das Datenbankschema und den Datenbankkatalog aus der Datei entfernt. Da Hibernate die Selects aus den Mapping-Dateien erstellt, die im Falle von Informix fehlerhaft sind. Anstatt ***SELECT ... FROM testi612:informix.tabelle*** wird ***SELECT ... FROM testi612.informix.tabelle*** generiert. Informix akzeptiert nur **:** als Trennzeichen des Datenbanknamens und Schemas. Dies führt dazu das man keine *SELECTs* erfolgreich ausführen kann. Dazu wurde ein Anwendung geschrieben welches die Mapping-Dateien anpasst.

```
...
<hibernate-mapping>
    <class name="model.An" table="an" schema="informix" catalog="testi612">
        <id name="anrefid" type="int">
            <column name="anrefid" />
            <generator class="assigned" />
        </id>
...
```

Listing 7.6: Fehlerhafte Mapping-Datei nach automatische Generierung für Informix

zu

```
...
<hibernate-mapping>
    <class name="model.An" table="testi612:informix.an">
        <id name="anrefid" type="int">
            <column name="anrefid" />
            <generator class="assigned" />
        </id>
...
```

Listing 7.7: Überarbeitete Mapping-Datei für Informix

Bei *SQLBase*, *SQL Server* und Informix treten keine Probleme bei der automatischen Generierung auf.

7.6 Cache

Für den Second-Level und Query-Cache wird der mitgelieferte Cache-Provider *EhCache* verwendet. Als erstes wird eine Konfigurationsdatei für den Cache erstellt. Dieser muss ehcache.xml heißen und sich im root-Verzeichnis befinden.

```
<ehcache>
    <diskStore path="java.io.tmp" />
    <defaultCache
        maxElementsInMemory="3000"
        timeToLiveSeconds="600"
        timeToIdleSeconds="300"
        overflowToDisk="true"/>
</ehcache>
```

Listing 7.8: EhCache.xml Einstellungen des Caches

Bei diesen Einstellungen wird eine *default*-Einstellung getätigt, zusätzlich kann für jede einzelne Tabelle eine eigene Einstellung eingetragen werden. Die erste *default*-Einstellung besagt, dass sich maximal 3000 Elemente im Speicher befinden dürfen.Die zweite und dritte Einstellung legt fest das die Daten 600 Sekunden Lebenszeit besitzen

und sich 300 Sekunden im Leerlauf befinden können, bevor sie aus dem Cache entfernt werden. Außerdem können die zwischengespeicherten Daten auf die Festplatte ausgelagert werden.

7.7 Benchmark

Als Performancetest wurde das Portlet im Originalzustand und nach dem Implementieren von *Hibernate* getestet. Es wurde der Second-Level- und der Abfrage-Cache bei *Hibernate* eingeschaltet. Die Suchkriterien waren bei allen Ausführungen gleich.

Aktion	**Original-Zustand**	**Hibernate**
Erste Abfrage Ressourcenauslastung aller Ressourcen des vergangenen Jahres	300 ms	650 ms
Zweite Abfrage Ressourcenauslastung aller Ressourcen des vergangenen Jahres	300 ms	50 ms
Erste Abfrage Ressourcenauslastung aller Ressourcen der letzten Woche	250 ms	600 ms
Zweite Abfrage Ressourcenauslastung aller Ressourcen der letzten Woche	250 ms	50 ms
Erste Abfrage Ressourcenauslastung einer Ressourcen der letzten Woche	200 ms	450 ms
Zweite Abfrage Ressourcenauslastung einer Ressourcen der letzten Woche	200 ms	30 ms

Tabelle 7.1: Übersicht der Abfrage-Performance durch Hibernate und dem Second-Level Cache

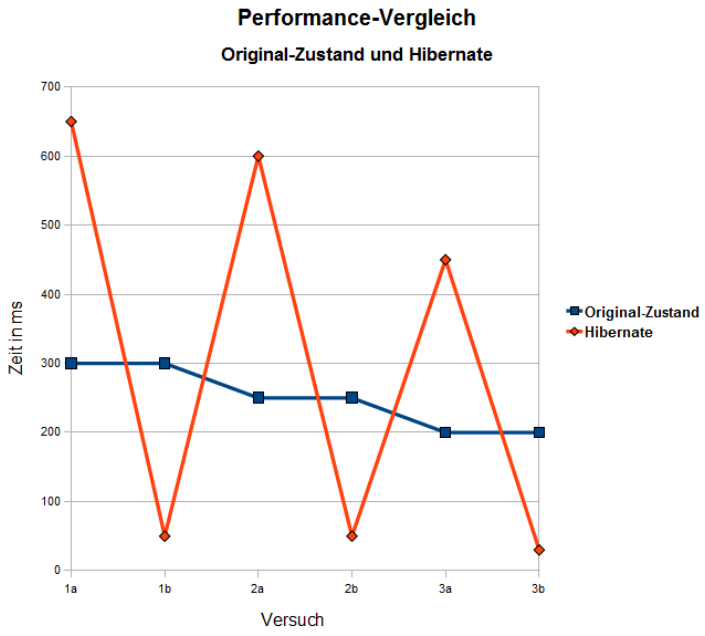

Abbildung 7.2: Performance-Vergleich Original-Zustand und Hibernate

Mit diesen Daten und der Graphik lässt sich erkennen, dass die erste Abfrage länger braucht als im original Zustand. Das ist Hibernate als Zwischenschicht geschuldet, da die, aus der Datenbank geholten Daten in den Cache gespeichert werden müssen. Ab der zweiten Abfrage werden die benötigten Daten aus dem Cache geholt und damit die benötigte Zeit deutlich verkürzt. Hingegen bei dem Ursprungszustand dauern die Abfragen im Durchschnitt gleich lang.

8 Weiterführende Maßnahmen

8.1 Caching

Mit der jetzigen Caching-Strategie werden nur die Abfragen gecached. Diese Daten werden nach 10 Minuten wieder gelöscht. Es wäre denkbar, die gecachten Daten unendlich zwischenzuspeichern und mit selbstgeschriebenen Funktionen die Gültigkeit der Daten anhand eines speziellen Attributes zu prüfen und gegebenenfalls nur verfallene Datensätze zu ersetzen.

8.2 Implementierung

Bei der jetzigen Implementierung werden bei jeder Abfrage eine einzelne Transaktion gestartet. Dies sollte geändert werden, da das Aufbauen, zu vieler Transaktion, zu Performanceverlust führen kann.

9 Fazit

Hibernate bietet mit seinen Möglichkeiten des objektrelationalen Mappings viele Vorteile. Vor allem im Bezug auf Datenbankabstraktion, welche im DELECO®-Umfeld, durch seine Unterstützung verschiedener Datenbanken, wichtig ist. Außerdem werden, durch diverse Caching-Funktionen, die Abfragen performant und mit geringer Belastung der Datenbank ausgeführt. Bei einer Neuentwicklung ist in diesem Umfeld Hibernate ohne Einschränkungen zu empfehlen. Bei einer Migration in ein bestehendes System ist der Auffand nicht zu vernachlässigen, da der komplette vorher implementierte SQL-Code mindestens angepasst, wenn nicht komplett ausgetauscht werden muss. Dieser Nachteil kann mitunter bei Zehntausenden Codezeilen einen enormen Zeitaufwand bedeuten.

Literaturverzeichnis

[King07] Gavin King, Christian Bauer:
Java Persistence mit Hibernate.
Carl Hanser Fachbuchverlag, 2007. - ISBN 978-3-446-40941-5

[King09] Pierre Henri Kuatè, Tobin Harris, Christion Bauer, Gavin King:
NHibernate in Action.
Manning Publications Co., 2009.
ISBN 978-1932394924

[Röder10] Daniel Röder:
JPA mit Hibernate: Java Persistence API in der Praxis.
Entwickler.Press 2010.
ISBN 978-3-868-02014-4

[King10] Gavin King, Christian Bauer, Max Rydahl Andersen, Emmanuel Bernard, Steve Ebersole:
Hibernate Reference Documentation 3.5.1-Final,
URL: <http://docs.jboss.org/hibernate/stable/core/reference/en/pdf/hibernate_reference.pdf>
, zuletzt abgerufen am 11.08.2010.

[Ullenboom09] Christian Ullenboom:
Java ist auch eine Insel, 8.Auflage.
Galileo Computing 2009.
ISBN 978-3-8362-1371-4

[Kühnel08] Andreas Kühnel:
Visual C# 2008 - Das umfassende Handbuch, 4. Auflage.
Galileo Computing 2008.
ISBN 978-3-8362-1172-7

[DELECO] *Übersicht der Module und Funktionen des DELECO® ERP* ,
URL: <http://www.debas.de/erp.html>
, zuletzt abgerufen am 07.09.2010.

Anhang A: SQLBase-Dialekt für Hibernate

```java
package dialect;

import org.hibernate.dialect.Dialect;
import java.sql.Types;
import org.hibernate.Hibernate;
import org.hibernate.dialect.function.SQLFunctionTemplate;
import org.hibernate.dialect.function.StandardSQLFunction;

/**
 * Hibernate-Dialect SQLBase
 *
 * @author storm
 */
public class SQLBaseDialect extends Dialect {

    public SQLBaseDialect() {

        registerColumnType(Types.BOOLEAN, "SMALLINT");
        registerColumnType(Types.SMALLINT, "SMALLINT");
        registerColumnType(Types.INTEGER, "INTEGER");
        registerColumnType(Types.DECIMAL, "DOUBLE(15,4)");
        registerColumnType(Types.DOUBLE, "DOUBLE(15,4)");
        registerColumnType(Types.VARCHAR, 255, "VARCHAR($1)");

        registerFunction("substring",
                new SQLFunctionTemplate(Hibernate.STRING, "@mid(?1, ?2, ?3)"));
        registerFunction("trim",
                new SQLFunctionTemplate(Hibernate.STRING, "@trim(?1)"));
        registerFunction("length",
                new SQLFunctionTemplate(Hibernate.INTEGER, "@length(?1)"));
        registerFunction("coalesce",
                new StandardSQLFunction("@coalesce(?1,?2)"));
        registerFunction("abs",
                new SQLFunctionTemplate(Hibernate.DOUBLE, "@abs(?1)"));
        registerFunction("mod",
                new SQLFunctionTemplate(Hibernate.INTEGER, "@mod(?1)"));
        registerFunction("sqrt",
                new SQLFunctionTemplate(Hibernate.DOUBLE, "@sqrt(?1)"));
        registerFunction("upper",
                new SQLFunctionTemplate(Hibernate.STRING, "@upper(?1)"));
        registerFunction("lower",
                new SQLFunctionTemplate(Hibernate.STRING, "@lower(?1)"));
        registerFunction("round",
                new SQLFunctionTemplate(Hibernate.DOUBLE, "@round(?1,?2)"));
        registerFunction("if",
                new SQLFunctionTemplate(Hibernate.DOUBLE, "@if(?1,?2,?3)"));
        registerFunction("msoDate",
                new SQLFunctionTemplate(Hibernate.DATE,
                "@datetochar(?1,'yyyyMMdd_hhmiss")));
    }
}
```

Listing A.1: SQLBase-Dialekt für Hibernate

Anhang B: Hibernate-Listener

```java
package com.deltabarth.deleco.portlet;

import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;
import org.hibernate.stat.Statistics;

/**
*   HibernateListener zum erzeugen und
*   beenden der SessionFactory beim start
*   und ende des Servletcontext
*
*/
public class HibernateListener implements ServletContextListener {

    /** Wenn ServletContext initialisiert wird,
    *   SessionFactory aufbauen und an den Context binden
    *
    *   @param ServletContextEvent
    */
    public void contextInitialized(ServletContextEvent sce) {
        // neue SessionFactory erzeugen
        SessionFactory sessionFactory =
            (new Configuration()).configure().buildSessionFactory();

        // Statistiken der SessionFactory erlauben
        Statistics stats = sessionFactory.getStatistics();
        stats.setStatisticsEnabled(true);

        // SessionFactory und Statistics an Context binden
        sce.getServletContext()
           .setAttribute("sessionFactory", sessionFactory);
        sce.getServletContext()
           .setAttribute("statistics", stats);
    }

    /** Wenn ServletContext beendet wird,
    *   SessionFactory beenden
    *
    *   @param ServletContextEvent
    */
    public void contextDestroyed(ServletContextEvent sce) {
        // SessionFactory aus dem Context beenden
        ((SessionFactory) sce
            .getServletContext()
            .getAttribute("sessionFactory")).close();
```

Listing B.1: Hibernate-Listener

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Stellen, die wörtlich oder sinngemäß aus Quellen entnommen wurden, sind als solche kenntlich gemacht.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt.

Mittweida, 10. 09 2010