Knowledge Environment Engineering

Datenanbindung über Hibernate in der E-Learning Open-Source-Plattform OLAT

Seminararbeit im Fach Dipl. Informatik

vorgelegt
von
Jia Xie
Matrikelnummer: 9213676

Angefertigt an
der Forschungsgruppe
Betriebliche Informationssysteme
Prof. Dr. Hans-Gert Gräbe

Version 2.1.7
05.02.2006

Inhaltsverzeichnis

1 LMS und OLAT

In diesem Kapitel werden die grundsätzliche Konzepte von Knowledge Environment Engineering und Learning Management System kurz eingeleitet. Als das Zielprojekt OLAT wird ein Überblick zu der Projektsgeschichte, der technischen Umgebung, dem Softwareaufbau, den technischen Schichten und den wichtigen Techinologien erworfen.

Die Informationen in diesem Kapitel stammen aus der Hibernate-Dokumentation [*OLAT 2006*].

1.1 Einleitung

1.1.1 Knowledge Environment Engineering, das Konzept

Bekanntermaßen beschäftigt sich *Knowledge Environment Engineering* als Teildisziplin der Softwaretechnik mit der Erstellung von Regelwerken, die u. a. die Fertigung von Learning Managmement Anwendungen systematisieren. Am Beispiel von *OLAT* möchte ich hier die verwendeten Technologien und Designansätze beschreiben, um daraus allgemeine Schlüsse für das Design entsprechender Anwendungen zu derivieren.

1.1.2 Learning management system, das Konzept

Beim *Learning Management System (LMS)* steht das Managen von Lernen und Lehren im Zentrum.

Eine eindeutige Definition des Begriffs *LMS* existiert nicht, der ungefähre Funktionsumfang lässt sich wie folgt beschreiben: Grundsätzlich ist jedes *LMS* eine Hülle, die Ihnen Funktionalitäten zur Verfügung stellt, damit Sie Ihr didaktisches Konzept einer Veranstaltung umsetzen, Ihren Lerninhalt bereitstellen und Ihre Teilnehmer verwalten können.

Ein *LMS* ist somit ein Werkzeug, das Sie im *"Blended Learning"* sowie bei virtuellen Veranstaltungen unterstützt. Dabei steht meistens die Administration und Betreuung der Teilnehmerinnen im Mittelpunkt. Dazu stellen Ihnen die jeweiligen *LMS* integrierte Funktionalitäten wie beispielsweise Zugangsregelungsmechanismen,

Kurs-Editoren, Teilnehmer- resp. Gruppenverwaltungen, Bewertungswerkzeuge, kollaborative Tools (wie *Foren*, *Chat*, *TWIKI*, *E-Mail* usw.), Lernkontrollumgebungen für Tests und Selbsttests sowie Lernmaterialdarbietungsmöglichkeiten zur Verfügung. Diese Funktionalitäten sind in den verschiedenen *LMS* unterschiedlich stark ausgebaut. Die meisten heutigen *LMS* sind Webapplikationen.

1.2 Die Geschichte des OLAT Projekts

"Online Learning And Training" kurz: *OLAT*, ist ein webbasiertes Lernmanagementsystem, das von den Informatik-Absolventen Franziska Schneider, Sabina Jeger und Florian Gnägi in Zusammenarbeit mit Professor Dr. Helmut Schauer an der Uni Zürich ins Leben gerufen wurde. *OLAT*, als nicht-kommerzielle Software, steht unter der *Apache*-Style Lizenz und kann somit für jeden Zweck gebraucht, verändert und vertrieben werden. Projektstart zur Umsetzung des Lernmanagementsystems war im Frühjahr 1999. Nach einer kurzen Entwicklungsphase konnte *OLAT* erstmalig im Wintersemester 1999/2000 erfolgreich eingesetzt werden. Seit dem wird an einer ständigen Weiterentwicklung, Verbesserung und Ergänzung gearbeitet. Ziel ist es, Dozenten und Studierenden eine plattformunabhängige und leicht zu benutzende Lernumgebung zur Verfügung zu stellen, welche sie beim Lernen und Lehren unterstützt.

Bei der Entwicklung des Systems wurde und wird besonders großer Wert darauf gelegt, offene Standards zu nutzen und dem Open-Source-Gedanken zu folgen. Anfänglich wurde die Skriptsprache *PHP* in Verbindung mit einer *MySQL-Datenbank* eingesetzt. *PHP* zeigte jedoch Schwächen mit steigender Komplexität des *OLAT*-Systems. Die Erweiterbarkeit der *PHP*-Skripte gestaltete sich problematisch und mit wachsender Zahl gleichzeitiger Benutzerzugriffe ergaben sich Schwierigkeiten in der Skalierbarkeit. Die Umstellung auf ein Java-basiertes System löste das Problem. Die Weiterentwicklung geriet zwar zunächst ins Stocken und bedeutete einen Rückschritt in der Funktionalität, die den *OLAT*-Nutzern angeboten werden sollte, doch es

wurde schnell klar, dass dies ein lohnenswerter Schritt war.

OLAT ist seit 2004 an mehreren Schweizer Hochschulen erfolgreich im Einsatz und trägt seinen Teil dazu bei, Studierenden und Lehrenden mittels einer Kommunikations- und Lernplattform eine Unterstützung zu bieten.

1.3 OLAT 5.x Übersicht

1.3.1 Software Umgebung

- *Java SDK* 1.5.x oder neuer
- *Tomcat* 5.x oder neuer
- Datenbank mit *UTF-8* unterstützt
- *Ant* 1.5 oder neuer (zur Konfiguration)

1.3.2 Architektur und Erweiterbarkeit

Das LMS OLAT wird unter Verwendung moderne Technologien und Hilfsmittel entwickelt und ist in der Programmiersprache Java geschrieben. Dies ermöglicht einen Betrieb unter verschiedenen Betriebssystemen wie *Windows*, *MacOSX*, *Linux*, *BSD*, *Unix* oder *Solaris* ohne Anpassungen vornehmen zu müssen. Zur persistenten Speicherung der Daten können ebenfalls verschiedene Datenbankmanagementsysteme wie zum Beispiel *MySQL*, *Postgres* oder *Oracle* zum Einsatz kommen.

OLAT setzt auf der *Java 2 Enterprise Edition (J2EE)* auf und verwendet eine Servlet basierte Architektur. Ein eigens entwickeltes

Model-View-Controller (MVC) Framework erlaubt eine moderne, fehlerreduzierte und schnelle Entwicklung mit strikter Trennung zwischen Darstellungslogik, Ablauflogik, Businesslogik und der Datenhaltung. Um eine möglichst hohe Wartbarkeit und Erweiterbarkeit zu erzielen wird bei der Entwicklung stets ein spezielles Augenmerk auf eine saubere Entkoppelung und eine hohe Wiederverwendbarkeit der verschiedenen Softwarekomponenten gelegt.

Verschiedene sogenannte *"Extension-Points"* erlauben das *LMS* nach kundenspezifischen Bedürfnissen zu erweitern, ohne dabei Änderungen am Grundsystem vornehmen zu können. Dies erleichtert die Installation von Updates und schützt somit getätigte Investitionen. Beispiele für solche Erweiterungen wären ein zusätzlicher Navigationspunkt in der Hauptnavigation oder ein neuer Kursbaustein. Die Erweiterungen können Layout Elemente und Übersetzungen mit sich bringen oder auf bestehende zurückgreifen.

1.3.3 Programmierungskonzept

Swing like - Komponenten basiert (MVC),
Business task (Controllers),
Business logic (Managers),

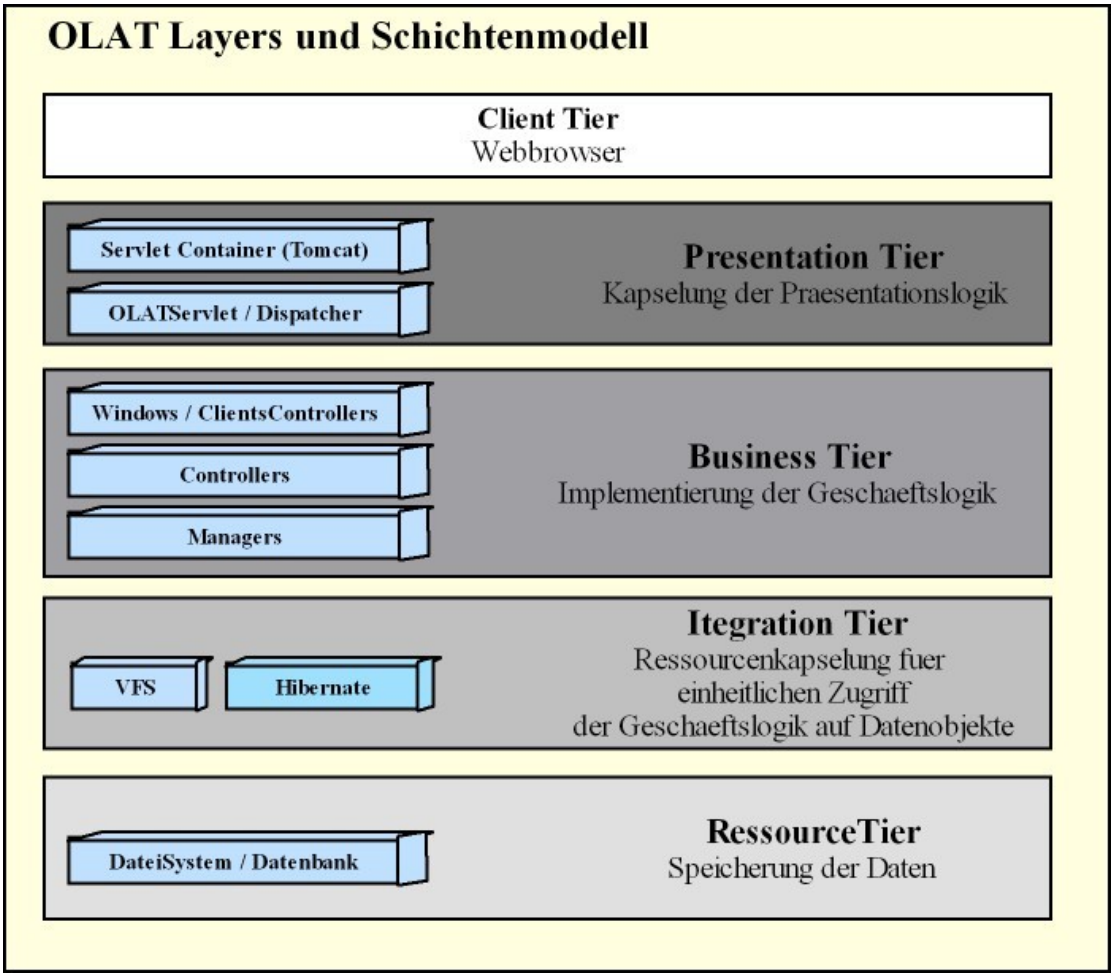
Verwendung der Opensource Libraries

- *hibernate* für Objekt/relationale Datenbank mapping
- *velocity* für Layout Prozess
- *spring* für Konfigurationsaufgaben

1.4 Software Architektur und Applikationsschichten des OLAT-Systems

1.4.1 Schichtenmodell, ein Übersicht

Das über *J2EE* spezifizierte Schichtenmodell vom internen Aufbau internetbasierter Anwendungen wird auch innerhalb des *OLAT*-Systems umgesetzt. Jede Schicht ist in sich geschlossen und nur über definierte Schnittstellen ansprechbar. Dadurch wird jede Schicht austauschbar und die Abhängigkeiten zwischen den Schichten werden minimiert, was zu einem entkoppelten System führt.



1.4.2 Aufgaben der OLAT Layers

Alle UserRequests, die durch den *Browser* geliefert werden, werden durch das *OLATServlet* geleitet. Dieses selbst verwendet *Dispatcher* um die *Request* entsprechend weiterzuleiten. Dies passiert innerhalb der *DispatcherAction* Klasse.

Jeder Benutzer wird durch eine *UserSession* repräsentiert, in der alle "*Windows*" eines Benutzers gespeichert werden. *Windows* sind innerhalb *OLAT* serverseitige Abbildungen der *Browser* Fenster des Users. Sie werden durch *ChiefController* gesteuert, die die *Navigation,* jedoch keine Inhalte, verwalten.

Allgemein sind *Controller* verantwortlich für Weiterleitung und Teile der *Business Logik*. Zu jedem gehört ein Objekt, welches die grafische Repräsentation der *Controllerlogik* zu jeder Zeit sichert, diese Abbildung wird später zu *HTML* Fragmenten verarbeitet.

Manager hingegen sind hier Klassen, die allgemeingültige Services ohne zugehörige grafische Repräsentation anbieten.

Der *VFS* wiederum ist eine Baumstruktur die einem Filesystem sehr ähnlich ist. Hier werden je nach Aufgabe verschiedene Services genutzt.

OLAT nutzt weiterhin das Hibernate Framework um Objekte aus *relationalen DB*'s zu gewinnen und diese wieder zu speichern. Besonders wird hierbei die *relationale Persistenz* der Daten gesichert.

1.4.3 Erläuterung zum Schichtenmodell

Den Sockel jeder Webapplikation nach *J2EE* bildet das *"Resource Tier"*, in dem alle zu speichernden Daten abgelegt werden. Das *"Integration Tier"* kapselt diese Ressourcen und sorgt dafür das die Geschäftslogik im *"Business Tier"* über eine einheitliche Schnittstelle

darauf zugreifen kann. Das *"Presentation Tier"* ist dafür zuständig, die von der Geschäftslogik übermittelten Inhalte in ein Format zu überführen, das der Client interpretieren und darstellen kann.

Das *OLAT*-System lässt sich unter Berücksichtigung dieses Schichtenmodells in sechs Ebenen gliedern. Die unterste Ebene bildet das Datenbankmanagementsystem. Auf dieses setzt das *Hibernate-Persistenz-Framework* auf, welches als Integrationsschicht zwischen Datenbank und Applikation fungiert. Eine zusätzliche Datenbank-Zugriffs-Ebene dient als Schnittstelle zwischen der eigentlichen Geschäftslogik im engeren Sinn und den persistenten Daten. Die Applikationslogik ist im Wesentlichen für die Umsetzung der Funktionalität verantwortlich. Die darauf aufbauende Ebene der Präsentationslogik bereitet mittels *Velocity* die Daten für den Client so auf, dass dieser innerhalb der Schicht der Präsentation die Daten lediglich darzustellen braucht. Jegliche Logik ist dem Client entzogen und orientiert sich somit am *J2EE*-Schichtenmodell.

1.4.4 Hibernate und Velocity

Für die Entkoppelung der einzelnen Systemschichten bedient sich OLAT einiger Standards und Frameworks. So dient in der Integrationsschicht das *Hibernate-Persistenz-Framework* der Speicherung von Objekten in relationalen Datenbanken und ermöglicht gleichzeitig die Austauschbarkeit des *DBMS*. In der Präsentationsschicht erfolgt über das *Velocity*-Konzept die Umsetzung des *Model-View-Controller-Prinzips*, welches eine strikte Trennung zwischen Präsentations- und Applikationslogik vorsieht.

In nächsten 2 Kapiteln werden die beide Technologien *Hibernate* vom Konzept zu praktischer Verwendung im *OLAT*-System weiter analysiert.

2 Persistents Frameworks

In diesem Kapitel wird das Konzept Object/Relational Mapping als Implementierung der Persistenzschicht vom Datenbank-basiert-system, das als Software aus Komponenten entworfen wurde, mit *Hibernate* als Beispiel diskutiert.

Hibernate ist ein Open-Source-Projekt und gehört zur Jboss Professional Open-Source Product Suite. Es ist ein an die *ODMG*-Spezifikation angelehntes, objektrelationales Mapping-Werkzeug.

Die Informationen in diesem Kapitel stammen aus der Hibernate-Dokumentation [*XiaXin 2005*].

2.1 Persistente Objekte und die Persistenzschicht

Um sich über die Funktionsweise von *Hibernate* Auskunft zu geben, ist es unerläßlich, einige Begriffe zu klären. Zum Ersten ist das das persistente Objekt. Persistente Objekte (und persistente Kollektionen) sind Objekte, die persistente Daten enthalten und damit eine Repräsentation im Externspeicher besitzen. In *Hibernate* ist jedes persistente Objekt, das unter Transaktionskontrolle bearbeitet wird, mit genau einer *Session* assoziiert. [3 *Hibernate Sessions und Transactions*]
Dies wird in nächstem Kapitel näher betrachtet.

2.2 Die Typen der Implementierung der Persistenzschicht

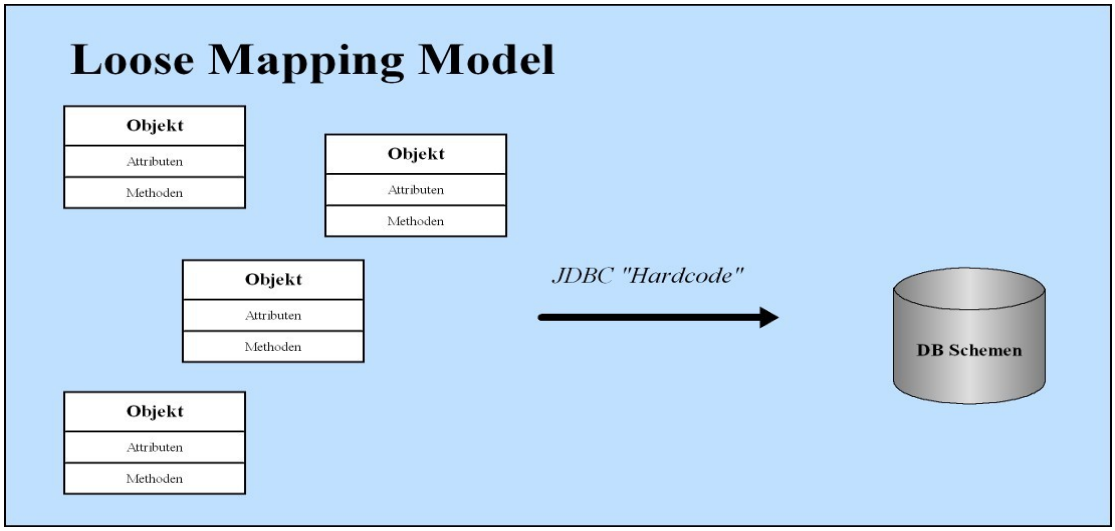
Im Allgemeinen werden drei Typen verwendet, um die Persistenzschicht zu realisieren.

2.2.1 Loose Mapping Model

Das *Loose-Mapping-Modell* ist die originäre Realisierung der Persistenzschicht. Der *JDBC*-Zugreiffscode, der die Persistent-funktionalitäten anbietet, wird je nach Bedürfnis innerhalb der Geschäftsklassen kombiniert.

```
public boolean reName(String id,String name){
    Connection conn=null;
    Statement statement=null;
    try {
        Class.forName("oracle.jdbc.driver.OracleDriver");
        conn = DriverManager.getConnection(
            "jdbc:oracle:thin:@dbserver:1521:forum",
            "MyUserName",
            "MyPassword"
            );
        String sqlStr =
            "Update user set name = '"
            + name
            + "' where user_id='" + id+"'";
            statement = conn.createStatement();
            return statement.execute(sqlStr);
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    } catch (SQLException e) {
        e.printStackTrace();
    }finally{
    if (conn!=null){
            try {
                if (statement!=null){
                    try {
                        statement.close();
                    } catch (SQLException e){
                        e.printStackTrace();
                    }
                }
            conn.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
    return false;
}
```

Wie der Beispielscode daoben zeigt, wird die auf-*JDBC*-Code-aufbauende Realisierung der Persistenzschicht direkt in einer Applikationsklasse hinzugefügt. Das dadurch erreichte Objekt/Relational Mappingsmodell ist dann ein so genanntes Loose-Mapping-Model.

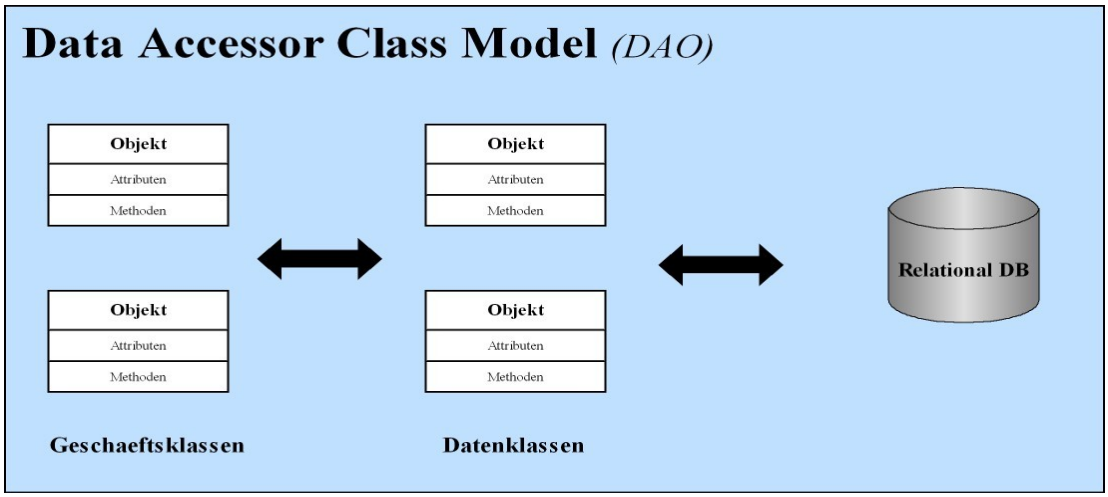


Vorteile dieses *Loose-Mapping-Models* sind Schnelligkeit, Einfachheit und Flexibilität. Für bereits erstellte DB Systeme und kleine Anwendungen ist das deutlich sinnvoll. Dagegen werden schlechte Erhaltbarkeit und Erweiterbarkeit eingebracht. Die Änderungen von Objektattributen und Datenbankstrukturen können nur mit hohem Aufwand realisiert werden.

Die logische Strukturen der Persistenzschicht wird in diesem Modell nicht deutlich getrennt.

2.2.2 Auf *Data Class* basiertes Modell der Persistenzschichtsrealisierung

In diesem Modell spielt *Data Class* eine Rolle als Brücke zwischen Geschäftsklassen und Persistenzschicht. Ein typisches Beispiel der Realisierung dieses Modells ist *DAO*. *Data Class* enthält hier *Domain Class/Object* und *Data Accessor Class* in *DAO*. Als die Abstraktion der realen Welt ist *Domain Class* in diesem Modell verantwortlich die Informationen zu tragen, um die Verbindung für Datenaustausch und -anpassung zu erstellen. Dazu dient *Data Accessor Class* um durch *JDBC*-Code eine direkte angepasste Verbindung von *Domain Class* zu den Datenbanktabellen zu generieren.



In diesem Modell ist die Trennung von Geschäftslogik und der in der Grundstufe liegenden Datenbankstruktur realisiert. Als eine relativ unbefestigte logische Schicht, verdeutlicht *Data Class* Das Hauptkonzept der Persistenzschicht. Die Änderungen in der Relational-DB-Schicht werden bis zur Data-Class-Schicht gekapselt, um den Einfluss auf obere Geschäftslogik möglichst zu vermeiden.

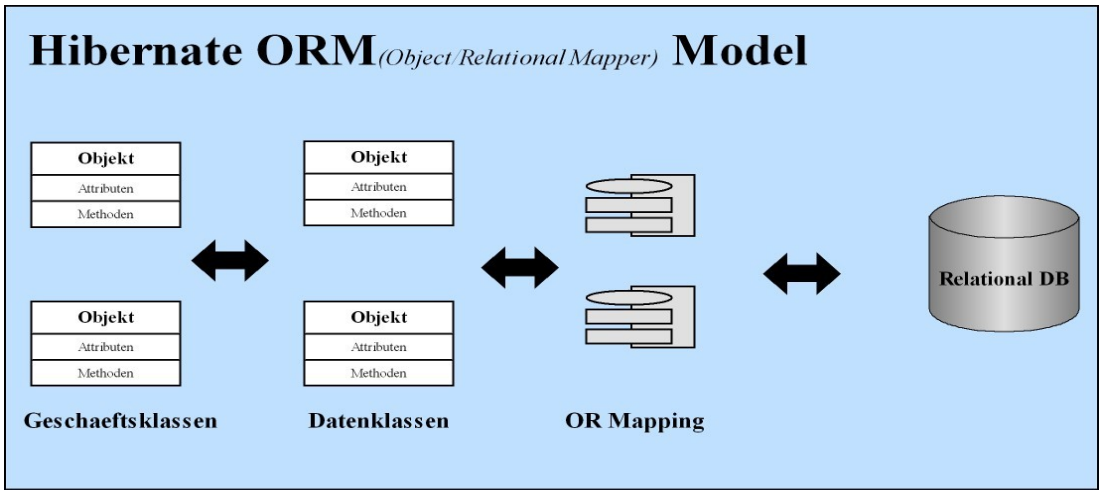
Beim Aufstieg der Anzahl dafür erworfener Schichten vergrößert sich gleichzeitig die Codemenge unglaublig schnell. Verglichen mit dem ersten Modell, bei der Verbesserung der Systemstruktur ist die Nebenläufigkeit auch klar:

Das Kosten der Entwicklung eines Projektes steigen.

2.2.3 Die auf existierenden Persistent-Framworks aufbauende Realisierung

Dieses Modell ist eingentlich eine Erweiterung des zweiten Modells. Die Anzahl der Data Accessors und Domain Classes von Data Class hat sich nicht reduziert. Die zeitaufwändigste Arbeit – auf *JDBC* basierte OR Mapping, ist schon von Third Party Entwicklern erfolgreich vorbreitet.

Die langwierige Codierungsarbeit im *Data Accessor* ist bemerkswert vereinfacht. Mit den Hilfswerkzeugen für Third Party Persistent-Frameworks sind Entwickler vom Druck der *Domain Class* Codierung fast befreit.



2.3 Vorteil vom Hibernate

Ein erfolgreiches Persistent-Framework wie Hibernate bietet:

- **Abnehmen langwieriger Codierung**

Die technischen Prozessen wie Angelegenheitsmanagement, Datenbankverbindung, SQL Generierung sind vom Persistent-Framework gekapselt. Entwicklern können mehr wichtigere Themen berücksichtigen.

- intensives Objektorientiertes Design

ORM bietet dem System natürlichere Realisierungsmethoden an und kann die Domain Objekte automatisch in verschiedene Datenbanktabellen abbilden. Zu beachten sind nur die Eingenschaften des Objektes, aber nicht mehr die Column Daten in JDBC ResultSet.

- bessere Leistung

Die Verbesserung der Leistung wird von Database Connection Pool, PreparedStatement Cache und Data Cache realisieren.

- bessere Portabilität

Unabhängigkeit von Betriebssystemen, starke Unterstützung für unterschiedliche Datenbanken (vollständige Kapselung), einfache Parametereinrichtung für Datenbankenwechsel.

3 Hibernate

In diesem Kapitel wird die Architektur und grundsätzliche Prinzipien von Hibernate weiter diskutiert.

Untersucht wird hier beispielsweise die Verwendung der Hibernate Technologie im OLAT Projekt, und danach verbreitet, die Version 3.2.0. Die Informationen in diesem Kapitel stammen aus der Hibernate-Dokumentation [*Hibernate 2006*] und [*XiaXin 2005*].

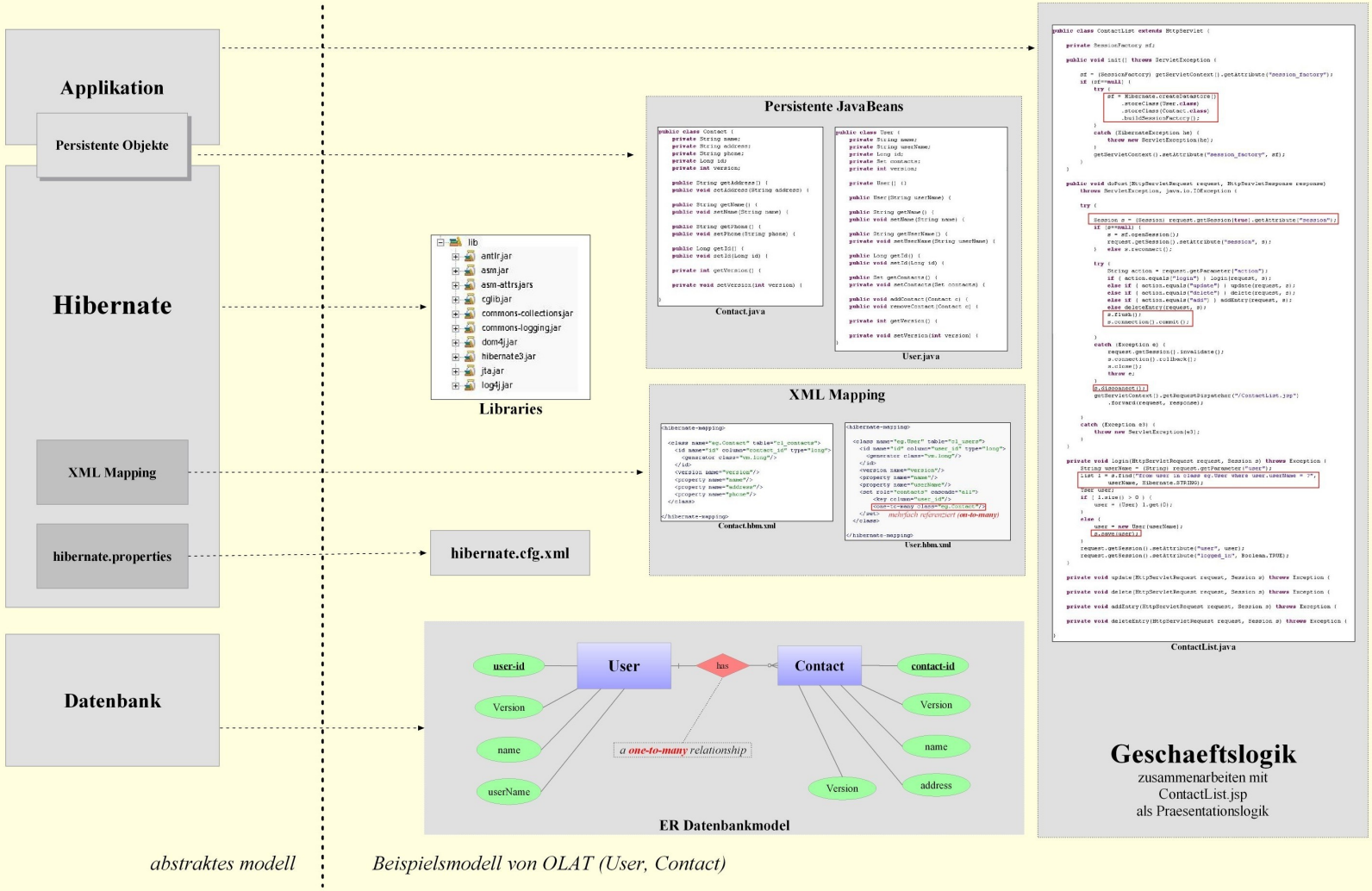
3.1 Hibernate Architektur

Um seine Funktionsweise vollständig entfalten zu können, benötigt das Framework folgende Komponenten:

- die Hibernate Java-Library
- eine Hibernate-Konfigurationsdatei (*hibernate.properties* bzw. *hibernate.cfg.xml*)
- Hibernate Query Language (*HQL*) für alle Datenbankabfragen
- die zu speichernden Java-Objekte (persistente Klassen)
- XML-Mapping-Dateien für die persistenten Klassen
- einer Datenbank samt Datenbankschema

Sehe auch die vollständige Abbildung in nächster Seit.

Hibernate Architektur *(Abstraktes Uebersicht und Abbildung zum Beispiel)*



Das Hibernate-Framework ermöglicht es, Objekte in einer relationalen Datenbank zu speichern und umgekehrt aus Datensätzen einer relationalen Datenbank Objekte zu erzeugen. Die Datenbank ist dabei beliebig wählbar und austauschbar, sofern ein *JDBC*-Treiber dafür existiert. Hibernate dient somit der Abstraktion der Datenbasis und steht als Mittlerschicht zwischen der eigentlichen Applikation und der Datenbank.

Die Konfiguration des Hibernate-Persistenz-Frameworks erfolgt textbasiert über die Datei *hibernate.properties*. Alternativ dazu kann dies auch mittels *XML* über hibernate.cfg.xml gelöst werden. Die *XML*-Mapping-Dateien bilden die persistenten Klassen auf Datenbankrelationen ab. Im besten Falle wird jede persistente Klasse durch genau eine Datenbankrelation repräsentiert, wobei die Membervariablen der Klasse die Attribute der Relation bilden. *OLAT* hinterlegt die Mapping-Dateien stets im gleichen Verzeichnis wie die persistente Klasse und benennt jeweils beide Dateien gleich. Eine Zuordnung von persistenter Klasse zur Mapping-Datei ist somit schneller möglich. Datenbankabfragen werden über die Hibernate-eigene Sprache *HQL* realisiert. Diese ist in ihrer Syntax sehr ähnlich zu *SQL*, im Unterschied dazu aber folgt *HQL* dem objektorientierten Paradigma und beherrscht Vererbung, Polymorphie und Assoziation.

Für die Realisierung der Datenbankzugriffe nutzt Hibernate eigene Sessions. Diese werden von der Klasse *SessionFactory* vergeben und verwaltet. Die *SessionFactory* ist threadsicher und von allen Threads in der Applikation nutzbar. Sessions hingegen sind nicht threadsicher und sollten deshalb nur einmal von einem "*Businessprocess*" benutzt und danach zerstört werden. Um Kollisionen zu vermeiden, sollte eine Session ebenso nie von zwei konkurrierenden Threads nutzbar sein. Innerhalb einer Session können mehrere Transaktionen durchgeführt werden. Die nachfolgende Übersicht zeigt beispielhaft einige grundlegende, über das Hibernate-Framework implementierte Funktionalitäten und Methoden.

Um mit dem Hibernate-Framework arbeiten zu können, sind eine Reihe von Schritten zu realisieren, welche zunächst allgemein erläutert und danach anhand der Umsetzung innerhalb von *OLAT* veranschaulicht werden sollen. Aus Sicht der Applikationsschicht sollten in einem ersten Schritt die persistenten Klassen angelegt werden. Darauf aufbauend sind die Mapping-Dateien zu erzeugen, anhand derer dann das Datenbankschema zu erstellen ist. Erst wenn diese Vorbedingungen erfüllt worden sind, lässt sich das Hibernate-Framework über olgende Anweisungen in die Anwendung einbauen:

a) Configuration-Objekt anlegen und alle persistenten Klassen übergeben
b) SessionFactory erzeugen

```
sf = (SessionFactory) getServletContext().getAttribute("session_factory");
if (sf==null) {
    try {
        sf = Hibernate.createDatastore()
            .storeClass(User.class)
            .storeClass(Contact.class)
            .buildSessionFactory();
    }
    catch (HibernateException he) {
        throw new ServletException(he);
    }
    getServletContext().setAttribute("session_factory", sf);
}
```

c) Session öffnen
d) Transaktion beginnen

```
Session s = (Session) request.getSession(true).getAttribute("session");
if (s==null) {
    s = sf.openSession();
    request.getSession().setAttribute("session", s);
}   else s.reconnect();
```

e) Hibernate-Methoden ansprechen, evtl. dafür notwendige HQL erzeugen

```
try {
    String action = request.getParameter("action");
    if ( action.equals("login") ) login(request, s);
    else if ( action.equals("update") ) update(request, s);
    else if ( action.equals("delete") ) delete(request, s);
    else if ( action.equals("add") ) addEntry(request, s);
    else deleteEntry(request, s);
    s.flush();
    s.connection().commit();

}
catch (Exception e) {
    request.getSession().invalidate();
    s.connection().rollback();
    s.close();
    throw e;
}
```

```
String userName = (String) request.getParameter("user");
List l = s.find("from user in class eg.User where user.userName = ?",
        userName, Hibernate.STRING);
User user;
if ( l.size() > 0 ) {
    user = (User) l.get(0);
}
else {
    user = new User(userName);
    s.save(user);
}
request.getSession().setAttribute("user", user);
request.getSession().setAttribute("logged_in", Boolean.TRUE);
```

f) Transaktion beenden
g) Session schließen
h) SessionFactory schließen

3.2 Datenbankkonstrukte

In Hibernate werden persistente Objekte mit einer sog. Session assoziiert. Jedes persistente Objekt kann dabei höchstens mit einer Session assoziiert werden. Ein Datensatz der Datenbank kann aber in mehreren Objekten und damit in mehreren Sessions bestehen. (*DB-ID* ungleich *JVM-ID*).

Jede Session besitzt einen eigenen lokalen Cache. Dieser unterstützt das Navigieren über den Objektgraphen und das Auffinden von Objekten via *OID*. Sie stellt ausserdem eine Factory für Transaktionen bereit und kann mehrere Transaktionen umfassen.

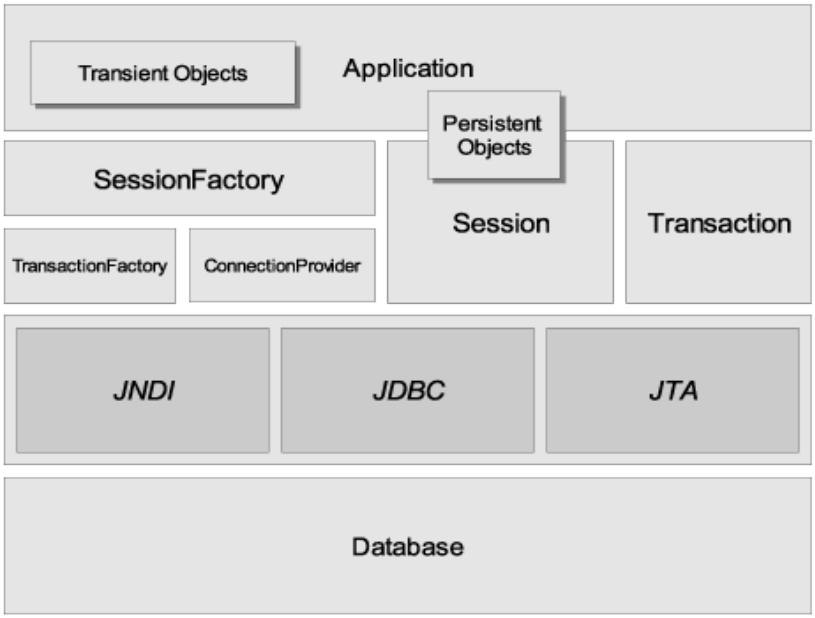
Des Weiteren wird von der Session eine Datenbankverbindung bereitgestellt und gleichzeitig verhüllt. Der Entwickler muss nicht explizit eine Verbindung aufbauen. Beginnt eine Transaktion, in der ein Zugriff auf persistente Daten erfolgt (z.B. Elemente lesen, ändern), wird automatisch die konfigurierte Verbindung zur Datenbank verwendet.

Die Klasse Session stellt Methoden zum Speichern, Ändern und Löschen von Daten
in der Datenbank bereit (*session.save(pO)*, *session.update(pO)*, *session.saveOrUpdate(pO)*, *session.delete(pO)* …)

Session-Objekte werden mit Hilfe der Klasse SessionFactory erzeugt. Auf der Ebene der SessionFactory kann zusätzlich zum lokalen (First Level) Cache ein Second Level Cache eingesetzt werden, der Objekte über Sessiongrenzen hinweg speichert (Prozess- oder Cluster-Level).

Die Transaktions-API definiert Methoden für *trx.begin()*, *trx.commit()*, und *trx.rollback()*.

Bei trx.commit() werden alle Änderungen in die Datenbank geschrieben. Die Verwendung der Transaktions-API ist optional. Hibernate kann auch zusammen mit einem *J2EE*-Server eingesetzt werden, der das sog. Java Transaction API (*JTA*) implementiert. Dann können Transaktionen ausserhalb von Hibernate gesteuert werden.



3.3 Zugriff auf Daten

Der Zugriff auf Daten in einer relationalen Datenbank erfolgt innerhalb von Hibernate-Transaktionen. Hibernate unterstützt den impliziten, navigierenden Zugriff auf die verschiedenen Datensätze. Sie werden bei Bedarf aus der Datenbank nachgeladen.

Beispiel für einen navigierenden Zugriff aus [Hibernate 2006]:

```
// load pk's state into cat
         sess.load( cat, new Long(pkId) );
         Set kittens = cat.getKittens();
```

Der explizite Zugriff auf die Daten im Datenspeicher wird in Hibernate durch eine objektorientierte, an *OQL* angelehnte Anfragesprache, *HQL* (Hibernate Query Language), und durch den direkten Zugriff mittels der *OID* unterstützt. Extents werden nicht definiert.

3.3.1 Expliziter Zugriff über eine OID

Ist die OID eines Objektes bekannt, kann das Objekt über die Methoden session.load()oder session.get() der Klasse Session direkt aus der Datenbank geladen werden. Die Methoden get() und load() unterscheiden sich wie folgt:

- session.get() greift direkt auf die Datenbank zu. Falls das Objekt auf der Datenbank nicht existiert, wird der Wert null zurückgeliefert.
- wird session.load() verwendet, so wird zunächst ein nicht initialisierter Proxy erzeugt. Das Objekt wird erst beim Zugriff auf eine der Methoden tatsächlich aus der Datenbank geladen. load() bietet die Möglichkeit die Daten direkt in eine neue Instanz zu laden oder aber eine bereits erzeugte Instanz zu befüllen.

2 Beispiele hierzu aus der Hibernate-Dokumentation [Hibernate 2006]
Beispiel zu get:

```
Cat cat = (Cat) sess.get(Cat.class, id);
if (cat == null) {
          cat = new Cat();
          sess.save(cat, id);
}
return cat;
```

Beispiel zu load:

```
        Cat fritz = (Cat) sess.load(Cat.class,
                  generatedId);
// you need to wrap primitive identifiers
        long pkId = 1234;
        DomesticCat pk = (DomesticCat) sess.load(
                  Cat.class,
                  new Long(pkId) );
        Cat cat = new DomesticCat();
// load pk's state into cat
        sess.load( cat, new Long(pkId) );
        Set kittens = cat.getKittens();
```

3.3.2 Expliziter Zugriff über eine Anfrage

Ist die OID eines Datensatzes nicht bekannt, so können Objekte über die Anfragesprache HQL (Hibernate Query Language) geladen werden. Hierfür bietet die Klasse Session unter anderem die Methode session.find(). Innerhalb dieser find()-Methode wird ein String nach dem SELECT-FROM-WHERE - Modell von SQL als Argument übergeben.

Ausserdem kann ein Ergebnisresultat definiert werden. Der navigierende Zugriff auf Attribute innerhalb der Anfrage ist erlaubt. Der Aufruf von Methoden innerhalb der Anfrage hingegen wird nicht unterstützt. Geschachtelte Anfragen (*Subqueries*) sind möglich.

HQL ist stark an OQL angelehnt und bietet unter anderem folgende Fähigkeiten:

- Projektionen
- Explizite Joins (inner join, left outer join, right outer join, full join)
- Aggregat-Funktionen (max, min, avg, sum und count)
- In der WHERE–Bedingung werden unter anderem mathematische Operatoren (+, -, *, /), Vergleichsoperatoren (=, >=, <=, <>, !=, like),
  logische Operatoren (and, or, not, in, between, is null)

und noch einige Andere unterstützt.
- order by und group by

Beispiele für Anfragen aus der Hibernate Dokumentation [Hibernate 2006]:

```
List cats = sess.find( "from Cat as cat where cat.birthdate = ?",
                        date, Hibernate.DATE
);
List mates = sess.find(
          "select mate from Cat as cat join cat.mate as mate " +
          "where cat.name = ?",
          name, Hibernate.STRING
);
List cats = sess.find(
          "from Cat as cat where cat.mate.bithdate is null"
);
List moreCats = sess.find(
          "from Cat as cat where " +
          "cat.name = 'Fritz' or cat.id = ? or cat.id = ?",
          new Object[]{id1, id2}, new Type[]{
                    Hibernate.LONG, Hibernate.LONG
          }
);
List mates = sess.find(
          "from Cat as cat where cat.mate = ?",
          izi, Hibernate.entity(Cat.class)
);
List problems = sess.find(
          "from GoldFish as fish " +
          "where fish.birthday > fish.deceased or fish.birthday
is null"
);
```

Neben dieser Form bietet Hibernate noch einige alternative Formen für Anfragen. Für grössere Mengen bei der Rückgabe, von denen nur ein Teil der Elemente genutzt werden soll, gibt es die Methode session.iterate(). In Hibernate können auch native SQL-Anfragen verwendet werden. Ausserdem wird eine API für java- asierte Anfragen angeboten (sog. Criteria Queries). Eine solche Anfrage kann mit Hilfe verschiedener Methoden und vordefinierter Ausdrücke (*Expressions*) zusammengesetzt und ausgeführt werden. Sie ähnelt *JDOQL*.

3.3.3 Polymorphismus bei Anfragen

Hibernate unterstützt Polymorphismus bei Anfragen in zwei Varianten, explizit und implizit. Unter *impliziten Polymorphismus* versteht Hibernate, dass in einer Query auch Instanzen aller Subklassen des gesuchten Elements zurückgeliefert werden. Beispiel: Die Anfrage select pet where … könnte Hunde, Katzen und weitere Haustiere als Ergebnis zurückgeben.

Expliziter Polymorphismus bedeutet, dass nur Instanzen des gesuchten Typs und speziell im Mapping-File definierter Subklassen zurückgeliefert werden. Subklassen können im XML-File bei der Klassendefinition explizit als <*subclass*> oder <*joined-subclass*> angegeben werden. Die Verwendung von implizitem oder explizitem Polymorphismus kann nicht dynamisch aus Java heraus festgelegt werden, sie wird statisch im XML-File definiert.

3.4 Manipulation von Daten

Daten werden mit Hilfe von Java im Code verändert, damit ist Java selbst OML. Datensätze werden hierfür innerhalb einer Transaktion in den Cache geladen, lokal verändert und bei einem erfolgreichen Beenden der Transaktion zurück in die Datenbank geschrieben.

Veränderte Daten können zusätzlich zu jedem Zeitpunkt mit *session.flush()* in die Datenbank propagiert werden. Wird die Transaktion in Hibernate zurückgesetzt und damit auch in der Datenbank abgebrochen, so werden die bereits in der Datenbank gemachten Änderungen von der Transaktionskontrolle der Datenbank zurückgesetzt. Die Änderungen an Objekten im Cache werden von Hibernate zurückgesetzt.

Ein erfolgreiches Beenden einer Transaktion ist nur möglich,

wenn auch die Transaktion in der Datenbank erfolgreich beendet werden kann. Ist dies nicht der Fall, weil es zu Konflikten kommt, gibt es einen Fehler und die Hibernate- Transaktion wird zurückgesetzt.

Im Folgenden wird zwischen einem erfolgreichen Beenden und einem Abbruch einer Transaktion in Hibernate und der Datenbank nicht mehr explizit unterschieden. Änderungen werden immer nur in der Datenbank wirksam, wenn eine Transaktion sowohl in Hibernate als auch in der Datenbank erfolgreich beendet werden kann. Ansonsten werden sie zurückgesetzt.

3.4.1 Insert (new)

Explizit

Objekte können explizit mit Hilfe der Methoden *session.save(o)* oder *session.saveOrUpdate(o)* persistent gespeichert werden. Wird die laufende Transaktion mit einem Commit beendet, werden sie in die Datenbank eingefügt.

(Erweitertes) Beispiel aus [Hibernate 2006]:

```
Cat potentialMate = new Cat();
firstSession.save(potentialMate);
firstSession.flush();
trx.commit();
```

Implizit

Die implizite Speicherung von Objekten in Hibernate ist nur möglich, wenn dies im Mapping-File für die Assoziationen der betroffenen Objekte so definiert wurde (Kaskadieren-Tag) oder wenn Kaskadieren global verwendet wird (wenn im Attribut <hibernate-mapping> default-cascade steht).

Beispiel aus [Hibernate 2006]. Einstellung im Mapping-File für einzelne Objekte:
cascade="all" oder cascade="save-update".

3.4.2 Delete

Explizit

Die Methode Session.delete(pO); löscht ein persistentes Objekt im Datenspeicher und macht es aus Sicht von Hibernate transient. Das Objekt selbst wird also zunächst einmal in Java nicht gelöscht. Anstatt viele einzelne Objekte zu löschen, können mit Hilfe eines Anfrage-Strings auch viele Objekte auf einmal gelöscht werden.

Implizit

Objekte können in Hibernate auch kaskadierend gelöscht werden. Eines der folgenden Attribute muss dafür im Mapping-File angegeben werden:
cascade="all", cascade="all-delete-orphan" oder cascade="delete".

3.4.3 Update

Explizit

Für Änderungen an Objekten, die eine Repräsentation in der Datenbank besitzen, aber von einer Session abgekoppelt wurden, muss die Methode session.update(o) oder aber session.saveOrUpdate(o) verwendet werden. Werden also Objekte optimistisch über Session-Grenzen hinweg verwendet und geändert, so muss ihre Aktualisierung explizit erfolgen. Änderungen von persistenten Daten in Objekten ausserhalb von Sessions werden nicht implizit bei einem Commit gespeichert.

Die Methode *session.flush()* sorgt dafür, dass Änderungen an Datenelementen in Java sofort in die Datenbank eingefügt werden, es erfolgt aber keine Aktualisierung von Änderungen auf der Datenbank im

Objekt, d.h. wurde ein Datensatz von einer anderen Applikation verändert, werden diese Änderungen nicht im Objekt sichtbar.

Implizit

Änderungen an persistenten Objekten werden beim Aufruf von trx.commit() automatisch erkannt und in die Datenbank geschrieben. Beim Aufruf von session.flush() werden sie ebenfalls in der Datenbank gespeichert. Persistente Objekte sind Objekte, die in der aktuellen Session mit Hilfe von session.save() oder session.saveOrUpdate()bereits gespeichert wurden oder die mit Hilfe von *session.load()*, *session.get()*, *session.iterate()* oder *session.filter()* aus der Datenbank geladen wurden. Zusätzlich gibt es die Möglichkeit des kaskadierenden Speicherns um erreichbare (nicht persistente) Objekte implizit in die Datenbank einzufügen oder zu aktualisieren.

Beispiel aus [Hibernate 2006]:

```
          DomesticCat cat = (DomesticCat) sess.load( Cat.class,
                                new Long(69));
          cat.setName("PK");
          sess.flush();
// changes to cat are automatically detected and persisted
```

3.5 Verwaltung von persistenten Instanzen

Änderungen an persistenten Objekten werden beim Aufruf von *trx.commit()* oder *session.flush()* (Zwischenspeichern) automatisch in die Datenbank geschrieben. Je nachdem, welche Einstellungen im Metafile bezüglich Kaskadierung eingestellt sind, werden alle Objekte des Graphs oder nur das konkrete Objekt gelöscht, eingefügt und/oder aktualisiert.

Wie genau Änderungen an Objekten erkannt werden geht aus

der Dokumentation nicht hervor. Nach der [Hibernate 2006] gilt aber, dass Änderungen automatisch erkannt und gespeichert werden. Unklar bleibt, ob nur geänderte Werte aktualisiert werden und wie erkannt wird, dass Werte sich verändert haben. Änderungen, die zwischenzeitlich gespeichert wurden, werden bei einem Rollback der Transaktion auch in der Datenbank (durch deren Transaktionskontrolle) rückgängig gemacht

Hibernate unterstützt sowohl eine pessimistische als auch eine optimistische Konkurrenzsteuerung. Für die Optimistische sind zusätzliche Felder in der Datenbank für Zeitstempel oder Versionsnummern nötig. Werden alle Objekte innerhalb einer Session neu erzeugt, geladen, geändert und/oder gelöscht, so entspricht dies dem pessimistischen Ansatz. In diesem Fall liegt die Konkurrenzsteuerung bei der Datenbank.

Möchte man aber Daten über einen längeren Zeitraum bearbeiten, so ist es möglich die transienten Objekte ausserhalb eines Transaktionskontextes für die Bearbeitung zu verwenden (sie können auch als Transfer-Objekte verwendet werden). Dies nennt sich *detaching*. Mit den Methoden session.update(o); und session.saveOrUpdate(o); können die transienten Objekte wieder einer Session zugeordnet werden (*attach*) und in einer weiteren Transaktion beim Aufruf der Methode trx.commit() gespeichert werden. Hierfür überprüft die

Laufzeitumgebung von Hibernate zunächst mit Hilfe von zusätzlichen Feldern in der Datenbank (Zeitstempel, Versionsnummer), ob ein erfolgreiches Beenden der Transaktion ohne Transaktionskonflikte überhaupt möglich ist. Haben sich die betroffenen Datenelemente in der DB nicht verändert, wird die Transaktion für ein endgültiges Beenden an die Konkurrenzsteuerung der Datenbank übergeben. Das Konzept optimistischer Transaktionen in Hibernate ist ähnlich wie in *JDO* . In Hibernate werden allerdings mehrere Hibernate-Transaktionen für eine semantische Benutzer-Transaktion verwendet.

3.6 Das Caching

Hibernate bietet verschiedene Caches an. Jede Session hat einen eigenen First Level Cache.

Ein optionaler Second Level Cache kann auf Ebene der SessionFactory für eine Prozess- oder Clusterübergreifende Nutzung eingesetzt werden (auch externe Cache-Implementationen).

Problematisch werden Caches in Hibernate, wenn Daten von anderen Applikationen (nicht über Hibernate selbst) verändert werden. Hier wird in Hibernate ein Timeout empfohlen, um das Arbeiten auf veralteten Werten weitgehend zu vermeiden. Der Einsatz eines solchen SLCs muss wohl bedacht werden und ist eine Geschäftsentscheidung.

3.7 Die transparente Persistenz

Auch Hibernate unterstützt die transparente Persistenz.

Der navigierende Zugriff auf persistente Objekte wird durch einen First Level Cache pro Session und das implizite Nachladen von Objekten die sich nicht im Hauptspeicher befinden, unterstützt. Dies erzeugt die Illusion, dass sich alle Objekte im Hauptspeicher befinden.

Wird eine Transaktion mit trx.commit() beendet, so werden alle persistenten Objekte implizit in der Datenbank eingefügt, aktualisiert oder gelöscht. Dieses Verhalten kann durch die Anwendung von Kaskadieren auch für nicht persistente Objekte (die persistenzfähig sind) definiert werden. Eine explizite Speicherung durch den Aufruf von session.flush() ist nicht nötig, wird aber bei der Verwendung eines Applikationsservers mit einer von aussen gesteuerten Transaktionskontrolle empfohlen.

Ist das Mapping-File einmal definiert, kann Hibernate zur Laufzeit die Abbildung der persistenten Objekte auf die Tabellen in der DB automatisch vornehmen.

Ergänzungen:

(Quelle von [Hibernate 2006])

Query Languages:

Es können beim Hibernate SQL-'SELECT'-Queries verwendet werden, entweder in "native SQL" oder vorzugsweise in der sehr ähnlichen aber objektorientierten SQL-Version "HQL (Hibernate Query Language)"

Beispiele:

1, HQL

```
select account, payment
from Account as account
  left outer join account.payments as payment
where :currentUser in elements(account.holder.users)
  and PaymentStatus.UNPAID = isNull(payment.currentStatus.name,
    PaymentStatus.UNPAID)
order by account.type.sortOrder, account.accountNumber,
    payment.dueDate
```

2, Cariteria

```
    List cats = sess.createCriteria(Cat.class)
  .add( Restrictions.sqlRestriction("lower({alias}.name) like lower(?)",
    "Fritz%", Hibernate.STRING) )
  .list();
```

Werkzeuge:

Top-down

mit XDoclet:

JavaBean --> Hibernate Mapping XML

mit hbm2ddl:

Hibernate Mapping XML --> Datenbank DDL

Bottom-up

mit Middlegen:

Datenbank DDL --> Hibernate Mapping XML

mit hbm2java:

Hibernate Mapping XML --> JavaBean

Produkte:

Hibernate Core for Java

Hibernate for Java, native APIs and XML mapping metadata

Hibernate Annotations

EJB 3.0 (JSR 220), Validator, Search

Hibernate EntityManager

Standard Java Persistence API for Java SE and Java EE

Hibernate Tools

Mapping editor, console, Ant task, wizards (Eclipse)

NHibernate

he NHibernate service for the .NET framework

JBoss Seam

Framework for JSF, Ajax, and EJB 3.0/Java EE 5.0 applications

Literatur:

[Hibernate 2006]:

HIBERNATE - Relational Persistence for Idiomatic Java
Hibernate Reference Documentation 3.2.1
http://www.hibernate.org/hib_docs/v3/reference/en/html/
Dezember, 2006

[OLAT 2006]:

OLAT 5.0 functional servey
Version 1.5, 2006-11-10
Florian Gnägi, frentix GmbH, www.frentix.com
Maya Schüssler, Zurich University,
www.id.unizh.ch/org/mels.html
http://www.olat.org/public/documentation/mainColumnParagraphs/05/document/OLAT_5_0_Functional_Survey_v2.pdf

[Xiaxin 2005]:

"The Hibernate Studies", Xia Xin
Publishing House of Electronics Industry, 2005, Shanghai
ISBN 7-121-00670-7
www.hibernate.org.cn

an sonst:

Objektrelationales Mapping (O/R-M) mit Hibernate 3
www.torsten-horn.de

Complete Hibernate 3.0 Tutorial
http://www.roseindia.net/hibernate/index.shtml

Ich bedanke mich für die sprachliche Hilfe von Herrn Stefan Bayer.

5. Feb. 2007, Leipzig