Teil I

Die Theorie des Object-Relational Impedance Mismatch

INSTITUT FÜR INFORMATIK
Fachbereich Informatik und Mathematik

Diplomarbeit

Analyse und Lösungen für den Object-relational Impedance-Mismatch

Philipp Scheit

Abgabe am 13. Dezember 2010

eingereicht bei
Prof. Dott.-Ing. Roberto V. Zicari

Datenbanken und Informationssysteme

Erklärung

gemäß DPO §11 ABS.11

Hiermit versichere ich, dass ich die vorliegende Diplomarbeit selbstständig verfasst habe und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Frankfurt am Main, den 13. Dezember 2010 Philipp Scheit

Abstract

Zusammenfassung

Danksagung

text

INHALTSVERZEICHNIS

I Die Theorie des Object-Relational Impedance Mismatch I

1 Einführung 1
1.1 Der Aufbau dieser Arbeit 2

2 Die Kernprobleme 3

3 Lösungen 5
3.1 Ersetzen der Datenbank 5
3.2 Object-relational Mapper 9

4 Object Relational Mapping 11
4.1 Speichern des Zustandes eines Objektes 11
4.2 Speichern von Klassenhierarchien 12
4.2.1 Horizontales Mapping 12
4.2.2 Vertikales Mapping 12
4.2.3 Filter-Mapping 13
4.2.4 Generelles Mapping 13
4.3 Speichern von Beziehungen zwischen Objekten 14
4.4 das Laden und `Marshalling` von Objekten 15
4.5 Vertiefung: Object Loading 15

5 Frameworks 21
5.1 Übersicht 21
5.2 Hibernate und Vererbung 24
5.3 Kohana, Doctrine und Hibernate und die Abfragesprache 25
5.4 Beziehungen zwischen Objekten 26
5.5 Object Loading 28
5.6 Abbildung von Klassen und Attributen 29
5.7 Zusammenfassung 30

6 Fazit 33
6.1 Während des Schreibens 33
6.2 Die Lösung des Impedance Mismatch 33

II Ein Framework zur Lösung des Object-Relational Impedance Mismatch **35**

7 Einleitung **37**
7.1 Aufbau dieses Teils 37

8 Implementierungen in PHP **39**

9 Anforderungen **41**
9.1 Objektidentifizierung 41
9.2 Cache 41
9.3 Abfragesprache 41
9.4 Change detection 41
9.5 Basis Klassen 41
9.6 Mapping Metadaten 41
9.7 Object Loading 41

10 Implementierung des Frameworks **43**
10.1 Cache 43
10.2 dynamischer Code 43
10.3 Datenbankabstraktion 43
10.4 Abbildung von Klassen und Attributen 43
10.5 Abbildung von Vererbung 43
10.6 Beziehungen zwischen Objekten 43
10.7 Object Loading 43

11 Fazit **45**

A Beispiel Datenbank Schema **47**

Index **48**

Literaturverzeichnis **49**

KAPITEL

EINS

Einführung

`Impedance matching` bezeichnet in der Elektrotechnik den Vorgang den Eingangswiderstand des Verbrauchers oder den Ausgangswiderstand der Quelle in einem Stromkreis so anzupassen, dass der Wirkungsgrad genau 50% beträgt. Abstrakter formuliert, muss man Leistungen zwischen zwei Punkten angleichen um das Optimale Ergebnis zur Übertragung von Signalen oder Energie zu erhalten.
Der Begriff wurde dann in die Informatik übernommen.

Definition 1 (Object-relational Impedance Mismatch)**:**
Der Object-relational Impedance Mismatch bezeichnet die Unverträglichkeit zwischen dem relationalem Datenmodell (definiert durch das relationale Schema) und dem objektorientierten Programmierparadigma.

Die objektorientierte Programmierung ist heute eine der bekanntesten Programmierungsparadigmen und beeinflusst das Design, die Methoden und den Entwicklungsprozess von moderner Software. Die Strukturen der objektorientierten Programmierung eignen sich am besten um die Anwendungslogik (`Business Logic`) und ein Model der realen Welt abzubilden. Polymorphie, Kapselung und Vererbung sind aus moderner Softwareentwicklung nicht mehr wegzudenken.

Relationale Datenbanken wurden designt riesige Mengen von elementaren Datentypen sicher abzuspeichern und schnell und effizient abfragen zu können. Im relationalen Schema wird ein Model der realen Welt durch Relationen dargestellt. Relationen unterliegen dem mathematischen Konzept der mengentheoretischen Relation, dem Kartesischen Produkt von einer Liste von Ausprägugnen der elementaren Datentypen.

Beide Paradigmen eignen sich am besten die Probleme auf ihrem Gebiet zu lösen, unterscheiden sich aber grundlegend in der Denkweise, der Methodik und im Design.

Heutzutage benötigt fast jede objektorientierte Applikation einen persistenten Datenspeicher. Muss der Zugriff auf die Daten auch für große Mengen performant bleiben, werden fast immer relationale Datenbankmanagementsysteme (`RDMBS`) hinzugezogen. Ein existiert also immer das Bedürfnis das Model der objektorientierten

Applikation und das Model des relationalen Schemas zu miteinander zu verbinden. Nach einer Studie werden 30-40% der Entwicklungszeit einer Applikation dafür verwendet Objekt-relationale Lösungen für die Daten zu finden [Kee04]. Wenn also der `IM` also einmal sorgfältig gelöst wurde, hat das große Vorteile für die Entwicklung jeder weiteren Anwendung.
Bevor man also anfängt über die Anwendungslogik und die Probleme, die sich bei jedem Projekt neu ergeben nachzudenken, sollte man zuerst genug Zeit für den `IM` aufwenden, um nicht in jedem Projekt 30-40% der verfügbaren Zeit zu vergeuden [New06].
Der `Object-Relational Impedance Mismatch` muss gelöst werden.

1.1 Der Aufbau dieser Arbeit

KAPITEL

ZWEI

Die Kernprobleme

Es stellt sich also die Frage, welche konkreten Probleme zur Überbrückung des `Impedance Mismatches` gelöst werden müssen. Um den Zustand der Objekte und das Schema der objektorientierten Applikation abdeckend in der Datenbank persistent speichern zu können, müssen zumindest folgende Teilprobleme bearbeitet werden:

Speichern der Struktur einer Klasse Ein Objekt im objektorientierten Model besitzt eine Klasse. Die Klasse definiert die Methoden und Eigenschaften des Objektes und kann sich in einer Klassenhierachie befinden. Im relationalen Schema sind Klassen nicht enthalten und können deshalb nicht natürlich abgebildet werden.

Speichern des Zustandes eines Objektes Eine Objektinstanz kann in der Applikation mehrere Zustände annehmen und wird von Methoden von einem in den nächsten übergeführt. Der Zustand des Objektes muss im relationalen Schema dargestellt werden. Wie kann dieser identifizierbar sein und wieviel müssen wir von der Objektinstanz in der Datenbank pflegen, um dies zu erreichen.

Speichern von Beziehungen zwischen Objekten Im objektorientierten Model haben Objektinstanzen die Möglichkeiten untereinander Daten auszutauschen. Objekte bestehen nicht nur allein sondern sind mit andern Objekten gekoppelt (Assoziation), aus anderen Objekten zusammengesetzt (Komposition) oder beeinhalten weitere Objekte (Aggregation). Diese Objekt-Beziehungen müssen im persistenten Datenspeicher festgehalten werden, denn auch sie bestimmen den Zustand eines Objektes. Ein Spezialfall einer Beziehung zwischen Objekten ist die Vererbung.

Speichern von Klassenhierarchien (Vererbung) Objektorientierte Modelle haben keine flachen Hierarchien. Klassen stehen durch das Prinzip der Vererbung in einer hierarchischen Beziehung untereinander. Im relationalem Schema ist es nicht vorgesehen Vererbung zu modellieren.

Abfrage von Objekten Mit einem `RDMBS` können komplexe Abfragen einfach gestellt und effizient bearbeitet werden. Diese Abfragen beziehen sich meist auf eine Menge von Objekten mit bestimmten Kriterien. Der Zugriff auf die Objekte wird also über Mengen erreicht. Im objektorientierten Model geschieht der Zugriff über die Navigation über die Beziehungen von Objekten. Eine Gesamtsicht auf alle Objekte oder das Bilden von konkreten Mengen von Objekten ist schwierig, da dann Pfade von Beziehungen analyisert werden müssen.

`Marshalling` von Objekten Das `Marhshalling` bezeichnet den Vorgang aus einem Abfragergebnis eine Menge von konkreten Objektinstanzen zu erstellen, die in der Applikation verwendet werden. Oft findet man auch die Begriffe `Hydrating` oder `Object Retrieving`. Ein Objekt kann eine beliebige Struktur haben, ein Abfrageergebnis ist aber immer ein einfaches Tupel. Wie erhält man aus dieser flachen Struktur eine Objektinstanz?

Aus diesen Teilproblemen ergeben sich Anforderungen für eine Lösung des `Impedance Mismatch`. Jedes Kernproblem erzeugt eine große Menge von Folgeproblemen, die sich auch überlappen können. Die Komplikationen lassen sich auch als Eigenschaften der Paradigmen wie in [IBNW09, S. 38] beschreiben. Diese sind dann:

- (Structure) Eine Klasse hat eine beliebige Struktur und eine beliebiges Verhalten definiert durch Methoden. Diese Struktur muss abgebildet werden
- (Instance) Der Objektzustand muss abgebildet werden.
- (Encapsulation) Auf ein Objekt wird über Methoden zugegriffen. Es kapselt sein Verhalten durch diese Methoden und wird somit nicht von außen definiert. Daten in der Datenbank haben keine Methoden und sind von überall modifizierbar.
- (Identity) Ein Objekt muss eine eigene Identität haben, die in beiden Modellen eindeutig ist
- (Processing Model) Der Zugriff auf Objekte innerhalb des objektorientierten Models geschieht über Pfade bestehend aus Objekten. Im relationalen Modell wird auf Objekte (bzw Daten) in Mengen zugegriffen.
- (Ownership) Das objektorientierte Model der Applikation wird vom Entwicklerteam der Software verwaltet. Das Datenbankschema vom Datenbankadministrator. Das Datenbankschema kann nicht nur von einer einzelnen Applikation benutzt werden, sondern von mehreren. Wie werden also Änderungen des Datenbankschemas oder der Applikation behandelt, sodass alle Applikationen auf die Änderungen reagieren können?

Die Liste der Probleme ließe sich noch weiter vergrößern, wenn man sich weitere Details von Datenbanksystemen und objektorientierten Programmiersprachen betrachtet. Eine detailiertere Vertiefung bieten Cook and Ibrahim [CI05].

KAPITEL

DREI

Lösungen

Es gibt verschiedene Lösungen für den IM. Zunächst möchte ich eine vorstellen, die dem eigentlichen IM aus dem Weg geht und einen anderen Ansatz verfolgt: Das Ersetzen der relationalen Datenbank mit einem anderen Datenbankmanagmentsystem. Danach beschäftige ich mich mit dem Object-Relational Mapping, welches die klassische Lösung des IM ist.

3.1 Ersetzen der Datenbank

Zunächst werden Objektdatenbankmanagementsysteme (OODBMS bzw ODBMS) betrachtet.

Definition 2 (ODBMS)**:**
An object-oriented database system must satisfy two criteria: it should be a DBMS, and it should be an object-oriented system, i.e., to the extent possible, it should be consistent with the current crop of object-oriented programming languages. The first criterion translates into five features: persistence, secondary storage management, concurrency, recovery and an ad hoc query facility. The second one translates into eight features: complex objects, object identity, encapsulation, types or classes, inheritance, overriding combined with late binding, extensibility and computational completeness. [ABD+ 89]

Der IM wird also überbrückt, indem das RDMBS mit einem System ersetzt wird, welches den Prinzipien des objektorientierten Paradigma folgt. Das bedeutet, dass viele Ausprägungen der Probleme die vorher genannt wurden auf natürliche Weise nicht mehr gelöst werden müssen. Gerade die Eigenschaften des zweiten Kriteriums wie: object identity, encapsulation und inheritance lösen die schwierigsten Probleme die in [IBNW09, S. 38] beschrieben werden. Das Objekt welches persistent gemacht werden soll, muss nicht mehr in atomare, in einer Datenbank speicherbare Datentypen zerlegt werden und beim Laden wieder zusammengesetzt werden. Gerade für komplexe Objekte mit vielen Abhängigkeiten untereinander ist dies ein enormer Vorteil. Der Grund warum dies mit ODBMSs so gut funktioniert ist, dass diese Systeme nicht mehr Relationen als Basis benutzen, sondern man sich veranschaulicht vorstellen kann, dass das Objekt „so wie es ist“ in der Datenbank gespeichert

wird.
Man könnte also annehmen, dass ODBMSs die sinnvollste, einfachste und ökonomischsten Lösungen für den IM sind. De facto ist es aber so, dass ODBMSs sich noch nicht so durchsetzen konnten, wie man anfänglich erwartet hatte. Wenn ein ODBMSs keine Relationen mehr benutzt, dann ist auch klar, dass man die Vorteile, die ein relationales DMBS mit sich bringt, verliert. In [Lea00] werden weitere folgende Gründe genannt, warum ODBMSs nicht so erfolgreich wurden:

- RDBMSs wurden mit objektorientierten Features bestückt und entwickelten sich so weiter. Dadurch gewannen die bereits in der Indstrie etablierten Datenbanksysteme wieder ihre verlorene Popularität gegenüber ODBMSs zurück.

- In ODBMSs ist es performanter komplexe Mengen von Datenobjekten zu speichern, ODBMSs können den Speicher des Clients direkt als Cache verwalten und der Optimierer des Systems wählt die beste Form für das physikalische Design der Datenbank. Diesen Vorteil reduzierten die RDBMSs indem sie die ihre Prozesse Daten aus Tabellen abzufragen, weiter entwickelten und optimierten.

- Zwar ist es vorteilhafter eine komplexes, objektorientiertes Schema einer Applikation mit einem ODBMSs persistent zu machen, doch die Bereiche in denen diese Schemas existieren und ODBMSs ihre Anwendungen finden sind nur kleine Nieschenmärkte.

- RDBMSs stützen sich auf den lang etablierten Standard SQL. Die Object Database Management Group entwickelt zwar Standards für ODBMSs und ORM-Produkte, aber der Standard ist nicht weit verbreitet und zu wenige Hersteller unterstützen den Standard, um ihn wirklich für die Industrie relevant zu machen. So konnte sich zum Beispiel der OQL-Standard (Object Query Language) nie richtig durchsetzen.

- Das Fehlen von SQL wurde zu spät vond en Herstellern als gravierender Nachteil der ODBMSs realisiert.

- Firmen die bereits ein bestehendes RDMBS, bzw die Abwandlung davon als ORDMBS als bevorzuge Datenbanklösung benutzen, wechseln gerechtfertigter Weise zu keiner neuen Lösung. Nur wenn die neuen Systeme einen unwiderstehlich lukrativen, wirtschaftlichen Vorteil gegenüber den alten bieten würden - was ODBMSs nicht können - gäbe es eine Chance, RDBMSs in ihrer Vormachtsstellung in Gefahr zu bringen. Hinzu kommt, dass bestehende Hersteller von RDBMSs mehr Ressourcen und Mittel z. B. fürs Marketing zur Verfügung haben, weil sie bereits eine größere Basis an verkauften Produkten besitzen.

Die genannte Erweiterung der relationalen Datenbankmanagmentsysteme mit objektorientierten Features bezeichnet man als Objektrelationales Datenbankmanagementsystem kurz ORDMBS. Die Definition eines ORDMBS ist schwer zu formulieren. Um genauer zu sein, gibt es überhaupt keinen Standard oder eine von allen anerkannte wissenschaftliche Arbeit, die einem erlauben würde ein ORDMBS zu charakterisieren.
Der Übergang von einem RDMBS zu einem ORDMBS kann also mehr oder weniger fließend sein. Somit können relationale Systeme schon als objektrelationale Systeme

bezeichnet werden, wenn sie nur ein objektorientiertes Feature umsetzen. In [Dev01] wird versucht die Charakteristika eines `ORDMBS` so zu beschreiben:

- Erweiterung von Basisdatentypen
- Unterstützung von komplexen Objekten
- Vererbung

Es ist also nicht leicht ein konkretes `DMBS` als `ORDMBS` oder `RDMBS` zu bezeichnen. Um trotzdem ein Gefühl entwickeln zu können, wie weit sich die Hersteller um die Entwicklung ihres `RDMBS` zu einem `ORDMBS` kümmern, zeigt Abbildung 3.1 eine mögliche Klassifizierung einiger bekannter Datenbankmanagmentsysteme (Quelle: Evans Data Corp: Users' Choice Database Servers (2008)). Desto weiter ein Datenbanksystem auf dem Strahl weiter links steht, desto eher ist es meiner Einschätzung nach als reines `ODBMSs` charakterisierbar. Desto weiter ein `DMBS` rechts auf dem Strahl erscheint, desto weniger objekt-relationale Features unterstützt es. Es ist zu beachten, dass die meisten `RDMBS` sich in Richtung einer Position weiter links auf dem Strahl entwickeln, während sich nur wenige `ODBMSs` in die relationale Richtung bewegen. Anbei noch ein paar Anmerkungen zur Platzierung der verschiedenen `DMBS` auf dem

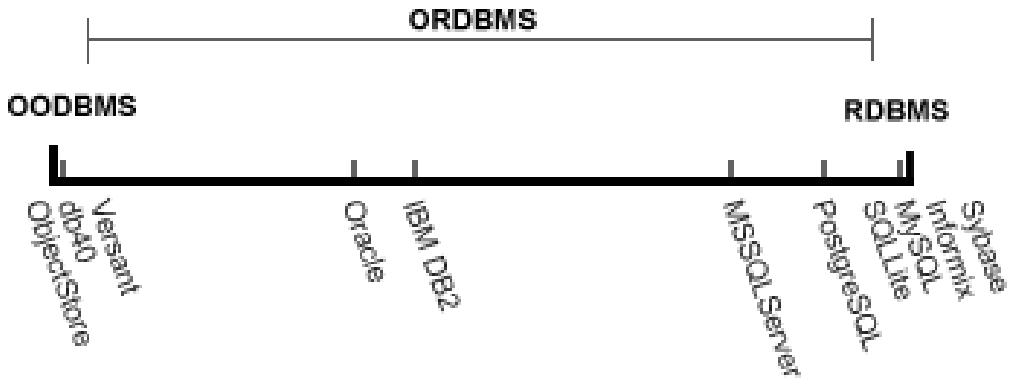

Abbildung 3.1: Klassifizierung beliebter Datenbankmanagmentsysteme

Strahl:

Oracle11 Mit `Oracle Objects` können Benutzerdefinierte Objekte (Objekttypen) in der Datenbank gespeichert werden. Diese Objekte werden aus Basisdatentypen oder anderen Objekten zusammengesetzt. Zusätzlich zu der Definition des Objekttypen können auch Methoden definiert werden. Danach ist es möglich diesen Typen in einer Tabelle zu verwenden. Dabei kann dieser Objekttyp mit verschiedenen anderen relationalen Informationen vermischt werden. Wird der Objekttyp als Typ einer Tabelle benutzt (`Object Tables`) kann sowohl auf das ganze Objekt (und dessen definierten Methoden), als auch auf jede Spalte einzeln zugegriffen werden. D. h. es ist möglich, das Objekt als Objekt oder auch als eine einzelne Zeile mit Spalten anzusprechen, um relationale Features nutzen zu können. Es sind Referenzen auf

Objekte in Tabellen möglich, es gibt Collections und Typen können voneinander abgeleitet werden. Extensions für die Spracherweiterung gibt es für C++, .NET und Java und natürlich über das eigene `Call Level Interface`; Zusätzlich gibt es eine SQL Spracherweiterung (inklusive DDL Statements) [Ora08].
Diese Sammlung von Eigenschaften und Fähigkeiten macht Oracle für mich zu dem DBMS welches man am ehesten als `ORDMBS` bezeichnen kann.

DB2 In DB2 ist es ebenfalls möglich benutzerdefinierte Typen zu erstellen. Dieses `distinct Types` können von Basisdatentypen abgeleitet werden und sind somit auch gut in den Database Manager integriert. Zusätzlich können diesen Typen auch benutzerdefinierte Funktionen zugewiesen werden. Es wird ein Grundstock von Multimedia Datentypen mitgeliefert: Audio, Bilder, Video und Text. Es kann eine Tabelle von Funktionen (statt Skalaren) zurückgegeben werden. Somit ist es möglich, jede Datenquelle als Tabelle darzustellen und so die gewohnten relationalen Features zu benutzen. Mit dem `structured Type` ist auch Vererbung möglich, denn `Structured Types` können von anderen Typen abgeleitet werden. Dadurch können dann Tabellen von einem `structured Type` ebenso wie in Oracle als Objekttabellen erstellt werden und diese wiederum von anderen Tabellen die Spalten und Methoden erben. Auch hier ist es möglich die Objekte in der Tabelle als separate Spalten anzusprechen. Zusätzlich werden Path-Expressions für die Abfrage von Beziehungen von Objekten genutzt und `Object Views` können erstellt werden. Dokumentation: [IBM01].

Microsoft SQL Server Features die man bei IBM und Oracle findet, findet man weniger in den Feature-Listen von Microsoft SQL Server ([Mic09]). Ich denke, das liegt daran, dass Microsoft andere Wege gefunden hat mit dem `IM` umzugehen [Mei06]. So findet sich z. B. LINQ als Objekt-relationales Feature wieder. Trotzdem liefert auch Microsoft eine Möglichkeit für Multimedia Datentypen (`Large User-Defined Types: UDTs`). Zu diesen Typen werden auch neue Definitionen von Aggregatfunktionen angeboten: `Large User-Defined Aggregates: UDAs`. Mit `ORDPATH` können Hierarische Daten als Baum abgespeichert werden.

PostgreSQL Als letztes rutscht PostgreSQL noch etwas in die `ODBMSs`-Richtung, da es versucht den SQL3 Standard (in Entwicklung) umzusetzen. Deshalb ermöglicht es jetzt schon z.B. in Funktionen Tabellen zurückzugeben, oder die Verwaltung von XML als Datentypen. Es erweitert auch einige seiner Basisdatentypen zu Multimediatypen.

Caché von Intersystems Caché von Intersystems [Int10] bezeichnet sich selbst als postrelationales Datenbanksystem. In der Abbildung taucht der Name nicht auf, da man es an jedem Punkt des Strahls zeichnen könnte. Caché bietet sowohl eine

relationale als auch eine objektorientierte Datenbankansicht. Die Datenbasis ist weder relational noch objektorientiert, sondern hierarchisch.

3.2 Object-relational Mapper

Zwar lösen ODBMSs die Schwierigkeiten die der IM mit sich bringt, doch je nach dem welches ORDMBS oder RDMBS benutzt wird, gibt es nach wie vor genug Probleme. Es lässt sich nur schwer vorraussagen, wann sich Datenbanksysteme so verändern, dass es in der praktischen Anwendung überhaupt keine relationalen Datenbanken mehr gibt. Deshalb gibt es genug Szenarien in denen der IM oder Teile davon nach wie vor eine Rolle spielen. Das bedeutet, es muss nach wie vor eine universellere Methode gefunden werden.
Wenn man die zu lösenden Schwierigkeiten betrachtet, fällt auf, dass die meisten dadurch entstehen, dass ein Element aus dem objektorientierten Paradigma im relationalen Schema nicht abgebildet werden kann. Es muss also ein Element (oder eine Kombination aus mehreren) des relationalen Models gefunden werden, das dieses Element des objektorientierten Models darstellen kann. Die Abbildungen zwischen diesen Elementen - auch Mapping genannt - können in verschiedenen Weisen realisiert werden. Eine Lösung des IM die eine Menge von Mappings definiert und relationale Elemente in objektorientierte Elemente oder vice versa überführt, nennt man einen Object Relational Mapper oder kurz ORM.
Im folgenden Kapitel konzentriere ich mich auf Object-relational Mappings. Ich stelle verschiedene schon erforschte Mappings für bestimmte Kernprobleme vor und erkläre die Probleme, die dabei entstehen.
Im Kapitel darauf betrachte ich ein paar Implementierungen dieser Strategien und zeige Vor- und Nachteile bei der Lösung dieser Probleme.

KAPITEL

VIER

Object Relational Mapping

Die folgenden Strategien haben sich für einen `ORM` bewährt und lassen sich auch größtenteils in der Literatur wiederfinden. Höchstwahrscheinlich existieren in der Praxis noch viel mehr implementierte Lösungen der Problematik. Dadurch, dass viele Hersteller eine eigene, unabhängige ORM-Lösungen implementieren mussten, weil es keine benutzbare, öffentliche Library oder Ähnliches gab, gibt es eine Vielzahl von unterschiedlichen Ansätzen. Es ist schwer sich für einen richtigen Lösungsansatz zu entscheiden, da sich von Fall zu Fall die Anforderungen stark ändern. Die möglichen Schwerpunkte durch die sich die Lösungen unterscheiden sowie Vor- und Nachteile sollen in späteren Kapiteln weiter untersucht werden. Deshalb werden hier zunächst die meist genutzen, in der Literatur am meisten genannten und in der Praxis etabliertesten Strategien vorgestellt.

4.1 Speichern des Zustandes eines Objektes

Die Konvention, die in den meisten `Mappings` getroffen wurde ist, dass eine Klasse als eine Relation (Tabelle) im Datenbankschema dargestellt wird. Der Name der Relation ist der Name der Klasse. Die Attribute (Spalten) der Relation sind die Attribute der Instanz der Klasse.
Den Zustand eines Objektes können wir also persistent machen, in dem wir die Ausprägungen aller Attribute des Objektes in die Relation einfügen. Eine Instanz eines Objektes wird dann durch ein Tupel in der Relation seiner Klasse repräsentiert.
Dadurch, dass ein Tupel in einer Relation eindeutig unterscheidbar sein muss, kann es sein, dass weitere Eigenschaften zur Klasse hinzugefügt werden müssen. Diese nennt man auch `Shadow information`. Meist ist die `Shadow Information` eine künstlicher numerischer Schlüssel, der sich für jedes neu eingefügte Tupel der Relation um 1 erhöht. Somit ist gewährleistet, dass jede Instanz der Klasse - auch wenn ihre Attribute gleich sind - eindeutig unterscheidbar ist. Dieses Prinzip wird Eindeutigkeit der Objekt-Identität genannt. Ich schreibe OID für `Object Identifier` und meine damit jene `Shadow information` die ein Objekt in der Applikation und in der Datenbank eindeutig identifiziert. Meist haben Objektinstanzen in objektorientierten Programmiersprachen eine eigene eindeutige Identität. Diese ist meist nicht als `Shadow information` verwendbar, da beim Laden eines Objektes aus der Daten-

bank meist eine neue Instanz der Klasse mit den Werten aus der Datenbank befüllt wird. Dabei besitzt dann das eben geladene Objekt A in der Applikation eine andere Identität als ein weiteres Objekt B, welches das selbe Tupel in der Datenbank repräsentieren kann. Dies ist in fast allen `Mappern` unerwünscht, da man nun nicht mehr Änderungen in Objekt A auf Änderungen in Objekt B beziehen kann und man unerweigerlich Konflikte beim Speichern von Objekt A und Objekt B erhält. Ich gehe also - wenn nicht anders genannt - davon aus, dass in der Applikation für jedes Tupel einer Klassenrelation zu jedem Zeitpunkt nur eine Objekt-Instanz existiert.

4.2 Speichern von Klassenhierarchien

Die Modellierung einer Klassenhierarchie im relationalen Schema ist eines der komplexen Probleme des `IM`. Trotzdem wurde mit der Zeit mehr als nur eine Variante gefunden. Es gibt mehrere `Mappings` für die Darstellung der Vererbung:

4.2.1 Horizontales Mapping

Jede Klasse der Hierarchie besitzt eine eigene Tabelle. In jeder Tabelle werden alle Attribute der eigenen Klasse A und die Attribute der Klassen, von denen Klasse A geerbt hat, als Spalten eingetragen. Die Attribute der Elternklassen werden also auf jede Kindklasse dupliziert.
Die Beziehung zwischen der Relation der Kindklasse und der Relation der Elternklasse wird meistens gar nicht abgebildet. Das heißt die Information, dass Klasse A Kind von Klasse B ist, ist nur im objektorientierten Model bekannt. Das Mapping ist also mit dieser Darstellung der Vererbung nicht bidirektional (Es gibt keine Möglichkeit ein ausschließlich relationales Model in ein objektorientiertes Model zu transformieren, wenn dieses Mapping benutzt wird). Um eine Klasse aus der Hierarchie zu laden, muss genau eine Tabelle abgefragt werden. Jedoch gibt es Probleme, wenn die Struktur einer Klasse weit unten in der Hierarchie geändert wird. Wird z. B. ein Attribut umbenannt, muss die Änderung in allen Kindklassen ebenfalls erfolgen. Dieses Ändern ist fehleranfällig und aufwendig.

4.2.2 Vertikales Mapping

Anmerkung (Tiefe einer Klassenhierarchie)**:**
Ich bezeichne die Tiefe der Klassenhierarchie als die Anzahl der Eltern und rekursiv dessen Eltern von einer Klasse. Wird also Klasse A von Klasse B abgeleitet und Klasse C von Klasse B dann ist die Tiefe der Hierarchie von A,B und C = 3.

Jede Klasse der Hierarchie besitzt eine eigene Tabelle, jedoch werden die Attribute der Elternklassen nicht in den Kindklassen dupliziert. Jede Relation der Kindklasse besitzt einen Fremdschlüssel auf den Schlüssel der Relation der Elternklasse (mehrere bei Mehrfachvererbung). Ein Objekt der Kindklasse K abgeleitet von Elternklasse E kann also nur aus der Datenbank geladen werden, wenn das Tupel aus der Relation K und das dazu passende Tupel der Relation E geladen wird. Die Eltern Klasse kann jedoch wiederum von einer weiteren Klasse abgeleitet sein. Je nachdem wie tief

die Klassenhierarchie ist, ist der gespeicherte Zustand eines Objektes auf mehr als eine Tabelle verteilt. Für eine sehr tiefe Klassenhierarchie müssen also für das Laden eines Objektes sehr viele Tabellen abgefragt werden. Das Ändern der Struktur einer Elternklasse ist kein Problem, da es keine Duplikate der Attribute gibt. Jede Klasse kapselt seine Attribute in seiner Relation.

4.2.3 Filter-Mapping

Um die komplette Hierarchie abzubilden wird nur eine Tabelle benutzt. Alle Instanzen der Klasse werden als Tupel in dieser Tabelle repräsentiert. Alle Attribute der kompletten Hierarchie befinden sich nur in dieser Tabelle. Da beim Einfügen eines Tupels von Klasse A nicht alle Spalten von Klasse B benötigt werden, muss es erlaubt sein dass ein Tupel der Klassentabelle leere Attribute der Klasse B bzw A besitzt. Da dadurch ein Tupel eingefügt werden kann, dessen Attribute alle leer sind (außer der OID natürlich), muss die Zugehörigkeit eines Tupels zu der Klasse in der Hierarchie als `Shadow information` gespeichert werden. Meist geschieht dies durch eine weitere Spalte der Tabelle mit dem Namen der Klasse zu der das Tupel gehört. Diese Spalte wird deshalb auch Filter (bei Hibernate `Discriminator`) genannt und gibt der Strategie den Namen. Dadurch dass jedes Feld (bei disjunkten Klassenattributen) leer sein darf, ist es nicht mehr möglich durch das relationale Schema die Integrität der Daten zu gewährleisten. Auch ist es möglich Attribute in der Datenbank zu speichern, die nicht zur Klasse des Tupels gehören. Die Applikation muss alle Abfragen mit dem Kriterium für den Filter erweitern, somit muss oft ein Index über den `Discriminator` gelegt werden, damit diese Abfrage performant bleiben.

4.2.4 Generelles Mapping

Anmerkung (Namen von Relationen)**:**
Namen von Relationen (Tabellen) eines Schemas werde ich rekursiv markieren.

Beim generellen `Mapping` erhält keine Klasse der Hierarchie eine eigene Relation. Die komplette Struktur einer Klasse (Name der Klasse, alle Attribute, der Name der Elternklasse(n)) wird in einem generellen Schema dargestellt. Eine Relation dieses Schemas speichert zum Beispiel die Namen der Klassen und die Referenz auf eine Elternklasse. Eine andere Relation modelliert die Namen und Typen der Attribute und die Zugehörigkeit zu einer Klasse.

Die Abbildung 4.1 zeigt ein Beispiel für so ein solches Schema. In *Klassen* sind die Namen aller Klassen der objektorientierten Applikation gelistet. In *Beziehungen* sind Vererbungen, Assoziationen und Kompositionen von Klassen gespeichert. Jede Klasse besitzt n Attribute, deshalb gibt es eine Beziehung zu Attribute die wiederum Objekte mit Werten verbindet. In *Werte* stehen alle Werte alle Objekte in der Applikation. Jede Instanz in der Tabelle *Instanzen* hat eine eigene OID (`Shadow Information`). Eine Instanz kann von einer anderen Instanz abgeleitet sein, oder mit einer anderen Instanz eine Beziehung haben (z. B. Vererbung).

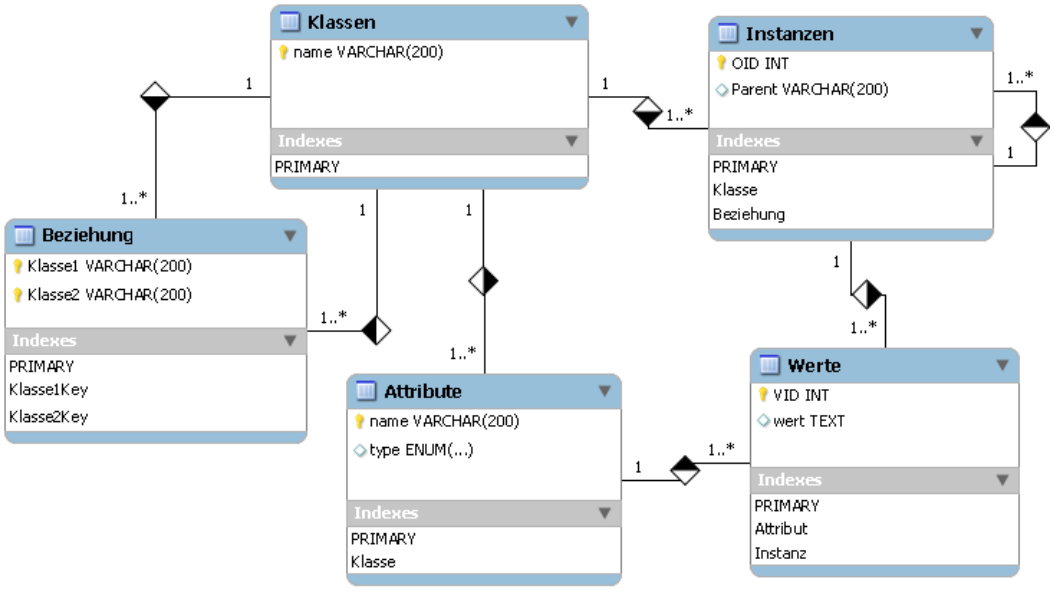

Abbildung 4.1: Beispiel einer Struktur für generelles Mapping

4.3 Speichern von Beziehungen zwischen Objekten

Die Beziehungen zwischen Objekten können so wie Beziehungen zwischen Relationen im klassischen relationalen Datenbankschema gelöst werden. Je nach Multiplizität der Beziehung (1:1 1:n oder n:m) werden Fremdschlüssel der Relation B in der Relation A angelegt, wenn A und B eine Beziehung eingehen. Bei einer Multiplizität von n:m muss eine weitere Beziehungs-Relation eingefügt werden die von Relation A und Relation B jeweils einen Fremdschlüssel speichert.
Dieser klassische Ansatz reicht für die meisten Applikationen aus und der Großteil der ORM-Software benutzt diesen auch. Dennoch wird z. B. in [LG07] noch ein andere Möglichkeit beschrieben: Für jede Beziehung zwischen Objekten wird eine Beziehungs-Relation eingefügt. Je nach Multiplizität werden in der Tabelle beide Fremdschlüssel einzeln (1:1), nur eine Seite (1:n) oder beide kombiniert mit Unique-Constraints belegt. Der Grund ist, dass z. B. bei einer Beziehung zwischen Relation A und Relation B mit Multiplizität 1:n im klassischen Schema der Fremdschlüssel in Relation B eingefügt werden muss. Betrachtet man Relation A also außerhalb dieser Beziehung ist nicht mehr zu erkennen, dass eine Beziehung von Relation A nach Relation B überhaupt besteht. Dies entspricht nicht dem Prinzip der Kapselung von Informationen im Objekt. Außerdem kann die Liste der Fremdschlüssel in einer Relation die an besonders vielen Beziehungen beteiligt ist, schnell sehr lang werden. Durch eine empirische Untersuchung kommt [LG07] zu dem Entschluss, dass die Lösung mit vielen Beziehungs-Relationen ungefähr gleich schnell im Vergleich zu der klassischen ist.

4.4 das Laden und Marshalling von Objekten

Das Laden von Objekten wird in den meisten ORM-Lösungen zu unausführlich behandelt. Mit einer naiven Idee lässt sich das Problem meist zufriedenstellend lösen. Es ist z.B. sehr einfach ein einzelnes Objekt mit einer gegebenen Objekt-ID aus der Datenbank zu laden und die passende Instanz in der objektorientierten Applikation zu erstellen:

- Die Datenbank muss nach dem Tupel für die Objekt-ID angefragt werden.
- Die Werte des Tupels müssen in die Attribute des Objektes überführt werden (Marshallling)

Kritischer wird es jedoch, wenn man eine größere Menge von Objekten gleichzeitig laden will. Der naive Ansatz - der zwar das Prinzip der Kapselung von Eigenschaften und Methoden einhält - scheitert hier an der Leistungsfähigkeit. Für eine Menge von n Objekten werden n - zugegeben sehr einfache - Abfragen an die Datenbank getätigt, anstatt eine „große“ Abfrage zu tätigen und über das Ergebnis zu iterieren. Hier kommt die Eigenart des IM besonders gut hervor.
Ein weiterer wichtiger Aspekt ist die Entscheidung wie die Anfragen an das System gestellt werden. Es stellt sich die Frage, ob es besser ist dem Anwender eine bekannte Anfragesprache als API zur Verfügung zu stellen (wie z.b. SQL) oder aber alle Details für das Laden von Objekten in abstrakten Methoden der objektorientierten Applikation zu Kapseln. Die Meinungen ob die Anfragesprache als SQL oder als abstrakte Query Sprache umgesetzt werden soll gehen weit auseinander. Somit wird in [Amb98, S. 7] die Benutzung von SQL als Verletzung der Kapselung angesehen. Wenn in der Applikation SQL direkt benutzt wird, bindet man die Applikation an das Datenbankschema. Man verliert also die Möglichkeit das Design des Schemas dem Mapper zu überlassen. Ebenso wird die Schnittstelle zur Datenbank nicht mehr abstrahiert. D.h. die Applikation ist von einem bestimmten DMBS abhängig. Wir das DMBS gewechselt muss auch die Applikation aufwendig geändert werden. Das Design der Abfragesprache wird also als besonders schwierig betrachtet. Die Sprache soll in der Domain des Objekt-Models designed werden, aber muss mächtig genug sein um alle komplexen SQL Fragen, die sonst geschrieben werden würden, behandeln zu können ([Rus08, S. 24]). Die Entscheidung SQL oder kein SQL zu benutzen ist deshalb sehr komplex und viele Lösungen unterscheiden sich durch das Design der Anfragesprache.

4.5 Vertiefung: Object Loading

Bevor ich im nächsten Kapitel auf verschiedene konkrete Implementierungen eingehen werde, möchte ich zunächst ein Thema noch weiter vertiefen: Das Laden von Objekten scheint nicht oft Inhalt von wissenschaftlichen Arbeiten zu sein, obwohl dort meiner Meinung nach, die besten Optimierungen bezüglich Performanz möglich sind.

Beginnt man eine ORM-Lösung naiv zu implementieren könnte man schnell zu folgender Struktur kommen:

- Jede Klasse, die persistent gemacht werden soll leitet eine Basisklasse ab in der die Persistensmechanismen als Methoden zur Verfüung stehen (save(), get(), delete())
- es gibt eine Methode get() mit dem Parameter OID welche den verknüpften Datensatz aus der Datenbank lädt (Verbindung herstellen, SQL absetzen, Daten auslesen, init() aufrufen)
- es gibt eine weitere Methode init() die die Werte aus dem Ergebnis der Abfrage von get() in die Attribute des Objektes überführt. Die Methode init() ordnet also den Attributen, eine Zeile des Abfrageergebnisses zu.

Diese Lösung scheint erst einmal sehr simpel und funktional zu sein. Die Klassen die persistent sein wollen, müssen die Basisklasse nur ableiten und init() so überschreiben, dass die speziellen Attribute der Klasse mit Daten gefüllt werden, sobald get() aufgerufen wird.

Es wird nun ein Teil aus dem Beispiel Datenbankschema (Anhang A) betrachtet. Wir wollen alle Verbindungen zwischen Aggregation und Project als konkretes Objekt erhalten. In diesem Objekt (wir nennen es Aggregation2Project) sind dann jeweils zwei Objekte aggregiert: Project und Aggregation. Alle drei Objekte leiten also unsere Basis-Klasse ab (BaseClass). Die init()-Funktionen von Project und Aggregation übertragen die Daten aus dem Resultset in ihre Attribute. Die init()-Funktion von Aggregation2Project tut das auch, aber zusätzlich instanziiert sie ein Projekt und eine Aggregation und ruft auf beiden jeweils die get() Methode auf. Dadurch werden dann die beiden Unterobjekte korrekt geladen und das Objekt Aggregation2Project ist insgesamt geladen.
Im Benchmark wird dieser Initialisierungsvorgang für Aggregation2Project für die gesamte Tabelle *aggregations_projects* ausgeführt. D.h. init() von Aggregation2Project wird so oft aufgerufen wie die Anzahl der Tupel in *aggregations_projects* ist. Demnach wird also auch für Project und für Aggregation get() sehr häufig aufgerufen. Sei x die Anzahl der Tupel in *aggregations_projects*, dann werden insgesamt x * 3 SELECT-Statements der Form:

```
SELECT * FROM :tabelle WHERE id = :id
```

aufgerufen: Einmal für das get() für Aggregation2Project dann jeweils für das get() von Project und von Aggregation.
Das Laden aus der Datenbank ist nun korrekt in allen Objekten gekapselt. Kein Objekt kennt die privaten Daten des anderen und jedes Objekt verwaltet seine eigenen Daten. Aber ist dieser Ansatz auch performant?
Die Anzahl der einfachen Querys scheint sehr groß zu sein. Doch nicht nur das, sondern viele der Queries werden auch mehrfach ausgeführt. In der Tabelle *projects* befinden sich nur ungefähr 180 Tupel. Die Beziehungen die in *aggregations_projects* gespeichert sind referenzieren also Aggregationen und Projekte mehrmals. Mit dem

naiven Ansatz gibt es eine Mehrzahl von Objekten die aus *aggregations_projects* geladen werden die dieselbe Projekt-ID aus der Datenbank besitzen und doppelt oder mehrfach geladen wurden. Da für jedes dieser Objekte ein neues erstellt wurde, verletzt dies natürlich das Prinzip der Objekt-Identität.
Das Problem ist einfach zu lösen (mit einem positiven Nebeneffekt). Aber zunächst ein anderer Ansatz:
Die Struktur bleibt weitesgehend gleich, es werden nur die init()- und get()-Methode der Klasse Aggregation2Projekt modifziert. Die get()-Methode wird so aus der BaseClass überschrieben, dass nicht nur die Daten von *aggregation_projects* sondern auch die Daten von *aggregations* und *projects* geladen werden. Dies kann z.B. mit einem LEFT JOIN im SELECT passieren. In der init()-Methode werden dann ganz normal die Attribute der Klasse gesetzt und die beiden Unterobjekte instanziiert. Im Gegensatz zu vorher wird nun aber nicht mehr die get()-Methode von Project und Aggregation aufgerufen, sondern das erhaltene Ergebnis an die Objekte weitergegeben. Dann wird nur init() auf beiden aufgerufen, sodass diese ihre Attribute nun aus dem Resultset des LEFT JOIN erhalten können.
Der Vorteil dieses Vorgehens ist, dass die Anzahl der Queries auf ein Vielfaches reduziert wird. Der Nachteil ist natürlich, dass nun die Klasse Aggregations2Project die Daten für Aggregation und Project aus der Datenbank geladen hat. Aggregation und Project haben nun nicht mehr das Laden der Daten gekapselt wie in der ersten Version. Aber auch hier haben wir das Problem, dass Objekte mit gleicher Shadowinformation nicht identisch sind.

In einem Benchmark über mehrere Testläufe der beiden Versionen ist die Tendenz eindeutig ablesbar: Die erste Variante braucht fast das doppelte an Zeit um alle Objekte zu laden und der Applikation bereit zu stellen. Die sogenannten `Roundtrips` zur Datenbank verlangsamen hier die Applikation (Kommunikationsoverhead, Ausführungsplan erstellen, etc). Man spart sozusagen mehr Performanz in dem man die Anzahl der Queries reduziert, denn die Menge der bezogenen Daten ist in beiden Versionen dieselbe (alle Objekte erhalten ihre Daten aus der Datenbank korrekt und vollständig).
Zum selben Ergebnis kommt [BPS99]: Das Team führte Abfragen auf einer Tabelle mit 100,000 Zeilen mit 100 Bytes pro Zeile: 16 Byte ObjectID (clustered Index), 3 mal 24 Byte Strings-Spalten, und 3 mal 4 Byte Integer-Spalten aus. Es wurden Abfragen in verschiedenen Blockgrößen ausgführt. Mit einem warmen Server-Cache wurden 580 Zeilen/Sekunde, 2700 Zeilen/Sekunde bzw 3200 Zeilen/Sekunde für die Blöckgrößen von 1, 20 bzw 100 gemessen. Das bedeutet, dass es ungefähr 5,5 mal schneller ist in einem Block von 100 Zeilen aus der Datenbank zu lesen, als Zeile für Zeile. Leider wurden die genauen Details des benutzen `RDMBS` und der benutzen Hardware nicht genannt, trotzdem ist die Tendenz gut zu erkennen.

Um unsere unfertigen Versionen vollständig zu machen, müssen wir noch eine weitere Struktur einfügen: Einen `Object-Cache`. Der `Object-Cache` speichert eine Referenz auf die Objekteinstanz einer bestimmten Klasse. Als Schlüssel dient dabei die OID und der Klassenname. Jedes mal wenn also ein Objekt aus der Datenbank geladen werden soll, wird überprüft ob der `Object-Cache` dieses Objekt bereits referenziert

hat. Dieser Ansatz hat die Vorteile, dass jetzt Abfragen an die Datenbank gespart werden (keine doppelten Queries mehr) und gleichzeitig keine neue Instanz für jedes neue Objekt aus der Datenbank geladen wird.
Beide Versionen profitieren natürlich von den gesparten Abfragen und werden performanter. Trotzdem ist die erste Version immer noch langsamer als die zweite. Erst wenn man alle Objekte für *projects* und alle Objekte von *aggregations* aus der Datenbank lädt bevor man mit dem Laden für die Objekte für *aggregations_projects* beginnt, laufen beide Versionen in ungefähr derselben Zeit. Diesen Vorgang nennt man `Prefetching`.
Dies ist einfach zu verstehen: Durch das Laden aller Objekte z.B. von *projects* wird eine einziger `Roundtrip` benötigt. Die Abfrage liefert direkt alle Instanzen der Objekte die im weiteren Verlauf sowieso geladen werden würden. Wir sparen also Abfragen in der Größenordnung der Instanzen der *projects*- und *aggregations*Tabelle.
Das `Prefetching` ist das zentrale Thema von [BPS99]. Es soll die Fragen beantwortet werden, wie herausgefunden werden kann, welche Objekte in der Applikation schon vorher geladen werden sollen, um die Anzahl der Abfragen zu minimieren. Ein paar Ideen sollen hier aufgegriffen werden:
Es müssen zwei Entscheidungen beim `Prefetching` von Objekten gefällt werden: Welche Objekte sollen geladen werden? Welche Daten sollen für die Objekte geladen werden?
Jedes Objekt kann Attribute besitzen. Ein Attribut kann entweder ein skalarer Wert (Strings, Integer, Booleans, etc), eine Beziehung zu einem anderen Objekt sein, oder eine Menge von Beziehungen zu mehreren anderen Objekten sein. Diese Menge von mehreren Objekten wird auch als `Set` von Objekten bezeichnet. Der Zustand eines Objektes wird durch die Menge der Attribute, die schon aus der Datenbank geladen wurden, beschrieben. D.h. ein Objekt kann als Instanz erzeugt werden, aber nicht alle Daten für alle Attribute werden aus der Datenbank ausgelesen und im Objekt gesetzt. Wenn die Applikation auf ein nicht geladenes Attribut zugreifen will, muss eine weitere Abfrage an die Datenbank gestellt werden und die Daten nachgeladen werden. Dies wird in der Literatur als `Lazy Loading` bezeichnet.
Lädt man immer alle Daten von einem Objekt verliert man Performanz, wenn die Applikation nur einen Teil der Daten eines Objektes benötigt. Man hätte also zu viele ungenutze Daten in den Speicher der Applikation geladen. Lädt man nur wenige Daten eines Objektes, muss man viele weitere `Roundtrips` zur Datenbank in Kauf nehmen. Idealerweise werden also alle Attribute, die in großen Blöcken geladen werden können mit `Prefetching` zur Verfügung gestellt.
Im Endeffekt, weiß der Entwickler selbst, welche Attribute er in welchem Fall braucht und muss selbst entscheiden, welches Set von Attributen er lädt. Die Software sollte hier versuchen möglichst wenig von den Prozessen zu verschleieren, damit der Entwickler die besten Optimierungen selbst vornehmen kann. Es gibt aber immer Szenarien, wo anfangst die beste Strategie für das Laden gewählt wird, aber später durch eine Änderung in der Applikation diese die schlechteste ist. Mein Ansatz ist deshalb, dass das System dem Entwickler nicht die Entscheidungen abnehmen, ihn aber dabei unterstützen sollte, die richtigen zu treffen. Zum Beispiel könnte das System bei der Entwicklung und beim Testen Tipps geben, wenn es eine schlechte Strategie erkennt. Diese Erkennung setzt eine Menge von Informationen voraus.

Z.b. muss die Anzahl und Art der Queries analysiert werden können. Dem System muss auch klar sein, was ein Query für Daten liefern will und ob es diese aus der Datenbank laden muss, oder diese bereits im Cache sind. Ich stelle mir eine Logdatei vor, die später nach der Suche nach schlechten Patterns durchsucht werden kann. Wichtig dabei auch ist, dass die Applikation mit den richtigen Kardinalitäten von Tabellen in der Datenbank rechnet. Es müssen also ebenso Statistiken über die Daten genutzt werden.
Dieser Ansatz soll im zweiten Teil der Arbeit weiter ausgeführt werden.

KAPITEL
FÜNF

Frameworks

Im folgenden Kapitel werden nun ein paar Frameworks vorgestellt, die den `Object-Relational Impedance Mismatch` lösen. Die verschiedenen Methoden den `IM` zu überbrücken liefern einen guten Eindruck über die Komplexität des Problems, die in einem späteren Kapitel betrachtet werden soll.

5.1 Übersicht

Zur Entwicklung von ORM-Software wurden viele Spezifikationen entwickelt. Einen Überblick verschafft die folgende Tabelle.

Enterprise JavaBeans 3.0 (EJB)	Spezifikation entwickelt unter dem Java Community Process. Die JavaBeans sind standardisierte Komponenten innerhalb eines Java-EE Servers. Teil von EJB ist die Java Persistence API.
Java Persistence API (JPA)	Ist eine Schnittstelle für die objekt-relationale Abbildung von POJOs (`Plain old Java Object`). Wurde im Rahmen der EJB 3.0 von der Software Expert Group als Teil der JSR 220 entwickelt und herausgegeben. Sie soll die besten Ideen der APIs von Hibernate, Toplink und JDO beeinhalten.
Java Data Objects (JDO)	Offizielle Spezifikation von Sun für ein Framework zur persistenten Speicherung von Java-Objekten. Die JPA wurde maßgeblich von JDO beeinflusst und wird deshalb auch ihr Nachfolger genannt. JDO war ebenfalls eine JSR Spezifikation und wird seit Version 2.0 von der Apache Software Foundation weiterentwickelt.

Die Masse der ORM-Tools ist schwer zu erfassen. Für fast jede Programmiersprache und Datenbank wurde mindestens ein kommerzielles Produkt, welches den `Object-Relational`

`Impedance Mismatch` löst, entwickelt. Zusätlich findet man meist eine Open-Source Implementierung. Ich versuche hier einen Überblick über die mir wichtigsten Tools, Frameworks und alternativen Lösungen für den `IM` zu geben. (Stand Ende 2010). Diese Liste ist jedoch bestimmt nicht vollständig.

EclipseLink	Implementiert die Java Persistence API (JPA) 2.0 und ist deshalb meist für Java genutzt.
TopLink	entwickelt von Oracle war TopLink die Referenzimplementierung der JPA 1.0 und wurde dann durch EclipseLink ersetzt.
Hibernate (Nhibernate für .NET)	Open-Source Framework für Java entwickelt von der JBoss Community. Setzt auch die JPA um (Hibernate EntityManager). Hibernate wird von den meisten als die Referenz für die Lösung des `IM` im Open-Source Bereich für Java gesehen.
DataObjects.NET	DataObjects.NET ist ein ORM Framework für das .NET Framework. DataObjects.NET wird der GPL Lizenz veröffentlicht. Es unterstütz LINQ und kann auch ohne Datenbank benutzt werden, Internationalisierung von Objekten möglich und vieles mehr.
Cayenne	Entwickelt von der Apache Software Foundation. Cayenne ist ein Open Source ORM Framework unter der Apache Lizenz. Zusätzlich zum Binden von Datenbankschemata zu Java Objekten unterstützt es auch Transaktionen, SQL Generierung und `remoting Services` (ein Webservice basierte Technologie für den Ersatz einer lokalen Datenbank)
Caché	Caché wird von InterSystems entwickelt und geht einen etwas anderen Weg als Objektdatenbanken und ORM Frameworks: Es bezeichnet sich als postrelationale Datenbank. Caché ermöglicht verschiedene Sichten auf die Objekte in der Datenbank per SQL (z.B. über ODBC/JDBC) aber auch durch objektorientiertem Zugriff. Es werden jeweils nur die Schnittstellen angesprochen und dieselben internen Datenbankbefehle genutzt.
Language Integrated Query (LINQ) to SQL	LINQ ist eigentlich eine Komponente des .NET Frameworks zur Abfrage von Datenquellen (XML, Datenbanken). Es ist eine Spracherweiterung und ermöglicht Abfragen in einer SQL-Select ähnlichen Struktur (select, from, where) an Datenstrukturen innerhalb des Programmes. Die Sprache kann dann so erweitert werden, dass die Transformierung in andere Anfragesprachen möglich ist. Deshalb löst die Umwandlung von LINQ nach SQL den `IM`.

LinqConnect	Eine Implementierung die nah an LINQ to SQL von Microsoft findet man LinqConnect. Es ist eine einfach zu benutzende ORM Lösung die als Server SQL Server, Oracle, MySQL, PostgreSQL und SQLLite unterstützt. Zusätzlich wird ein Model Designer-Tool für Visual Studio geliefert.
OpenJPA	Wie der Name schon sagt ist die OpenJPA eine Open-Source (Apache Lizenz) Implementierung der Java Persistence API (2.0). OpenJPA wird von der Apache Software Foundation entwickelt.
JPOX	In der Version 1.2 implementiert JPOX die JPA 2.0 und die JDO 2.0 Spezifikation. Es ist also ein weiteres Persistenz-Framework für Java.
Kohana	Kohana bezeichnet sich selbst als lightweight PHP Framework. Ein Modul dieses Frameworks überbrückt den `IM`.
Doctrine	Doctrine ist ein weiteres PHP Framework. Es geht ähnliche Lösungswege wie Hibernate und ist damit eins der wenigen Frameworks in PHP die eine eigene Query Language (DQL) zur Verfügung stellen.
Propel	Als Teil vom Symphonie Framework für PHP5 ist Propel für den ORM Bereich zuständig. Propel generiert automatisch Code, der die Klassen des objektorientierten Models persistent machen kann. Dabei werden auch einzelne Querys pro Klasse definiert. Durch dieses kompilieren spart Propel zur Laufzeit jede Menge Resourcen. Zusätzlich werden Prepared Statements, Transaktionen und Validatoren sowie ein Object-Cache unterstützt.
ActiveRecord	Die Lösung für Ruby in Ruby on Rails (ein Ruby Framework) heißt ActiveRecord. Es implementiert auch das ActiveRecord-Pattern.
GORM	GORM ist die ORM Lösung für Grails. Grails ist ein Framework basierend auf Groovy, welches an Ruby on Rails angelehnt ist. Groovy ist eine dynamische Open-Source Programmiersprache für die Java Virtual Machine. Groovy ermöglicht durch seine dynamischen Features interessante und neue Herangehensweisen an alte Probleme, deshalb finden sich in GORM auch neue und innovative Lösungen für den `IM`.

Einen Eindruck in die Performanzleistungen ein paar dieser Frameworks, die für .NET entwickelt wurden bietet [ORM09]. Dort wird auch beleuchtet inwiefern diese

Frameworks alle Features von LINQ implementieren.
Eine Betrachtung aller Details für jedes Framework würde den Rahmen dieser Arbeit sprengen, deshalb wird das Augenmerk auf Besonderheiten von einigen für diese Arbeit relevanten Implementierungen gerichtet. (Das bedeutet nicht, dass nichtgenannte nicht besonders relevant in anderen Bereichen sind). Am meisten konzentriere ich mich also auf die Frameworks in die ich den meisten Einblick erhalten konnte.
Ich betrachte folgende Frameworks:

- Oracle TopLink 11.g (11.1.1) für Java [Ora09]
- Hibernate 3.6.0.CR2 für Java [JBo10a]
- Kohana 2.4.0 RC2 für PHP [Koh10b]
- Doctrine 2.0.0BETA4 für PHP [Doc10a]

5.2 Hibernate und Vererbung

Hibernate ([JBo10b]) ist ein `ORM-Tool` für Java und .NET (dort heißt es NHibernate). Es ist eine Open-Source Software und hat sich als die eine Alternative gegen kommerzielle Produkte durchgesetzt.
Die Konfigurtion eines Mappings wird in Hibernate durch den Benutzer eingestellt. Zu jeder Klasse, die persistent werden soll, wird eine XML-Steuerungsdatei angelegt die das Mapping spezifiziert. Die XML-Dateien können auch automatisch durch Tools erzeugt werden, indem `XDoclet-Tags` aus dem Java-Quellcode ausgelesen werden. In JDK 5.0 kann dies mit Annotations erreicht werden.
Die meisten Frameworks konzentrieren sich auf genau eine Möglichkeit ein `Mapping` von der Domain des Objekt-Models zum relationalen Schema zu realisieren. Dabei wird oft vernachlässigt, dass ein `Mapping` in einem Kontext besonders effektiv und performant sein kann, aber für einen anderen Kontext fast unbrauchbar ist. Am besten wird dies am Beispiel des `Mappings` für Vererbung deutlich:
In einer sehr tiefen Klassenstruktur ¿ 5 Ableitungen pro Klasse Anders als andere Frameworks entscheidet sich Hibernate nicht für nur ein Mapping (z.b. horizontales Mapping), sondern lässt dem Benutzer die Wahl zwischen allen drei Möglichkeiten:

- „Table per class hierarchy“ (`Filter-Mapping`)
- „Table per subclass“ (`vertikales Mapping`)
- „Table per concrete class“ (`horizontales Mapping`)

[JBo10a, Kapitel: inheritance]
Nicht nur zwischen den verschiedenen Strategien kann gewählt werden, sondern es können auch Kombinationen der verschiedenen benutzt werden. Somit ist es möglich einen Teil der Hierarchie als `Filter Mapping` und den anderen Teil als `vertikales Mapping` abzubilden. Dies bietet höchste Flexibilität und lässt je nach Kontext die Benutzung der besten Strategie zu.

5.3 Kohana, Doctrine und Hibernate und die Abfragesprache

Kohana [Koh10a] und Doctrine [Doc10b] sind beides PHP Frameworks. Beide haben sich gegen die wirklich großen PHP-Frameworks wie z.B. Zend noch nicht durchgesetzt, dennoch sind die Ideen und verwendeten Algorithmen aus eigener Erfahrung sehr vielversprechend.
Wie im Kapitel zuvor betont ist das Design der `Query-Language`, mit der Objekte aus der Datenbank geladen werden können, eins der anspruchsvollsten Aufgaben des `ORM`. Hibernate entscheidet sich für eine eigene Implemetierung genannt HQL (`Hibernate Query Language`). Kohana implementiert eine NoSQL ähnliche API Version, die sich aber wie SQL schreiben lässt. Doctrine nennt seine Abfragesprache DQL (`Doctrine Query Language`). Die DQL wird wie SQL geschrieben, allerdings wird diese Sprache dynamisch erweitert, sodass es möglich ist z.b. Klassennamen als Tabellen und `JOINs` ohne `ON`-Bedingung zu benutzen. Da Doctrine einen eigenen Parser benutzt um das DQL zu bearbeiten, sind auch eigene Erweiterungen dieser Sprache möglich und die Kapselung zur Datenbank bleibt erhaltent. Das DQL wird in objektorientierte Strukturen umgewandelt und dann an den `Database Abstraction Layer` (DBAL) weitergegeben. Ein DBAL erlaubt dann ohne den Code der Applikation zu ändern das Datenbanksystem zu wechseln. Es ist aber nicht nur möglich in DQL Abfragen zu schreiben, ebenso stellt Doctrine einen `QueryBuilder` zur Verfügung, der ähnlich wie in Kohana eine NoSQL API zur Verfügung stellt.
HQL ist eben wie DQL an SQL angelehnt. Es ermöglicht aber ebenfalls eine einfachere Schreibweise für die Abfrage von Objekten.
Als Tabellen im FROM-Teil der Abfrage können Objektnamen genutzt werden.
`JOINs` können ohne `ON`-Bedingung benutzt werden:

```
select cust
from Product prod,
    Store store
    inner join store.customers cust
}
```

Die erweiterte Punkt-Notation wie z.b. `store.location.name` erlaubt implizit auf Unterobjekte des Objektes zuzugreifen (in dem Beispiel ist `location` ein Unterobjekt von `store` und auf dessen Eigenschaft `name` wird zurückgegriffen).
Es können - von der unterliegenden Datenbank unterstützte - Skalare Funktionen und Join-Typen benutzt werden. Es werde somit nahezu alle Features von SQL durch HQL abgebildet. Eine eigene Abfragesprache zu entwickeln ist aber durchaus sinnvoll - anstatt SQL zu benutzen. Denn dadurch dass das Selektieren von einzelnen Attributen eines Objektes, das Erstellen von Bedingungen für `JOINs` und das Erstellen von eigenen Traversierungen der Beziehungen wegfällt, wirkt eine geschriebene HQL Anfrage viel übersichtlicher und ist viel kürzer als eine herkömmliche SQL Abfrage.
Zum Beispiel:

```
select cust
from Product prod,
    Store store
```

```
    inner join store.customers cust
where prod.name = 'widget'
    and store.location.name in ( 'Melbourne', 'Sydney' )
    and prod = all elements(cust.currentOrder.lineItems)
```

würde in SQL ungefähr so aussehen:

```
SELECT cust.name, cust.address, cust.phone, cust.id, cust.
   current_order
FROM customers cust,
    stores store,
    locations loc,
    store_customers sc,
    product prod
WHERE prod.name = 'widget'
    AND store.loc_id = loc.id
    AND loc.name IN ( 'Melbourne', 'Sydney' )
    AND sc.store_id = store.id
    AND sc.cust_id = cust.id
    AND prod.id = ALL(
        SELECT item.prod_id
        FROM line_items item, orders o
        WHERE item.order_id = o.id
            AND cust.current_order = o.id
    )
```

Die Vorteile sind also ersichtlich: Es ist viel einfacher große, komplexe Anfragen in einem bereits bekannten Standard zu schreiben, als eine neue (NoSQL)-Sprache zu lernen, die möglicherweise nicht vollständig alle Konstrukte von SQL übernommen hat (wie in Kohana). Trotzdem wird mit den `HQL-Queries` die Applikation nicht an das Schema der relationalen Datenbank gekoppelt, da die Anfrage zuerst geparst wird und dann an die Datenbank-Verwaltung weitergegeben wird, die dann den Befehl in SQL transformiert. Einzig allein die Performanz eine solchen Abfrage zuerst auf Textbasis zu analysieren, dann wieder in SQL zu transformieren und dann erst auszuführen, ist in Frage zu stellen. (Dieses Problem lässt sich mit Bytecode Caches für PHP relativ gut umgehen). Dennoch scheint hier das Entwickeln einer eigenen Abfragesprache den Aufwand wert zu sein.

5.4 Beziehungen zwischen Objekten

Bevor ich kurz Beschreiben werde, wie Oracle Toplink [Ora09] Beziehungen (`Mappings`) zwischen Klassen realisiert, möchte ich kurz die grundlegende Struktur einer ORM Lösung, die mit TopLink arbeitet, vorstellen. TopLink unterstützt generell 3 verschiedene Arten von ORM Konfigurationen, die sich durch die Datenquellen unterscheiden (XML, `EIS`-Datenquellen über einen `JCA` Adapter, relationale Datenbanken). Ich konzentriere mich hier nur auf die Konfiguration mit relationalen Datenbanken. Die weiteren Möglichkeiten für die anderen Datenquellen sind sehr interessant, würden aber den Rahmen dieser Arbeit sprengen.

In den sogenannten relationalen Projekten kann jede Datenbank genutzt werden, die über `JDBC` erreichbar ist. Die Konfiguration eines Projektes erfolgt über `Descriptors`. Ein `(relational) Descriptor` beeinhaltet eine Menge von Mappings und Einstellungen, die dafür benötigt werden um eine übliche Java-Klasse persistent zu machen. `Descriptors` sind hierchisch, werden bei der Kompilierung in XML umgewandelt und können durch Java selbst, das Tool „TopLink Workbench“ oder durch den „Oracle JDeveloper Editor“ erstellt werden. TopLink Workbench ermöglicht das zusammenstellen der `Descriptors` und `Mappings` durch Klicks und Bearbeiten von Eigenschaften, ohne dass XML- oder Javacode geschrieben werden muss. Ob der Workbench der JDeveloper Editor oder Java selbst genommen wird, ist unrelevant, da alle Methoden die XML Files, die zur Laufzeit von TopLink gelesen werden, erzeugen. Deshalb werde ich das Vorgehen verschiedene Mappings zu konfigurieren nur erklären und nicht für eine bestimmte Methode explizit ausformulieren.

TopLink unterscheidet generell bidirektionale und unidirektionale Beziehungen. Bei bidirektionalen Beziehungen, die während der Laufzeit geändert werden, muss die Referenz der jeweils andere Seite auch modifiziert werden, wenn dieses Objekt bereits im Speicher vorhanden ist. In TopLink ist es möglich dies entweder automatisch erledigen zu lassen, oder die bidirektionale Beziehung selbst (in den Settern und Gettern der Klassen) zu verwalten. Es wird jedoch davor gewarnt automatisches Verwalten zu aktivieren, wenn ein Objekt der Beziehung in einem anderen Kontext verwendet werden kann, wo es diese Beziehung nicht besitzt.
Das Vorgehenfür ein bidirektionales Mapping für die Tabellen *timeslices* und *users* im Beispiel-Datenbankschema A wäre dann:

1. Es müssen `Descriptors` für *timeslices* und für *users* erstellt werden. Die passenden Java-Klassen heißen `Timeslice` und `User`.
2. Es wird für beide `Descriptors` ein neues, relationales `OneToMany - Mapping` erstellt.
3. In dem Mapping in `Timeslice` wird der `Referencediscriptor` auf `User` gesetzt
4. In dem Mapping in `User` wird der `Referencediscriptor` auf `Timeslice` gesetzt
5. Im Mapping von `Timeslice` wird die Spalte `user_id` als `ForeignKeyFieldName` definiert.
6. Im Mapping von `User` wird die Spalte `id` als `TargetForeignKeyFieldName` definiert.

Dies ist besonders Flexibel, da man entscheiden kann, ob das Mapping auf der einen Seite ebenfalls gesetzt werden soll. Es würde auch funktionieren, wenn wir nur in `Timeslice` das `OneToMany - Mapping` setzen und dies in `User` nicht tun. Allerdings könnten wir dann in `User` auf keine Collection von `Timeslices` zugreifen, sondern nur für jeden `Timeslice` auf den passenden `User`. Der Entwickler kann somit selbst

bestimmen, welche Navigationspfade er in seiner Applikation zwischen Objekten anlegen will und welche nicht.

Hibernate geht denselben Weg, nur natürlich mit Annotations oder XML Files als Konfigurationsmöglichkeiten. Auch hier können bidirektionale Beziehungen oder unidirektionale Beziehungen gepflegt werden. Zusätzlich lassen sich für jede Beziehung noch weitere Optionen einstellen: Z. B. ob die `OneToMany`-Beziehung mit einer Relation (Tabelle) im relationalen Schema abgebildet werden soll (Ich nenne diese Tabelle im weiteren Verlauf auch Zwischentabelle). Das bedeutet, dass nicht eine Relation einen Fremdschlüssel für die andere beeinhaltet, sondern beide Fremdschlüssel in dieser Zwischtentabelle existieren, so wie es bei einer `ManyToMany`-Beziehung wäre. Es ist möglich eigene Join-Bedingungen zu formulieren, doppelte Fremdschlüssel (Schlüssel auf beiden Seiten der Beziehung) zu definieren und die Art der Verknüpfung genauer zu spezifizieren. Letzteres meint, das Hibernate angewiesen werden kann, Objekt B zu löschen wenn Objekt A gelöscht wird und mit B in einer Beziehung mit der Option `Cascade: delete` steht. Analog gibt es Optionen für das kaskadierte Update. Das `Lazy Loading` kann hier ebenso eingestellt werden. Jedes Objekt kann so sehr genau für seinen Lebenszyklus in der objektorientierten Applikation konfiguriert werden. Der Entwickler muss nicht mehr darauf achten, dass Objekte in einem Kompositionsmuster als unreferenzierte Objekttupel in der Datenbank bleiben.
Auch das Collection-Handling von Hibernate ist mit mehr Funktionalität ausgestattet. So ist es zum Beispiel bei einer sortierten Collection möglich, den Sortierschlüssel in der Zwischentabelle zu definieren, oder ein eigenes Kriterium zur Sortierung anzugeben, welches dann im Abfrage-SQL benutzt wird. Auch in Collections ist kaskadiertes Löschen möglich.

5.5 Object Loading

Das klassische `Lazy Loading` ist in Toplink elegant gelöst. Es gibt drei verschiedene Möglichkeiten `Lazy Loading (Indirection)` zu erreichen. Die erste ist an der Stelle des Attributes im Objekt A, welches auf ein anderes Objekt B verweisen soll einen `value holder` einzufügen, statt das Objekt direkt zu laden. Ein `value holder` ist eine Instanz einer Klasse die das Interface `ValueHolderInterface` implementiert. Greift nun der Getter von Objekt A auf das Attribut zu, in dem der `value holder` liegt, wird dieser mit dem konkreten Objekt ergänzt. Bei einer weiteren Anfrage des Getters, wird dann einfach der gespeicherte Wert innerhalb des `value holder` zurückgegeben.
Die zweite Methode ist einen `Proxy` zu benutzen, der zunächst das gewünschte Objekt simuliert, indem es ein Interface dieses Objektes implementiert. Das bedeutet, dass jedes Objekt der Applikation, welches als `Indirection Proxy` genutzt werden soll ein Interface für die eigenen Methoden benutzen muss, man aber den Vorteil hat, dass keine spezielle Klasse von TopLink (wie in der ersten Methode) verwendet werden muss.
Die dritte Möglichkeit ist ausschließlich für Collections (Hashtable, List, Map, Vec-

tor, etc) und nennt sich `transparent container indirection`. Wird ein Mapping als `transparent container indirection` auf der Many-Seite einer Beziehung konfiguriert, benutzt TopLink eine konkrete Klasse aus dem `indirection` Package, welches das gewünschte Interface der Collection implementiert (Hashtable, List, Map, etc). Die Objekte auf die dann in der Applikation zugegriffen wird, werden dynamisch nachgeladen.
Das bietet dem Entwickler eine hohe Flexibilität fürs `Lazy Loading`. Er kann nun selbst entscheiden, welche Objekte der Applikation von Anfang an komplett erstellt werden und welche durch weitere Abfragen aus der Datenbank geladen werden. Da der `ORM` nicht selbst entscheidet, welche Objekte er `lazy loaded`, ermöglicht dies die beste Performanz zu erreichen.

In Doctrine ist `Lazy Loading` das Standardverhalten. D.h. jedes Unterobjekt eines Objektgraphen wird mit `Lazy Loading` geladen. Doctrine benutzt dafür Proxys, die sich wie die Originalobjekte verhalten, aber automatisch Abfragen an die Datenbank schicken, wenn auf ihre noch nicht geladenen Attribute zugegriffen wird. Wenn man also nicht alle Objekte in Doctrine laden will, muss man mit der DQL eine größere Abfrage formulieren und sicherstellen, dass alle Objekte die man in diesem Teil der Applikation benötigt auch hydrated werden. Als Default werden alle Objekte aus dem Cache geladen, wenn sie schon einmal geladen wurden. Dieses Verhalten kann mit `Query-Hints` überschrieben oder modifiziert werden.

In Kohana sind alle Unterobjekte standardmäßig nicht geladen. Erst wenn man auf ein Attribut zugreift, wird dieses mit einem weiteren Query an die Datenbank geladen. Die ganze Verwaltung dieser Logik passiert durch eine PHP-Magic Funktion (eine Funktion die jedes mal von PHP aufgerufen wird, wenn ein Attribut des Objektes aufgerufen wird). Wenn man nicht alle Objekte des Graphen mit `Lazy Loading` haben will, muss man wie bei Doctrine ein größeres Query erstellen. Im `NoSQL-Query-Builder` kann man die gewünschten Unterobjekte mit einer Funktion `with()`hinzufügen. Kohana muss dazu alle Assoziationen zwischen diesen Objekten bereits kennen. Man ist selbst dafür verantwortlich, dass das Laden von mehreren Unterobjekten funktioniert, denn die Applikation gibt im Normalfall keinen Fehler aus und alle Objekte werden mit `Lazy Loading` geladen, was man ja in den meisten Fällen nicht will. Wird die Abfrage richtig groß (mehr als 5 Unterobjekte), leidet auch die Performanz. Der Grund ist, dass an jede Initialisierungsfunktion der Unterobjekte die gesamte Zeile der Abfrage (die natürlich sehr breit ist) weitergegeben wird. Außerdem sind Magic Methoden in PHP richtige Performanzvernichter.

5.6 Abbildung von Klassen und Attributen

Hibernate und TopLink benutzen beide ein eigenes System um jegliche Form von Mappings für Attribute zu ermöglichen. Ein Attributmapping ist eine Regel, die den Wert aus der objektorierten Applikation eines Objektattributs in das relationale Schema überträgt. Dabei gibt es in beiden Lösungen eine fast unbegrenzte Anzahl von Möglichkeiten diese Regeln umzusetzen:

- Mit `Transformation Mappings` können Datentypen der Applikation (z. B. Java.util.date) in Datentypen des relationalen Schemas (z. B. DATE und TIME) umgewandelt werden
- Mit read und write Expressions (Hibernate),`Attribute Transformers` (TopLink) oder `Custom Mapping Types` (Doctrine) kann jede beliebige Funktion für die Transformierung von Attributen benutzt werden
- Es gibt vorgefertigte Attributmappings, die Objekte in BLOBS oder Strings einer Tabelle serialisieren
- Es gibt Attributmappings auf Objekt-Rationale Datentypen (TopLink)
- Es existieren XML-Type Mappings, mit denen ganze Objekte oder kombinierte Attribute als XML serialisiert werden können (Hibernate, Toplink)
- Es können eigene Interfaces implementiert werden, die die Werte transformieren

In Kohana ist es nicht möglich Attributmappings selbst zu schreiben. Nur elementare Datentypen werden durch fixe (durch Kohana definierte) Mappings übertragen. Möchte man zum Beispiel einen `Unixtimestamp` nicht als `DATETIME`-Datentyp in der Datenbank speichern, ist dies nicht möglich. Die Transformationen müssen in den Gettern und Settern des persistenten Objektes geschrieben werden, sodass bei jedem Zugriff die Transformation erneut ausgeführt werden muss.
Doctrine ermöglicht ebenso wie TopLink das definieren von eigenen Mappings. Es ist sogar möglich einen eigenen Metadatentreiber zu schreiben, der dann die Metadaten aus irgendeiner Datenquelle lesen kann (es ist z. B. XML / YAML möglich). Als einfache Mappings sind Transformationen von elementaren Datentypen aus PHP in elementare Datentypen aus dem relationalen Schema möglich. Diese Mappings können jedoch durch eigene Klassen modifziert werden - ebenso wie bei TopLink und Hibernate. Es müssen jediglich die Funktionen des Interfaces (convertToPHPValue() und convertToDatabaseValue() implementiert werden).

5.7 Zusammenfassung

Generell ist in Hibernate sehr viel konfigurierbar. Auch wenn eine komplexe Konfiguration meistens bedeutet, dass die Benutzung der Software komplizierter wird, ist dies in Hibernate nicht so. Die XML-Konfigurationsdateien und die Annotations bleiben größtenteils sehr übersichtlich und wirken auf mich sehr intuitiv. Werden Konfigurationsoptionen weggelassen, benutzt Hibernate immer einen Standard, der meist schon genau der Wert wäre, denn man eingestellt hätte. Somit sind z. B. Fremdschlüssel immer als ¡Tabellenname¿_id gekennzeichnet. In den meisten Fällen lassen sich sogar diese globalen Standards einstellen.
TopLink konzentriert sich hingegen auf das eigene Workbench Tool, welches ermöglicht alle Optionen der Mappings und Relationen per Klick in einer GUI einzustellen. Zwar ist dies sicherlich sehr komfortabel, aber wenn man weitere Optionen benötigt, die noch nicht in der GUI vorhanden sind, muss man auf das Schreiben von Java Code

zurückfallen. Und leider sind mehr Beispiele mit Screenshots für das Workbench Tool als für Java Code in der Dokumentation zu finden. Insgesamt sind die Optionen auch nicht so detailliert wie bei Hibernate. Das liegt aber unter Umständen auch daran, dass man in Java jede Klasse die ein bestimmtes TopLink-Interface implementiert als Ersatz für die TopLink Klassen mit wenigen Optionen benutzen kann. Man kann also die Mappings durch eigenen Java Code anpassen und ist somit ebenfalls flexibel, auch wenn man dann die API besser kennen muss.
Vergleicht man die ORM Lösung von Kohana mit der Vielfalt von Optionen von Hibernate und TopLink ist klar, warum Kohana sich nur als kleines Framework bezeichnet. Zwar lässt auch Doctrine viele Features aus, die bei Hibernate und TopLink stark ausgereift scheinen, es hat diese aber dennoch in der Roadmap stehen. (Man muss auch beachten, dass es sich noch um eine Betaversion handelt). Vermutlich liegt es aber auch daran, dass Java eine weitaus mächtigere Sprache als die einfache Skriptsprache PHP ist und es somit für Hersteller generell mehr Potential auszuschöpfen gibt.
Ich möchte Doctrine fast als PHP-Äquivalent für Hibernate bezeichnen. Die Annotations und Prinzipien mit den verschiedenen Problemen umzugehen sind starke Analogien zu Hibernate. Obwohl ich Doctrine noch nie in einer realen Applikation testen konnte, wirken die Konzepte sehr vielversprechend. Auch eine konsequente objektorierte Programmierung (die nicht immer in PHP Applikationen gegeben ist) überzeugt. Überall dort wo Optionen fehlen könnten, ist die Applikation durch Implementierung von Interfaces zu erweitern.

KAPITEL

SECHS

Fazit

6.1 Während des Schreibens

Der `IM` ist immer eins der Probleme gewesen, die mich am meisten interessiert haben. Bevor ich den richtigen Namen des Problems wusste, welches ich hatte, als ich meine erste objektorientierte Applikation mit einer MySQL-Datenbank schrieb, war mir klar, dass dies ein sowohl praktisches also auch theoretisch interessantes Problem ist. Bei der Vorbereitung für diese Arbeit dachte ich zuerst, dass Material über dieses Problem zu finden ein großes Problem sei. Mittlerweile habe ich über 40 Papers und Veröffentlichungen gefunden, die sich genau mit diesem Thema befassen. Leider war es dennoch schwer Publikationen zu finden, die das Offensichtliche weiter vertiefen. Somit begann ich das Kapitel mit der Vertiefung über das `Object Marshalling` ohne eine passende Arbeit darüber gefunden zu haben. Ich implementierte die ersten zwei naiven herangehensweisen und kam zu dem Ergebnis, dass die Roundtrips der Schlüssel für die Performanz beim Laden sind.
Ein paar Wochen später habe ich auf http://www.odbms.org/ dann das entscheidene Whitepaper [BPS99] für dieses Kapitel gefunden. Zu meiner Beruhigung kamen Bernstein, Pal und Shutt zu demselben Ergebnis wie ich und entwickelten dabei noch die beste Idee für das Prefetching von Objekten, die ich bisher kannte.

6.2 Die Lösung des Impedance Mismatch

Alles in allem, ist die eine richtige, korrekte, performante Lösung des `Object-Relational Impedance Mismatch` schwer zu bestimmen. Viele Systeme haben sich in der Praxis bewährt und nicht alle benutzen unterschiedliche Methoden. In den meisten Fällen, die ich entdeckt habe, ist es vorallem wichtig, dass der Programmierer der Applikation weiß, wie er seine ORM-Lösung einzusetzen hat. Eine Kenntnis über das relationale Schema ist durch die Natur des Problemes immer nötig. Wenn - wie von allen gewünscht - die Applikation von der eigentlichen Datenbanklösung durch eine Abstraktionsebene getrennt wird, verliert man zu oft zu viel Performanz. Der Tradeoff findet zwischen zwei Parametern statt: Der Abstraktion und der Performanz. In

unserem Falle ist die Abstraktion der Grad, wie sorgfältig die Prinzipien der objektorientierten Programmierung umgesetzt werden. Die Performanz lässt sich dadurch bewerten, wenn man sie mit einer komplett relational programmierten Lösung oder einer komplett objektorientierten Lösung vergleicht [CJ10, Kap. 3]. Viele Beispiele aus den Problemen des `IM` haben genau diese genannten Tradeoff-Parameter:

- Benutzt man lieber horizontales Mapping (performanter bei tiefen Klassenhierarchien), oder vertikales Mapping (flexibler gegen Änderungen)
- Schreibt man SQL Befehle als String eingebettet in der Programmiersprache (performanter), oder entwickelt man eine eigene Abfrage-Sprache oder eine NoSQL-Lösung (flexibler, sauberer, fehlerunanfälliger)
- Lässt man dem Entwickler den Einblick in die Struktur der Daten, hat dieser die Möglichkeiten seine Abfragen so zu formulieren, dass sie vom Datenbanksystem optimiert werden können. Implementiert man einen `Database Abstraction Layer` kann man zwar das unterliegende `DMBS` jederzeit austauschen, aber man verschleiert die Sicht des Entwicklers auf die Stuktur der Daten.

Die Frage die sich also stellt, wenn man sich für eine Lösung des `IM` entscheiden will, ist: Wieviel Performanz will man dafür einbüßen, dass man agiler, flexibler und fehlerfreier an objektorientierten Anwendungen mit relationalen Datenbanken als Persistenzsystem arbeiten kann? Diese Frage ist denke ich nicht für jeden Anwender pauschal zu beantworten und somit wird es auch nie die einzige Lösung für den `IM` geben.

Teil II

Ein Framework zur Lösung des Object-Relational Impedance Mismatch

KAPITEL

SIEBEN

Einleitung

7.1 Aufbau dieses Teils

KAPITEL

ACHT

Implementierungen in PHP

KAPITEL
NEUN

Anforderungen

9.1 Objektidentifizierung

9.2 Cache

9.3 Abfragesprache

9.4 Change detection

9.5 Basis Klassen

9.6 Mapping Metadaten

9.7 Object Loading

KAPITEL
ZEHN

Implementierung des Frameworks

10.1 Cache

10.2 dynamischer Code

10.3 Datenbankabstraktion

10.4 Abbildung von Klassen und Attributen

10.5 Abbildung von Vererbung

10.6 Beziehungen zwischen Objekten

10.7 Object Loading

KAPITEL

ELF

Fazit

ANHANG

A

Beispiel Datenbank Schema

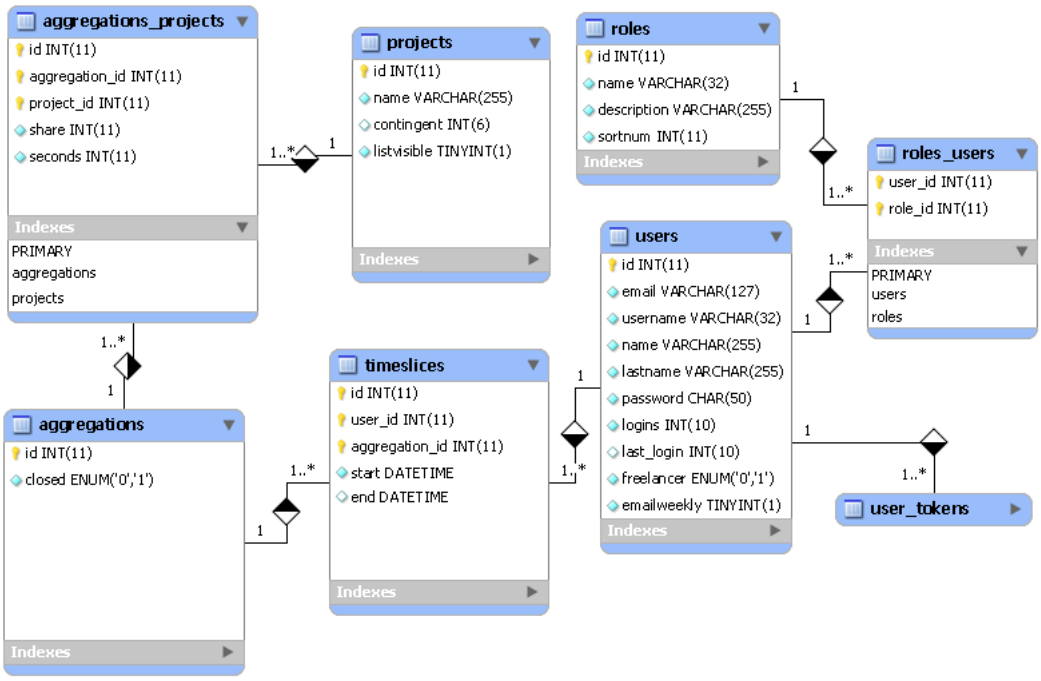

Abbildung A.1: Die Beispieldatenbank

Das Schema modelliert eine Applikation um die Arbeitszeiten von Mitarbeiten zu bestimmen. In *timeslices* sind dabei Zeitspannen, die der Benutzer im Betrieb gearbeitet hat. Er loggt sich Morgens ein und Abends wieder aus. Nach dem gearbeiteten Tag erstellt er eine Aggregation. Eine Aggregation kann mehrere Timeslices besitzen und wird Projekten zugeordnet. Es wird dann die Zeit in einer Aggregation die von den Timeslices kommt, auf die zugeordneten Projekte verteilt.

Relation	Anzahl der Zeilen
projects	193
aggregations_projects	11.862
aggregations	3.742
timeslices	4.151
users	20

LITERATURVERZEICHNIS

[ABD+89] Malcolm Atkinson, François Bancilhon, David DeWitt, Klaus Dittrich, David Maier, and Stanley Zdonik. The object-oriented database system manifesto, 1989. 5

[Amb98] S W Ambler. The design of a robust persistence layer for relational databases. *AmbySoft Inc. White Paper*, 1998. 15

[BPS99] Philip A. Bernstein, Shankar Pal, and David Shutt. Context-based prefetch for implementing objects on relations. In *In Proceedings of the 25th International Conference on Very Large Data Bases*, pages 327–338. Morgan Kaufmann, 1999. 17, 18, 33

[CI05] William R. Cook and Ali H. Ibrahim. Integrating programming languages and databases: What is the problem. In *In ODBMS.ORG, Expert Article*, 2005. 4

[CJ10] Stevica Cvetković and Dragan Janković. A comparative study of the features and performance of orm tools in a .net environment. In Alan Dearle and Roberto Zicari, editors, *Objects and Databases*, volume 6348 of *Lecture Notes in Computer Science*, pages 147–158. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-16092-9_14. 34

[Dev01] Ramakanth Subrahmanya Devarakonda. Object-relational database systems — the road ahead. *Crossroads*, 7(3):15–18, 2001. 6

[Doc10a] Doctrine Project. *Doctrine Documentation*, 2010. 24

[Doc10b] Doctrine Project. *Doctrine: Object Relational Mapper*, 2010. 25

[IBM01] IBM. *DB2's object-relational highlights: User-defined structured types and object views in DB2*, 2001. 8

[IBNW09] Christopher Ireland, David Bowers, Michael Newton, and Kevin Waugh. A classification of object-relational impedance mismatch. *Advances in Databases, First International Conference on*, 0:36–43, 2009. 4, 5

[Int10] Intersystems. *Intersystems Caché*, 2010. 8

[JBo10a] JBoss Community. *Hibernate*, 2010. 24

[JBo10b] JBoss Community. *Hibernate: Relational Persistence for Java & .NET*, 2010. 24

[Kee04] C Keene. Data services for next-generation soas. *SOA WebServices Journal*, 1(4):2004, 2004. 2

[Koh10a] Kohana Team. *Kohana*, 2010. 25

[Koh10b] Kohana Team. *Kohana Documentation*, 2010. 24

[Lea00] Neal Leavitt. Whatever happened to object-oriented databases? *Computer*, 33:16–19, 2000. 6

[LG07] Fakhar Lodhi and Muhammad Ahmad Ghazali. Design of a simple and effective object-to-relational mapping technique. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, pages 1445–1449, New York, NY, USA, 2007. ACM. 14

[Mei06] Erik Meijer. There is no impedance mismatch: (language integrated query in visual basic 9). In *OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 710–711, New York, NY, USA, 2006. ACM. 8

[Mic09] Microsoft. *What's New in SQL Server 2008*, 2009. 8

[New06] Ted Neward. The vietnam of computer science, 2006. 2

[Ora08] Oracle. *Oracle Database Object-Relational Developer's Guide 11g Release 1 (11.1)*, 2008. 8

[Ora09] Oracle. *Oracle Fusion Middleware Developer's Guide for Oracle TopLink 11g Release 1 (11.1.1)*, 2009. 24, 26

[ORM09] ORMBattle.NET Team. *ORM Comparison and Benchmarks on ORMBattle.NET*, 2009. 23

[Rus08] Craig Russell. Bridging the object-relational divide. *Queue*, 6(3):18–28, 2008. 15