Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Heinrich Latreider

Vergleich und Bewertung von OR-Mapping-Frameworks
am Beispiel eines Kundenverwaltungssystems

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Heinrich Latreider

Vergleich und Bewertung von OR-Mapping-Frameworks am Beispiel eines Kundenverwaltungssystems

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Stefan Sarstedt
Zweitgutachter: Prof. Dr. Olaf Zukunft

Abgegeben am 05.02.2016

Heinrich Latreider

Thema der Bachelorarbeit

Vergleich und Bewertung von OR-Mapping-Frameworks am Beispiel eines Kundenverwaltungssystems

Stichworte
Hibernate, Entity Framework, ActiveJDBC, OR-Mapper

Kurzzusammenfassung

Während sich in der Softwareentwicklung das Paradigma der objektorientierten Programmierung durchgesetzt hat, sind in der Welt der Datenbanken relationale Datenbanksysteme weit verbreitet. Wegen der Unverträglichkeit dieser beiden Ansätze werden oft OR-Mapper als wichtige Hilfsmittel zur Persistierung von Objekten aus objektorientierten Programmiersprachen in relationale Datenbanken eingesetzt.

Das Ziel der Bachelorarbeit ist, drei ausgewählte Werkzeuge zur objektrelationalen Abbildung anhand zuvor erarbeiteter Bewertungskriterien miteinander zu vergleichen. Bewertungsgrundlage ist dabei die Konzeption und Realisierung eines fiktiven Kundenverwaltungssystems unter Einsatz dieser Werkzeuge.

Heinrich Latreider

Title of the paper

Comparision and evaluation of O/R mapping frameworks on the example of a customer management system

Keywords
Hibernate, Entity Framework, ActiveJDBC, O/R mapper

Abstract

While in software development the object oriented programming paradigm has established, relational database systems are common in the world of databases. Because of the mismatch between these both approaches, object relational mappers are used for persisting objects from object oriented programming languages in relational databases.

The aim of the bachelor thesis is to compare three selected tools for object relational mapping based on previously elaborated evaluation criteria. Basis of valuation is the conception and realization of a fictional customer management system with the use of these tools.

Inhaltsverzeichnis

1 Einführung 7

1.1 Problemstellung und Zielsetzung 7

1.2 Abgrenzungen 8

2 Grundlagen 9

2.1 Objektorientierte Programmierung 9

2.1.1 Objekte und Klassen 9

2.1.2 Assoziationen 10

2.1.3 Vererbung und Polymorphie 11

2.1.4 Object Lifecycle 12

2.2 Relationale Datenbanken 12

2.2.1 Relationales Datenmodell 12

2.2.2 Schlüssel 13

2.2.3 Beziehungen 14

2.2.4 SQL 14

2.2.5 Transaktionen 16

2.2.6 Datenbankschnittstellen 18

2.3 Objektrelationale Abbildung 18

2.3.1 Impedance Mismatch 18

2.3.2 OR-Mappingframeworks 20

2.4 Architektur- und Entwurfsmuster 20

2.4.1 Architekturmuster 21

2.4.2 Verhaltensmuster 23

2.4.3 Strukturmuster ... 24
2.4.3.1. Vererbung ... 24
2.4.3.2. Wertetypen ... 25
2.4.4 Weitere Muster und Prinzipien ... 25
3 Bewertungskriterien ... 26
3.1 Mapping-Features ... 26
3.2 Performance ... 26
3.2.1 Fetching-Strategien ... 27
3.2.2 Batch-Operationen ... 27
3.2.3 Caching ... 27
3.3 Mapping-Richtungen ... 28
3.4 Datenbankunterstützung ... 28
3.5 Datenbankaustausch ... 29
3.6 Transaktionen ... 30
3.7 Nebenläufigkeit ... 30
3.8 Schemaänderungen ... 31
3.9 Abfragemittel ... 31
4 Konzeption und Realisierung der Referenzanwendung ... 32
4.1 Spezifikation ... 32
4.1.1 Kurzbeschreibung des Systems ... 32
4.1.2 Anwendungsfälle ... 32
4.1.3 Datenmodell ... 33
4.2 Fachliche Architektur ... 34
4.2.1 Komponenten und Schnittstellen ... 34
4.2.2 Einfluss der OR-Mapper auf die Architektur ... 36
4.3 Realisierung ... 36
4.3.1 Vorgehen und Erfahrungen ... 36
4.3.2 Verwendete Werkzeuge ... 37
5 Evaluierung ... 38
5.1 NHibernate ... 38
5.1.1 Kurzbeschreibung ... 38

5.1.2 Evaluation anhand Kriterien 40
5.2 Entity Framework 49
5.2.1 Kurzbeschreibung 49
5.2.2 Evaluation anhand Kriterien 51
5.3 ActiveJDBC 58
5.3.1 Kurzbeschreibung 58
5.3.2 Evaluation anhand Kriterien 58
6 Fazit 65
6.1 Ergebnisse 65
6.2 Ausblick 66
Glossar 67
Literaturverzeichnis 68
Anhang A 71
Anhang B 74

1 Einführung

1.1 Problemstellung und Zielsetzung

Die objektorientierte Programmierung ist ein Programmierparadigma, in welchem Objekte bzw. Entitäten eine wichtige Rolle spielen. In objektorientierten Programmiersprachen werden diese durch Instanzen von Klassen repräsentiert, die unter anderem Attribute enthalten, welche die Eigenschaften des Objektes abbilden.

Zur Speicherung (Persistierung) dieser Objekte werden vor allem relationale Datenbanken verwendet, die auf dem relationalen Datenmodell basieren, welches in den 1970er Jahren von Edgar F. Codd entwickelt wurde. In diesem Modell werden Daten in Tabellen gespeichert, die wiederum miteinander in Relation stehen können.

Das Problem ist nun, dass sich Objekte nicht ohne weiteres auf relationale Datenbanken abbilden lassen, was auch als Impedance Mismatch bezeichnet wird. Die Abbildung per Hand ist zumeist sehr aufwändig, vor allem bei komplexen Datenmodellen. Zum Persistieren werden deshalb oft objektrelationale Mapper eingesetzt, welche die Abbildung von Objekten auf Tabellen und umgekehrt kapseln und vom verwendeten Datenbanksystem abstrahieren.

Das Ziel dieser Bachelorarbeit ist der Vergleich und die Bewertung von NHibernate für .NET, das Entity Framework für .NET und ActiveJDBC für Java.

Dazu werden in **Kapitel 2** Grundlagen zur objektorientierten Programmierung und der relationalen Datenbanken sowie des daraus entstehenden Impedance Mismatch erläutert, Funktion und Aufgaben der objektrelationalen Mapper beschrieben und bewährte Muster und Prinzipien erläutert, die in Bezug auf die objektrelationale Abbildung entstanden sind.

In **Kapitel 3** werden Bewertungskriterien erarbeitet, anhand derer sich die OR-Mapper objektiv miteinander vergleichen lassen sollen.

Kapitel 4 beschäftigt sich mit der Konzeption und Realisierung einer Referenzanwendung unter Zuhilfenahme der OR-Mapper, die als Bewertungsgrundlage der in Kapitel 3 aufgestellten Kriterien dient.

Kapitel 5 widmet sich der Evaluation der OR-Mapper anhand der zuvor erarbeiteten Bewertungskriterien, in **Kapitel 6** werden abschließend die Ergebnisse der Evaluation diskutiert.

1.2 Abgrenzungen

Die in Kapitel 4 konzeptionierte Referenzanwendung dient lediglich zu explorativen Zwecken und hat nicht den Anspruch, ein System für den Produktiveinsatz zu sein. Vielmehr sind Anwendungsfälle so gewählt, dass Konzepte der objektorientierten Programmierung zum Einsatz kommen und Techniken bzw. Konzepte der OR-Mapper veranschaulicht werden.

2 Grundlagen

In diesem Kapitel sollen die Grundlagen erläutert werden, die für das Verständnis und für den Einsatz objektrelationaler Mapper relevant sind.

Dazu soll zunächst ein Überblick über die Konzepte der objektorientierten Programmierung und der relationalen Datenbanken verschafft werden. Darauf aufbauend soll der Impedance Mismatch genauer erläutert und Konzepte und Nutzen von objektrelationalen Mappern dargestellt werden. Am Ende dieses Kapitels werden gängige Architektur- und Entwurfsmuster erläutert, die in Zusammenhang mit objektrelationaler Abbildung entstanden sind und sich durchgesetzt haben.

2.1 Objektorientierte Programmierung

Die objektorientierte Programmierung ist ein Programmierparadigma, das auf der Objektorientierung basiert. Diese befasst sich mit der Modellierung und Abbildung von Objekten in Software zur Beherrschung der Komplexität großer Softwaresysteme (vgl. [Lahres u.a. 2016]).

Sie baut auf dem objektorientierten Design auf, in welchem die Objekte und ihre Beziehungen zueinander modelliert werden.

2.1.1 Objekte und Klassen

Objekte

Eine zentrale Rolle bei der objektorientierten Programmierung spielt der Begriff des Objektes. Ein Objekt ist eine konkrete Ausprägung eines Gegenstandes von Interesse, den man in Software abbilden möchte. Ein Objekt kann beispielsweise eine Person, ein Fahrzeug oder eine Bestellung sein. Ein wichtiges Merkmal von ihnen ist, dass sie Attribute besitzen, die ihren Zustand repräsentieren. Außerdem sind sie in der Lage, Nachrichten zu empfangen, mit denen ihr Zustand abgerufen oder geändert werden kann.

Klassen

Klassen stellen ein Sprachkonstrukt objektorientierter Programmiersprachen dar und definieren Struktur und Verhalten von Objekten. Sie dienen außerdem zur Erzeugung konkreter Objekte bzw. Instanzen zur Laufzeit eines Programmes.

2.1.2 Assoziationen

In Programmen stehen Objekte meist mit anderen Objekten in Beziehung. Durch Assoziationen zwischen Klassen wird festgelegt, welche Beziehungen zwischen Objekten möglich sind und in welcher Ausprägung. In der UML werden Assoziationen durch eine Linie zwischen beiden Klassen dargestellt.

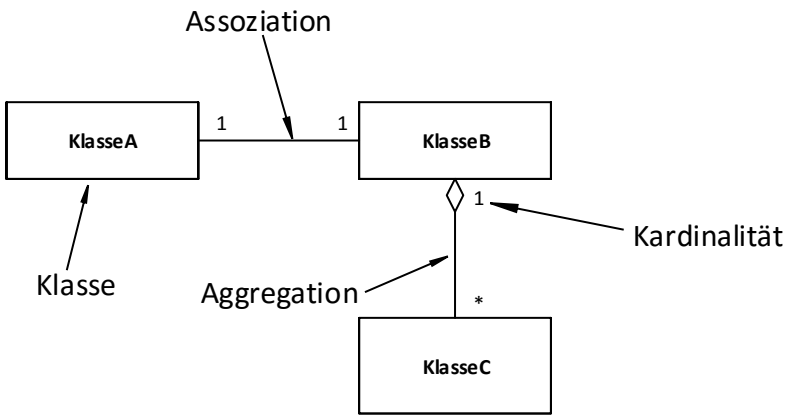

Abbildung 1: Assoziationen und Aggregation in UML

Assoziationen werden in zwei Arten eingeteilt, in Kann-Beziehungen und in Muss-Beziehungen. Eine Kann-Beziehung gibt an, dass zwischen Objekten eine Beziehung existieren kann, aber nicht unbedingt muss, wohingegen in einer Muss-Beziehung ein Objekt zu mindestens einem anderen Objekt in Beziehung stehen muss. Am Beispiel von Kunden und Bestellungen betrachtet existiert vom Kunden eine Kann-Beziehung zu Bestellungen, eine Bestellung hingegen muss aber eindeutig einem Kunden zugeordnet werden können, ist somit also eine Muss-Beziehung.

Kardinalitäten

Kardinalitäten (auch Multiplizitäten) geben an, mit wie vielen Objekten einer anderen Klasse ein Objekt in Beziehung stehen kann. Aus ihnen folgt unmittelbar, ob es sich um eine Kann- oder Muss-Beziehung handelt. Ein Objekt kann mit genau einem anderen Objekt in Beziehung stehen (1), mit mindestens einem (1..*), mit höchstens einem (0..1), oder beliebig vielen (*). Die ersten beiden Kardinalitäten geben Muss-Beziehungen an, die letzten beiden dagegen Kann-Beziehungen.

Aggregation

Die Aggregation ist eine Sonderform der Assoziation und drückt eine *Teile-Ganze* bzw. eine *has-a*-Beziehung aus. Sie sagt aus, dass ein Objekt Teil (Komponente) eines anderen Objektes (Aggregat) ist. Wenn zusätzlich die Existenz eines Teils vom Ganzen abhängt, also eine Komponente ohne sein Aggregat nicht existieren kann, wird die Aggregation als echte Aggregation oder auch als Komposition bezeichnet. In der UML wird dies durch eine Raute auf der Seite des Aggregates dargestellt (s. Abbildung 1).

Navigierbarkeit

Als Navigierbarkeit bezeichnet man die Möglichkeit eines Objektes, auf seine assoziierten Objekte zugreifen zu können. Wenn zwischen zwei assoziierten Objekten jeweils zum anderen Objekt navigiert werden kann, wird die Assoziation als bidirektional bezeichnet, wenn dagegen die Navigation nur in eine Richtung möglich ist, ist die Assoziation unidirektional.

2.1.3 Vererbung und Polymorphie

Ein wichtiges Konzept der objektorientierten Programmierung ist die Vererbung. Mit ihr lassen sich *Ist ein* bzw. *is-a* Beziehungen modellieren.

Vererbung dient in der objektorientierten Programmierung dazu, gemeinsame Attribute und Methoden verschiedener Klassen in einer Oberklasse zusammenzufassen (Generalisierung) bzw. aus einer Klasse Unterklassen abzuleiten, die zusätzliche Attribute und Methoden aufweisen (Spezialisierung).

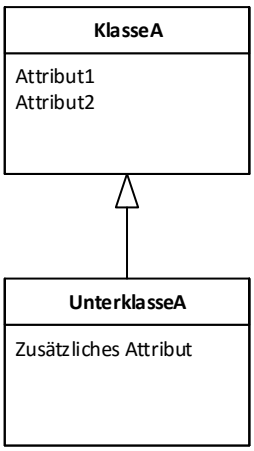

Abbildung 2: Vererbung in UML

Dies hat den Vorteil, dass Attribute und vor allem Methoden, die mehrere Klassen gemeinsam haben, nur einmal implementiert werden müssen und somit duplizierter Code reduziert und die Wartbarkeit des Codes erhöht wird.

Wenn eine Oberklasse lediglich gemeinsame Attribute und Methoden enthält, aber zur Laufzeit keine Objekte dieser Klasse erzeugt werden sollen, so wird diese Klasse als abstrakt bezeichnet, ansonsten handelt es sich um eine konkrete Klasse.

Polymorphie

Polymorphie (griechisch für Vielgestaltigkeit) ist ein weiteres Konzept der objektorientierten Programmierung und tritt in Zusammenhang mit Schnittstellen (Interfaces) und Vererbung

auf. Die Polymorphie sagt aus, dass bei einem Methodenaufruf einer Variablen oder eines Parameters ihr konkreter Typ nicht bekannt sein muss, sondern erst zur Laufzeit durch dynamische Bindung ermittelt wird.

Dies wird zum einen dadurch ermöglicht, da alle Klassen, die ein Interface implementieren, auch dessen Methoden implementieren müssen, und zum anderen dadurch, dass eine Unterklasse Methoden aus der Oberklasse überschreiben kann, also deren Funktionalität neu implementiert. Aus beidem folgt, dass eine Methode, die in einem Interface oder in einer Oberklasse definiert wurde, mehrmals implementiert sein kann.

2.1.4 Object Lifecycle

Der Object Lifecycle (oder Objekt-Lebenszyklus) beschreibt die Zeit zwischen der Erstellung eines Objektes und seiner Zerstörung. Im Folgenden werden grob die Phasen des Lebenszyklus für ein Objekt beschrieben (vgl. [Lahres u.a. 2016]):

- Speicherallokation: Die Größe des Objektes wird berechnet und entsprechender Speicherplatz reserviert.
- Erzeugung: Mithilfe des Konstruktors wird ein neues Objekt (Instanz) einer Klasse erzeugt und seine Attribute initialisiert. Die Klasse dient dafür als Schablone. Die Referenz auf das Objekt wird in einer Variablen gespeichert.
- Nutzung: Das Objekt kann genutzt werden, indem Nachrichten an ihn geschickt werden (Methodenaufrufe).
- Zerstörung: Wenn das Objekt nicht mehr referenziert wird, z.B. am Ende einer Funktion (lokale Variable), wird es gelöscht und vom Garbage Collector der zuvor allokierte Speicherplatz für das Objekt wieder freigegeben.

2.2 Relationale Datenbanken

Nachfolgend soll das relationale Datenmodell, die Anfragesprache SQL, Transaktionen und Datenbankschnittstellen näher erläutert werden.

2.2.1 Relationales Datenmodell

Das relationale Datenmodell ist ein Modell, in welchem Daten logisch in Form von Relationen abgebildet werden. Dieses Modell wurde erstmals in den 1970 von Edgar F. Codd beschrieben ([Codd 1970]) und ist sehr stark an das mathematische Konzept der Relationen angelehnt.

Mathematisch gesehen ist eine Relation eine Teilmenge eines kartesischen Produktes mehrerer Mengen (vgl. [Faeskorn et. al. 2007). Im Sinne des relationalen Datenmodells kann sie aber auch anschaulich als Tabelle verstanden werden. Die Zeilen beschreiben dabei Tupel bzw. Datensätze, die Spalten beschreiben einzelne Elemente des Tupels bzw. die Attribute

des Datensatzes. Genau wie in der Mengenlehre Relationen nur Tupel mit Elementen aus den in Relation stehenden Mengen enthalten dürfen, müssen die Werte der Tabellenspalten in einem bestimmten Wertebereich (Domäne) liegen, dessen Werte zudem atomar sein müssen, d.h. die Werte dürfen keine strukturierten Daten sein (vgl. [Kemper u.a. 2006]).

Das Ziel des relationalen Datenmodells ist es, Daten auf logischer Ebene zu strukturieren, unabhängig von ihrer internen Repräsentation auf sie zugreifen zu können und sie möglichst frei von Redundanzen zu halten (vgl. [Codd 1970]).

Das relationale Datenmodell gilt als mathematisch fundiert und hat sich in der Datenbankentwicklung als Standard etabliert (vgl. [Wikipedia RDB]). Datenbanksysteme, die auf dem relationalen Datenmodell basieren, haben momentan mit Abstand die größte Popularität im Vergleich zu NoSQL oder anderen Technologien (Stand: 2016, vgl. Abbildung 3).

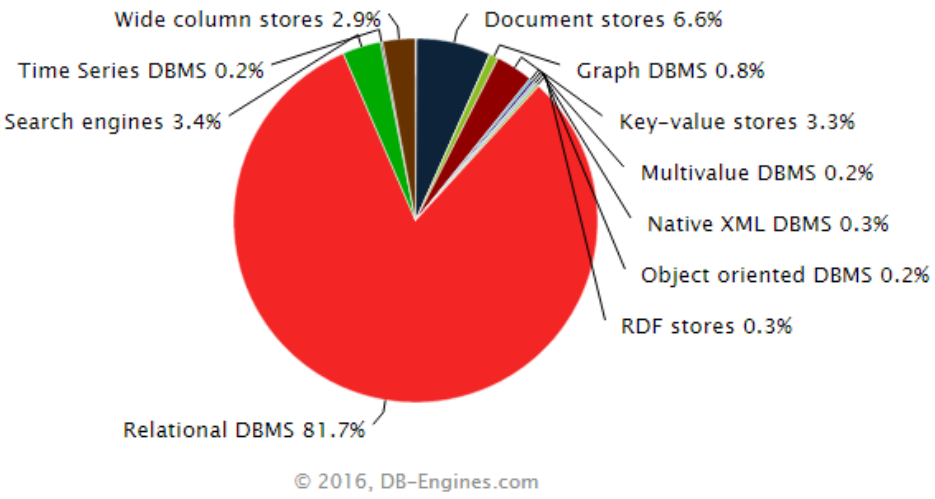

Abbildung 3: Popularität von Datenbanksystemen nach Kategorie[1]

2.2.2 Schlüssel

Primärschlüssel

Als Primärschlüssel bezeichnet man eine Teilmenge von Attributen in einer Tabelle, die zur eindeutigen Identifizierung eines Datensatzes dienen. Primärschlüssel können natürliche Attribute sein, zum Beispiel Kontonummern oder KFZ-Kennzeichen, oder auch künstlich erzeugt werden.

[1] Quelle: http://db-engines.com/en/ranking_categories (letzter Zugriff: 30.01.2016)

Künstliche Primärschlüssel werden auch als Surrogatschlüssel bezeichnet und werden meist vom Datenbanksystem vergeben. Sie stellen die einfachste Form von Primärschlüsseln dar, da sie immer eindeutig sind, nur aus einer Spalte bestehen und somit über Fremdschlüssel einfacher referenziert werden können.

Fremdschlüssel

Fremdschlüssel sind Attribute in einer Tabelle, die auf den Primärschlüssel eines anderen Datensatzes verweisen, der entweder zur selben oder zu einer anderen Tabelle gehört. Diese Verweise drücken Beziehungen zwischen Datensätzen aus.

Durch Referenzielle Integrität wird zudem sichergestellt, dass Fremdschlüssel nur auf existierende Datensätze verweisen und Datensätze nicht gelöscht werden dürfen, solange sein Primärschlüssel durch Fremdschlüssel referenziert wird.

2.2.3 Beziehungen

Im relationalen Datenmodell können mehrere Relationen bzw. Tabellen miteinander in Beziehung stehen, die über Fremdschlüssel abgebildet werden. In relationalen Datenbanken treten Beziehungen in folgenden Formen auf:

1:1-Beziehung

Die 1:1-Beziehung beschreibt eine Beziehung eines Datensatzes einer Tabelle A zu genau einem Datensatz einer anderen Tabelle B. Diese Beziehung wird über eine Fremdschlüsselspalte in einer der beiden Tabellen A oder B realisiert, die auf den Primärschlüssel der anderen Tabelle verweist.

1:n-Beziehung

Die 1:n-Beziehung beschreibt eine Beziehung eines Datensatzes einer Tabelle A zu beliebig vielen Datensätzen einer anderen Tabelle B. In diesem Fall wird die Beziehung über eine Fremdschlüsselspalte in der Tabelle B realisiert, die auf den Primärschlüssel der Tabelle A verweist.

n:m- Beziehung

Die n:m-Beziehung beschreibt eine Beziehung zwischen zwei Tabellen A und B, in der die Datensätze beider Tabellen beliebig vielen Datensätzen der jeweils anderen Tabelle zugeordnet sein können. In diesem Fall wird die Beziehung über eine Zwischentabelle mit zwei Fremdschlüsseln realisiert, die jeweils auf den Primärschlüssel der Tabelle A und B verweisen.

2.2.4 SQL

SQL (Structured Query Language bzw. Strukturierte Abfragesprache) ist eine Datenbanksprache zur Erzeugung, Manipulation und Abfrage relationaler Datenbanken. Sie

ist ein ISO- und ANSI-Standard und wird von den gängigen relationalen Datenbanksystemen als Abfragesprache genutzt (vgl. [Wikipedia SQL]).

Die Sprache basiert auf der relationalen Algebra sowie auf dem relationalen Kalkül (vgl. [Kemper u.a. 2006]). Ersteres beschreibt eine formale Sprache, mit der sich Anfragen an Datenbanken stellen lassen. Der relationale Kalkül dagegen bildet einen deklarativen Ansatz zur Beschreibung von Ergebnismengen. SQL gilt damit als deklarativ, da angegeben wird, welche Ergebnismenge man erwartet, aber keinen Algorithmus angibt, nach dem die Datenbank durchsucht wird.

Im Folgenden werden einige der wichtigsten SQL-Befehle erläutert:

Selektion

```
SELECT * FROM Kunden
```

Selektiert alle Zeilen der Tabelle *Kunde*.

Selektion mit Where-Klausel

```
SELECT * FROM Kunden WHERE Vorname = 'Max' AND Nachname = 'Mustermann'
```

Selektiert alle Zeilen der Tabelle *Kunden*, in denen der Vorname den Wert „Max“ und der Nachname den Wert „Mustermann“ hat.

Projektion

```
SELECT Vorname, Nachname, Geburtsdatum FROM Kunden
```

Gibt eine Tabelle mit den Spalten *Vorname, Nachname* und *Geburtsdatum* zurück.

Umbenennung

```
SELECT Vorname, Adresse as Rechnungsadresse FROM Kunden
```

Gibt eine Tabelle mit den Spalten *Vorname* und *Adresse* aus der Tabelle *Kunden* zurück, wobei der Spaltenname *Adresse* in *Rechnungsadresse* umbenannt wurde.

Joins

```
SELECT Kunden.Kunden_ID, Kunden.Vorname, Bestellungen.Bestellungsnummer
FROM Kunden INNER JOIN Bestellungen on Kunden.Kunden_ID =
Bestellungen.Kunden_ID
WHERE Kunden.Vorname LIKE 'A%'
```

Verknüpft die beiden Tabellen *Kunden* und *Bestellungen* über die Kunden-ID und gibt eine Tabelle mit den Spalten *Kunden-ID, Vorname* und *Bestellungsnummer* aller Kunden zurück, deren Vorname mit „A“ beginnt. Der Primärschlüssel der Tabelle *Kunde* sowie der Fremdschlüssel in der Tabelle *Bestellungen* haben dabei den gleichen Wert. Diese Art der Verknüpfung wird auch als Inner-Join bezeichnet.

Wenn zusätzlich Datensätze ohne Fremdschlüssel bzw. mit nicht referenzierten Primärschlüsseln berücksichtigt werden sollen, müssen die Tabellen über ein Outer-Join verknüpft werden.

Datensatz einfügen

```
INSERT INTO Kunden (Vorname, Nachname) VALUES ('Max', 'Mustermann')
```

Fügt einen neuen Datensatz mit *Vorname = „Max“* und *Nachname = „Mustermann“* in die Tabelle *Kunden* ein.

Datensatz aktualisieren

```
UPDATE Kunden SET Vorname = 'Neuer Name' WHERE Kunden_ID = 123
```

Setzt für alle Zeilen der Tabelle *Kunden*, welche die Bedingung *„Kunden_ID = 123“* erfüllen, den Wert der Spalte *Vorname* auf „Neuer Name“.

Datensatz löschen

```
DELETE FROM Kunden WHERE Kunden_ID = 321
```

Löscht alle Zeilen der Tabelle *Kunden*, welche die Bedingung *„Kunden_ID = 321“* erfüllen.

Datensätze zählen

```
SELECT COUNT(*) FROM Kunden
```

Gibt die Anzahl der Zeilen in der Tabelle *Kunden* zurück. *COUNT* zählt zu den Aggregatsfunktionen, die auf Spalten ausgeführt werden. Andere Aggregatsfunktionen sind z.B. *SUM* zum Addieren der Werte einer Spalte, *AVG* zur Berechnung des Durchschnittswertes oder *MIN* bzw. *MAX* zur Bestimmung des minimalen bzw. maximalen Werts in einer Spalte.

2.2.5 Transaktionen

Als Transaktion bezeichnet man eine Sequenz von Datenbankoperationen, die als eine logische Einheit verstanden wird und somit vollständig, ununterbrechbar und fehlerfrei ausgeführt werden muss.

Ein Beispiel für eine Transaktion ist eine Überweisung eines Geldbetrags g von Konto A nach Konto B (vgl. [Kemper u.a. 2006]):

1. read(A, a): lese den aktuellen Kontostand von Konto A und lege ihn in a ab
2. a := a – g: vermindere a um den Betrag g
3. write(A, a): schreibe a als neuen Kontostand für Konto A in die Datenbank
4. read(B, b): lesen den aktuellen Kontostand von Konto B und lege ihn in B ab

5. b := b + g: erhöhe b um den Betrag g
6. write(B, b): schreibe b als neuen Kontostand für Konto b in die Datenbank

Für diese Transaktion muss sichergestellt werden, dass entweder alle Operationen oder keine davon ausgeführt wird (Atomicity), die Datenbank nach Beendigung der Transaktion in einem konsistenten Zustand hinterlassen wird (Consistency), die Transaktion nicht durch nebenläufige Transaktionen beeinflusst wird (Isolation) und alle Änderungen nach Beendigung der Transaktion dauerhaft sind (Durability). Diese vier Eigenschaften werden auch als ACID-Eigenschaften bezeichnet.

Atomarität der Transaktionen wird durch die Transaktionsklammer sichergestellt, die aus den Befehlen *Begin*, *Commit* und *Rollback* besteht. *Begin* ist der Befehl zum Starten einer Transaktion, mit *Commit* werden alle Änderungen nach der *Begin*-Klammer dauerhaft festgeschrieben. Wenn eine Transaktion hingegen aus einem Grund nicht korrekt ausgeführt werden kann, müssen alle Änderungen innerhalb der Transaktionen durch ein *Rollback* rückgängig gemacht werden. Dadurch wird garantiert, dass am Ende der Transaktion entweder alle Befehle korrekt ausgeführt wurden oder die Transaktion wirkungslos bleibt.

Um die Isolation von nebenläufig laufenden Transaktionen sicherzustellen, kommen Lese- und Schreibsperren zum Einsatz. Da vollständige Isolation jedoch nur gewährleistet ist, wenn die Transaktionen serialisiert[2] ausgeführt werden, dadurch aber nicht die Vorteile der Nebenläufigkeit ausnutzt werden, kann das Isolations-Kriterium aufgeweicht und in sogenannte Isolationsebenen unterteilt werden. Mit diesen wird festgelegt, wann Lese- und Schreibsperren zum Einsatz kommen (vgl. [Kuaté u.a. 2009]). Das hat aber auch zur Folge, dass bei nebenläufigen Transaktionen folgende Anomalien auftreten können:

1. Lost Update: Zwei Transaktionen ändern einen Datensatz, die Änderungen der ersten Transaktion werden dabei von der zweiten Transaktion überschrieben.
2. Dirty Read: Eine Transaktion liest Daten einer anderen nicht abgeschlossenen Transaktion (die nach dem Lesevorgang eventuell zurückgesetzt wird).
3. Non-repeatable Read: Mehrmaliges Auslesen desselben Datensatzes, der durch eine andere Transaktion verändert wird, liefert jeweils andere Ergebnisse.
4. Phantom Read: Zwischen zwei Lesevorgängen in einer Transaktion werden Datensätze von einer anderen Transaktion verändert, hinzugefügt oder gelöscht. Die beiden Lesevorgänge der ersten Transaktion beziehen sich somit auf verschiedene Ergebnismengen, was zu Inkonsistenzen führt.

Durch die Wahl der Isolationsebene wird festgelegt, welche dieser Anomalien beseitigt werden, aber auch welche potenziell auftreten können. Bei *Read Uncommited* treten alle Anomalien außer *Lost Updates* auf, *Read Commited* verhindert zusätzlich *Dirty Reads*, mit *Repeatable Read* sind nur noch *Phantom Reads* möglich, bei *Serializable* tritt keine der Anomalien mehr auf. Höhere Isolation geht bei einem hohen Grad an Nebenläufigkeit aber

[2] Im Sinne von hintereinander

auch zu Lasten der Performance, es muss also eine Abwägung zwischen Isolation und Performance stattfinden.

2.2.6 Datenbankschnittstellen

Datenbankschnittstellen sind Programmierschnittstellen (APIs) und ermöglichen es, über ein Anwendungsprogramm auf Datenbanken zuzugreifen, ohne die technischen Details wie Verbindungsaufbau zur Datenbank zu kennen. Als wichtige Schnittstellen seien hier ADO.NET für das .NET-Framework sowie JDBC für Java genannt.

Innerhalb von Anwendungsprogrammen können mithilfe der Schnittstellen SQL-Anfragen an die Datenbank gestellt werden und daraus folgend Daten gelesen bzw. geschrieben werden oder auch Datenbanktransaktionen ausgeführt werden.

2.3 Objektrelationale Abbildung

Die objektrelationale Abbildung beschäftigt sich mit der Abbildung von Objekten in Software auf relationale Datenbanken und umgekehrt. Im Folgenden werden die Probleme erläutert, die in Verbindung mit OR-Mapping auftreten können, und die Herangehensweise von Persistenzframeworks an dieses Problem.

2.3.1 Impedance Mismatch

Als Impedance Mismatch wird die Unverträglichkeit von Objekten einer objektorientierten Programmiersprache mit dem relationalen Datenmodell bezeichnet. Er tritt auf, wenn Objekte aus einer objektorientierten Anwendung in relationalen Datenbanken persistiert werden sollen. Dieses Problem hat folgende Ursachen:

1. Strukturdifferenzen

Während das Paradigma der objektorientierten Programmierung auf bewährten Prinzipien des Software-Engineerings basiert, entstand das relationale Datenmodell auf Grundlage der Mengenlehre aus der Mathematik[3]. Ein Problem liegt somit in den unterschiedlichen Datenstrukturen von Objekten und Datensätzen.

- Klassen vs. Relationen: Klassen definieren Struktur und Verhalten von Objekten, welche bei Erzeugung im Hauptspeicher des Programmes abgelegt werden. Relationen dagegen sind Mengen, die aus Tupeln bestehen, welche wiederum Daten repräsentieren.
- Assoziationen vs. Fremdschlüsselbeziehungen: Objekte enthalten Referenzen auf andere Objekte in Referenzvariablen, über die zu anderen Objekten navigiert werden

[3] vgl. http://www.agiledata.org/essays/impedanceMismatch.html (letzter Zugriff: 28.01.2016)

kann. Die Objekte und ihre Beziehungen zueinander werden deshalb meist auch als Graph aufgefasst. In relationalen Datenbanken sind Beziehungen über Fremdschlüssel abgebildet. Erst durch Verknüpfung der Tabellen (Joins) können in Beziehung stehende Datensätze miteinander in Verbindung gebracht werden.

- Datentypen: In Programmiersprachen wie Java oder C# werden Datentypen wie String, Boolean oder Bytes genutzt, relationale Datenbanksysteme nutzen dagegen Datentypen wie *nvarchar, bit und binary* (am Beispiel vom Microsoft SQL-Server[4]).
- Fachliche Datentypen: In fachlichen Datentypen (auch abstrakte Datentypen oder Wertetypen) werden zusammengehörige Attribute mit fachlicher Bedeutung in eigenen Objekten zusammengefasst. Im Gegensatz zu Objekten haben sie keine Identität und sind unveränderbar (immutable). Diese Art von Objekten muss aber dennoch persistierbar sein. Im Gegensatz zu primitiven Datentypen wie Strings oder Integer existiert für Wertetypen jedoch kein entsprechendes Äquivalent in relationalen Datenbanken, da nur atomare Daten erlaubt sind.

2. Vererbung

Die Vererbung ist ein objektorientiertes Konzept und wird im relationalen Datenmodell nicht direkt unterstützt. Für die Abbildung von Vererbungshierarchien müssen deshalb geeignete Ansätze gefunden werden.

3. Identität

Ein Objekt definiert seine Identität über seine Adresse im Hauptspeicher. Das bedeutet, dass zwei Objekte derselben Klasse mit gleichen Werten, aber verschiedenen Speicheradressen, niemals identisch sind. In der objektorientierten Programmierung unterscheidet man deshalb auch zwischen Wertgleichheit für die Gleichheit aller Attribute und Referenzgleichheit für die Gleichheit der Adresse im Hauptspeicher.

Im relationalen Datenmodell hingegen definiert sich die Identität eines Datensatzes ausschließlich über den Primärschlüssel. Zwei Datensätze mit gleichem Primärschlüssel sind somit auch immer identisch. Um diese Art der Identität in objektorientierten Sprachen umzusetzen, wird meist die Equals-Methode der Klasse überschrieben, damit sie auf Wertgleichheit statt auf Referenzgleichheit prüft.

4. Kapselung

In objektorientierten Programmiersprachen kann mit Zugriffsmodifikatoren angegeben werden, auf welche Art Daten eines Objektes gelesen bzw. verändert werden dürfen. In Java oder C# kann zum Beispiel über die Schlüsselwörter *private* und *public* bestimmt werden, ob Attribute nur für das Objekt selbst sichtbar sind, um diese vor unberechtigtem Zugriff zu

[4] https://msdn.microsoft.com/de-de/library/ms187752(v=sql.120).aspx (letzter Zugriff: 31.01.2016)

schützen, oder diese öffentlich sein sollen, und somit von allen anderen Objekten gelesen und verändert werden dürfen.

In relationalen Datenmodell existieren solche Techniken jedoch nicht. Dennoch bieten viele relationale Datenbanken über Rechtevergabe ähnliche Schutzmechanismen an, um Daten vor unberechtigtem Zugriff zu schützen (vgl. [Kemper u.a. 2006]). Über diese können z.B. Rechte wie Operationen auf Tabellen vergeben werden, diese gelten dann jedoch auf Nutzer bzw. Rollenebene.

2.3.2 OR-Mappingframeworks

OR-Mappingframeworks, auch OR-Mapper oder Persistenzframeworks genannt, sind Werkzeuge für objektorientierte Programmiersprachen und haben die Funktion, den Impedance Mismatch zu überbrücken bzw. zu minimieren, indem sie als Schnittstelle zwischen dem objektorientierten Programm und den relationalen Datenbanken fungieren. Dadurch können zum einen die Vorteile der objektorientierten Programmierung ausgenutzt werden und komplexe Anwendungslogik realisiert werden, zum anderen werden aber auch die Vorteile relationaler Datenbanken genutzt.

Im Folgenden werden die wichtigsten Aufgaben der Mapper aufgelistet:

- Abbildung von Klassen, Objekten und Attributen auf Tabellen, Zeilen und Spalten.
- Abbildung von Vererbungshierarchien in der Datenbank.
- Abbildung aller Arten von Beziehungen zwischen Objekten. Dazu zählen 1:1, 1:n und n:m-Beziehungen.
- CRUD: Speichern, Laden, Aktualisieren und Löschen von Objekten in Datenbanken
- Caching, Lazy-/Eager-Loading

Martin Fowler erwähnt in seinem Blogeintrag über OR-Mapper[5], dass OR-Mapper etwa 80-90% der objektrelationalen Abbildung behandeln, die restlichen 10-20% also Wissen über die Funktionsweise von relationalen Datenbanken erfordern. Objektrelationale Mapper sind also nicht als Werkzeuge zu betrachten, die es überflüssig machen, sich mit relationalen Datenbanken zu beschäftigen.

2.4 Architektur- und Entwurfsmuster

Während Architekturmuster der Strukturierung von Software auf oberster Ebene dienen, sind Entwurfsmuster Lösungsansätze für wiederkehrende Teilprobleme, die sich in der Praxis bewährt haben. Auch im Bereich der objektrelationalen Abbildung haben sich Entwurfsmuster bewährt. Im Folgenden werden die wichtigsten Muster sowie weitere

[5] http://martinfowler.com/bliki/OrmHate.html (Letzter Zugriff: 27.01.2016)

Prinzipien erläutert. Die meisten dieser Muster können [Fowler u.a. 2002] entnommen werden.

2.4.1 Architekturmuster

Schichtenarchitektur

Die Schichtenarchitektur beschreibt die Strukturierung von Softwaresystemen in mehrere Schichten bzw. Komponenten, die konzeptionell voneinander getrennt sind und denen jeweils eigene Aufgabenbereiche zugeordnet sind, zu deren Erfüllung sie nur Schnittstellen niedrigerer Schichten nutzen dürfen, aber nicht die der höheren Schichten (vgl. [Fowler u.a. 2002]). Damit soll erreicht werden, dass Abhängigkeiten zwischen Schichten minimiert werden und höhere Schichten einfach ausgewechselt werden können, ohne untere Schichten zu berühren. Außerdem wird dadurch die Verständlichkeit des Systems erhöht.

Eine der bekanntesten Schichtenarchitekturen ist die Aufteilung in die drei Schichten Präsentation, Geschäftslogik und Datenhaltung. Sie wird deshalb auch als Drei-Schichten-Architektur bezeichnet.

- Präsentation: Die Präsentationsschicht ist für die visuelle Darstellung der Daten sowie für die Interaktion mit dem Benutzer verantwortlich. Dies kann über eine grafische Benutzeroberfläche (GUI) geschehen, genauso gut kann aber auch eine Konsole zum Einsatz kommen, die per Kommandozeile bedient wird.
- Geschäftslogik: Die Geschäftslogik fasst jegliche Anwendungslogik des Systems zusammen. Darunter fällt die fachliche, also domänenspezifische Logik wie die Prüfung von Eingabedaten oder Abbildung von Geschäftsprozessen.
- Datenhaltung: Die Datenhaltungsschicht ist dafür verantwortlich, Daten aus Datenquellen zu laden und in diesen zu speichern. So kümmert sich diese Schicht um die Kommunikation mit Datenbanken, Message-Queues oder Nachbarsystemen.

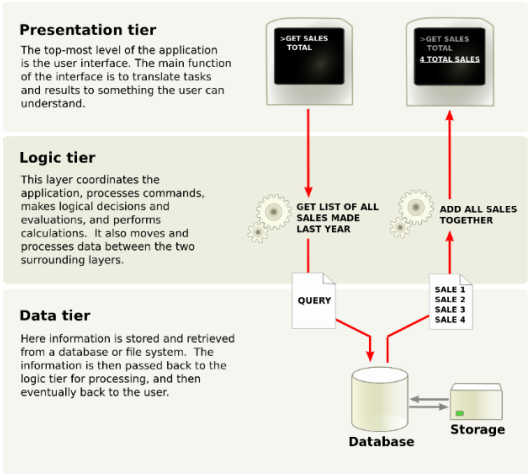

Abbildung 4: klassische Drei-Schichten-Architektur (Quelle: Wikipedia)

Data-Mapper

Das Data-Mapper-Muster ist ein Architekturmuster für objektrelationale Abbildung. Um eine Trennung zwischen Geschäftslogik und Persistenzlogik zu erreichen, kommen Mapper-Objekte zum Einsatz, welche sowohl die Struktur der Objekte als auch ihre Struktur in der Datenbank kennen und die CRUD-Logik für diese Objekte implementieren.

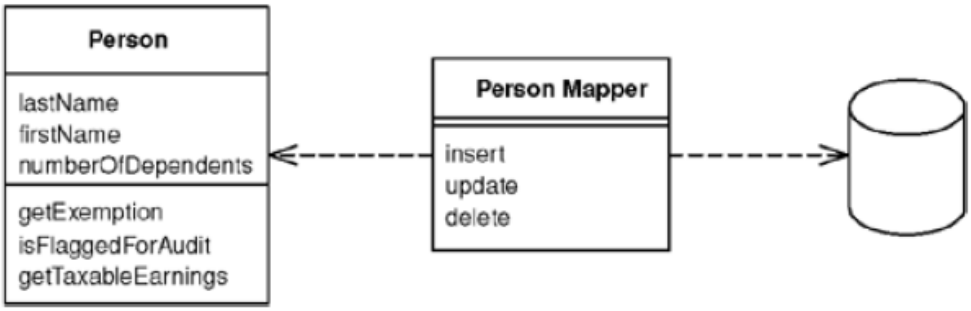

Abbildung 5: Data Mapper für ein Person-Objekt (Quelle: [Fowler u.a. 2002])

Durch Data-Mapper können Objekte persistent gemacht werden, ohne dass sie Wissen über ihre Repräsentation in der Datenbank haben müssen bzw. von der Datenbank abhängig sind. Sie können somit als normale Objekte behandelt werden.

Active-Record

Das Active-Record-Muster ist ebenfalls ein Architekturmuster zur objektrelationalen Abbildung. Im Gegensatz zum Data-Mapper ist bei Active-Record die Persistenzlogik aber in der Objektklasse selbst implementiert. Dafür benötigt diese notwendigerweise Wissen über ihre Repräsentation in der Datenbank und ist folglich an das darunterliegende Datenbankschema gekoppelt.

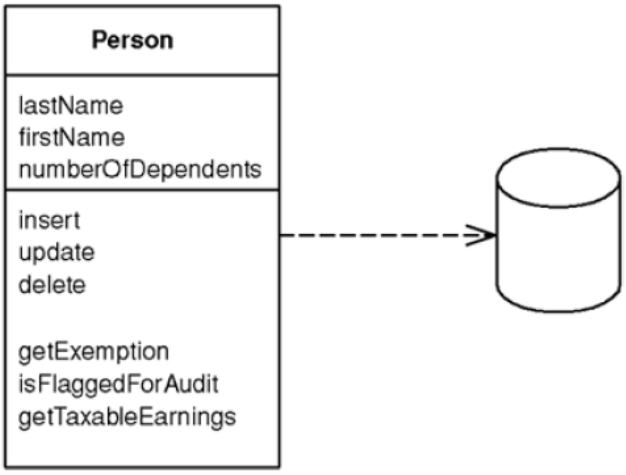

Abbildung 6: Active-Record-Pattern für ein Person-Objekt (Quelle: [Fowler u.a. 2002])

Aus diesem Grund eignet sich Active-Record eher unter der Voraussetzung, dass die Geschäftslogik nicht zu komplex ist, z.B. wenn die Klasse nur CRUD-Logik implementieren muss (vgl. [Fowler u.a. 2002]). Denn durch die Vermischung von Geschäfts- und Persistenzlogik in einer Klasse steht das Active-Record-Muster in direktem Widerspruch zum Single-Responsibility-Prinzip, welches vorschreibt, dass eine Klasse nur für eine fest definierte Aufgabe zuständig ist.

2.4.2 Verhaltensmuster

Unit-of-Work

Das *Unit-of-Work* ist ein Objekt, welches dafür zuständig ist, neue Objekte sowie Änderungen bzw. Löschvorgänge von persistenten Objekten innerhalb einer Geschäftstransaktion zu verfolgen und alle dafür notwendigen Datenbankaufrufe erst am Ende dieses Vorgangs auszuführen, anstatt jeden Aufruf direkt auszuführen. Dadurch werden viele kleine Datenbankaufrufe reduziert, was für die Performance von Vorteil ist. Außerdem können überflüssige Datenbankoperationen vermieden werden, da bei mehrmaliger Änderung eines Objektes nur ein Datenbankaufruf für das Update am Ende der Geschäftstransaktion ausgeführt werden muss.

Lazy-Loading

Lazy-Loading ist ein Verhaltensmuster und beschreibt das Verhalten eines Objektes, assoziierte Objekte erst aus einer Datenquelle zu laden, wenn auf diese zugegriffen wird. Damit soll vermieden werden, dass komplette Objektgraphen aus der Datenbank geladen werden, obwohl nur das Objekt selbst benötigt wird. Dadurch wird in der Regel ein Performancegewinn erzielt.

Allerdings muss auch beachtet werden, dass Lazy-Loading in bestimmten Fällen mehr Datenbankabfragen generieren kann als nötig und somit hohe Kosten entstehen. Folgendes Beispiel soll dies verdeutlichen:

```
foreach (Kunde kunde in kunden)
{
    List<Bestellung> bestellungen = kunde.Bestellungen;
    ...
}
```

Listing 1: Beispiel für das Select N+1-Problem

Es soll durch eine Liste von Kunden iteriert werden, wobei für jeden Kunden auf seine Bestellungen zugegriffen wird. Wenn Lazy-Loading aktiviert ist, wird bei jedem Durchlauf eine Datenbankabfrage generiert, um die Bestellungen für einen Kunden zu laden. Dieses Verhalten ist in der Regel nicht erwünscht und wird auch als *Select N+1*-Problem bezeichnet, da eine Abfrage zum Laden von *n* Objekten ausgeführt wird und darauf folgend *n* Abfragen zum Laden von *n* Assoziationen.

Identity-Map

Identity-Map ist ein Muster, welches sicherstellen soll, dass ein Datensatz zur Laufzeit bzw. innerhalb eines Kontexts nur einmal geladen wird. Dadurch sollen zum einen unnötige Datenbankabfragen, zum anderen aber auch Inkonsistenzen vermieden werden, die dadurch auftreten können, dass zwei oder mehrere Objekte denselben Datensatz repräsentieren, aber Änderungen nur an einem Objekt vorgenommen werden.

Um dies zu vermeiden, wird eine Datenstruktur verwendet, die als Identity-Map bezeichnet wird, da sie die Identität eines Datensatzes (in der Regel den Primärschlüssel) auf das zugehörige Objekt im Hauptspeicher abbildet. Wenn ein Objekt über seinen Primärschlüssel angefragt wird, wird zuerst die Identity-Map durchsucht und bei Vorhandensein eines Eintrages die Referenz auf das Objekt zurückgegeben. Bei Nichtvorhandensein wird das Objekt aus der Datenbank geladen und in der Identity-Map abgelegt.

Die Identity-Map Muster lässt sich einfach realisieren, da Programmiersprachen wie Java oder C# bereits passende Datenstrukturen anbieten (Map bzw. Dictionary), die eine effiziente Suche ermöglichen.

2.4.3 Strukturmuster

2.4.3.1. Vererbung

Table per Hierarchy

Beim Ansatz *Table per Hierarchy*, auch *Single Table Inheritance*, wird eine komplette Klassenhierarchie auf eine Datenbanktabelle abgebildet. Um zu unterscheiden, welcher Klasse ein Datensatz angehört, wird eine Diskriminatorspalte erzeugt, der die Klasse des Objekts angibt, welches durch den Datensatz repräsentiert wird.

Ein Merkmal dieser Abbildungsstrategie ist, dass die Tabelle die Vereinigungsmenge aller Attribute der zur Hierarchie zugehörigen Klassen als Spalten enthält. Folglich können die Tabellen bei großen Hierarchien ziemlich groß werden und viele *NULL*-Werte in den Feldern enthalten, da alle nicht zur Klasse gehörigen Attribute NULL sein müssen. Diese NULL-Felder muss das DBMS dennoch verwalten. Außerdem werden *NOT-NULL*-Constraints damit unmöglich bzw. muss diese Einschränkung anderweitig sichergestellt werden.

Table per Class

Bei diesem Ansatz wird für jede Klasse einer Hierarchie eine eigene Tabelle erzeugt. Eine Unterklasse kann somit auf mehrere Tabellen aufgeteilt sein, was Join-Operationen für das Auslesen und Zusammenfügen von Objekten zur Folge hat.

Table per Concrete Type

Bei diesem Ansatz wird für jede konkrete Klasse eine Tabelle mit allen Attributen erzeugt. Dadurch wird sofort ersichtlich, welche Klasse durch die Tabelle abgebildet wird, außerdem

können Datensätze ohne Join-Operationen ausgelesen werden. Dafür müssen aber Änderungen in Oberklassen wie neue Attribute in allen Tabellen der Unterklassen nachgepflegt werden.

2.4.3.2. Wertetypen

Serialized-LOB

Beim *Serialized-LOB*-Muster werden Wertetypen serialisiert und als Large Object (LOB) in einer Datenbankspalte abgelegt. Wie diese serialisierte Form aussieht, ist nicht vorgeschrieben, wodurch der Wertetyp in seiner Binärform (BLOB) oder in einer langen Zeichenkette (CLOB) in der Datenbank gespeichert werden kann. Wird der Typ wieder aus der Datenbank geladen, muss er deserialisiert, d.h. in seine objektorientierte Form zurückübersetzt werden. Bei diesem Ansatz können jedoch Attribute des Datentyps bei Datenbank-Querys nicht berücksichtigt werden, da die Datenbank die Struktur des Datentyps nicht kennt.

Embedded-Value

Bei *Embedded-Value* wird für jedes Attribut des Wertetyps eine eigene Spalte in der Tabelle der Objektklasse erzeugt, zu die der Wertetyp gehört. Wenn zum Beispiel die Klasse Kunde den Wertetyp Adresse enthält, wird für jedes Feld der Adresse (Straße, Hausnummer, PLZ, Ort usw.) eine eigene Spalte in der Kundentabelle erzeugt und die Felder des Adresse-Typs jeweils in den zugehörigen Spalten gespeichert. Dieser Ansatz ermöglicht es, einzelne Attribute des Wertetyps bei Querys abzufragen, etwa die Filterung von Kunden nach Postleitzahl.

2.4.4 Weitere Muster und Prinzipien

Repository

Das Repository Muster ist ein Entwurfsmuster und kapselt den Zugriff auf persistente Objekte unabhängig von der Datenquelle. Mit ihnen soll ein einheitlicher und objektorientierter Zugriff auf Daten ermöglicht werden, die aus unterschiedlichen Datenquellen wie Datenbanken, Webservices oder XML-Dateien stammen.

Repositorys eignen sich vor allem, wenn intensiv mit Querys gearbeitet wird (vgl. [Fowler u.a. 2002]), da sie die Query-Logik kapseln.

Separation of Concerns

Separation of Concerns ist ein Entwurfsprinzip und beschreibt die Trennung von Zuständigkeiten innerhalb von Software. Zuständigkeiten sind etwa Anwendungslogik, Datenquellenanbindungen, Anbindungen zu Nachbarsystemen oder Logging. Durch Trennung dieser Zuständigkeiten soll die Verständlichkeit und Testbarkeit der einzelnen Komponenten erhöht und die Kopplung zwischen den Komponenten reduziert werden.

3 Bewertungskriterien

Dieses Kapitel beschäftigt sich mit der Bildung und Begründung von Bewertungskriterien, anhand derer die verwendeten OR-Mapper objektiv miteinander verglichen werden sollen.

3.1 Mapping-Features

Primäre Aufgabe von OR-Mappern ist die objektrelationale Abbildung. Dazu zählt die Abbildung von Objekten und Attributen auf Zeilen und Spalten einer Datenbanktabelle, aber auch die Abbildung von Assoziationen zwischen Objekten auf Fremdschlüssel und Zwischentabellen. Darüber hinaus existieren allerdings auch komplexere Mapping-Szenarien wie die Abbildung von Vererbungshierarchien und fachlichen Datentypen auf das relationale Datenmodell. In Kapitel 2.4.3 wurden dazu Ansätze vorgestellt.

Ein weiteres Feature ist Cascading. Cascading beschreibt das Verhalten von Create-, Update- und Delete-Operationen auf Objektgraphen.

- Create: Beim Speichern eines neuen Objekts in der Datenbank werden alle assoziierten Objekte, die nicht persistent sind, ebenfalls in der Datenbank gespeichert.
- Update: Wenn einem persistenten Objekt eine neue Assoziation zu einem nicht-persistenten Objekt hinzugefügt wird, wird beim Update des Objektes das neue Objekt ebenfalls gespeichert.
- Delete: Wenn ein Objekt gelöscht wird, von dem andere Objekte abhängig sind, z.B. bei einer Komposition, werden die abhängigen Objekte mitgelöscht.

Welches Verhalten bezüglich Cascading erwünscht ist, hängt vor allem vom Datenmodell, aber auch von den Anforderungen an das System ab. Deshalb sollten geeignete Möglichkeiten angeboten werden, Cascading-Eigenschaften zu beeinflussen bzw. zu konfigurieren.

3.2 Performance

Eine wichtige Rolle bei der Entwicklung und Nutzung von Softwaresystemen spielt die Performance, also die Leistungsfähigkeit eines Systems. Da Performance in der Informatik ein weitläufiger Begriff ist und von vielen Faktoren wie zum Beispiel Rechenleistung, aber auch von anderen Systemen bzw. Technologien wie verwendeten Datenbanksystemen oder

Middleware abhängt, soll hier nun abgegrenzt werden, anhand welcher Punkte sich die Performance von OR-Mappern vergleichen lässt.

3.2.1 Fetching-Strategien

Fetching ist das Laden von Objekten und ihren Beziehungen aus einer Datenbank. Dafür existieren mehrere Ansätze. Mit Lazy-Loading wurde in Kapitel 2.4.2 bereits ein Ansatz für das verzögerte Laden von assoziierten Objekten vorgestellt.

Im Folgenden soll nun eine weitere Strategie, das Gegenstück zum Lazy-Loading, vorgestellt werden, nämlich das Eager-Loading. Beim Eager-Loading werden alle auf absehbare Zeit benötigten Daten auf einmal geladen, um den Nachteil des *Select N+1* Problems vom Lazy-Loading entgegenzuwirken und einen Performancegewinn zu erzielen.

Unter dem Punkt Fetching-Strategien soll untersucht werden, welche Möglichkeiten zur Konfiguration von Lazy- bzw. Eager-Loading angeboten werden und wie dies realisiert ist.

3.2.2 Batch-Operationen

Eine Batch-Operation im Sinne von Datenbanken beschreibt eine Folge von Datenbankoperationen, die mit einem Datenbankaufruf ausgeführt werden, im Gegensatz zu einer Transaktion aber nicht zwingend eine atomare Operation ist. Ein Beispiel ist etwa das Masseneinfügen neuer Datensätze wie Kreditkartenabrechnungen zum Ende eines Monats, bei der für jeden Kunden alle im Abrechnungsmonat getätigten Umsätze in der Rechnung aufgelistet werden. Datenbanktechnisch betrachtet hat das viele Insert-Befehle in eine Rechnungstabelle zur Folge.

Ein Ansatz dafür ist, für jede einzelne Rechnung einen eigenen Insert-Befehl an die Datenbank zu schicken. Das kostet allerdings eine Round Trip Time (RTT) pro Anfrage, was sich negativ auf die Performance auswirkt, da sich die RTT addieren. Aus diesem Grund bieten Datenbankschnittstellen die Möglichkeit, mehrere SQL-Befehle als Batch an das Datenbanksystem zu schicken, wodurch die RTT pro Anfrage entfällt. Die Datenbankschnittstellen bieten meist auch die Möglichkeit, die Größe des Batchs, d.h. die Anzahl gleichzeitig abgeschickter Befehle, zu konfigurieren.

Unter dem Punkt Batch-Operationen soll untersucht werden, welche Möglichkeiten zum Erstellen und Ausführen von Batch-Operationen durch die OR-Mapper angeboten werden.

3.2.3 Caching

Ein weiterer Mechanismus, der sich positiv auf die Performance auswirkt, ist Caching. Caching im Sinne der objektrelationalen Abbildung hat das Ziel, dass bereits geladene Objekte kein weiteres Mal aus der Datenbank geladen werden, wenn sie ein weiteres Mal angefragt werden. Stattdessen werden Objekte aus dem Cache zurückgegeben, was einen Performancegewinn zur Folge hat.

Unter dem Punkt Caching soll untersucht werden, welche Caching-Strategien verwendet werden.

3.3 Mapping-Richtungen

Bei der Auswahl eines OR-Mappers müssen unter anderem folgende beiden Szenarien betrachtet werden:

1. Es existiert kein Datenbankschema und es ist von Seiten der Datenbankentwickler keines vorgegeben. In diesem Fall ist es wünschenswert, aus Objekt-Mappings im Code bzw. Konfigurationen das Datenbankschema mit Tabellen, Attributen und Fremdschlüsseln generieren zu lassen. Dieser Ansatz wird auch als Forward-Mapping bezeichnet.
2. Es existiert bereits ein Datenbankschema, sei es durch die Entwicklung einer Anwendung mit Zugriff auf eine bereits existierende Datenbank oder durch die Vorgabe des Schemas durch die Datenbankentwickler. In diesem Fall sollte es möglich sein, aus dem existierenden Schema die Objektklassen mit ihren Attributen und Assoziationen zueinander zu generieren. Dieser Ansatz wird als Reverse-Mapping bezeichnet.

Unter diesem Hintergrund soll untersucht werden, welche Möglichkeiten angeboten werden, Objektklassen im Code und das Datenbankschema miteinander zu synchronisieren.

3.4 Datenbankunterstützung

Ein weiteres wichtiges Kriterium stellt die Unterstützung verschiedener Datenbanksysteme dar. Obwohl SQL eine standardisierte Sprache ist, existieren verschiedene SQL-Dialekte, die sich unter anderem auch in der Syntax unterscheiden. Diese verschiedenen Dialekte müssen in OR-Mappern berücksichtigt werden, um mit verschiedenen Datenbanksystemen kompatibel zu sein.

Ein Beispiel für Syntaxunterschiede ist ein SQL-Statement zur Auswahl der obersten n Reihen einer Tabelle[6]:

MySQL

```
SELECT * FROM Kunden LIMIT n
```

Oracle

```
SELECT * FROM Kunden WHERE ROWNUM <= n;
```

[6] vgl. http://www.w3schools.com/sql/sql_top.asp (letzter Zugriff: 28.01.2016)

Microsoft SQL-Server

```
SELECT TOP n * FROM Kunden;
```

Des Weiteren sind auch die von den jeweiligen Datenbankherstellern angebotenen Datenbankschnittstellen relevant, da der Mapper sich mit allen unterstützten Datenbanksystemen verbinden können muss. Während die Java-Datenbankschnittstellen für MySQL oder Oracle die JDBC Spezifikation implementieren, und somit über eine einheitliche API auf verschiedene Datenbanksysteme zugegriffen werden kann, sind in ADO.NET jeweils eigene Klassen für den Zugriff auf verschiedene Datenbanksysteme notwendig.

Als Beispiel sind in folgender Tabelle die Connection-Klassen und Command-Klassen der jeweiligen Datenbankschnittstellen aufgelistet:

Datenbanksystem	Connection-Klasse	Command-Klasse
Microsoft SQL-Server	SqlConnection	SqlCommand
MySQL	MySQLConnection	MySQLCommand
Oracle	OracleConnection	OracleCommand

Diese Unterschiede müssen ebenfalls vom Mapper berücksichtigt werden, um mit verschiedenen Datenbankschnittstellen kompatibel zu sein.

Dieser Punkt soll lediglich eine Auflistung offiziell unterstützter Datenbanksysteme bilden.

3.5 Datenbankaustausch

Nach Einführung eines Anwendungssystems kann es passieren, dass Datenbanken ausgetauscht werden müssen. Ein Beispiel ist etwa die strategische Entscheidung eines Unternehmens, auf ein anderes Datenbanksystem umzusteigen, zum Beispiel vom Microsoft SQL-Server auf MySQL. Ein anderer Grund kann sein, den Lock-In-Effekt zu vermeiden und sich von einem Datenbankhersteller abhängig zu machen.

Inwieweit der eigene Code von einem Datenbankaustausch beeinträchtigt ist, hängt unter anderem davon ab, wie stark er an das darunterliegende Datenbanksystem gekoppelt ist, etwa durch SQL-Syntax.

Um das Kriterium hinsichtlich Datenbankunabhängigkeit von OR-Mappern beurteilen zu können, muss deshalb der Aufwand betrachtet werden, der investiert werden muss, um die zugrundeliegende Datenbank auszutauschen. Es soll nicht betrachtet werden, wie viel Aufwand für die Migration der existierenden Daten nötig ist, da dies nicht in den Aufgabenbereich der OR-Mapper fällt.

3.6 Transaktionen

Datenbankschnittstellen bieten eine API an, mit der Datenbanktransaktionen im Code abgebildet werden können. Da OR-Mapper jedoch vom Datenbanksystem abstrahieren, sollten diese eine eigene Schnittstelle zur Abwicklung von Transaktionen anbieten, etwa das Öffnen (Begin/Open), Committen und Zurücksetzen (Rollback) von Transaktionen.

Unter diesem Punkt soll untersucht werden, wie Transaktionen in den OR-Mappern abgebildet werden.

3.7 Nebenläufigkeit

Da Anwendungssysteme meist von vielen Nutzern gleichzeitig genutzt werden und diese damit auch gleichzeitig auf Datenbestände zugreifen, ist das Risiko von auftretenden Konflikten dementsprechend hoch. Als Beispiel sei ein Hotelreservierungssystem genannt, bei dem mehrere Rezeptionisten Reservierungen für Hotelzimmer hinterlegen können. Wenn zwei oder mehr Rezeptionisten gleichzeitig eine Reservierung hinterlegen wollen, lassen sie sich die Liste der freien Zimmer anzeigen, um dann ein freies Zimmer auszuwählen und für dieses Zimmer eine Reservierung zu tätigen. Das kann im schlimmsten Fall dazu führen, dass alle Rezeptionisten eine Reservierung für dasselbe Zimmer tätigen, was zwangsläufig zu Inkonsistenzen führt.

Um diese nebenläufigen Änderungen und potenziell entstehende Konflikte zu erkennen, gibt es pessimistisches und optimistisches Sperren. Pessimistische Sperren nutzen Sperren in der Datenbank, wodurch nebenläufige Transaktionen die gesperrten Datensätze nicht mehr lesen/ändern dürfen. Wenn eine Transaktion jedoch länger dauert, z.B. durch einen Benutzerdialog (s.o. Beispiel mit Hotelreservierung), geht dies zu Lasten der Nebenläufigkeit, da die Datensätze in der Datenbank solange gesperrt bleiben, bis der Nutzer die Transaktion abschließt und die Datenbank die Sperren wieder freigibt.

Deshalb existiert als Alternative zur Nebenläufigkeitskontrolle das optimistische Sperren. Optimistisches Sperren geht davon aus, dass die meisten Transaktionen Datensätze nur lesen, aber selten ändern. Wenn Datensätze dennoch geändert werden, muss eine Transaktion sicherstellen, dass die ursprünglich gelesenen Daten zwischenzeitlich nicht zuvor durch eine andere Transaktion in der Datenbank geändert wurden. Dafür enthält jedes Objekt als zusätzliches Attribut ein Token, z.B. einen Versionszähler oder einen Zeitstempel, der jedes Mal inkrementiert bzw. aktualisiert wird, wenn der Datensatz in der Datenbank geändert wird. Eine Transaktion muss dann bei der Speicherung des Objekts den Versionszähler/Zeitstempel mit dem aus der Datenbank abgleichen.

Unter diesem Punkt soll untersucht werden, welche Hilfsmittel zur Nebenläufigkeits-kontrolle durch die Mapper angeboten werden.

3.8 Schemaänderungen

Auch während des Produktiveinsatzes einer Anwendung wird diese stetig weiterentwickelt. Es können neue Anforderungen hinzukommen, die neue Objekte bzw. Attribute notwendig machen und somit auch Schemaänderungen in der Datenbank wie neue Tabellen und Spalten nach sich ziehen. Je nach Mapping-Richtung müssen dann aus neu hinzugefügten Klassen und Attributen neue Tabellen und Spalten in der Datenbank erzeugt werden (Forward-Mapping) oder die Schemaänderungen in neuen Code überführt werden (Reverse-Mapping), um das Datenmodell im Code und das Datenbankschema synchron zu halten.

Die OR-Mapper sollen dahingehend untersucht werden, wie hoch der Aufwand für das Hinzufügen neuer Objektklassen bzw. Attribute und damit auch neuer Tabellen und Spalten zu einem bereits existierenden Datenmodell ist.

3.9 Abfragemittel

Abfragemittel stellen wichtige Werkzeuge zum Durchsuchen von Datenbeständen einer Datenbank dar. Ohne OR-Mapper erfolgen Datenbankabfragen über das Schreiben eigener SQL-Strings im Code, die von der Datenbankschnittstelle an das Datenbanksystem geschickt und dort ausgeführt werden. Wenn jedoch ein OR-Mapper zum Einsatz kommt, sollten diese eine möglichst datenbankunabhängige Schnittstelle zum Abfragen der Datenbank anbieten, um die Abhängigkeit zur Datenbank möglichst gering zu halten.

Die OR-Mapper sollen deshalb hinsichtlich der angebotenen Möglichkeiten untersucht werden, die dem Entwickler für Abfragen an die Datenbank zur Verfügung stehen. Diese Mittel sollen hinsichtlich folgender Kriterien untersucht werden:

- Suchen von Datensätzen nach Suchkriterien (where, like etc.)
- Aggregation von Attributen (sum, count, avg etc.)
- Wie lesbar ist dieser Code?
- Wie abhängig ist der Code von der Datenbank?

4 Konzeption und Realisierung der Referenzanwendung

In diesem Kapitel soll ein Anwendungssystem, genauer ein Kundenverwaltungssystem, konzeptioniert und anschließend realisiert werden, welches zum Persistieren der Objekte die zum Vergleich ausgewählten OR-Mapper zu Hilfe nimmt.

Dabei liegt der Fokus nicht bei der Spezifikation und Realisierung einer Präsentationsschicht, sprich einer GUI, sondern viel mehr bei der Formulierung von Anwendungsfällen sowie der objektorientierten Modellierung und Objektpersistenz.

4.1 Spezifikation

Als Referenzanwendung kommt ein fiktives Kundenverwaltungssystem zum Einsatz, in dessen Mittelpunkt Anwendungsfälle stehen, die mithilfe der Objektorientierung in Software abgebildet werden. Die Anwendungsfälle sollen so gewählt werden, dass Mittel der objektorientierten Programmierung wie Vererbung, Komposition und Assoziationstypen, aber auch Mittel der relationalen Datenbanken wie CRUD, Transaktionen und Batch-Operationen zum Einsatz kommen sollen.

An dieser Stelle sei nochmals erwähnt, dass es sich nicht um ein System handelt, das für den Produktiveinsatz entwickelt wird, sondern lediglich zu explorativen Zwecken gilt.

4.1.1 Kurzbeschreibung des Systems

Es soll ein Informationssystem zur Verwaltung von Kunden eines Sportstudios entwickelt werden. Mithilfe des Systems soll es möglich sein, Kundendaten im System zu hinterlegen, Kurse zu erstellen, die von einem Trainer geleitet werden, die Anmeldung und Teilnahme der Kunden an einzelnen Kursen zu verwalten und am Monatsende Rechnungen für die vom Kunden gebuchten Kurse zu verschicken.

4.1.2 Anwendungsfälle

Im Folgenden werden die Anwendungsfälle bzw. Anforderungen an das System aufgelistet:

1. Der System-Administrator soll einen neuen Mitarbeiter mit Vor- und Nachnamen eintragen können. Bei einem Mitarbeiter handelt es sich um einen Rezeptionisten oder einen Trainer. Ebenso soll der Administrator Daten der Mitarbeiter im System einsehen, ändern und löschen können.
2. Ein Rezeptionist soll im System einen neuen Kunden anlegen können. Zu diesem Kunden gehören folgende Kundendaten: Vorname, Nachname, Adresse, Emailadresse, Telefonnummer, Geburtsdatum und ein Kundenstatus.
3. Zum Kundenstatus zählt einer der folgenden Zustände: Basic, Premium, Gekündigt.
4. Ein Rezeptionist soll im System einen neuen Kurs anlegen können. Dieser Kurs hat Titel, Beschreibung, eine maximale Anzahl an Teilnehmern, eine Veranstaltungszeit und einen Kursleiter (Trainer).
5. Ein Rezeptionist kann für einen oder mehrere Kunden einen Kurs buchen. Voraussetzung ist, dass dieser Kurs noch genügend freie Plätze hat.
6. Ein Rezeptionist kann einen oder mehrere Kunden auf einen anderen Kurs umbuchen. Es gelten die gleichen Voraussetzungen wie in Punkt 5.
7. Das System soll zum Ende eines Abrechnungszeitraumes (Monat) Rechnungen für die Kunden erstellen. In dieser Rechnung werden alle in diesem Monat vom Kunden gebuchten Kurse als Rechnungspositionen aufgelistet und in Rechnung gestellt.

4.1.3 Datenmodell

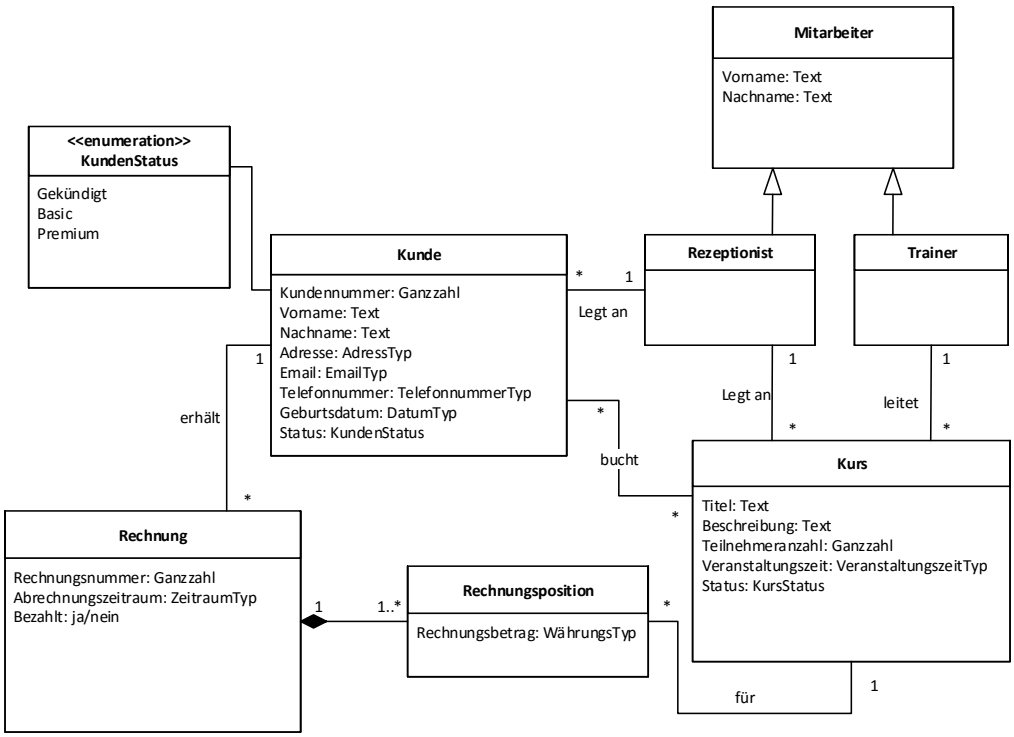

Abbildung 7: Datenmodell der Referenzanwendung

Aus den formulierten Anwendungsfällen lässt sich ein Datenmodell ableiten, wie es in Abbildung 7 dargestellt ist.

Rezeptionisten und Trainer sind unter Mitarbeiter zusammengefasst und bilden somit eine Vererbungshierarchie. Unter den verschiedenen Klassen existieren 1:n- und n:m-Beziehungen, mit der Beziehung zwischen Rechnung und Rechnungsposition wird eine Kompositionsbeziehung dargestellt.

Zusätzlich existieren verschiedene fachliche Datentypen wie Email-Adresse, Veranstaltungszeit und Abrechnungszeitraum, durch den Kundenstatus enthält das Modell eine Aufzählung.

All diese auftretenden Fälle gilt es durch die OR-Mapper in das relationale Modell abzubilden.

4.2 Fachliche Architektur

4.2.1 Komponenten und Schnittstellen

Basis der Architektur der Referenzanwendung ist die von Professor Sarstedt entwickelte Q3/HAW-Referenzarchitektur (vgl. [Sarstedt 2014]).

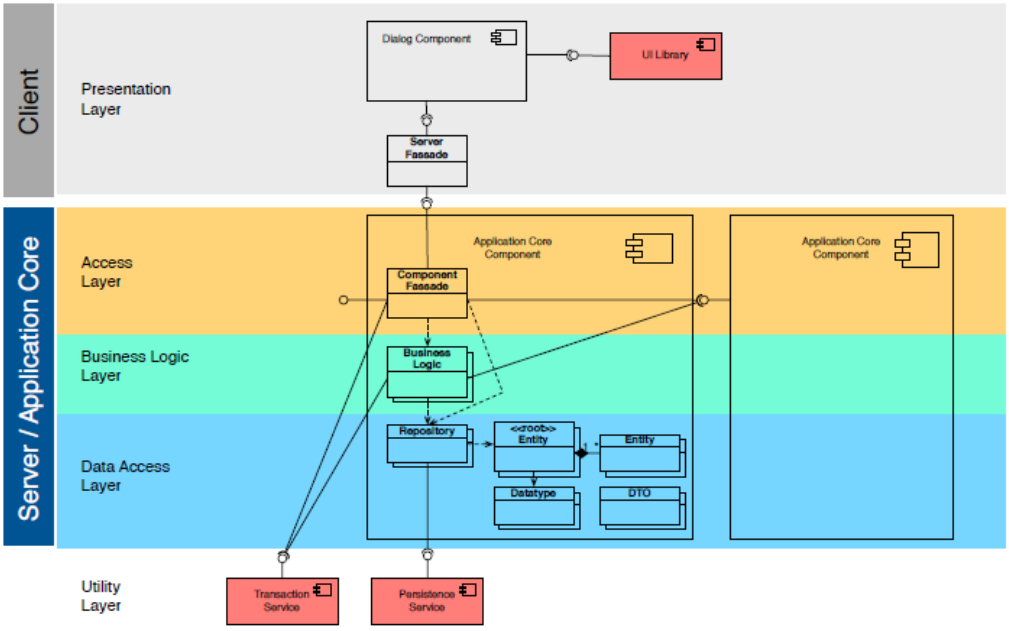

Abbildung 8: Q3/HAW-Referenzarchitektur (Quelle: [Sarstedt 2014])

Diese Architektur sieht auf oberster Ebene eine Aufteilung zwischen dem Anwendungskern, der Präsentationsschicht (Client) und technischen Diensten wie Persistenz und

Transaktionsdiensten vor. Dadurch soll der Anwendungskern frei von technischer Logik gehalten werden und somit nur die Fachlichkeit des Systems vereinigen.

Der Anwendungskern selbst besteht aus mehreren Komponenten, die einzelne Bereiche der Fachlichkeit implementieren und miteinander interagieren, indem sie sowohl Schnittstellen anbieten als auch die der anderen Komponenten nutzen. Jede Komponente des Anwendungskerns ist wiederum in drei Schichten unterteilt:

- **Access-Layer**: dient als Zugriffschicht und enthält Schnittstellen der Komponente sowie eine Fassade, die diese Schnittstellen implementiert. Sie überprüft übergebene Parameter und delegiert an die folgenden beiden Schichten oder an andere Komponenten.
- **Business-Logic-Layer**: implementiert die fachliche Logik der Komponente und nutzt Schnittstellen anderer Komponenten sowie den Data Access Layer.
- **Data-Access-Layer**: enthält die Objektklassen, Wertetypen sowie Repositorys für den Zugriff auf persistente Daten. Repositorys greifen auf Persistenzdienste zu.

In Abbildung 9 sind die Komponenten und ihre Abhängigkeiten zueinander dargestellt. Im Mittelpunkt steht der Anwendungskern, der die fachliche Logik sowie die Objektklassen des Systems enthält.

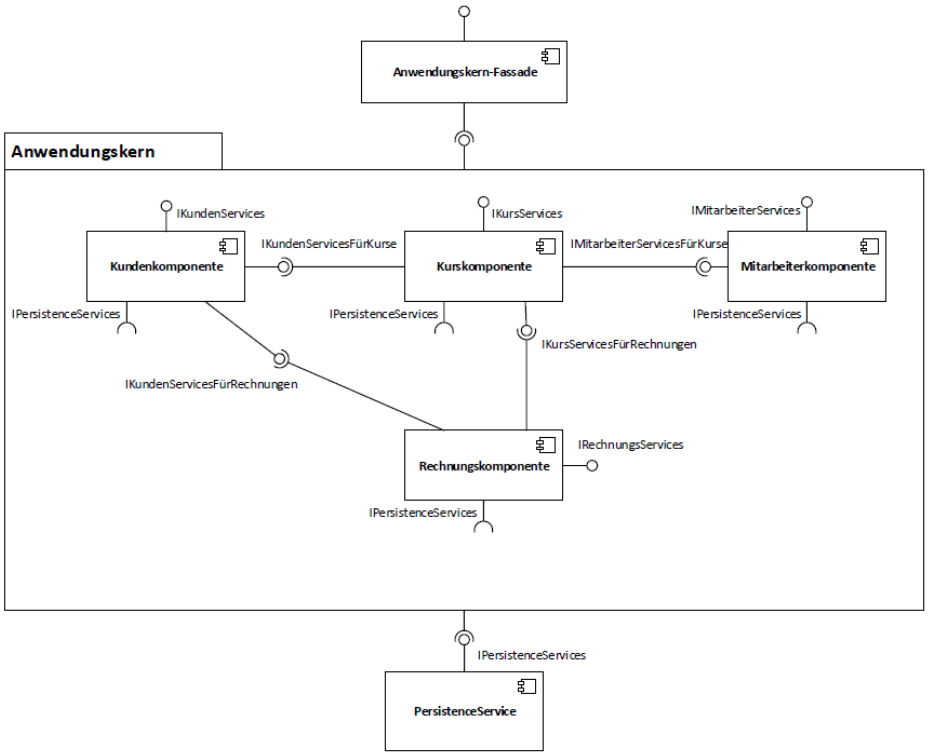

Abbildung 9: Komponentendiagramm der Referenzanwendung

Aus Platzgründen sind die Innensichten der einzelnen fachlichen Komponenten dem Anhang zu entnehmen. Diese sind jedoch alle gemäß der Q3/HAW-Referenzarchitektur aufgebaut.

4.2.2 Einfluss der OR-Mapper auf die Architektur

In einer klassischen Dreischichtenarchitektur sind OR-Mapper in der Persistenzschicht angesiedelt, da sie für die Datenhaltung zuständig sind. Im Falle der ausgewählten Architektur kommen für die Integration des Mappers nur die *PersistenceService*-Schnittstellen infrage. Ein Sonderfall bildet hier aber das Framework ActiveJDBC, da durch Active-Record die Persistenzlogik bereits im Objekt liegt, eine Auslagerung dieser Logik in einen Persistence-Service somit nicht möglich ist. Es könnten hier lediglich Helferklassen angeboten werden, welche technische Funktionalität wie die Konfiguration der Datenbank kapseln.

Einen großen Einfluss auf die Architektur bzw. auf den Entwurf hat außerdem das Muster, welches durch den Mapper implementiert wird. Während NHibernate und Entity Framework Data-Mapper zur Abbildung von Objekten nutzen, implementiert ActiveJDBC das Active-Record-Pattern. Das hat Einfluss darauf, wo das Mapping konfiguriert wird. Im Falle von Active-Record geschieht dies nämlich in der Objektklasse selbst, bei Data-Mappern muss dagegen festgelegt werden, in welchen Komponenten diese Mapping-Konfigurationen liegen sollen.

Einen nicht so großen Einfluss auf die Architektur hat dagegen der Nachteil von Active-Record bezüglich dem Single-Responsibility-Prinzip, da komplexe Geschäftslogik bereits in Business-Logic-Layern implementiert wird und die Persistenzlogik der Klassen in die Oberklasse (siehe 5.3.2) ausgelagert ist.

4.3 Realisierung

4.3.1 Vorgehen und Erfahrungen

Die in 4.2.1 definierten Komponenten wurden der Reihe nach realisiert. Begonnen wurde mit den Komponenten mit den kleinsten Abhängigkeiten. Damit sollte erreicht werden, dass das Zusammenspiel der Komponenten direkt getestet werden konnte. Um die Korrektheit des Systems sicherzustellen, wurden automatisierte Komponententests entwickelt, welche die öffentlichen Schnittstellen der Komponenten testen.

Bei der Implementation in C# hat sich herausgestellt, dass aufgrund der Architektur des Systems das Framework NHibernate relativ schnell durch Entity Framework ersetzt werden konnte. Hier war es lediglich notwendig, die Implementation der *IPersistenceService* und *ITransactionService*-Schnittstellen auszutauschen sowie die Mapping-Konfigurationen für das Entity Framework anzupassen.

Bei ActiveJDBC stellte sich heraus, dass Unit-Tests der Objektklassen ohne Datenbank nicht möglich sind, da die Objekte Datensätze repräsentieren und durch die Abhängigkeit vom ActiveJDBC-Framework nicht als POJOs (Plain Old Java Object) behandelt werden können.

4.3.2 Verwendete Werkzeuge

Im Folgenden werden Werkzeuge aufgelistet, die neben den Mappern selbst für die Realisierung der Referenzanwendung genutzt wurden:

NHibernate und Entity Framework:

Werkzeug	**Beschreibung**	**Version**
Windows 10	Betriebssystem von Microsoft	10 (64-Bit)
Visual Studio	Entwicklungsumgebung (IDE) von Microsoft für die Entwicklung von .NET-Anwendungen	2015 Enterprise (14.0.23107.0)
C#	Programmiersprache zur Entwicklung von .NET-Anwendungen aller Art	5.0
Microsoft SQL-Server	Relationales Datenbanksystem von Microsoft	2012 64-Bit (11.0.5058.0)
MySQL-Server	Relationales Datenbanksystem (Open Source)	5.7.10
Nuget	Paket-Manager für Visual Studio	3.3.0.167

ActiveJDBC:

Werkzeug	**Beschreibung**	**Version**
Windows 10	Betriebssystem	10 (64-Bit)
IntelliJ IDEA	Entwicklungsumgebung (IDE) für die Entwicklung von Java-Anwendungen,	15.0.1 Ultimate
Java	Objektorientierte Programmiersprache für die Entwicklung plattformunabhängiger Programme	8 (Update 66)
Microsoft SQL-Server	Relationales Datenbanksystem von Microsoft	2012 64-Bit (11.0.5058.0)
MySQL-Server	Relationales Datenbanksystem (Open Source)	5.7.10
Maven	Build-Tool für Java zur Ausführung von Builds und Auflösung von Abhängigkeiten	3.3.3

5 Evaluierung

5.1 NHibernate

Hersteller: NHibernate Community

Verwendete Version: 4.0.0.4000

Link: nhibernate.info

Plattform: .NET Framework (1.1 – 4.5), Mono (Quelle: Wikipedia)

5.1.1 Kurzbeschreibung

NHibernate ist ein OR-Mapper für .NET und entstand ursprünglich aus dem Framework Hibernate für Java, es handelt sich somit um eine Portierung. Das Projekt wurde im Jahr 2003 gestartet und hat sich zu einem bekannten und ausgereiften Werkzeug für die objektrelationale Abbildung von .NET-Objekten entwickelt.

Die zentrale Schnittstelle von NHibernate für den Entwickler bildet dabei die *ISession.* Diese dient in NHibernate als *Unit-of-Work* und ist damit für die Verwaltung der Objekte zuständig. Innerhalb dieser Session haben die Objekte einen Lebenszyklus, der auch als Entity-Lifecycle bezeichnet wird. Folgende Abbildung zeigt ein Zustandsdiagramm eines Objekts:

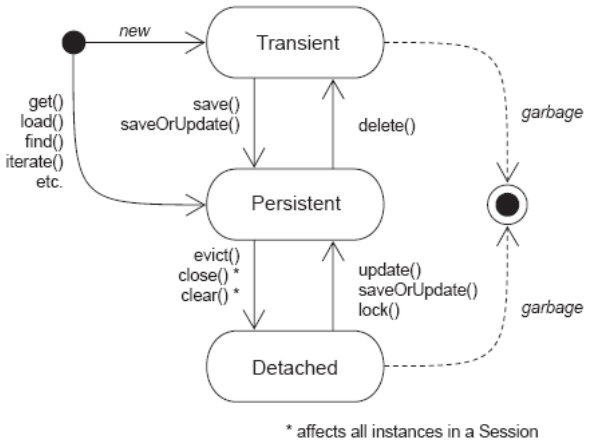

Abbildung 10: Persistence Lifecycle in NHibernate (Quelle: [Kuaté u.a. 2009])

Objekte, die keine Repräsentation in der Datenbank haben, haben den Zustand *transient.* Dies ist der Fall, wenn das Objekt neu erzeugt oder von der Session als gelöscht markiert wurde. Wenn ein transientes Objekt mithilfe von *save()* in der Datenbank gespeichert oder ein Objekt aus einer Datenbank geladen wird, handelt es sich um ein persistentes Objekt. Objekte im Zustand *persistent* sind die Objekte, die durch eine Session verwaltet werden.

Ein weiterer Zustand ist *detached.* Dieser Zustand bezeichnet ein Objekt, das zwar persistent ist, aber nicht durch eine Session verwaltet wird. Dies ist der Fall, wenn eine Session geschlossen oder zurückgesetzt wird oder wenn das Objekt manuell aus der Session entfernt wird. Diese Objekte können aber wieder mit einer anderen Session verknüpft werden.

5.1.2 Evaluation anhand Kriterien

Mapping-Features

In NHibernate werden Klassen und Attribute auf Datenbanktabellen und Spalten abgebildet. Assoziationen zwischen Klassen werden auf Fremdschlüssel und Zwischentabellen abgebildet. Diese Abbildung kann in NHibernate auf mehrere Weisen konfiguriert werden:

- XML-Code: Hier erfolgt die Abbildung über XML-Code in separaten Dateien. Diese Möglichkeit erweist sich im Vergleich zu den beiden anderen Möglichkeiten jedoch als aufwändig und fehleranfällig, da Fehler erst zur Laufzeit entdeckt werden.
- Annotations: .NET bietet die Möglichkeit, Klassen und Felder mit Annotations zu versehen, die Metadaten über diese enthalten. So kann NHibernate die Metadaten auslesen und aus ihnen dann die Konfiguration ableiten.
- Mapper-Klassen: Die dritte Möglichkeit ist das Mapping über separate Mapper-Klassen. Bis Version 3.2 war dies nur über externe Erweiterungen wie Fluent NHibernate[7] möglich, seit 3.2 steht dafür die *Mapping-by-Code*-API zur Verfügung.[8] In den Mapper-Klassen werden Primärschlüssel, Attribute und Beziehungen über Lambda-Ausdrücke angegeben (vgl. Listing 2).

```
public class KursMap : ClassMap<Kurs>
{
    public KursMap()
    {
        Id(x => x.ID).GeneratedBy.Increment();
        Map(x => x.Titel);
        Map(x => x.Beschreibung);
        Map(x => x.MaximaleTeilnehmeranzahl);
        Component(x => x.Veranstaltungszeit, m =>
        {
            m.Map(v => v.StartZeitpunkt);
            m.Map(v => v.EndZeitpunkt);
        });
        Map(x => x.Kursstatus);
        HasManyToMany(x => x.Teilnehmer).Not.LazyLoad();
        References(x => x.Kursleiter);
        References(x => x.AngelegtVon);
    }
}
```

Listing 2: Mapping mit Fluent NHibernate

[7] http://www.fluentnhibernate.org/ (letzter Zugriff: 28.01.2016)

[8] https://en.wikipedia.org/wiki/NHibernate#NHibernate_3.2 (letzter Zugriff: 28.01.2016)

In den Objektklassen müssen alle Felder für das Mapping als `virtual` deklariert werden. Dies hat den Grund, dass NHibernate intern Unterklassen aus den Objektklassen ableitet, die zur Erzeugung von Proxy-Objekten für das Lazy-Loading zum Einsatz kommen (vgl. [Kuaté u.a. 2009]). Die Mapping-Konfigurationen (ob über XML, Annotations oder Mapper-Klassen) müssen NHibernate über die Klasse *Configuration* bekannt gemacht werden.

Bei der Persistierung von Wertetypen unterstützt NHibernate sowohl Serialisierung als auch die Speicherung als Embedded-Value (siehe 2.4.3.2). Dies kann individuell für alle Wertetypen konfiguriert werden. Wenn der Wertetyp als Attribut definiert wird (über die Methode *Map*, vgl. Listing 2), serialisiert NHibernate das Objekt implizit und speichert es als BLOB in einer eigenen Spalte. Die Klasse des Wertetyps muss dafür mit der Annotation *Serializable* gekennzeichnet sein. Wenn dagegen jedes Attribut des Wertetyps in einer eigenen Spalte gespeichert werden soll, kann der Wertetyp als *Component* definiert werden (vgl. Listing 2). In beiden Ansätzen kapselt NHibernate die Übersetzung zwischen Objekt- und Datenbankdarstellung.

Bei der Abbildung von Vererbungshierarchien werden alle drei der in Kapitel 2.4.3.1 genannten Ansätze unterstützt. Dafür muss das Mapping entsprechend angepasst werden. Die Basisklasse wird bei Konfiguration durch Mapper-Klassen mithilfe von *ClassMap* gemappt, während für Unterklassen *SubclassMap* zum Einsatz kommt.

Bei *Table per Hierarchy* muss in der Basisklasse eine Diskriminatorspalte konfiguriert werden. Dies geschieht folgendermaßen:

```
DiscriminateSubClassesOnColumn("Discriminator");
```

Listing 3: Table-per-Hierarchy Mapping mit Fluent NHibernate

In den Unterklassen kann dann optional mit der Methode *Discriminator* der Wert für die Diskriminatorspalte angegeben werden, als Standardwert wird der Name der Klasse verwendet.

Für *Table per Concrete Type* muss in das Mapping der Basisklasse folgendes eingefügt werden:

```
UseUnionSubclassForInheritanceMapping();
```

Listing 4: Table-per-Concrete-Type Mapping mit Fluent NHibernate

Diese Zeile sorgt dafür, dass jede konkrete Unterklasse auf eine eigene Tabelle abgebildet wird, erlaubt zusätzlich aber auch polymorphe Abfragen auf Oberklassen.

Beim Ansatz *Table per Class* muss außer dem Mapping von Attributen und Assoziationen nichts weiter konfiguriert werden.

Bezüglich Cascading bietet NHibernate bietet verschiedene Cascading-Optionen an. Standardmäßig ist Cascading deaktiviert, d.h. wenn ein Objekt mit einer Assoziation zu einem nicht-persistenten Objekt gespeichert werden soll, muss das assoziierte Objekt erst

persistent gemacht werden. Die Cascading-Eigenschaft lässt sich global, aber auch für jede Assoziation separat konfigurieren:

```
HasMany(x => x.Rechnungspositionen).Cascade.All();
```

Es sind folgende Einstellungen möglich:

1. Save-Update: Speichert assoziierte Objekte beim Speichern/Updaten eines Objektes
2. Delete: Löscht assoziierte Objekte beim Löschen eines Objektes
3. All: Kombination aus Save-Update und Delete
4. Delete-Orphan: Wie Delete, es werden aber zusätzlich Objekte, die keine Beziehung zu einem anderen Objekt haben, gelöscht.
5. All-Orphan: Kombination aus All und Delete-Orphan

Performance

Fetching-Strategien

NHibernate unterstützt Lazy-Loading und Eager-Loading. Standardmäßig ist Lazy-Loading für alle Assoziationen aktiviert. Wenn jedoch für einzelne Assoziationen die Lazy-Loading-Eigenschaft gesetzt werden soll, lässt sich dies über das Mapping individuell konfigurieren. Ein Beispiel ist in Listing 2 zu sehen.

Wenn Lazy-Loading deaktiviert ist, wird das referenzierte Objekt implizit über Eager-Loading mitgeladen, für das sich sogar konfigurieren lässt, auf welche Art das Eager-Loading erfolgen soll. Die Standardeinstellung ist *Select*, damit wird das *Select N+1*-Problem jedoch nicht behoben, sondern nur auf den Zeitpunkt des Ladens des Objektes selbst verlagert. Um Assoziationen effektiv zu laden, sollte für das Fetching deshalb *Join* oder *Subselect* konfiguriert werden.

Batch-Operationen

Batch-Operationen lassen sich in NHibernate auf verschiedene Weisen realisieren. Dafür muss zum einen in der Datenbankkonfiguration die Eigenschaft *adonet.batch_size* konfiguriert werden, die angibt, aus wie vielen Befehlen ein Batch bestehen soll.

Danach können einzufügende Objekte beispielsweise per Schleife in der Datenbank gespeichert werden. NHibernate wird diese Operationen dann als Batch ausführen. Im Falle von Batch-Inserts muss aber beachtet werden, dass eine passende Strategie für die ID-Generation ausgewählt werden muss. Wenn die ID z.B. von der Datenbank generiert wird, führt NHibernate die Insert-Befehle nicht als Batch-Operation aus.

```
using (var trans = session.BeginTransaction())
{
    entities.ForEach(e => session.SaveOrUpdate(e));
    trans.Commit();
}
```

Listing 5: Transaktion in NHibernate

Da alle persistenten Objekte durch NHibernate gecacht werden, kann eine große Menge von einzufügenden Objekten jedoch leicht zu Speicherproblemen führen. In der NHibernate-Dokumentation wird deshalb empfohlen, regelmäßig *Flush*() und *Clear*() auf dem Session-Objekt aufzurufen, damit alle Objekte mit der Datenbank synchronisiert werden und der Cache zurückgesetzt wird[9].

Ein alternativer Ansatz ist die Verwendung von *Stateless-Sessions*, die sich unter anderem insofern von einer Session unterscheidet, dass die Objekte hier keinen Persistenzkontext haben und damit auch nicht gecacht werden.

Caching

NHibernate bietet mehrere Caching-Arten an, den First-Level-Cache, den Second-Level-Cache und den Query-Cache.

Der *First-Level-Cache* ist ein Cache, in dem Objekte abgelegt werden, die innerhalb einer Session in irgendeiner Form persistent gemacht wurden (vgl. Persistence-Lifecycle). Dieser Cache ist an eine Session gebunden. Wenn ein Objekt innerhalb einer Session mehrmals angefragt wird, gibt der First-Level-Cache eine Referenz auf das Objekt im Cache zurück (vgl. 2.4.2, Identity-Map). Außerdem werden Änderungen nicht sofort mit der Datenbank synchronisiert, sondern im Cache gehalten und gemäß Unit-of-Work-Pattern erst beim Aufruf von *Flush()* oder beim Commiten einer Transaktion in die Datenbank geschrieben (vgl. [Kuaté u.a. 2009]).

Der *Second-Level-Cache* gilt im Gegensatz zum First-Level-Cache nicht für eine Session, sondern für die gesamte Laufzeit des Programmes bzw. im Rahmen einer *SessionFactory*, die beim Start des Programmes erzeugt wird. Jede Session hat zur Laufzeit Zugriff auf diesen Cache. Welche Entitäten in diesem Cache abgelegt werden, lässt sich über das Mapping konfigurieren. Der Second-Level-Cache wird allerdings nur empfohlen, wenn das Programm exklusiven Schreibzugriff auf eine Datenbank hat und ist nur für Entitäten sinnvoll, auf die größtenteils nur lesend zugegriffen wird[10]. Das hat den Grund, dass Änderungen in der Datenbank nicht in den Cache propagiert werden und Daten im Cache somit veraltet sein können.

Der *Query-Cache* cacht Ergebnisse von Abfragen. Dieser Cache empfiehlt sich für Querys, die regelmäßig mit gleichen Parametern ausgeführt werden. Da jedoch keine Objekte, sondern nur deren Schlüssel gecacht werden, sollte der Query-Cache nur in Kombination mit dem Second-Level-Cache verwendet werden[11].

---

[9] http://nhibernate.info/doc/nhibernate-reference/batch.html (letzter Zugriff: 21.01.2016)

[10] http://nhibernate.info/doc/nhibernate-reference/performance.html#performance-cache (letzter Zugriff: 21.01.2016)

[11] http://nhibernate.info/doc/nhibernate-reference/performance.html#performance-querycache (letzter Zugriff: 21.01.2016)

Mapping-Richtungen

In NHibernate ist es möglich, aus dem Code bzw. aus den Mapping-Konfigurationen das Datenbankschema zu generieren. Dafür ist die Klasse *SchemaExport* zuständig, welche die Konfiguration mit allen Mapping-Informationen entgegennimmt und aus der Konfiguration das Schema generiert.

```
var schemaExport = new SchemaExport(cfg);
schemaExport.Create(true, true);
```

Listing 6: Generierung des Datenbankschemas in NHibernate

Die Generierung von Code aus einem existierenden Schema, also Reverse-Mapping, wird von NHibernate allerdings nicht unterstützt.

Datenbankunterstützung

Eine Liste der offiziell unterstützten Datenbanksysteme der Homepage des NHibernate-Projekts zu entnehmen[12]. Unterstützt werden die Datenbanksysteme SQL-Server, Oracle, DB2, Firebird, Informix, MySQL, PostgreSQL, SQLite und Sybase. Dafür muss dem Projekt der jeweils passende ADO.NET-Treiber hinzugefügt werden. Dieser kann über den Paket-Manager Nuget[13] für .NET relativ schnell und einfach nachinstalliert werden.

Datenbankaustausch

In NHibernate müssen für die Datenbank-Konfiguration mindestens der SQL-Dialekt und der Connection-String konfiguriert werden. Für den Austausch einer Datenbank reicht es dann aus, den Connection-String und bei einem anderen Datenbanksystem den Dialekt sowie den ADO.NET-Treiber an die neue Datenbank anzupassen. Bei Bedarf muss das Schema in der neuen Datenbank allerdings noch erzeugt werden. Dies wird durch die Klasse *SchemaExport* bewerkstelligt. Beim nächsten Start der Anwendung wird NHibernate dann die neue Datenbank nutzen.

Folgendes Beispiel soll den Wechsel vom SQL-Server 2012 auf MySQL demonstrieren:

```
fluentConfig.Database(MsSqlConfiguration.MsSql2012
                      .ConnectionString(ConnString);
```

Listing 7: Datenbank-Konfiguration für SQL-Server in NHibernate

```
fluentConfig.Database(MySQLConfiguration.Standard
                      .ConnectionString(ConnString);
```

Listing 8: Datenbank-Konfiguration für MySQL in NHibernate

[12] http://nhibernate.info/doc/nhibernate-features.html (letzter Zugriff: 21.01.2016)

[13] https://www.nuget.org/ (letzter Zugriff: 31.01.2016)

Unter Umständen müssen per Hand geschriebene SQL-Strings ebenfalls an das neue Datenbanksystem angepasst werden, z.B. bei Nutzung datenbankspezifischer SQL-Syntax und Funktionalität.

Transaktionen

In NHibernate werden Transaktionen über die ITransaction-API ausgeführt. Diese API dient als Abstraktionsschicht zur Datenbankschnittstelle, da sie das Transaktionsmanagement über die Datenbankschnittstelle direkt an das Datenbanksystem weiterdelegiert.

```
using (var transaction = _session.BeginTransaction())
{
    try
    {
        // transactional code
        transaction.Commit();
        _session.Flush();
    }
    catch(StaleObjectStateException e)
    {
        transaction.Rollback();
        throw e;
    }
    ...
}
```

Listing 9: Transaktion in NHibernate

Nebenläufigkeit

Um optimistisches Sperren zu realisieren, bietet NHibernate mehrere Möglichkeiten an. Eine davon ist die Konfiguration eines Versionszählers. Dafür muss jede Klasse, für die optimistisches Sperren genutzt werden soll, ein Versionsattribut enthalten und in der Konfiguration gemappt werden (s. Listing 10).

```
public class KundeMap : ClassMap<Kunde>
{
    public KundeMap()
    {
        ...
        Version(x => x.Version);
        ...
    }
}
```

Listing 10: Optimistisches Sperren in NHibernate mit Versionszähler

Dadurch wird bei jedem Update eines Objektes dessen Version mit der aus der Datenbank abgeglichen.

Eine Alternative zum Versionszähler ist das Abgleichen der Attribute des Objekts mit den Werten in der Datenbank. Um festzustellen, ob das zu aktualisierende Objekt verändert wurde, werden statt eines Versionszählers alle Attribute des Objekts mit der Datenbank abgeglichen (*All*) bzw. nur die modifizierten Attribute (*Dirty*). Dies kann dann erforderlich sein, wenn aus bestimmten Gründen keine Versionsspalten im Datenbankschema auftauchen sollen. Gleichzeitig muss dann aber die Eigenschaft *Dynamic-Update* aktiviert sein, da NHibernate die SQL-Statements zum Updaten der Objekte dynamisch generieren muss, anstatt nur die Versionsspalte zu überprüfen.

Schemaänderungen

In NHibernate lässt sich ein bereits existierendes Schema automatisch aktualisieren. Dies funktioniert ähnlich wie das Erstellen eines Schemas. Einziger Unterschied ist lediglich, dass Updates des Schemas über die Klasse *SchemaUpdate* ausgeführt werden, der genau wie beim *SchemaExport* über den Konstruktor die Konfiguration übergeben wird.

```
var schemaUpdate = new SchemaUpdate(cfg);
schemaUpdate.Execute(true, true);
```

Listing 11: Schema-Update mit NHibernate

So kann NHibernate bei jedem Start des Systems die Konfiguration mit dem Datenbankschema abgleichen und neue Tabellen und Spalten einfügen, wenn diese nicht vorhanden sind. Schema-Updates sind in NHibernate konstruktiv, d.h. es werden niemals Tabellen oder Spalten gelöscht. Dies muss, wenn erforderlich manuell vorgenommen werden.

Abfragemittel

In NHibernate werden mehrere Möglichkeiten angeboten, Abfragen zum Durchsuchen der Datenbank auszuführen. Diese sollen hier aufgezählt und beschrieben werden.

SQL: Abfragen im Code können direkt als SQL-Strings formuliert und ausgeführt werden, indem sie von NHibernate an die Datenbank weiterdelegiert werden. Dabei muss jedoch beachtet werden, dass die Strings kompatibel mit dem SQL-Dialekt des Datenbanksystems sein müssen und somit unter Umständen an ein Datenbanksystem gebunden sein können. Um SQL-Injections zu verhindern, können Eingabedaten mithilfe von parametrisierten Abfragen bereinigt werden.

HQL: HQL ist eine für Hibernate entwickelte Abfragesprache und wurde für NHibernate übernommen. Folgendes Beispiel demonstriert eine HQL-Query zur Suche nach Rechnungen:

```
from Rechnung as rechnungen
where rechnungen.AbrechnungsZeitraum.Monat = :monat
and rechnungen.AbrechnungsZeitraum.Jahr = :jahr
```

Listing 12: HQL-Query in NHibernate

Es ist ersichtlich, dass HQL wie SQL deklarativ ist und eine ähnliche Syntax verwendet, aber im Vergleich zusätzlich über objektorientierte Elemente wie den Zugriff auf Attribute über eine objektorientierte Notation verfügt. Somit können Querys ausgeführt werden, ohne das Datenbankschema zu kennen.

ICriteria-API: Über die ICriteria-API werden Abfragen über eine API ausgeführt, in der Suchkriterien angegeben werden können. Der Query aus Listing 12 würde mit der ICriteria-API folgendermaßen formuliert:

```
session.CreateCriteria<Rechnung>()
    .Add(Restrictions.Eq(
        Projections.Property<Rechnung>(
        x => x.AbrechnungsZeitraum.Monat), zeitraum.Monat))
    .Add(Restrictions.Eq(
        Projections.Property<Rechnung>(
        x => x.AbrechnungsZeitraum.Jahr), zeitraum.Jahr))
    .List<Rechnung>();
```

Listing 13: Query mit der ICriteria-API

Abfragen mit der ICriteria-API sind im Vergleich zu HQL typsicher und Syntaxfehler werden somit schon zur Compilezeit erkannt, der Nachteil ist jedoch eine geringere Lesbarkeit der Abfragen.

LINQ: LINQ steht für *Language-Integrated Query* und ist ein Programmiermodell für .NET zum einheitlichen Zugriff auf interne Datenquellen wie Listen und Felder, aber auch auf externe Datenquellen wie XML-Dokumente, Textdateien oder Datenbanktabellen. Da LINQ in .NET eingebunden ist, sind die Abfragen typsicher und Fehler können im Gegensatz zu String-basierten Abfragen wie SQL oder HQL bereits durch den Compiler entdeckt werden.

Um Abfragen über LINQ auszuführen, implementiert NHibernate das *IQueryable<T>*-Interface von Microsoft, die neben Methoden wie dem All- bzw. Existenzquantor und Collection-Funktionen wie *Contains* auch Methoden zum Durchsuchen von Datenquellen und Aggregationsfunktionen wie *Count*, *Sum* oder *Average* spezifiziert.

Eine Query für Rechnungen wird in LINQ folgendermaßen formuliert:

```
session.Query<Rechnung>()
.Where(x => x.AbrechnungsZeitraum.Monat == zeitraum.Monat
        && x.AbrechnungsZeitraum.Jahr == zeitraum.Jahr)
.ToList();
```

Listing 14: LINQ-Query für Rechnungen

Einige .NET-Sprachen wie C# bieten über eigene Schlüsselwörter sogar die Möglichkeit, LINQ-Abfragen im Code in einer SQL-ähnlichen Syntax zu formulieren, jedoch mit dem Vorteil der Typsicherheit durch den Compiler. Die in Listing 14 formulierte Abfrage sieht dann so aus:

```
(from rechnungen in session.Query<Rechnung>()
 where rechnungen.AbrechnungsZeitraum. Monat == zeitraum.Monat
        && x.AbrechnungsZeitraum.Jahr == zeitraum.Jahr
 select rechnungen).ToList();
```

Listing 15: LINQ-Query mit C#-Schlüsselwörtern

5.2 Entity Framework

Hersteller: Microsoft

Verwendete Version: 6.1.3 (10. März 2015)

Plattform: .NET, Mono

Link: msdn.microsoft.com/en-us/data/ef.aspx

5.2.1 Kurzbeschreibung

Das Entity Framework (früher ADO.NET Entity Framework) ist ein OR-Mapper, welcher von Microsoft für die .NET-Plattform entwickelt wurde und im Jahr 2008 als Teil der Datenbankschnittstelle ADO.NET mit dem .NET-Framework 3.5 erschien.

Seit 2012 steht das Entity Framework unter der Apache Lizenz und ist somit Open Source und unter der Plattform CodePlex[14] einsehbar. Seit Version 6.0 (2013) ist das EF nicht mehr Teil des .NET-Frameworks, sondern wird unabhängig davon weiterentwickelt.

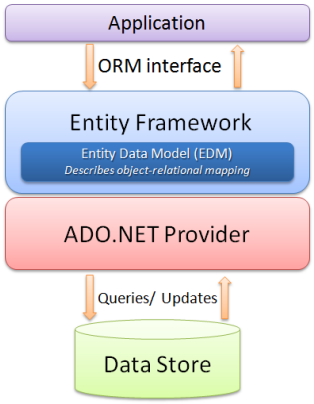

Abbildung 11: Rolle des Entity Frameworks (Quelle: Microsoft)

Mit dem Entity Framework kann objektrelationale Abbildung in .NET-Anwendungen realisiert werden. Für den Datenbankzugriff wird dabei der ADO.NET Treiber genutzt.

Ein wichtiger Bestandteil im Entity Framework ist die DbContext-API, die Konfigurationseinstellungen, das Datenmodell und vieles andere enthält. Mit ihr werden Querys ausgeführt, CRUD-Operationen ausgeführt und Transaktionen abgewickelt. Sie bildet somit die zentrale Schnittstelle für den Entwickler.

[14] http://entityframework.codeplex.com/ (letzter Zugriff: 28.01.2016)

Objekte in Entity Framework haben genau wie in NHibernate einen Lebenszyklus mit verschiedenen Zuständen. Folgende Abbildung veranschaulicht den Lebenszyklus:

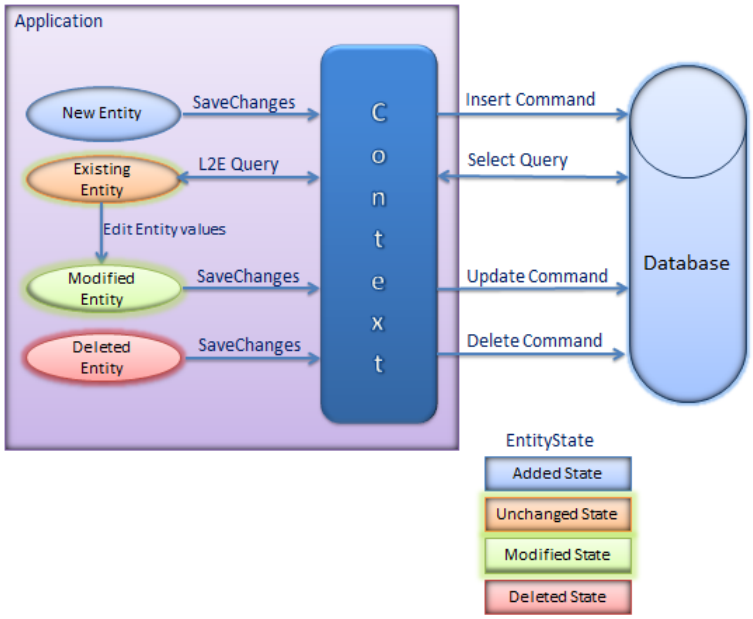

Abbildung 12: Entity Lifecycle im Entity Framework[15]

Neue Entitäten werden mit *Add* dem Context bekanntgemacht und mit *SaveChanges* in die Datenbank eingefügt. Wenn eine Entität mit der Datenbank synchron ist, befindet sie sich im Zustand *unverändert*, solange bis ihre Werte verändert werden oder sie mit *Remove* auf *gelöscht* gesetzt wird. Mit *SaveChanges* wird das Objekt dann in der Datenbank aktualisiert bzw. gelöscht. Der Context bildet somit das *Unit-of-Work* von Entity-Framework.

[15] http://www.entityframeworktutorial.net/entity-lifecycle.aspx (letzter Zugriff: 28.01.2016)

5.2.2 Evaluation anhand Kriterien

Mapping-Features

In Entity Framework werden Objektklassen und ihre Attribute auf Tabellen und Spalten abgebildet. Assoziationen werden über Fremdschlüssel und Zwischentabellen dargestellt. Das Mapping erfolgt entweder direkt im DbContext oder über Mapper-Klassen, in der ähnlich wie in NHibernate über eine Fluent-API Attribute, Beziehungen sowie weitere Eigenschaften wie Constraints oder Spaltennamen angegeben werden können, die abgebildet werden sollen. Mapper-Klassen bieten jedoch den Vorteil, das Mapping auf mehrere Komponenten aufzuteilen und diese dynamisch zur Laufzeit einzubinden[16].

Hierbei muss jedoch beachtet werden, dass alle Klassen, Attribute und Assoziationen implizit durch Code-First-Konventionen[17] gemappt werden, sobald die Klassen dem DbContext bekanntgemacht werden. Attribute, die „ID“ im Namen enthalten, werden etwa als Primärschlüssel abgebildet, Klassen ohne Attribute mit dieser Eigenschaft werden als Wertetypen behandelt. Das explizite Mapping ist somit redundant, sofern nicht weitere Eigenschaften wie Spaltennamen oder Constraints angegeben werden. Um das Mappen eines Attributes zu vermeiden, muss es explizit mit *Ignore* gekennzeichnet werden (analog zu *Property*, vgl. folgendes Listing).

```
public class KundeMap : EntityTypeConfiguration<Kunde>
{
    public KundeMap()
    {
        HasKey(x => x.Kundennummer);
        Property(x => x.Vorname);
        Property(x => x.Nachname);
        Property(x => x.Adresse.Strasse);
        Property(x => x.Adresse.Hausnummer);
        ...
        HasRequired(x => x.AngelegtVon);
        ToTable("Kunden");
    }
```

Listing 16: Fluent Mapping mit Entity Framework

Wie im Listing außerdem dargestellt, ist die Konfiguration von Wertetypen ebenfalls über die Fluent-API möglich. Das Entity Framework verwendet dabei den Ansatz *Embedded-Value* und

[16] vgl. http://romiller.com/2012/03/26/dynamically-building-a-model-with-code-first/ (letzter Zugriff: 28.01.2016)

[17] https://msdn.microsoft.com/de-de/data/jj679962.aspx (letzter Zugriff: 28.01.2016)

legt für jedes Attribut des Wertetyps eine eigene Spalte an. Serialisierung wird dagegen nicht direkt unterstützt bzw. muss manuell vorgenommen werden.

Über die Fluent-API kann außerdem das Mapping von Vererbungshierarchien konfiguriert werden. Dabei sind alle drei der in Kapitel 2.4.3.1 beschriebenen Ansätze möglich:

- *Table per Hierarchy*: Hierfür müssen die Mappings für alle Klassen in der Klassenhierarchie konfiguriert werden. Ausnahme sind dabei Klassen, die keine Attribute enthalten. Entity Framework legt dann eine Tabelle mit dem Namen der Basisklasse an, in der Objekte der kompletten Hierarchie persistiert werden.
- *Table per Class:* Zusätzlich zu den Mapping-Klassen für jede Klasse in der Hierarchie muss in jeder Klasse die Zieltabelle explizit angegeben werden (vgl. Listing 16).
- *Table per Concrete Type*: Mapping-Klassen müssen nur für konkrete Klassen geschrieben werden.

Cascading gilt in Entity-Framework standardmäßig für Save- und Update-Operationen von Objekten. Wenn zusätzlich eine Muss-Beziehung von einem Objekt A zu einem anderen Objekt B existiert, also die Referenz zu Objekt B nicht *null* sein darf, wird Objekt A wird beim Löschen von B standardmäßig mitgelöscht. Das Verhalten bezüglich Delete-Operationen kann sowohl global als auch für jede Assoziation mit *WillCascadeOnDelete* konfiguriert werden.

Performance

Fetching-Strategien

Entity Framework unterstützt Lazy-Loading und Eager-Loading. Um Lazy-Loading nutzen zu können, müssen alle Assoziationen mit dem Schlüsselwort `virtual` (C#) versehen werden (vgl. NHibernate). Assoziationen, die nicht `virtual` sind, werden nicht durch Lazy-Loading nachgeladen, sondern müssen bei Bedarf explizit nachgeladen werden. Die Lazy-Loading Eigenschaft kann im *DbContext* global für alle Klassen gesetzt werden. Standardmäßig ist Lazy-Loading aktiviert.

```
Configuration.LazyLoadingEnabled = false;
```

Listing 17: Deaktivierung von Lazy-Loading für alle Klassen

Um das *Select N+1*-Problem zu vermeiden (vgl. 2.4.2, Lazy-Loading), müssen assoziierte Objekte mitgeladen werden, da ansonsten bei jedem Schleifendurchlauf Elemente aus der Datenbank nachgeladen werden. In diesem Fall kommt Eager-Loading zum Einsatz.

```
(from kurse in session.Query<Kurs>()
select kurse).Include(x => x.Teilnehmer)
.ToList();
```

Listing 18: LINQ-Query zum Laden von Kursen inklusive Teilnehmern

Wie in Listing 18 dargestellt kann in einer Query mit *Include* angegeben werden, welche assoziierten Objekte mitgeladen werden sollen. Entity Framework lädt dann den Objektgraphen über ein Outer-Join aus der Datenbank.

Batch-Operationen

Batch-Operationen werden in der verwendeten Version des Entity-Frameworks (6.1.3) nicht unterstützt. Allerdings hat Microsoft die Möglichkeit für Batch-Operationen mit der Version 7 angekündigt (vgl. [Microsoft 2015a]).

Caching

- **Objektcache:** Genau wie der First-Level-Cache von NHibernate bietet Entity Framework einen Objektcache, um unnötige Datenbankaufrufe für bereits geladene Objekte zu vermeiden. Der Objektcache ist nur für eine DbContext-Instanz gültig.
- **Abfrageplan-Cache:** Wenn ein Query (s. Abfragemittel) ausgeführt wird, übersetzt Entity Framework ihn mithilfe eines Plancompilers in einen SQL-Befehl. Wenn diese Query an anderer Stelle erneut ausgeführt wird, wird der dazugehörige SQL-Befehl aus dem Abfrageplan-Cache geladen, anstatt die Abfrage erneut zu kompilieren.
- **Ergebniscache:** Im Ergebniscache (Level-2-Cache) werden Abfrageergebnisse gecacht. Entity Framework implementiert diesen Cache jedoch nicht direkt, daher muss die Funktionalität über Tools von Drittanbietern hinzugefügt werden (vgl. [Microsoft d])

Mapping-Richtungen

Entity Framework bietet sowohl die Möglichkeit des Forward-Mappings (Code First) als auch des Reverse-Mappings (Database First).

In Visual Studio wählt man dazu „Projekt“ → „Neues Element hinzufügen…“ und als Element „ADO.NET Entity Data Model“. Dort besteht dann jeweils die Möglichkeit, Code First oder Database First jeweils mit und ohne Designer auszuwählen (s. Abbildung 13).

Bei Database First werden über die folgenden Dialoge die Datenbank sowie die Tabellen ausgewählt, für die im Code Klassen generiert werden sollen. Entity Framework generiert dann aus den angegebenen Tabellen alle Objektklassen inkl. Attribute und Assoziationen. Wird dagegen Code First ausgewählt, wird lediglich eine Model-Klasse erstellt, die von *DbContext* erbt. Dieser Klasse können dann die Objektklassen bekanntgemacht werden, die Entity Framework abbilden soll. Aus den Klassen bzw. den Metainformationen leitet Entity-Framework das Datenmodell ab und generiert daraus das Datenbankschema.

Der Designer ist eine optionale Möglichkeit für Code-First bzw. Database-First. Mit ihm wird das Datenmodell visuell in einer UML-ähnlichen Notation dargestellt und kann dort direkt per Mausklick bearbeitet werden. Intern wird das Datenmodell im XML-Format in einer EDMX-Datei gespeichert und aus diesem dann per Codegenerierung die Entitätsklassen generiert. Laut offiziellen Ankündigungen soll dieses Feature aber mit der Version 7 entfernt werden (vgl. [Microsoft 2015a]).

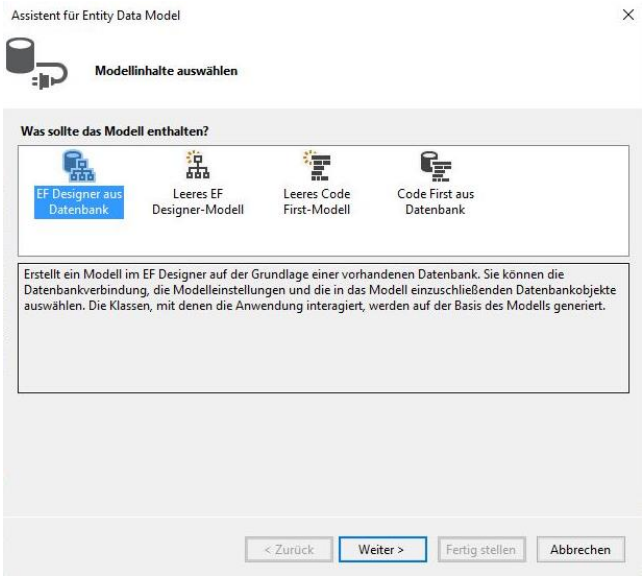

Abbildung 13: Auswählen der Mapping-Richtung

Datenbankunterstützung

Neben dem hauseigenen Microsoft SQL-Server werden auch andere relationale Datenbanksysteme unterstützt. Die Liste der offiziell unterstützten Datenbanksysteme ist dem MSDN zu entnehmen[18]. Zu den bekanntesten zählen dabei MySQL, Oracle, DB2 (IBM), SQL-Anywhere (SAP), SQLite und PostgreSQL.

Datenbankaustausch

Um die Datenbank auszutauschen, muss lediglich der Connection-String geändert werden sowie eventuell der ADO.NET-Provider ausgetauscht werden. Entity Framework nutzt ab dann die neue Datenbank zur Speicherung und generiert auch das Schema für die neue Datenbank.

Per Hand geschriebene SQL-Abfragen müssen gegebenenfalls angepasst werden.

Transaktionen

Zentrale Schnittstelle für den Entwickler ist die DbContext-Klasse. Neue Objekte werden dem *DbContext* per *Add*() bekannt gemacht, Änderungen an persistenten Objekten werden durch den *DbContext* verfolgt. Durch den Aufruf von *SaveChanges*() werden alle geänderten und neuen Objekte mit der Datenbank synchronisiert. Entity Framework führt dafür implizit eine Transaktion aus.

[18] https://msdn.microsoft.com/en-us/library/dd363565.aspx (letzter Zugriff: 14.01.2015)

Wenn jedoch geänderte Daten auf zuvor gelesenen Daten basieren, müssen Lese- und Schreiboperationen in einer Transaktion stattfinden. Dafür bietet die DbContext-API die Methoden der Transaktionsklammer an.

```
using (var transaction = _context.Database.BeginTransaction())
{
    try
    {
        // transactional CRUD
        transaction.Commit();
    }
    catch(DBConcurrencyException e)
    {
        transaction.Rollback();
        throw e;
    }
    catch ...
}
```

Listing 19: Transaktionen in Entity Framework

Genau wie in NHibernate wird die Transaktionsverwaltung über die Datenbankschnittstelle an das Datenbanksystem weiterdelegiert und bei Fehlern eine Exception ausgelöst, die im Code behandelt werden muss.

Nebenläufigkeit

In Entity Framework wird optimistische Synchronisation unterstützt. Dafür müssen Klassen ein Versions-Attribut vom Typ *Byte-Array* hinzugefügt werden und im Mapping als *IsRowVersion()* konfiguriert werden.

```
Property(x => x.Version).IsRowVersion();
```

Listing 20: Konfiguration eines Versions-Attributs

Entity Framework legt für dieses Attribut in der Datenbank eine Spalte vom Typ *Timestamp* an und nutzt den Zeitstempel des Datensatzes, um Änderungen zu erkennen. Ein Versionszähler, der automatisch inkrementiert wird, wird dagegen nicht direkt unterstützt bzw. muss dieser manuell inkrementiert werden.

Als Alternative zum Versions-Attribut gibt es die Möglichkeit, bereits vorhandene Attribute zur optimistischen Synchronisation zu nutzen. Dafür werden die Attribute im Mapping mit *IsConcurrencyToken*() konfiguriert, wodurch ihre Werte beim Updaten des Objekts mit der Datenbank abgeglichen werden und unterschiedliche Werte als Änderung durch eine andere Transaktion aufgefasst werden.

Schemaänderungen

Im *DbContext* kann konfiguriert werden, welche Strategie zur Initialisierung der Datenbank verfolgt wird. Standardeinstellung ist die Erzeugung der Datenbank, wenn sie nicht existiert. Andere Einstellungen sind das Neuerzeugen der Datenbank bei jedem Start bzw. bei jeder Änderung am Datenmodell. Dies ist vor allem für Entwicklungszwecke sinnvoll. Änderungen am Datenmodell werden dann jedoch nicht auf ein existierendes Datenbankschema übertragen, was im Produktiveinsatz unumgänglich ist. Um das Datenbankschema bei jedem Start der Anwendung mit dem Datenmodell zu synchronisieren, muss die Ausführung von Migrationen explizit im *DbContext* aktiviert werden.

```
Database.SetInitializer(new MigrateDatabaseToLatestVersion<Model1,
SchemaMigration>());
```

Bei der Initialisierung des *DbContext* zur Laufzeit wird dann das Datenmodell im Code mit dem Schema der Datenbank abgeglichen. Dafür legt Entity Framework in der Datenbank eine eigene Migrationstabelle an, in neben anderen Informationen wie den Namen der *DbContext*-Klasse das Datenmodell (genauer die Metadaten) in serialisierter Form gespeichert ist.

Wenn sich Datenmodell und Schema unterscheiden, werden automatisch die fehlenden Tabellen und Spalten erzeugt. Änderungen am Schema, die Datenverluste nach sich ziehen würden, z.B. durch Entfernung von Tabellen oder Spalten, müssen explizit zugelassen werden, da Entity-Framework ansonsten die Änderungen verweigert.

Abfragemittel

Zum Abfragen der Datenbank werden von Entity-Framework mehrere Mittel angeboten. Diese sollen im Folgenden genauer erläutert werden:

SQL: Es können native SQL-Abfragen formuliert werden, die durch Entity Framework an die Datenbank delegiert werden. Dies kann erforderlich sein, wenn datenbankspezifische Funktionalität genutzt werden soll oder Ergebnisse von Abfragen nicht in Objekte übersetzt werden sollen.

Entity-SQL: Die zweite Möglichkeit bildet Entity Frameworks eigene Abfragesprache Entity-SQL, die sich am ehesten mit HQL vergleichen lässt, da sie ebenfalls datenbankunabhängig ist und genau wie HQL Klassennamen und Attribute statt Tabellen und Spalten angegeben werden.

```
select value r from KundenverwaltungContext.Rechnungen as r
where r.AbrechnungsZeitraum.Jahr == @jahr
and r.AbrechnungsZeitraum.Monat == @monat
```

Listing 21: Abfrage in Entity-SQL

Listing 21 veranschaulicht eine Query in Entity-SQL zur Suche nach Rechnungen in einem bestimmten Abrechnungszeitraum. Die Abfrage kann außerdem unter Definition von Parametern mit dem @-Präfix parametrisiert werden, um Eingabeparameter zu bereinigen.

LINQ-to-Entities: Mithilfe von LINQ-to-Entities können Abfragen mithilfe von LINQ erstellt und ausgeführt werden. Hier gelten die gleichen Konzepte wie bei LINQ für NHibernate.

5.3 ActiveJDBC

Hersteller:	Open Source
Verwendete Version:	1.4.11 (17. Juni 2015)
Plattform:	Java
Link:	javalite.io

5.3.1 Kurzbeschreibung

ActiveJDBC ist ein OR-Mapper für die Programmiersprache Java und somit ein alternatives Werkzeug zum Persistenzframework Hibernate. ActiveJDBC implementiert das Active-Record-Pattern und ist stark an das ActiveRecord-Framework von Ruby[19] angelehnt.

ActiveJDBC folgt dem *Convention over Configuration* Prinzip, welches aussagt, dass durch Einhaltung von Konventionen der Konfigurationsaufwand reduziert werden soll. So wird in ActiveJDBC etwa erwartet, dass jede Tabelle eine Primärschlüsselspalte mit dem Namen *id* hat, Tabellen als Namen den Plural des Klassennamens haben und bei *One-to-Many*-Beziehungen die *Many*-Tabelle einen Fremdschlüssel mit dem Namen <*One-Klasse*>_*id* auf den Primärschlüssel der *One*-Tabelle enthält.

Im Gegensatz zu NHibernate und Entity Framework nutzt ActiveJDBC nicht das *Unit-of-Work*-Pattern, die Objekte sind also selbst für ihre Persistenz verantwortlich.

5.3.2 Evaluation anhand Kriterien

Mapping-Features

Da ActiveJDBC das Active-Record-Pattern implementiert, findet das Mapping in den Objektklassen statt. Dafür wird die abstrakte Klasse *Model* zur Verfügung gestellt, welche die notwendige CRUD-Logik implementiert. Eine Objektklasse, die gemappt werden soll, muss dann lediglich von *Model* erben, ist somit selbst von technischen Details befreit.

Unabhängig vom Prinzip Convention over Configuration lassen sich in den Objektklassen auch die Namen für die Tabelle, die ID-Spalte und Assoziationen per Java-Annotations konfigurieren. Die Konfiguration des Tabellennamens macht vor allem bei nicht-englischen Klassennamen Sinn, da die Konvention, dass der Tabellenname der Plural des Klassennamens ist, nur im Englischen funktioniert[20].

---

[19] http://guides.rubyonrails.org/active_record_basics.html (letzter Zugriff: 31.01.2016)

[20] http://javalite.io/english_inflections (letzter Zugriff: 31.01.2016)

```
@Table("kunden")
@IdName("kundennummer")
@BelongsTo(parent = Rezeptionist.class,
foreignKeyName = "angelegtVon_id")
public class Kunde extends Model{
    ...
}
```

Listing 22: Konfiguration der Klasse Kunde in ActiveJDBC

Auf Attribute und assoziierte Objekte wird über Akzessor-Methoden der Model-Klasse zugegriffen. Dafür müssen jedoch die Spaltennamen in der Datenbank bekannt sein, da ActiveJDBC aus dem Datenbankschema das Datenmodell ableitet.

Um Datenbankoperationen auszuführen, werden Methoden der Model-Klasse genutzt. Um zum Beispiel ein neues Objekt zu persistieren, wird die Methode *save()* auf dem Objekt aufgerufen. Dasselbe gilt für das Updaten und Löschen von Objekten. Für Suchmethoden werden statische Methoden wie *find*() oder *where*() verwendet (mehr dazu unter Abfragemittel).

```
Kunde k = new Kunde();
k.setString("vorname", "Max");
k.setString("nachname", "Mustermann");
k.save();
```

Listing 23: Erzeugung und Persistierung eines neuen Kunden

Es fällt auf, dass Attribute eines Objekts von außen beliebig gelesen und verändert werden können, da alle Akzessor-Methoden als public deklariert sind. Dadurch bestehen keine Schutzmöglichkeiten, Attribute vor unberechtigtem Zugriff zu schützen, außerdem wird damit die Kopplung zur Persistenzschicht erhöht, da jeder, der von außen auf das Objekt zugreift, die Spaltennamen kennen muss. Eine Möglichkeit, dies zu vermeiden, wäre die Spezifikation von eigenen Getter- und Setter-Methoden in Interfaces, die von den Model-Klassen implementiert werden, um einen einheitlichen Zugriff auf Attribute und Assoziationen zu ermöglichen und Implementierungsdetails zu verstecken.

Bei der Abbildung von Vererbungshierarchien wird nur *Table per Concrete Type* unterstützt, da in ActiveJDBC nur eine 1:1-Abbildung zwischen Model-Klasse und Tabelle möglich ist.

Die Abbildung von Wertetypen wird seitens ActiveJDBC nicht direkt unterstützt und muss stattdessen manuell vorgenommen werden. Folgende Abbildung demonstriert das Mapping eines Wertetyps nach dem Embedded-Value-Pattern. Es sei dabei vorausgesetzt, dass in der Tabelle des Models zwei Spalten mit den Namen *veranstaltungszeit_start* und *veranstaltungszeit_ende* existieren.

```
private final String startZeitpunkt = "veranstaltungszeit_start";
private final String endZeitpunkt = "veranstaltungszeit_ende";

public VeranstaltungszeitTyp getVeranstaltungszeitpunkt() {
    return new VeranstaltungszeitTyp(getDate(startZeitpunkt),
                                     getDate(endZeitpunkt));
}

public void setVeranstaltungszeitpunkt(VeranstaltungszeitTyp vaz){
    set(startZeitpunkt, vaz.getStartZeitpunkt(), endZeitpunkt,
        vaz.getEndZeitpunkt());
}
```

Listing 24: Abbildung fachlicher Datentypen in ActiveJDBC

Serialisierung bzw. Deserialisierung muss ebenfalls manuell vorgenommen werden. Die Datenbankspalte muss dafür einen passenden Datentyp haben (beispielsweise Binary für BLOBs oder nvarchar für CLOBs).

Cascading wird lediglich für Delete-Operationen über die Methode *deleteCascade()* unterstützt, d.h. beim Speichern und Updaten müssen nicht-persistente abhängige Objekte manuell gespeichert werden. Außerdem gilt Cascading beim Löschen für alle Assoziationen und kann nicht wie in NHibernate für einzelne Beziehungen konfiguriert werden. Wenn bestimmte Assoziationen nicht vom Löschvorgang betroffen werden sollen, müssen diese mit *deleteCascadeExcept*() explizit ausgeschlossen werden.

Alternativ kann auch die Methode *save()* überschrieben werden, um Cascading für Save- und Update-Operationen zu implementieren.

Performance

Fetching-Strategien

Standardmäßig werden alle Assoziationen per Lazy-Loading nachgeladen. Querys, deren Rückgabewert in einer Liste gespeichert wird, werden ebenfalls erst ausgeführt, wenn auf die Liste zugegriffen wird.

Um das *Select N+1*-Problem zu vermeiden, können ähnlich wie in Entity Framework über die Methode *include*() explizit die Klassen angegeben werden, die mitgeladen werden sollen. Eager-Loading wie in NHibernate, das für einzelne Assoziationen konfiguriert und dann implizit beim Laden von Objekten ausgeführt wird, ist auch hier nicht möglich.

Batch-Operationen

Batch-Operationen werden in ActiveJDBC über Prepared-Statements ausgeführt. In diesen werden die auszuführenden Befehle angegeben und die Parameter ergänzt. Folgendes Beispiel soll dies veranschaulichen:

```
PreparedStatement ps = Base.startBatch(
    "insert into kunden (vorname, nachname) values (?,?)");
for(Kunde k : kunden){
    Base.addBatch(ps, k.getVorname(),k.getNachname());
}
Base.openTransaction();
Base.executeBatch(ps);
Base.commitTransaction();
```

Listing 25: Batch-Insert in ActiveJDBC

Batch-Operationen für mehrere Objekte sind dagegen nicht möglich, da jedes Objekt selbst für seine Persistenz zuständig ist und nicht durch ein Unit-of-Work o.ä. verwaltet wird.

Caching

ActiveJDBC bietet die Möglichkeit an, Abfragen zu cachen. Die Implementation des Cache erfolgt durch das externe Framework EHCache[21], welches zuvor konfiguriert werden muss. Caching kann pro Klasse konfiguriert werden. Dazu muss die Klasse mit der Annotation *@Cached* versehen werden.

Wenn ein neues Objekt einer Klasse in die Datenbank eingefügt wird, wird der Cache für diese Klasse sowie der abhängigen Klassen geleert, da er ab dem Moment des Einfügens eines Objekts nicht mehr synchron mit der Datenbank ist. Es wird deshalb empfohlen, Daten zu cachen, die größtenteils gelesen, aber selten geändert werden[22].

Mapping-Richtungen

In ActiveJDBC werden keine Möglichkeiten angeboten, aus Objektklassen das Datenbankschema bzw. aus dem Datenbankschema Objektklassen zu generieren. Das Schema mit Tabellen, Spalten und Fremdschlüsselbeziehungen muss per Hand durch SQL-Skripte erzeugt werden.

Im Code müssen per Hand entsprechende Klassen erstellt werden, die von der abstrakten Klasse *Model* erben. In diesen Klassen werden dann per Annotations Tabellenname, ID-Spalte (falls die Primärschlüsselspalte nicht „id“ heißt) und Assoziationen konfiguriert (vgl. Listing 22). Die Attribute der Objekte werden zur Laufzeit dynamisch anhand der vorhandenen Spalten in der Tabelle bestimmt und müssen nicht explizit konfiguriert werden. Es können in den Klassen aber optional aber Getter und Setter implementiert werden, um den Zugriff auf Attribute zu kapseln[23].

---

[21] http://www.ehcache.org/ (Letzter Zugriff: 28.01.2016)

[22] http://javalite.io/caching (Letzter Zugriff: 31.01.2016)

[23] http://javalite.io/setters_and_getters (Letzter Zugriff: 31.01.2016)

Datenbankunterstützung

Die Liste der offiziell unterstützten Datenbanksysteme ist der Homepage des ActiveJDBC-Projekts zu entnehmen[24].

Aktuell werden offiziell die Datenbanksysteme Microsoft SQL-Server, MySQL, Oracle, PostgreSQL, H2 und SQLite3 unterstützt.

Die Verbindung zu diesen Datenbanken erfolgt über den entsprechenden JDBC-Treiber.

Datenbankaustausch

Um eine Datenbank auszutauschen, muss der Datenbanktreiber sowie der Connection-String angepasst werden. Diese werden neben den Login-Daten zum Öffnen der Datenbankverbindung benötigt. Da die Persistenzlogik in den Model-Klassen liegt, muss außerdem sichergestellt sein, dass das neue Schema mit den Model-Klassen im Code kompatibel ist, bei einer neuen Datenbank muss das komplette Schema also erst durch SQL-Skripte erzeugt werden.

Da die Query-API außerdem auf SQL basiert (in Abfragemittel genauer beschrieben), müssen eventuell Abfragen angepasst werden, die datenbankspezifische Syntax enthalten.

Transaktionen

Für Transaktionen bietet ActiveJDBC einen Wrapper zu der Transaktions-API von JDBC an. Folgendes Beispiel soll eine Transaktion in ActiveJDBC veranschaulichen.

```
ActiveJDBCHelper.openTransaction();
if (kursNach.HatFreiePlaetze()) {
    // Buche Kunde um
    ActiveJDBCHelper.commitTransaction();
} else {
    ActiveJDBCHelper.rollbackTransaction();
    throw new KursUeberfuelltException("Zielkurs ist bereits
ausgebucht");
}
```

Listing 26: Transaktion in ActiveJDBC

In diesem Fall sorgt die Transaktion dafür, dass ein Kunde zu einem anderen Kurs umgebucht wird, wenn noch freie Plätze vorhanden sind. Ansonsten werden alle bereits erfolgten Änderungen in der Datenbank durch ein Rollback rückgängig gemacht.

Nebenläufigkeit

ActiveJDBC unterstützt optimistisches Sperren per Versionszähler. Dafür erwartet das Framework per Konvention eine Spalte vom Typ Ganzzahl mit der Bezeichnung

[24] http://javalite.io/activejdbc#supported-databases (Letzter Zugriff: 28.01.2016)

record_version[25]. Alternativ kann der Spaltenname ebenfalls per Java-Annotation der Klasse konfiguriert werden.

```
@VersionColumn("version")
public class Kunde extends Model{...}
```

Abbildung 14: Konfiguration der Versionsspalte

Beim ersten Speichern eines Objekts wird der Versionszähler mit dem Wert 1 initialisiert und bei jedem Update automatisch inkrementiert.

Wenn ein Datensatz an anderer Stelle geändert wurde und der Versionszähler einen anderen Wert als den des Objekts aufweist, wird ActiveJDBC entsprechend eine *StaleModelException* beim Speichern des Objekts auslösen, die dann vom Entwickler zu behandeln ist.

Schemaänderungen

Da ActiveJDBC aus dem Datenbankschema das Datenmodell ableitet, sind Änderungen am Datenmodell nur durch Änderung des Datenbankschemas möglich. Wenn nur neue Attribute hinzukommen, müssen keine weiteren Änderungen im Code vorgenommen werden, da über die Akzessor-Methoden des *Models* dynamisch auf diese zugegriffen werden kann. Wenn dagegen neue Tabellen und Fremdschlüsselbeziehungen hinzukommen, müssen diese im Code durch neue Model-Klassen abgebildet werden.

Änderungen am Schema müssen wie die Erzeugung ebenfalls durch eigene SQL-Skripte erfolgen, die Befehle zum Erzeugen, Umbenennen bzw. Löschen von Tabellen, Spalten und Fremdschlüsselbeziehungen müssen per Hand geschrieben und ausgeführt werden.

Abfragemittel

ActiveJDBC bietet direkten Zugriff auf die darunterliegende Datenbank durch Abfragen über eigene SQL-Befehle. Diese werden eins zu eins an die Datenbank weitergereicht, können aber auch als parametrisierte Abfragen ausgeführt werden, um übergebene Parameter wie Eingabedaten zu bereinigen und SQL-Injections zu vermeiden.

Als Alternative zu nativem SQL bietet ActiveJDBC eine Query-API an, mit der Abfragen gegen die Datenbank ausgeführt werden können. Dennoch erweist diese sich als sehr SQL-nah, da Suchkriterien als SQL-Klauseln formuliert werden.

```
List<Kunde> kunden = Kunde.where("vorname like ?", "M%");
```

Listing 27: Query in ActiveJDBC

In Listing 27 ist ein Beispiel zur Suche nach Kunden zu sehen, deren Vorname mit *M* beginnt. Durch die Verwendung von SQL-Syntax in der Query-API wird somit eine hohe Kopplung zur

[25] http://javalite.io/optimistic_locking (letzter Zugriff: 31.01.2016)

Datenbank erzeugt. Dies ist insofern von Bedeutung, da ansonsten keine datenbankunabhängigen alternativen Möglichkeiten angeboten werden.

Mit *count()* wird außerdem nur eine einzige Aggregationsfunktion angeboten. Andere Aggregationen wie *sum oder avg* werden nicht unterstützt und müssen per Hand implementiert werden.

6 Fazit

6.1 Ergebnisse

Bei der Evaluierung hat sich herausgestellt, dass bestimmte Frameworks in bestimmten Bereichen mehr Funktionalität anbieten, dafür in anderen Bereichen weniger. Ein Beispiel ist der Vergleich zwischen NHibernate und Entity Framework. In den meisten Bereichen ähneln sich die beiden Frameworks. So nutzen beide Frameworks Data-Mapper, um Objektklassen frei von Persistenzlogik zu halten und Objekte dieser Klassen als Plain Objects behandeln zu können. Außerdem bieten beide über die *ISession* bzw. über den *DbContext* ein Unit-of-Work zur Verwaltung der Objektpersistenz. Unterschiede sind aber zum Beispiel, dass Entity Framework im Gegensatz zu NHibernate neben Forward-Mapping auch Reverse-Mapping unterstützt, dafür aber auf direktem Wege keine Serialisierung von Wertetypen und keine optimistische Synchronisation mit einem automatisch inkrementierenden Versionszähler unterstützt. Vor allem die letzten beiden Unterschiede mögen zwar nicht gravierend sein, trotzdem können sie aufgrund bestimmter Anforderungen einen Einfluss auf die Entscheidung für ein Framework haben.

Als größter Unterschied hat sich jedoch unabhängig von der Programmiersprache der Gegensatz zwischen NHibernate/Entity Framework und ActiveJDBC erwiesen. Dass ActiveJDBC nicht zuletzt durch Active-Record einen komplett anderen Ansatz als die beiden anderen Frameworks verfolgt, wurde auch bei der Realisierung der Referenzanwendung ersichtlich. Angefangen vom Zugriff auf Attribute über Akzessor-Methoden der Oberklasse über die Query-API bis hin zur Erzeugung und Änderung des Datenbankschemas hat sich die Programmierung mit ActiveJDBC im Vergleich zu NHibernate und Entity Framework als sehr datenbanknah erwiesen. Ob dies als Nachteil angesehen wird, hängt aber auch von den eigenen Anforderungen bezüglich Datenbankunabhängigkeit ab, dennoch führt dies vor allem bei Abfragen zu mehr datenbanktechnischem Code als bei den anderen beiden Frameworks.

Dennoch haben sich alle drei Frameworks bei der Realisierung der Referenzanwendung als wertvolle Hilfsmittel erwiesen, da sie dem Entwickler jede Menge Arbeit bezüglich objektrelationaler Abbildung abnehmen und mithilfe von ihnen der Programmieraufwand und vor allem die Menge an Code im Gegensatz zur Programmierung auf der Datenbankschnittstelle mit SQL deutlich reduziert wird.

6.2 Ausblick

In dieser Bachelorarbeit wurden drei OR-Mapper untersucht. Da auf dem Markt jedoch sehr viel mehr Mapper für jede erdenkliche objektorientierte Sprache existieren, ist diese Arbeit nicht als Empfehlung eines bestimmten Frameworks zu betrachten, vor allem da aufgrund des Umfangs nicht alle Frameworks miteinander verglichen werden konnten und somit keine Garantie gegeben werden kann, dass sich ein Framework für bestimmte Fälle am besten eignet. Außerdem werden die Frameworks stetig weiterentwickelt und um neue Funktionalität ergänzt (vgl. [Microsoft 2015a]), wodurch die Ergebnisse der Evaluierung zumindest teilweise an Gültigkeit verlieren können und somit nur als Momentaufnahme zu betrachten sind.

Dennoch kann der aufgestellte Kriterienkatalog neben eigenen Anforderungen eine Hilfestellung zur Evaluierung von zum Beispiel Hibernate für Java, ActiveRecord für Ruby on Rails, aber auch vieler weiterer Frameworks bieten und somit die Entscheidung für ein bestimmtes Framework erleichtern.

Glossar

Akzessor	Ein Akzessor ist eine Zugriffsfunktion, mit der ein Attribut eines Objektes abgerufen bzw. verändert werden kann. Diese Funktionen werden auch als Getter und Setter bezeichnet.
CRUD	CRUD ist ein Akronym und bezeichnet die Operationen Create, Read, Update und Delete, also das Erstellen, Lesen, Aktualisieren und Löschen von Datensätzen in einer Datenbank.
Geschäftstransaktion	Eine Geschäftstransaktion bezeichnet im Gegensatz zu einer Datenbanktransaktion eine Transaktion im fachlichen Sinne, z.B. eine Bestellung von Waren.
Garbage Collector	Der Garbage Collector kümmert sich um die Speicherverwaltung in einem Programm und sorgt dafür, dass nicht mehr benötigter Speicherplatz wieder freigegeben wird.
Interface	Ein Interface dient in objektorientierten Programmiersprachen zur Spezifikation von Schnittstellen, in der Signaturen von Methoden angegeben werden, die von den Klassen zu implementieren sind.
NoSQL	NoSQL ist ein Begriff für Datenbanken, die nicht dem relationalen Datenmodell folgen, sondern andere Strukturierungsansätze nutzen.
Plain Object	Ein Plain Object ist ein Objekt im Sinne der Objektorientierung, also ein Objekt ohne Abhängigkeiten zu externen Werkzeugen.
Round Trip Time	Die Round Trip Time im Kontext von Datenbanken bezeichnet die Zeit, die vom Abschicken einer Abfrage bis zur Antwort vom Datenbanksystem vergeht.
SQL Injection	Eine SQL-Injection ist ein Angriff auf SQL-Datenbanken, der dadurch ermöglicht wird, wenn Benutzereingaben ohne Überprüfung bzw. Bereinigung an das Datenbanksystem weitergegeben werden und somit die Semantik von Abfragen manipuliert werden kann.

Literaturverzeichnis

[Agile Data] Agile Data**:** *The Object-Relational Impedance Mismatch*, URL: http://www.agiledata.org/essays/impedanceMismatch.html (letzter Zugriff: 28.01.2016)

[Balzert 2014] Balzert, Helmut: *Java - Objektorientiert programmieren*, 3. Auflage, W3L-Verlag, 2014

[Codd 1970] Codd, Edgar F.: *A Relational Model of Data for Large Shared Data Banks* [PDF], 1970, URL: https://www.seas.upenn.edu/~zives/03f/cis550/codd.pdf (Letzter Zugriff: 15.10.2015)

[Codeproject 2012] Igor Ignatov: *Inheritance mapping strategies in Fluent Nhibernate,* 2012, URL: http://www.codeproject.com/Articles/232034/Inheritance-mapping-strategies-in-Fluent-Nhibernat (letzter Zugriff: 28.01.2016)

[EFTutorial] Entity Framework Tutorial: *Database Initialization Strategies in Code-First,* URL: http://www.entityframeworktutorial.net/code-first/database-initialization-strategy-in-code-first.aspx (letzter Zugriff: 28.01.2016)

[Faeskorn u.a. 2007] Faeskorn-Woyke, Heide; Bertelsmeier, Birgit; Riemer, Petra; Bauer, Elena: *Datenbanksysteme - Theorie und Praxis mit SQL2003, Oracle und MySQL*, Pearson Studium, 2007

[Fowler u.a. 2002] Fowler, Martin u.a.: *Patterns of Enterprise Application Architecture*, Addison Wesley, 2002

[Kemper u.a. 2006] Kemper, Alfons; Eickler, André: *Datenbanksysteme - Eine Einführung*, 6. Auflage, Oldenbourg Verlag, 2006

[Kuaté u.a. 2009] Kuaté, Pierre Henri; Harris, Tobin; Bauer, Christian; King, Gavin: *NHibernate in Action*, Manning, 2009

[Kühnel 2016] Kühnel, Andreas: *C# 6 mit Visual Studio 2015*, 7. Auflage, Rheinwerk Verlag, 2016

[Lahres u.a. 2016] Lahres, Bernhard; Raýman, Gregor; Strich, Stefan: *Objektorientierte Programmierung – Das umfassende Handbuch*, 3. Auflage, Rheinwerk-Verlag, 2016

[Mertins u.a. 2012] Mertins, Dirk; Neumann, Jörg; Kühnel, Andreas: *Microsoft SQLServer 2012 - Das Programmierhandbuch*, 5. Auflage, Galileo Computing, 2012

[Microsoft 2015a] Lerman, Julie: *Ausblick auf Entity Framework 7*, MSDN-Blogpost, URL: https://msdn.microsoft.com/de-de/magazine/dn890367.aspx (letzter Zugriff: 28.01.2016)

[Microsoft b] MSDN: *Code First-Migrationen,* URL: https://msdn.microsoft.com/de-de/data/jj591621.aspx (letzter Zugriff: 28.01.2016)

[Microsoft c] MSDN: *Fluent-API – Konfigurieren/Zuordnen von Eigenschaften und Typen mit der Fluent-API*, URL: https://msdn.microsoft.com/de-de/data/jj591617.aspx (letzter Zugriff: 28.01.2016)

[Microsoft d] MSDN: *Leistungsüberlegungen zu Entity Framework 5,* URL: https://msdn.microsoft.com/de-de/data/hh949853.aspx (letzter Zugriff: 31.01.2016)

[Microsoft 2014e] ASP.NET: *Handling Concurrency with the Entity Framework 6 in an ASP.NET MVC 5 Application,* 2014, URL: http://www.asp.net/mvc/overview/getting-started/getting-started-with-ef-using-mvc/handling-concurrency-with-the-entity-framework-in-an-asp-net-mvc-application (letzter Abruf: 28.01.2016)

[Sarstedt 2014] Sarstedt, Stefan: *Vorlesung: Das Softwareprojekt,* Sommersemester 2014

[Wikipedia DB] Wikipedia – Die freie Enzyklopädie: *Datenbankschnittstelle*, URL: https://de.wikipedia.org/w/index.php?oldid=132089121 (letzter Zugriff: 28.01.2016)

[Wikipedia Isolation] Wikipedia – Die freie Enzyklopädie: *Isolation*, URL: https://de.wikipedia.org/w/index.php?oldid=147447457 (letzter Zugriff: 28.01.2016)

[Wikipedia Impedance Mismatch] Wikipedia – Die freie Enzyklopädie: *Impedance Mismatch*, URL: https://de.wikipedia.org/w/index.php?oldid=138335493 (Letzter Zugriff: 28.01.2016)

[Wikipedia NHibernate] Wikipedia – The Free Encyclopedia: *NHibernate*, URL: https://en.wikipedia.org/w/index.php?oldid=646879943 (letzter Zugriff: 28.01.2016)

[Wikipedia Objektorientierung] Wikipedia – Die freie Enzyklopädie: *Objektorientierung*, URL: https://de.wikipedia.org/w/index.php?oldid=138591884 (letzter Zugriff: 28.01.2016)

[Wikipedia Polymorphie] Wikipedia – Die freie Enzyklopädie: *Polymorphie*, URL: https://de.wikipedia.org/w/index.php?oldid=150064667 (letzter Zugriff: 28.01.2016)

[Wikipedia RDB] Wikipedia – Die freie Enzyklopädie: *Relationale Datenbank*, URL: https://de.wikipedia.org/w/index.php?oldid=149338316 (letzter Zugriff: 28.01.2016)

[Wikipedia Repository] Wikipedia – Die freie Enzyklopädie: *Repository*, URL: https://de.wikipedia.org/w/index.php?oldid=143670462 (letzter Zugriff: 28.01.2016)

[Wikipedia SoC] Wikipedia – The Free Encyclopedia: *Separation of Concerns*, URL: https://en.wikipedia.org/w/index.php?oldid=694825758 (letzter Zugriff: 28.01.2016)

[Wikipedia SQL] Wikipedia – The Free Encyclopedia: SQL, URL: https://de.wikipedia.org/w/index.php?oldid=150703502 (letzter Zugriff: 28.01.2016)

[Wikipedia Surrogatschlüssel] Wikipedia – Die freie Enzyklopädie: *Surrogatschlüssel*, URL: https://de.wikipedia.org/w/index.php?oldid=146231093 (letzter Zugriff: 28.01.2016)

[Wikipedia Vererbung] Wikipedia – Die freie Enzyklopädie: *Vererbung*, URL: https://de.wikipedia.org/w/index.php?oldid=150428444 (letzter Zugriff: 28.01.2016)

Anhang A

Innensichten der Referenzanwendung:

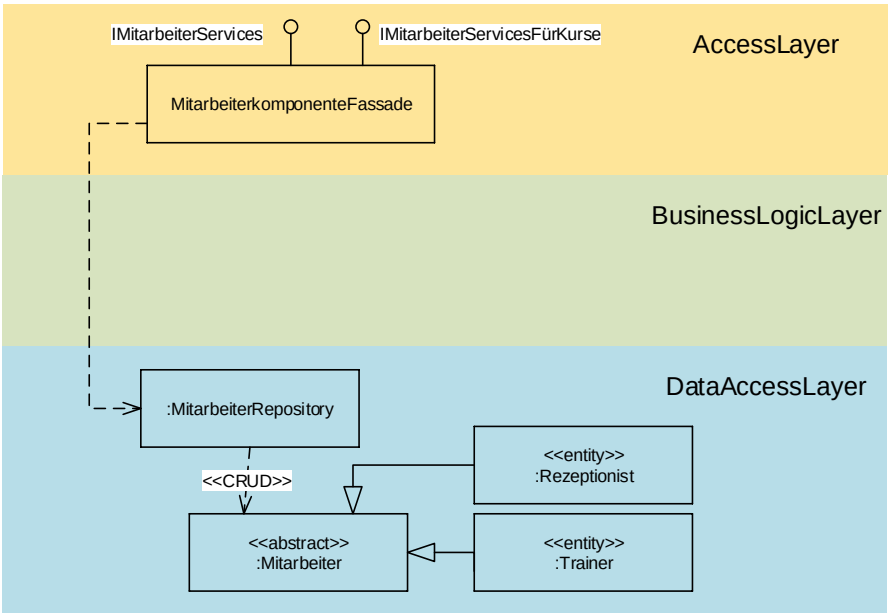

Abbildung 15: Innensicht Mitarbeiterkomponente

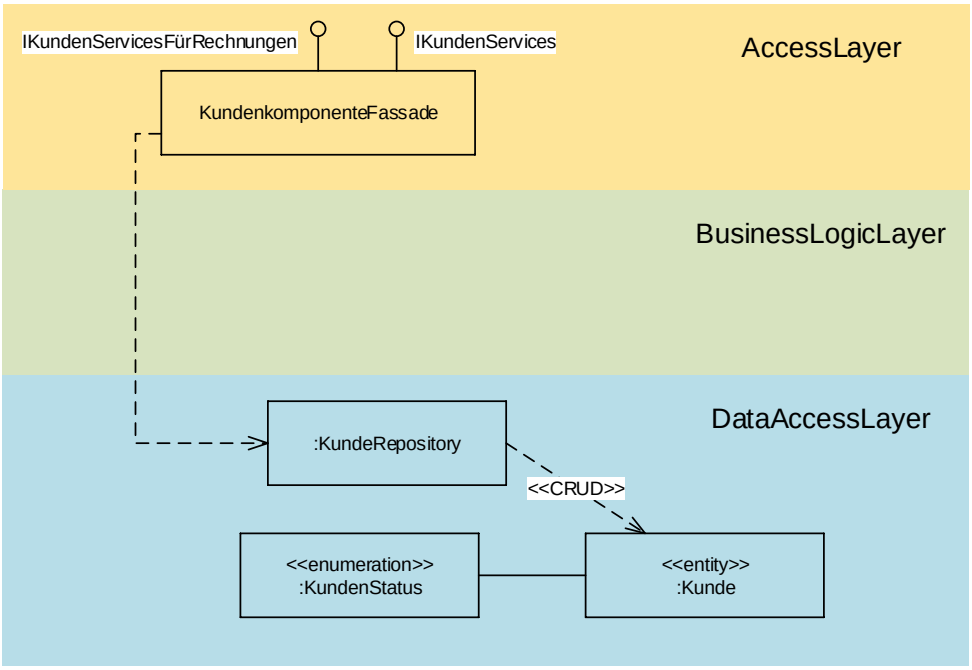

Abbildung 16: Innensicht Kundenkomponente

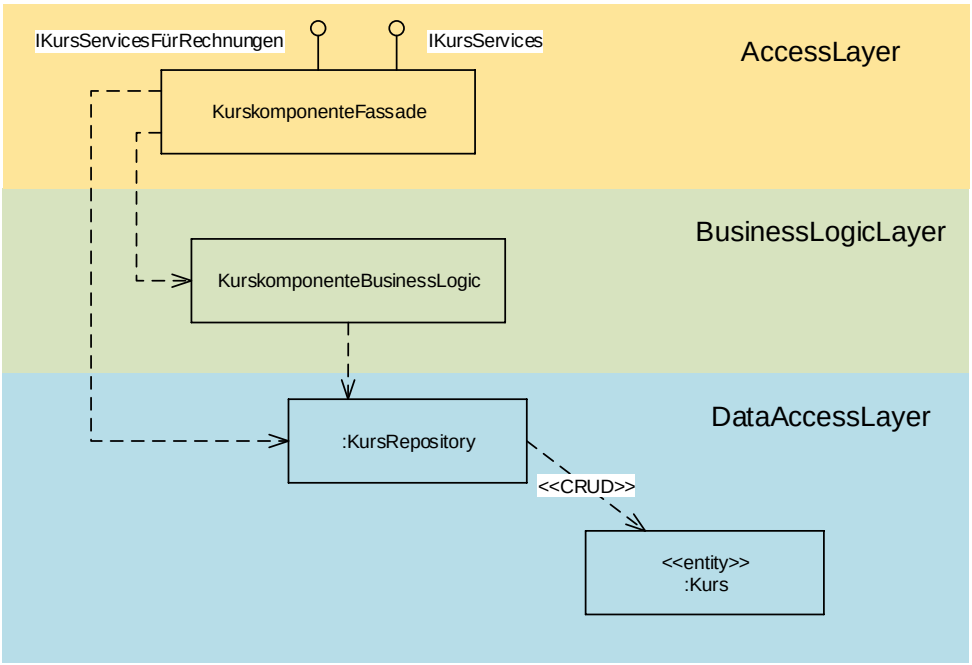

Abbildung 17: Innensicht Kurskomponente

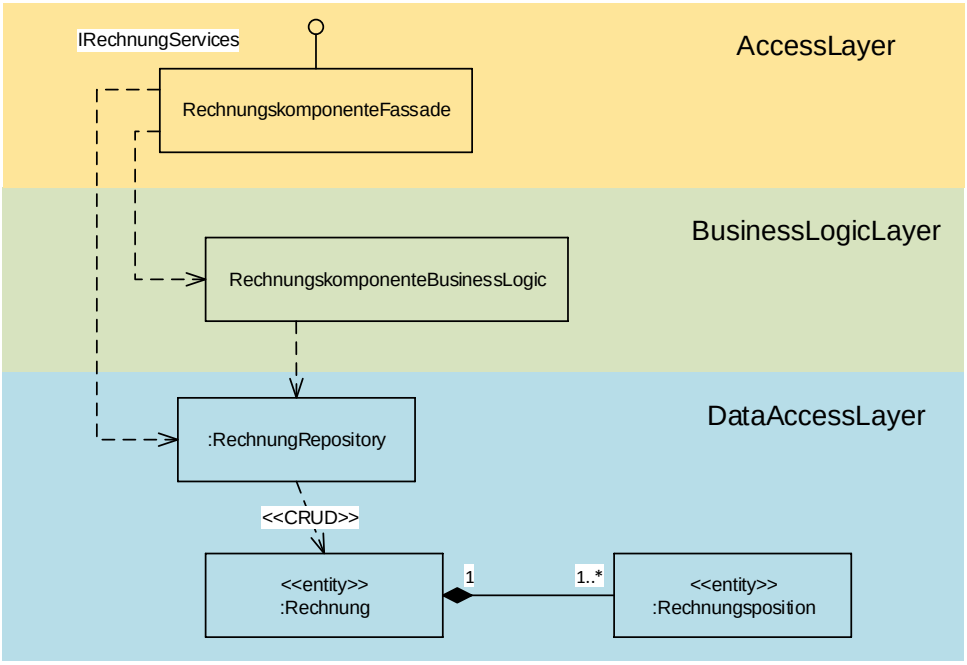

Abbildung 18: Innensicht Rechnungskomponente

Anhang B

Inhalte der CD-ROM:

- Implementation:
    - NHibernate: Implementation der Referenzanwendung in C# unter Verwendung von NHibernate (Visual Studio Projektmappe)
    - Entity Framework: Implementation der Referenzanwendung in C# unter Verwendung von Entity Framework (Visual Studio Projektmappe)
    - ActiveJDBC: Implementation der Referenzanwendung in Java unter Verwendung von ActiveJDBC (IntelliJ-IDEA Projekt)
- Bachelorarbeit als PDF-Dokument

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, den 05.02.2016 _____

Heinrich Latreider