Wissenschaftliche Arbeit

Implementierung eines Hibernate Envers Klon

Andreas Rau

22. Juli 2014

Bearbeitungszeitraum	13 Wochen
Matrikelnummer	418694
Kursbezeichnung	Tinf12A
Betreuer	Daniel Pittner
Studienreferent	Jörg Eissler

Selbstständigkeitserklärung

gemäß § 5 (2) der „Studien- und Prüfungsordnung DHBW Technik“ vom 18. Mai 2009.

Ich habe die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Stuttgart, September 2014

Andreas Rau

Abstract

Software is getting more complex and the need of archiving rises. In order to fulfill legal compliances a tool for archiving has to be developed. Regular database archiving is restricted by configs given by the database providers. A reusable tool for archiving following the example of hibernate envers is the solution.

Abbildungsverzeichnis

Akronyme

JPA

Java Persistence API

AOP

aspektorientierte Programmierung

OSGI

open service gateway initiative

JPQL

Java Persistence Query Language

SQL

structured query language

POJO

plain old Java object

Glossar

Audit-Log

Ein Audit Log oder auch **Audit Trail** ist eine chronologische Aufzeichnung von Informationen.

Persistenzprovider

Persistenzprovider sind Implementationen der JPA Schnittstelle wie Hibernate, Eclipselink und OpenJPA.

Java Persistence API

Die Java Persistence API ist eine Schnittstelle für Java-Anwendungen. Sie vereinfacht das Objektrelationale Mapping von Objekten auf Datenbanken.

Envers

Envers ist ein Feature von Hibernate welches sich Entity Versionierung kümmert.

1.Einleitung

1.1. Auditierung

Der Begriff der Auditierung kommt aus dem Lateinischen und bedeutet so viel wie „Anhörung“. Dabei gehört zu einem Audit die Ist-Analyse des aktuellen Zustands und der Vergleich des erwarteten Wertes mit dem Analysierten, wobei in dieser Arbeit das Auswerten des erwarteten Wert mit dem Analysierten unbehandelt bleibt.

Die Informationen, die bei der Analyse einer Auditierung festgehalten werden, sind das Was, das Wer und das Wann. Informationen, die in dieser Zusammenstellung abgespeichert werden, nennt man Audit-Log. Diese Analyse findet bei jeder Änderung statt. Eine Umsetzung des Audit-Logs in Datenbanken findet sich in Versionsspalten, Userspalten und Spalten die den letzten Zugriff sichern.

Eine intelligentere Lösung sind Audit-Trails die über die abgespeicherten Informationen des Audit-Logs hinaus gehen und in separaten Tabellen komplette Tupel samt Zusatzinformationen über Benutzer und Zeit festhalten. Ein solcher Trail ermöglicht es ein Tupel vom aktuellen Zustand samt aller User die daran gearbeitet haben bis hin zu der Erstellung zurück zu verfolgen und unter Umständen wiederherzustellen. Damit wird eine nachvollziehbare Datenverarbeitung garantiert, da alle Änderungen einem Datum und einem User zugeordnet werden können.

Anwendungsfälle finden sich in Sicherheitsroutinen wie dem Vier-Augen-Prinzip, das besagt, dass ein Benutzer keine zwei Bearbeitungsschritte nacheinander ausführen darf. Nur ein anderer Benutzer darf den nächsten Bearbeitungsschritt durchführen.

Bei der Implementierung einer auf Datenbanken beruhenden Software bietet es sich an Auditierungsprozesse mit einzubeziehen. Die meisten Datenbanksysteme sind objektrelational aufgebaut. Üblicherweise wird mittels objetrelationalen Abbildungstechnologien eine Kommunikation zwischen Software und Datenbank ermöglicht.
Dabei werden sogenannte Persistenzprovider verwendet um die Verbindung zwischen Datenbanken und Software zu vereinfachen. Sehr häufig kommt hierbei Hibernate oder seit einigen Jahren eine Java Persistence API - JPA Implementierung zum Einsatz. Die bekanntesten JPA Anbieter sind Hibernate, EclipseLink, früher auch unter dem Namen TopLink bekannt, und Open JPA.

1.2. Aspektorientierte Programmierung

Die Speicherung von Objekten in einer relationalen Datenbank bringt unvermeidbar redundante Codeschnipsel mit sich. Schreib- und Lesetransaktionen haben mit der eigentlichen Programmlogik nichts zu tun und verschlechtern lediglich die Lesbarkeit des Codes. Das Paradigma der AOP liegt darin, logische Prozesse von der eigentlichen Anwendungslogik zu trennen.
Die Auditierung ist ein typisches Anwendungsbeispiel der aspektorientierten Programmierung, hierbei werden die datenbankspezifischen Transaktionen von der restlichen Logik soweit wie möglich getrennt.

1.3. Aufgabenstellung

Das Ziel dieses Praxiseinsatzprojekts ist eine Applikation mit ähnlichem Umfang wie Hibernate Envers zu schreiben. Die Notwendigkeit liegt in der Tatsache dass Websphere liberty Server, welche auf OSGI aufbauen, durch die Isolation der Klassen nicht mit der Hibernate JPA Implementierung von Envers harmonieren. Dabei unterliegt die Anwendung einigen Kriterien. Er muss in Java geschrieben, auf der Java Persistence API aufbauen sowie OSGI kompatibel implementiert werden. Als Persistence Provider wird Eclipse Link vorausgesetzt.

Die Anwendung soll eine einfache Datenbanktabellen-Historie erstellen ohne dabei ManyToOne, OneToMany, ManyToMany oder OneToOne Relationen zu sichern, wobei letztere ohne viel Aufwand als Plugin verfügbar gemacht werden können soll. Die Beziehung zwischen E-Mails und deren Attachments wurden als Beispiel genannt. Dabei sollte stets die AOP umgesetzt werden.

2. Grundlagen

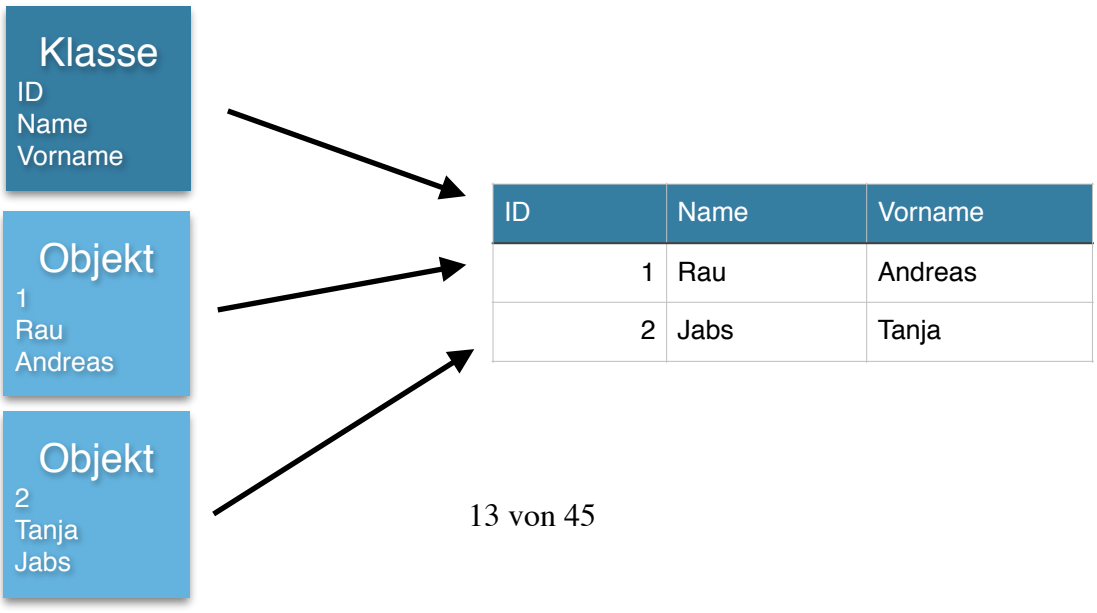
2.1. JPA

Die Java Persistence API ist eine Schnittstelle für Java-Anwendungen und vereinfacht die Abbildung von Objekten in Datenbanktabellen. Dadurch ist die Zustandsspeicherung eines Programms realisierbar, da Laufzeitobjekte einer Java-Anwendung in relationalen Datenbanken abgebildet werden können. Bei einem Ausfall des Programms oder der Hardware können Objekte aus Datenbanktabellen in die Anwendung geladen werden, um den letzten gespeicherten Zustand wieder herzustellen.

2.1.1. Entitäten

Die Persistenz-Entitäten sind gewöhnliche Java Objekte welche mit objekrelationalen Meta Daten vermerkt werden. Diese Metadaten sind JPA spezifische Java Annotationen.
Jede Klasse ist logisch auf eine Tabelle innerhalb einer Datenbank abgebildet, wobei jedes Objekt der Klasse ein Tupel innerhalb der Tabelle ist.

Abbildung 1:
Die Abbildung zeigt das objektrelationale Mapping von Klassen auf Datenbanktabellen.



2.1.2.Persistenz Metadaten

JPA benötigt Informationen über die zu persistierenden Entitäten. Diese Entitäten werden in der persistence.xml eingetragen. Des Weiteren enthält die XML wichtige Informationen wie die URL, welcher JDBC Treiber verwendet wird und ob Tabellen in der Datenbank erstellt werden oder nur vorhandene benutzt werden sollen.
Hier eine einfache persistence.xml als Beispiel, die erste Box zeigt an welche Klassen persistiert werden sollen, die zweite die Datenbankeinstellungen.

Abbildung 2:
Die Abbildung zeigt die Einstellungen in der persistence.xml. Unwichtige Bereiche sind ausgegraut.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/per-
sistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence http://
xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd“>

   <persistence-unit name="Auditing" transaction-type="RE-
SOURCE_LOCAL">
         <class>entities.Setting</class>

         <properties>
               <property name="javax.persistence.jdbc.url"
value="jdbc:mysql://localhost:3306/audit"/>
               <property name="javax.persistence.jdbc.user"
value="root"/>
               <property name="javax.persistence.jdbc.driver"
value="com.mysql.jdbc.Driver"/>
               <property name="eclipselink.ddl-generation"
value=„none"/>

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

2.1.3.JPQL

Die Java Persistence Query Language ist ein Teil der Java Persistence API. JPQL ist eine Datenbanksprache welche sich von normaler SQL Syntax unterscheidet. Die Unterschiede werden in 2.1.4 angesprochen. Der größte Vorteil liegt darin, dass JPQL Datenbanken unabhängig einsetzbar sind und somit ein Datenbankwechsel nicht unbedingt voraussetzt den Quellcode zu aktualisieren. Die Unabhängigkeit wird dadurch erreicht, dass JPA über die Einstellungen in der persistence.xml ausließt, welcher JDBC Treiber verwendet wird. Bei der Kompilierung der Java Klassen wird dann der JPQL Code in die entsprechende Datenbanksyntax geändert. JPQL unterstützt SELECT, DELETE und UPDATE Queries, die neben der eigentlichen Persistierung durch JPA oder einen Persistenzprovider wie Hibernate oder Eclipselink den vollen Funktionsumfang einer normalen Quert einnehmen können.

2.1.4.Unterschiede zu nativem SQL

JPQL Syntax ähnelt nativem SQL ist aber auf JPA Entitäten ausgelegt anstatt auf Datenbank Tabellen.
Eine Datenbankanfrage der Form:

```
SELECT s FROM Setting s
```

gibt eine Liste von allen Einstellungen zurück. Dabei ist egal ob sich nur ein Element in der Liste befindet oder mehrere, es wird immer eine Liste zurück gegeben. Es ist jedoch möglich sich mittels der Methode getFirstResult() das erste Element der Liste zu holen.

2.2. Hibernate

Das objektrelationale Abbilden von Java Objekten auf Datenbanktabellen und wiederum aus Datenbanktupeln Objekte zu erstellen spezifiziert in der Java Persistence API ist der Grundstein für das Hibernate Framework.

2.2.1. Annotations

Java Annotationen werden in Hibernate dazu verwendet Objekte und Methoden mit Metadaten zu versehen. Dabei werden Hibernate spezifische Annotationen verwendet. Folgend werden einige Annotationen erklärt:

@Entity

Jedes persistierte POJO ist eine Entität und muss für Hibernate als solche sichtbar gemacht werden.

@Id

Die „@Id“ Annotation markiert das nachfolgende Feld als Primärschlüssel. Dabei werden oft weitere Annotationen wie „@GeneratedValue“, welche sich um die Generierung von einzigartigen Werten kümmert, verwendet. Vergleich bar mit dem SQL Statement AUTO INCREMENT.

Abbildung 3:
Die Abbildung zeigt Beispielhaft die Annotierung eine Entitäts Klasse.

```
@Entity
public class Setting {

   @Id
   int id;
…
}
```

2.2.2.Envers

Envers steht für Entity Versioning und kümmert sich um die Historierung von Entitäten. Beachtet werden dabei insert, update und delete Queries. Bei jeder Veränderung eines Datenbanktupels werden mittels einer Revisionsnummer und der Art der Revisionierung Revisionstabellen angelegt. Dabei berücksichtigt envers korrekt die Beziehungen zwischen einzelnen Tabellen und bildet diese ebenso nach. Um besagte Beziehungen zu berücksichtigen muss der jeweilige Fremdschlüssel mit einer entsprechenden Annotation markiert werden.
Die folgende Abbildung zeigt drei Tabellen wobei die Tabelle Student als zu persistierend und auditierend markiert wurde. Bei einem Insert legt Envers die Tabelle Student_AUD an. Der Postfix „_AUD“ wird von Envers automatisch generiert falls nicht genauer spezifiziert. Die Tabelle Student_AUD unterscheidet sich von seinem Original in den Spalten „REV“ welche die Revisions Id und „REVTYPE“ die Art der Revisionierung beinhaltet. Dabei steht die Null für einen Insert, die Eins für ein Update und die Zwei für ein Delete.

Die REV Spalte ist ein Fremdschlüssel der zu der Tabelle „REVINFO“ gehört. Jedes Tupel in besagter Tabelle hat noch ein Revisions Kennzeichen.

Abbildung 4:
Die Abbildung zeigt die von Hibernate und Envers erstellten Datenbanktabellen.

Tabelle Student

ID	Name	Vorname
1	Rau	Andreas
2	Jabs	Tanja

Tabelle Student_AUD

ID	Name	Vorname	REV	REVTYPE
1	Rau	Andreas	1	0
2	Jabs	Tanja	2	0

Tabelle REVINFO

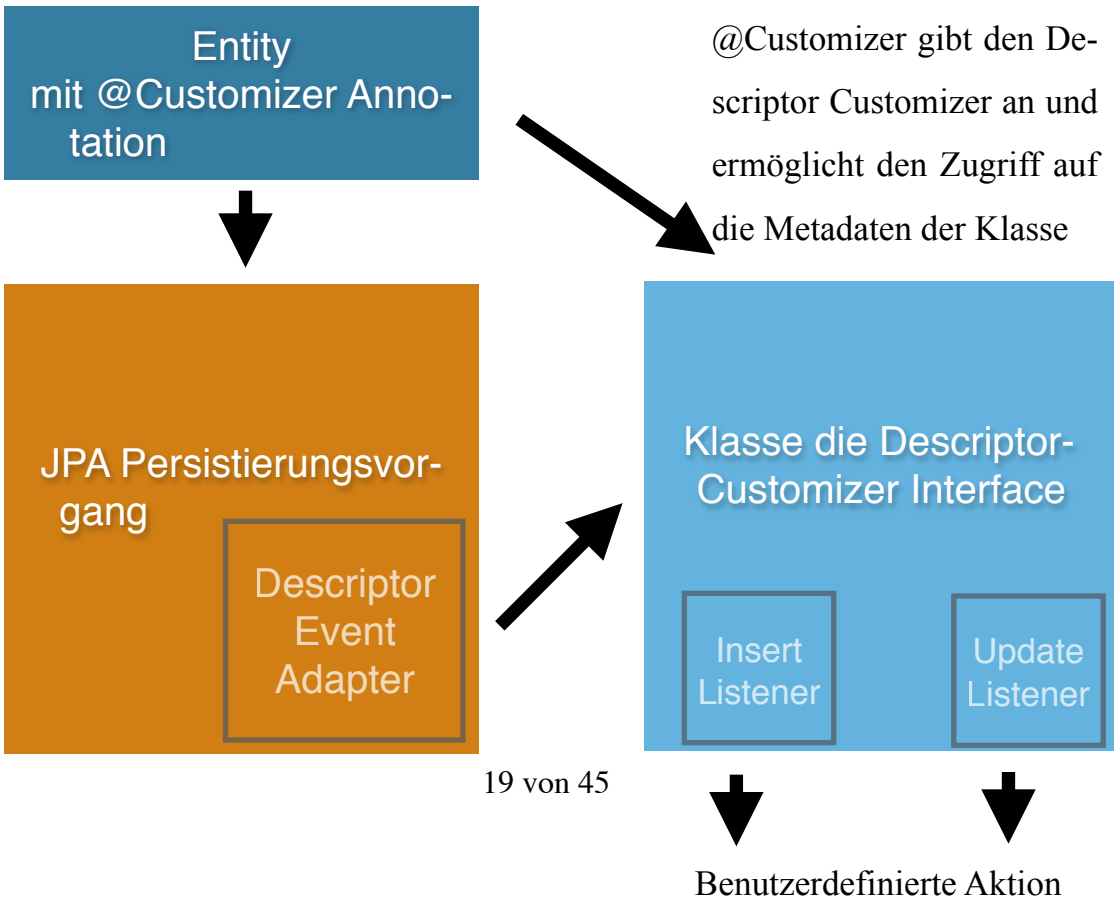
REV	REVSTMP
1	1278451
2	1792649

2.3. Eclipse Link

2.3.1. Customizer Annotation

Eclipselink ermöglicht es die Persistenz Meta Daten mit Hilfe von Java Code zu erweitern. Die Schnittstelle dazu bildet der DescriptorCustomizer. Informationen über eine Klasse werden per @Customizer Annotation an den Deskriptor weiter gegeben. in Kombination mit den aboutToUpdate und aboutToInsert Methoden kann bei jedem Persistierungsvorgang der markierten Klasse eine beliebige Aktion gestartet werden.
Dieses Verfahren hat den Vorteil dass mit einer Annotation einer Klasse in den eigentlichen von JPA verschleierten Persistierungsvorgang eingegriffen werden kann. Ohne besagtes Verfahren wäre es nicht oder nur mit großen Umständen möglich einen Envers Klon zu erstellen.

Abbildung 5:
Die Abbildung zeigt die Beziehungen zwischen den Customizer Elementen.



2.4. OSGI

Bei der Entwicklung einer Applikation wird auf eine saubere Programmierweise geachtet, es wird versucht Ordnung zu halten um unnötige Komplikationen zu vermeiden. Von zeit zu Zeit ändern sich jedoch Bedingungen für die Applikation, es werden neue Features hinzugefügt, manche Funktionsweisen umgeändert und auf ein Mal hat man ein kompliziertes ‚Spaghettikonstrukt' aus Code, der alles andere als transparent ist. Das Pflegen einer solchen Applikation erweist sich als schwierig und zeitintensiv. Eine Lösung wäre ein Refactoring, jedoch sind die meisten Manager davon wenig überzeugt, da weder neue Verbesserungen eingebaut werden noch sich ein direkter Profit erzielen lässt.

Die Open Service Gateway Initiative (OSGI) bietet ein Konzept zur Modularisierung von Java Applikationen. Im Gegensatz zu einem Jar, welches in Java ebenso der Modularität dient, enthält ein OSGI Bundle Metadaten über benutzte und zur Verfügung gestellte Schnittstellen sowie Versionsinformationen und User beziehungsweise Hersteller. Diese Metainformationen erlauben einem OSGI Baustein modularer aufzutreten als einem reinen Jar. Ermöglicht wird dies durch ein einzigartigen Classloading Mechanismus. Der Classloader definiert einen isolierten Bereich für die Klasse. Dabei werden einkommende und ausgehende Schnittstellen explizit gesetzt.

Ein Bundle sieht von einem anderen Bundle nur die Informationen welche explizit exportiert werden, somit werden interne Angelegenheiten von externen getrennt. Ein bekanntes Beispiel ist die Entwicklungsumgebung Eclipse. Eclipse baut auf OSGI auf und demnach lassen sich Plugins unabhängig voneinander installieren und deinstallieren. Des Weiteren sind alle Bundles

mit einer Versionsnummer versehen somit kann eine Applikation mehrere Versionen eines Bundles gleichzeitig führen. Jedes Bundle gibt in seinen Dependencies an auf welche anderen Bundles es angewiesen ist.

3. Implementierung

3.1. Zielsetzung

Zur Lösung komplexer Problemstellungen, die meist aus mehreren, voneinander unabhängigen Abschnitten bestehen, sind Vorgehensweisen zu entwickeln, welche die Lösung aller Teilprobleme vorsehen. Für die Lösungswege der Teilprobleme wird eine Reihenfolge festgelegt, wodurch die Ergebnisse der einzelnen Phasen zu einer ganzheitlichen Lösung führen. Daher besteht das primäre Ziel dieser Praxisarbeit in der Entwicklung eines Workflows, der den gesamten Prozess von Listenern bis hin zur Datenbankanbindung abdeckt und ordnet.

3.2. Umsetzung

Die Herangehensweise zur Lösung des gestellten Problems, ist die Zerlegung in Teilprobleme, für welche jeweils Ziele definiert werden. Wird während der Implementierung ein neues Problem entdeckt, so findet für dieses ebenfalls eine Analyse statt. Nach der Analyse werden die einzelnen Bestandteile geordnet und dokumentiert.

3.3. Struktur

Um die Arbeit zu vereinfachen werden von vorneherein Prioritäten vergeben, die sich durch die Reihenfolge der Teilprobleme widerspiegeln. Die höchste Priorität ist damit dem Listener zuzuschreiben. Er ist dafür zuständig eine von EclipseLink durchgeführte Transaktion abzufangen und sie dem Programm zugänglich zu machen. Dabei werden Informationen zum persistierten Objekt direkt an die Auditierungsinstanz weitergegeben, die sich

mittels verschiedener Algorithmen und Helferklassen um das ORM der History kümmern. Eine besondere Schwierigkeit liegt in der vorausgesetzten Flexibilität des Codes, welcher jede annotierte Klasse mit einer Transaktions Historie versieht. In 3.5 ist eine Übersicht der Klassen, die implementiert werden sollen dargestellt. Die Setting Klasse ist dabei die zu persistierende Klasse.

3.4. Libraries

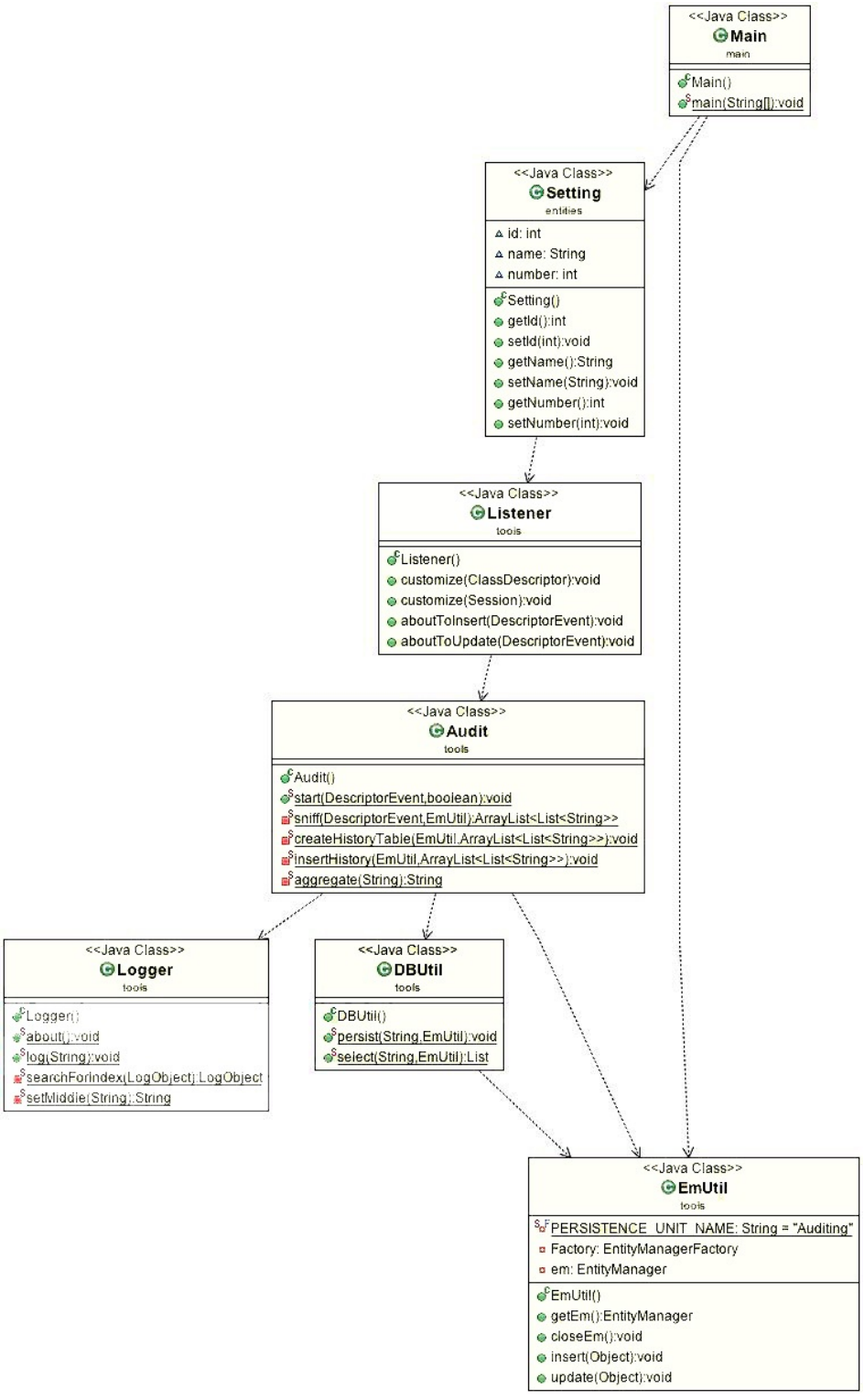
In der Envers Klon Applikation werden die jeweiligen für die Datenbank spezifischen JDBC Treiber verwendet, gefunden werden können diese in der JDBC Userlibrary. Des Weiteren wird die Java Persistence Jar in der Version 2.1 ebenso die Eclipselink Jar in der Version 2.5 benutzt.

3.5. UML (Datenmodell)

Das Datenmodell enthält die Klassen „Listener“, „Audit“, „DBUtil“, „EmUtil“, „Main“ und „Logger“. Für die einzelnen Logs, die sehr häufig auftreten , wurde mit Beachtung der AOP eine Logger Klasse erstellt, welche spezialisierter auf die Anwendung ist als die reine „print“ Methode. Der Envers Klon wird objektorientiert implementiert. In der Abbildung ist ein UML der kompletten Anwendung samt einer zu persistierenden Klasse zu sehen.

Abbildung 6:

Die Abbildung zeigt das Datenmodell des Envers Klons.



3.6. Listener

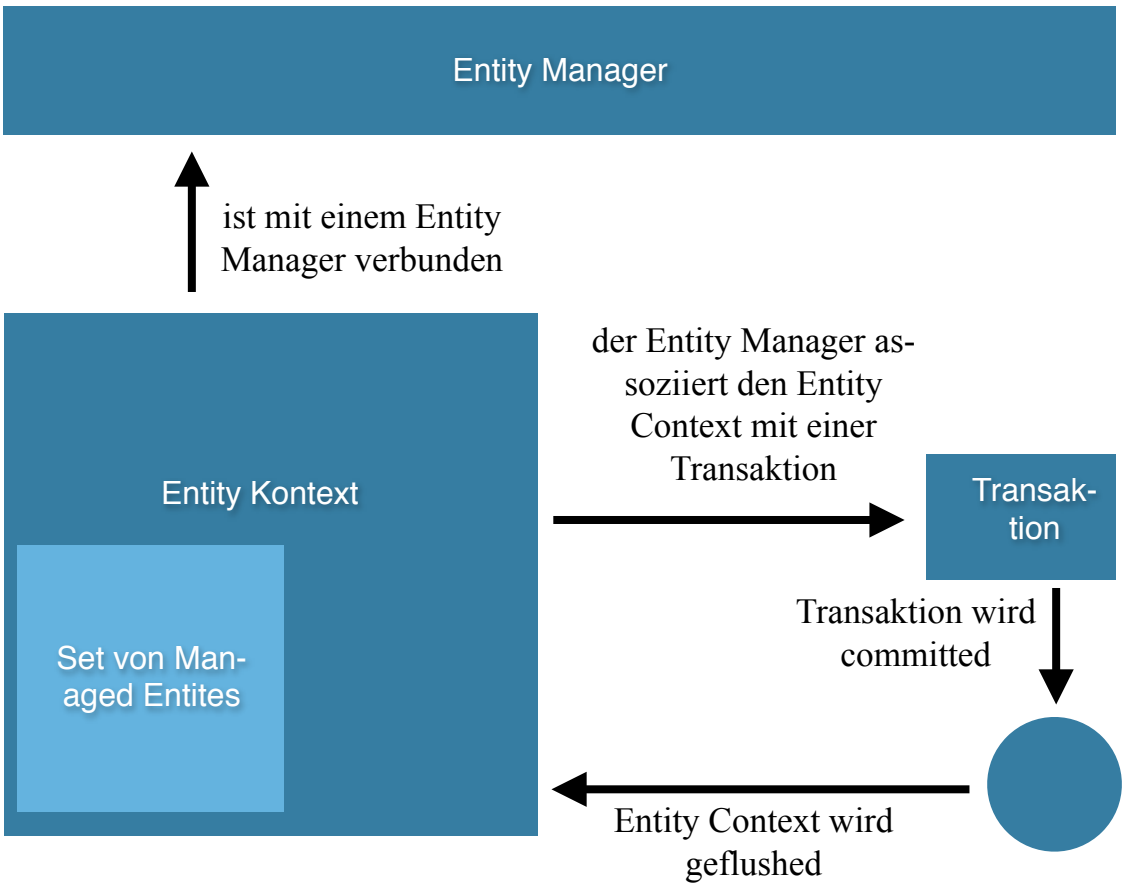
Zu aller erst sind Begrifflichkeiten zu klären, die im folgenden Abschnitt verwendet werden. Eine Entity Manager Klasse kümmert sich um den Lebenszyklus eines Entity Kontexts (create, update, delete). Die Persistence Unit kümmert sich um die Konfiguration der Entitätsklassen (persistence.xml[1]). Der Persistence Kontext ist zuständig für die Konfiguration eines Sets von Managed Entities. Zuletzt gibt es noch eine Managed Entity die nichts anderes ist als eine Instanz einer Entität welche als zu persistierend in der persistence.xml eingetragen ist (POJO).
Die Hierarchische Struktur ist in Abbildung zu sehen.

Ein Persistence Kontext wird vom Entity Manager mit einer Datenbanktransaktion verbunden. Es gibt zwei verschiedene Transaktionsschemen, wichtig für den Envers Klon ist das ‚ressource local' Schema, bei dem eine Transaktion auf dem vorhandenen JDBC Treiber aufbaut.
Das erste und wichtigste zu lösende Teilproblem ist zu erkennen wann ein Persistierungsvorgang startet. Dabei ist zu unterscheiden zwischen der Prähandlung, eigentlichen Handlung und Posthandlung. Während der Prähandlung schaut der Entity Manager ob eine Entity Kontext Instanz vorhanden ist, falls nicht wird eine erstellt (Prähandlung). Danach verbindet der Entity Manager den Entity Kontext mit einer Transaktion (eigentliche Handlung). Sobald die Transaktion committed ist wird der Entity Kontext geflushed (Posthandlung).

[1] wie beschrieben in 2.1.2

Jetzt ist zu klären welcher Schritt für den Envers Klon am wichtigsten ist. Eine Transaktion ist vollkommen abgeschlossen sobald ein Commit erfolg reich ausgeführt wurde. Um eine sichere und intelligente Historisierung zu gewährleisten sind Flüchtigkeitsfehler wie ‚Datenleichen' zu vermeiden. Deshalb wird der Historisierungsvorgang nur nach einem erfolgreichen Commit durchgeführt. Wenn eine Historisierung schon mit der Präphase starten würde ergäbe sich das Risiko, dass eine Historisierung stattgefunden hat, ohne dass das historisierte Tupel überhaupt persistiert wurde.

Abbildung 7:
Die Abbildung zeigt wie Entity Manager mit Entity Kontext und einer Transaktion verbunden sind.



Methoden

Die Methoden, die in der Listener Klasse verwendet werden um einen Persistierungsvorgang zu entdecken, müssen wie in 3.4 beschrieben nach dem Persistierungsvorgang die Historisierung einleiten. In Eclipse Link kommen damit nur folgende Methoden in Frage: **postUpdate** und **postInsert** des Descriptor Customizer Interfaces. Bei jedem Commit einer Insert und Update Transaktion einer mit @Customizer annotierten Klasse wird postInsert beziehungsweise postUpdate aus der Listener Klasse aufgerufen. Dabei muss die Annotation der Klasse auf den Listener verweisen. Mit dem Commit einer Transaktion wird im Anschluss ein Descriptor Event Objekt an die Audit Klasse weiter gegeben. Dieses Event enthält alle Informationen über die persistierte Entität sowie deren Transaktion. Diese Informationen werden in der Audit Klasse benötigt um eine Transaktion durchzuführen.

3.7. EmUtil

Die EmUtil Klasse wird einmal beim Start der Applikation in der Audit Klasse initialisiert. Sie beinhaltet alle eine Entity Manager Factory welche wiederum ein Entity Manager[2] Objekt erstellt.
Die Klasse hat die Funktion eines Containers, welche alle notwendigen Schritte beinhaltet, um einen Entity Manager zu erzeugen. Mit einer Getter Methode lässt sich aus dem EmUtil Objekt der Entity Manager abrufen. Des Weiteren enthält der Container eine Methode um den Entity Manager zu schließen ebenso wie eine insert und update Funktion, welche auf JPA aufbauen und ein zu persistierendes Objekt als Parameter haben. Dabei wird eine Transaktion gestartet das angegebene Objekt persistiert und im Anschluss die Transaktion commited.

Der Entity Manager sowie alle vorherig genannten Methoden wurden in ein einzelnes Objekt gepackt um der AOP gerecht zu werden und die objekorientierte Programmierweise anzuwenden.

[2] wie in 3.5

3.8. DBUtil

Bei der Erstellung einer Historietabelle in der Datenbank muss der Anwendung bekannt sein welche Spalten die zu historisierende Entität aufweist. Dies erweist sich als einfach solange man ein festgelegtes Entitätsschema hat bei dem alle Entitäten bekannt und in die persistence.xml eingetragen sind. Dabei würde mit der eigentlichen zu persistierenden Entität direkt eine namentlich abgeglichene Historie Entität als Klasse angelegt werden und auf die Persistierungsmethoden von Eclipselink zurückgegriffen werden. Bei jedem instantiieren einer Entitätsklasse würde direkt eine entsprechende Historie Klasse mit den selben Feldern gefüllt werden.

Da der Envers Klon aber modular eingesetzt wird ist vorher nicht bekannt welche Entitäten persistiert werden. Dabei werden diese Entitäten auch nicht direkt importiert sondern über ein Descriptor Event des Descriptor Customizer geholt.
Eclipselink ermöglicht nur das persistieren von bereits bestehenden Klassen, deswegen muss bei dem Envers Klon auf eine direkte SQL Persistierung[3] zurückgegriffen werden.
Dabei übernimmt DBUtil die komplette Datenbankkommunikation für schreibende und lesende Zugriffe.
Die Methode „persist“ hat als Parameter einen QueryString der in der Audit Klasse erstellt wird sowie ein EntityManagerUtil[4] Objekt. Bei einem Aufruf der persist Methode wird eine Transaktion im Entity Manager gestartet, eine Query erstellt und ausgeführt.

[3] wie in 2.1.3

[4] wie in 3.6

3.9. Audit

Die eigentliche Hauptfunktion der Anwendung liegt in dem Auslesen des Descriptor Events sowie die Aufbereitung der Daten bis hin zur manuellen Persistierung der generierten Historie Entität. Dabei spielt die Audit Klasse eine wichtige Rolle.

Aufgerufen wird zuerst die Start Methode durch den Listener, welcher als Parameter mit gibt ob es sich um ein Update oder ein Insert handelt. Ein EmUtil wird initialisiert und für den User wird ein Log erstellt, in dem zu erkennen ist, um was für eine Art der Persistierung es sich handelt. Als nächstes wird eine Arraylist aus Stringlisten initialisiert und mittels der „Sniff“ Methode gefüllt. Dabei wird als Parameter das Descriptor Event als auch das EmUtil Objekt mitgegeben.

Sniff

Um die Historie einer Entität festzuhalten muss zuerst festgelegt werden wie die Tabelle heißen soll, welche Spalten die Tabelle in der Datenbank haben wird, welchen Typ diese Spalten haben werden und mit welchen Werten die Spalten gefüllt werden sollen. Die Sniff Methode erstellt durch das Füllen der Arraylist ein Abbild der persistierten Entität.

Der erste Schritt im Füllen beinhaltet das Herauslesen der persistierten Entität. Dabei wird aus dem Descriptor Event die Entität einem neuen Objekt zugewiesen, um es leichter zugänglich zu machen. Als nächstes werden alle Getter Methoden, die sich in der eigentlichen Entitätsklasse befinden, invoked um den Wert der persistierten Felder herauszufinden. Fortlaufend werden der Name des Feldes sowie der Typ in die Arraylist geschrieben. Die Daten werden nicht ‚raw‘ eingetragen sondern an die SQL Syntax (upper-

case) angepasst sowie kleinere Stringoperationen ausgeführt. Am Ende wird von der Sniff Methode die gefüllte Arraylist zurückgegeben.

Abbildung 8:
Die Abbildung zeigt eine gefüllte Arraylist.

tableName	columnList	typeList	valueList
SETTING	NAME	VARCHAR(255)	default
	ID	INT	201
	NUMBER	INT	0

Create History

Die zuvor gefüllte Arraylist wird weiter an die „createHistory“ Methode weitergegeben welche anfangs prüft, ob in der Datenbank schon eine History Tabelle der persistierten Entität vorhanden ist. Dabei wird eine SQL Query erstellt die mittels der „SHOW TABLES LIKE“ Funktion nach der Historie Tabelle schaut. Die Query wird an die DBUtil.select Methode weitergegeben. Im Fall dass noch keine Historie Tabelle angelegt ist, wird diese erstellt. Dabei wird ebenso wie für die Suchfunktion eine Query erstellt. Der Unterschied liegt darin, dass es sich um eine „Create Table“ Funktion handelt. Die SQL Query wird mittels der zuvor gefüllte Arraylist erstellt. Danach wird die Query von der DBUtil.persist Methode ausgeführt.
Die SQL Befehle werden vom Logger ausgegeben, beziehungsweise informiert er den User, ob das Erstellen einer Historie Tabelle von Nöten war oder nicht.

Insert History

Als zweitletzter Schritt wird die Methode „insertHistory“ aufgerufen und ihr werden die EmUtil sowie die Arraylist mit den Informationen über die Entität mitgegeben. Darauf wird eine Insert SQL Query erstellt, die alle Informationen der Entität enthalten.

3.10.Logger

Der Logger verbessert die Übersichtlichkeit der gesamten Applikation indem er statt der einfachen print Funktion mehrzeilige Logs generiert. Dabei wird der Zeilenumbruch in der Klasse festgelegt. Innerhalb der verwendeten Klassen muss nur der umformatierte String für die Ausgabe eingegeben werden und in der Konsole erscheint ein formatierter Text der die Übersichtlichkeit des Logs erheblich steigert.

Abbildung 9:
Die Abbildung zeigt einen Beispiel Log, bei dem keine Historie Tabelle angelegt werden musste.

```
------------------------------------------------------------
----------            insert detected             ----------

------------------------------------------------------------
----------       persisted entity : SETTING       ----------

------------------------------------------------------------
----------            invoked methods:            ----------

------------------------------------------------------------
---------- getName(): return : name=NAME, type :  ----------
----------     VARCHAR(255), value : default      ----------

------------------------------------------------------------
---------- getId(): return : name=ID, type : INT, ----------
----------              value : 201               ----------

------------------------------------------------------------
----------getNumber(): return : name=NUMBER, type ----------
----------            : INT, value : 0            ----------

------------------------------------------------------------
----------   no need to create SETTING_HISTORY,   ----------
----------         proceeding with insert         ----------

------------------------------------------------------------
----------  SETTINGHISTORY gets inserted, query:  ----------

------------------------------------------------------------
---------- INSERT INTO SETTING_HISTORY (NAME, ID, ----------
---------- NUMBER) VALUES ("default", "201", "0") ----------

------------------------------------------------------------
----------           SETTING persisted            ----------
```

Abbildung 10:

Die Abbildung zeigt einen Log, bei dem noch keine Historie Tabelle vorhanden war

```
------------------------------------------------------------
----------            insert detected             ----------

------------------------------------------------------------
----------       persisted entity : SETTING       ----------

------------------------------------------------------------
----------            invoked methods:            ----------

------------------------------------------------------------
---------- getName(): return : name=NAME, type :  ----------
----------     VARCHAR(255), value : default      ----------

------------------------------------------------------------
---------- getId(): return : name=ID, type : INT, ----------
----------               value : 1                ----------

------------------------------------------------------------
----------getNumber(): return : name=NUMBER, type ----------
----------            : INT, value : 0            ----------

------------------------------------------------------------
----------     need to create SETTING_HISTORY     ----------

------------------------------------------------------------
----------   CREATE TABLE SETTING_HISTORY( NAME   ----------
----------   VARCHAR(255), ID INT, NUMBER INT,    ----------
----------  REV_ID INT NOT NULL AUTO_INCREMENT,   ----------
----------         PRIMARY KEY (REV_ID))          ----------

------------------------------------------------------------
----------        SETTING_HISTORY created         ----------

------------------------------------------------------------
----------  SETTINGHISTORY gets inserted, query:  ----------

------------------------------------------------------------
---------- INSERT INTO SETTING_HISTORY (NAME, ID, ----------
----------  NUMBER) VALUES ("default", "1", "0")  ----------

------------------------------------------------------------
----------           SETTING persisted            ----------
```

4. Zusammenfassung und Ausblick

Abschließend kann gesagt werden, dass das gesamte Projekt ein Erfolg war und eine gute Basis für eine zukünftige Weiterführung bietet. Eine mögliche Erweiterung wäre bei der Implementierung der Audit Klasse eine postDelete Funktion einzuführen, um neben einem Insert und Update auch ein Delete eines Datenbanktupels festzuhalten. Dadurch könnte man eine Fremdschlüsselimplementierung zwischen Auditierungstabellentupel und Entitätstupel erstellen, was bis jetzt nicht möglich war. Der Grund dafür ist dass ein Tupel in der Historie Tabelle welches eine Fremdschlüssel Referenzierung auf ein Entitäts Tupel hat nur existieren kann, solange das Entitäts Tupel besteht. Dabei sind zwei Betrachtungsweisen einer Historisierung zuerst zu klären.
Die Erste beinhaltet den Gedanken, dass eine Historie auf ein Original referenziert ist um ein Nachvollziehen der Historie besser zu verwirklichen. Dabei müsste dann der vorher benannte Vorgang des postDelete implementiert werden, da sonst ein original Tupel nicht gelöscht werden kann ohne einen Integritätsfehler in der Datenbank auszulösen. Aufgrund des Löschens des OriginalTupels kann die Historie nicht mehr bestehen, und müsste auch gelöscht werden, was aber sinnlos wäre da damit die Historie verloren geht.
Die Zweite besteht darin dass die Historie Tabelle über keine Fremdschlüssel verfügt, eine Übersicht zum Nachvollziehen der Historie kann mit einem Join realisiert werden.

Literaturverzeichnis

Alexandre de Castro Alves :

OSGI in depth.

1. Auflage, 2011, Manning Publications

Christian Bauer, Galvin King, Gary Gregory :

Java Persistence with Hibernate.

2. Auflage, 2014, Manning Publications

Eclipselink Documentation Center

JPA Specification

Envers Documentation

Anhang

Listener Class

```
package tools;

import org.eclipse.persistence.config.DescriptorCustomizer;
import org.eclipse.persistence.config.SessionCustomizer;
import org.eclipse.persistence.descriptors.ClassDescriptor;
import org.eclipse.persistence.descriptors.DescriptorEvent;
import org.eclipse.persistence.descriptors.DescriptorEven-
tAdapter;
import org.eclipse.persistence.sessions.Session;

public class Listener extends DescriptorEventAdapter implements
DescriptorCustomizer {
	/**
	 * This will audit a specific class.
	 */
	@Override
	public void customize(ClassDescriptor descriptor) {
		descriptor.getEventManager().addListener(this);
	}
	/**
	 * this is executed whenever an entity is inserted
	 */
	@Override
	public void aboutToInsert(DescriptorEvent event) {
		Audit.start(event, true);
	}

	/**
	 * this is executed whenever an entity is updated
	 */
	@Override
	public void aboutToUpdate(DescriptorEvent event) {
		Audit.start(event, false);
	}
}
```

DBUtil Class

```
package tools;

import java.util.List;

import javax.persistence.Query;

public class DBUtil{

	//static final String[] dialect = {"mySQL", "Derby"};

	public static void persist(String queryString, EmUtil emU-
til) {
		emUtil.getEm().getTransaction().begin();
		Query query =
emUtil.getEm().createNativeQuery(queryString);
		query.executeUpdate();
		emUtil.getEm().getTransaction().commit();
	}

	public static List select(String queryString, EmUtil emU-
til) {
		emUtil.getEm().getTransaction().begin();
		Query query =
emUtil.getEm().createNativeQuery(queryString);
		List result = query.getResultList();
		emUtil.getEm().getTransaction().commit();
		return result;
	}
}
```

EmUtil Class

```
package tools;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

public class EmUtil {

	private static final String PERSISTENCE_UNIT_NAME = "Au-
diting";
	private EntityManagerFactory Factory;

	private EntityManager em;

	public EmUtil(){
		this.Factory = Persistence.createEntityManagerFacto-
ry(PERSISTENCE_UNIT_NAME);
	    this.em = Factory.createEntityManager();
	}


	public EntityManager getEm(){
		return em;
	}

	public void closeEm(){
	    this.em.close();
	    this.Factory.close();
	}

	public void insert(Object object){
		this.em.getTransaction().begin();
		this.em.persist(object);
		this.em.getTransaction().commit();
	}

	public void update(Object object){
		this.em.getTransaction().begin();
		this.em.merge(object);
		this.em.getTransaction().commit();
	}
}
```

Audit Class

```
package tools;

import java.lang.reflect.Field;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.lang.reflect.Modifier;
import java.util.ArrayList;
import java.util.Collection;
import java.util.List;

import javax.persistence.Query;

import org.eclipse.persistence.annotations.Customizer;
import org.eclipse.persistence.descriptors.DescriptorEvent;
import org.eclipse.persistence.sessions.Record;
/**
 *
 * this class audits every object that is annotated with "@Cus-
tomizer(tools.Listener.class)"
 * @author andy
 *
 */
public class Audit {

      public static void start(DescriptorEvent event, boolean
mode){
            EmUtil emUtil = new EmUtil();
            if(mode)
                  Logger.log("insert detected");
            else
                  Logger.log("update detected");
            /**
             */
            ArrayList<List<String>> data = sniff(event, emUtil);
            createHistoryTable(emUtil, data);
            insertHistory(emUtil, data);
            /**
             */
            emUtil.closeEm();
            emUtil = null;
      }
```

```
private static ArrayList<List<String>> sniff(DescriptorEvent
event, EmUtil emUtil){
           ArrayList<List<String>> data = new
ArrayList<List<String>>();
                 List<String> tableName = new
ArrayList<String>();
                 List<String> columnList = new
ArrayList<String>();
                 List<String> typeList = new ArrayList<String>();
                 List<String> valueList = new
ArrayList<String>();
           data.add(tableName);
           data.add(columnList);
           data.add(typeList);
           data.add(valueList);

           tableName.add(event.getObject().getClass().getSimple-
Name().toUpperCase());
           Logger.log("persisted entity : " +
tableName.get(0).toString());
           Object entity = event.getObject();

           for(Method method : entity.getClass().getDeclared-
Methods()){
                 if(Modifier.isPublic(method.getModifiers())
                       && method.getParameterTypes().length == 0
                       && method.getReturnType() != void.class
                       && (method.getName().startsWith("get")))
                 {
                       try {
                             String column = method.getName().sub-
string(3, (method.getName().length())).toUpperCase();
                             String value =
method.invoke(entity).toString();
                             String type = aggregate(method.getRe-
turnType().toString()).toUpperCase();

                             if(column != null && value != null &&
type != null){
```

```
Logger.log(method.getName() + "(): return :"

            + " name=" + column

            + ", type : " + type

            + ", value : " + value);
                                    data.get(1).add(column);
                                    data.get(2).add(type);
                                    data.get(3).add(value);
                                }
                            } catch (IllegalAccessException e) {
                                Logger.log("the 'get' method on the
persisted object cant't be invoked");
                                e.printStackTrace();
                            } catch (IllegalArgumentException e) {
                                Logger.log("the 'get' method on the
persisted object cant't be invoked");
                                e.printStackTrace();
                            } catch (InvocationTargetException e) {
                                Logger.log("the 'get' method on the
persisted object cant't be invoked");
                                e.printStackTrace();
                            }
                        }
            }
            return data;
      }
```

```
private static void createHistoryTable(EmUtil emUtil,
ArrayList<List<String>> data){
           String table = data.get(0).get(0).toString().toUpper-
Case();
           String showTables = "SHOW TABLES LIKE '" + table +
"_HISTORY'";

           if(!(DBUtil.select(showTables,
emUtil).toString().contains(table + "_HISTORY"))){
                 Logger.log("need to create " + table + "_HISTO-
RY");

                 String createHistoryTable = "CREATE TABLE " +
table + "_HISTORY(";
                 for(int i = 0; i < data.get(1).size(); i++){
                            createHistoryTable = createHistory-
Table + " " + data.get(1).get(i);
                            createHistoryTable = createHistory-
Table + " " + data.get(2).get(i) + ",";
                       }
                 createHistoryTable = createHistoryTable + "
REV_ID INT NOT NULL AUTO_INCREMENT,"

           + " PRIMARY KEY " + "(REV_ID)"

           //+ ", FOREIGN KEY " + "(ID)" + " REFERENCES " + ta-
ble + "(ID)"

           + ")";
                 Logger.log(createHistoryTable);
                 DBUtil.persist(createHistoryTable, emUtil);
                 Logger.log(table + "_HISTORY created");
           }
           else{
                 Logger.log("no need to create " + table + "_HIS-
TORY, proceeding with insert");
           }

     }
```

```
private static void insertHistory(EmUtil emUtil,
ArrayList<List<String>> data){
            String table = data.get(0).get(0).toString().toUpper-
Case();
            String insertQuery = "INSERT INTO " + table + "_HIS-
TORY (";

            for(int i = 0; i < data.get(1).size(); i++){
                  if(i + 1 < data.get(1).size())
                        insertQuery = insertQuery +
data.get(1).get(i) + ", ";
                  else
                        insertQuery = insertQuery +
data.get(1).get(i);
            }
            insertQuery = insertQuery + ") VALUES (";
            for(int i = 0; i < data.get(1).size(); i++){
                  if(i + 1 < data.get(1).size())
                        insertQuery = insertQuery + "\"" +  da-
ta.get(3).get(i) + "\", ";
                  else
                        insertQuery = insertQuery + "\"" +  da-
ta.get(3).get(i) + "\"";
            }
            insertQuery = insertQuery + ")";

            Logger.log(insertQuery);
            DBUtil.persist(insertQuery, emUtil);
            Logger.log(table + " persisted");
      }

      private static String aggregate(String type) {
            if (type.contains("String")) {
                  return "varchar(255)";
            }
            if (type.contains("int")) {
                  return "int";
            }

            return null;
      }

}
```