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

Die objektorientierte Programmierung ist in der heutigen Softwareentwicklung nicht mehr wegzudenken. Auf Sprachelemente wie Kapselung, Polymorphie und Vererbung kann nur schwer verzichtet werden. Ebenso sind relationale Datenbanken noch immer die erste Wahl für die persistente Speicherung von Daten. Sie skalieren für große Datenmengen, sind sicher, verwalten Transaktionen zwischen Benutzern und besitzen einen großen Anteil am wirtschaftlichen Markt.
Die Sammlung der Probleme die auftreten, wenn man objektorientierte Datenstrukturen in einer relationalen Datenbank speichern möchte werden als `Object-relational Impedance Mismatch` bezeichnet.
In dieser Arbeit werden die entstehenden Probleme, die durch den Unterschied zwischen den beiden Paradigmen enstehen, erläutert und bisher gefundene Lösungen für den `Impedance Mismatch` vorgestellt. Es wird gezeigt, warum es nicht eine einzige perfekte Lösung für den `IM` geben kann, sodass die bisher entwickelten Persistenzmechanismen alle einem bestimmten Anwendungsfall dienen.
Die Strategien werden kritisch betrachtet und eigene Verbesserungen und Konzepte ausgearbeitet, die eine Basis für ein weiteres Framework für Webapplikationen legen. Die Effizienz dieser verbesserten Konzepte wird durch einen Vergleich mit ähnlichen Produkten untersucht.

INHALTSVERZEICHNIS

Teil I

Die Theorie des Object-Relational Impedance Mismatch

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.

Beim `Impedance matching` in der Informatik sind die zwei Punkte, zwischen denen die Übertragung optimiert werden soll, die Welt der relationalen Datenbanken und die Welt der Objektorientierung.

Die objektorientierte Programmierung ist heute eine der bekanntesten Programmierparadigmen 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 Ausprägungen 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. Es 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 objektrelationale Lösungen für die Daten zu finden [Kee04]. Wenn der IM einmal sorgfältig gelöst wurde, hat das große Vorteile für die Entwicklung jeder weiteren Anwendung.
Bevor man sich also mit der Anwendungslogik und den Problemen beschäftigt, die sich in jedem Projekt neu ergeben, sollte man Zeit investieren den IM einmal zufriedenstellend zu lösen, um nicht in jedem Projekt erneut ein Drittel der verfügbaren Zeit verbrauchen zu müssen [New06].

1.1 Der Aufbau dieser Arbeit

Im nächsten Kapitel werden die Kernprobleme des Impedance Mismatch vorgestellt. Diese Probleme werden sich in allen Bereichen dieser Arbeit wiederfinden und unter verschiedenen Gesichtspunkten betrachtet werden.

Im dritten Kapitel werden allgemeine Lösungen für den IM vorgestellt. Entweder ersetzt man das relationale Datenbanksystem mit einem alternativen System, oder man versucht das Problem auf Softwareebene anzugehen. Es werden objektorientierte Datenbankmanagementsysteme betrachtet, sowie die Weiterentwicklung von relationalen Datenbankmanagementsystemen mit objektrelationalen Features.

Im vierten Kapitel konzentriere ich mich dann auf die Softwarelösung die sich Object-relational Mapper nennt. Die Strategien eines Object-relational Mapper werden vorgestellt, die Anforderungen definiert und mögliche Schwierigkeiten veranschaulicht.

Im fünften Kapitel wird ein kleiner Überblick über die große Masse von Frameworks, die sich als Lösung für den IM bezeichnen gegeben. Auch die Standards für die Speicherung von Objekten, die größtenteils durch den Java Community Process entwickelt wurden, werden erwähnt.
Zusätzlich werden zwei PHP Frameworks (Kohana und Doctrine) und zwei Java Frameworks (Hibernate und TopLink) untersucht. Ich beschreibe, wie diese die Anforderungen aus Kapitel 4 implementieren.

Der erste Teil wird mit einem kurzen Fazit in Kapitel 6 abgeschlossen.

Im zweiten Teil der Arbeit wird ein selbstentwickeltes Framework in PHP namens *PSCORM* vorgestellt. Anhand dieses Beispieles soll eine Möglichkeit gezeigt werden, wie man ein solches Framework umsetzen könnte.

In der Einführung werden spezielle Eigenarten der Webentwicklung mit PHP vermittelt und erklärt, warum sich diese so stark auf die praktische PHP Programmierung und auf PHP Frameworks auswirken.

Im zweiten Kapitel werden Besonderheiten der Implementierung von *PSCORM* kurz skizziert und Ideen aus dem theoretischen Teil weiter ausformuliert. Es werden Strategien aus den untersuchten Frameworks wieder aufgegriffen und eigene erklärt.

In einer Evaluation wird *PSCORM* mit Kohana und Doctrine, die schon im ersten Teil der Arbeit untersucht wurden, verglichen. Es werden zwei Tests für das Laden von Objekten ausgewertet. Danach werden diese Ergebnisse analysiert und kritisch bewertet, ob die neuen Konzepte für das Framework wirkliche Verbesserungen bedeuten können.

Im Schlusskapitel werden die Erkenntnisse dieser Arbeit zusammengefasst.

KAPITEL

ZWEI

Die Kernprobleme

Es stellt sich zuerst die Frage, welche konkreten Probleme zur Überbrückung des `Impedance Mismatches` gelöst werden müssen. Um den Zustand der Objekte und das Modell der objektorientierten Applikation abdeckend in der Datenbank persistent speichern zu können, müssen zumindest folgende Teilprobleme behandelt 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 Klassenhierarchie 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 überführt. 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 anderen Objekten gekoppelt (Assoziation), aus weiteren Objekten zusammengesetzt (Komposition) oder beeinhalten Objekte (Aggregation). Diese Objektbeziehungen 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 zueinander. Im relationalen Schema ist es nicht vorgesehen Vererbung natürlich 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 von Objektbeziehungen. Eine Gesamtsicht auf alle Objekte oder das Bilden von konkreten Mengen von Objekten ist schwierig, da immer 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`. Ich werde des öfteren auch „Hydrieren“ verwenden. Ein Objekt kann eine beliebige Struktur haben, aber ein Abfrageergebnis ist 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 (Objektidentität kurz OID).
- (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 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 ihre Prozesse Daten aus Tabellen abzufragen, weiter 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 von 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 [SM95]. 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 für die Auswahl der Datenbanksysteme: 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 objektrelationale 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

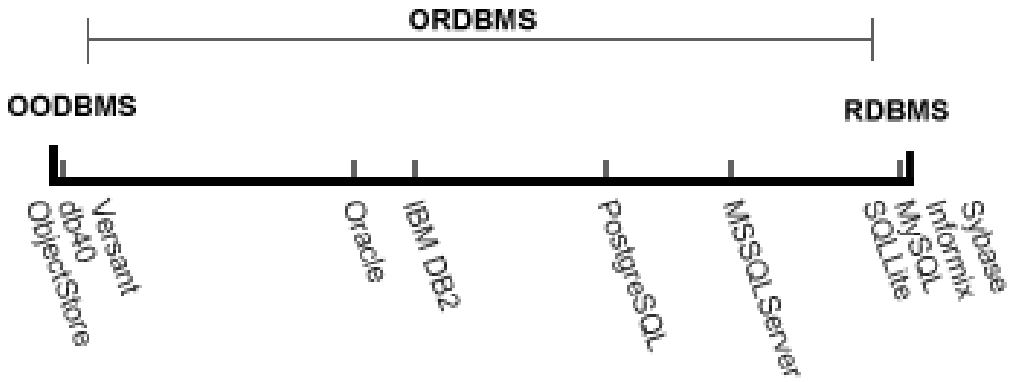

Abbildung 3.1: Klassifizierung beliebter Datenbankmanagmentsysteme

Platzierung der verschiedenen `DMBS` auf dem 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. Diese `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. Von Funktionen können Tabelle statt Skalare 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 deren 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. [IBM01].

PostgreSQL PostgreSQL tendiert in die `ODBMSs`-Richtung, da es auch die Vererbung in Tabellen unterstützt. Bei einem `CREATE TABLE` kann die Option `INHERITS` mit angegeben werden. Die Attribute der Elterntabelle werden dann auf die Kindtabellen übertragen. Zusätzlich versucht PostgreSQL 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.

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 objektrelationales 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.

Caché von Intersystems (nicht abgebildet) 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 nachdem welches ORDMBS oder RDMBS benutzt wird, gibt es noch genug Probleme die auf Softwareebene gelöst werden müssen. Es lässt sich nur schwer vorraussagen, wann sich Datenbanksysteme so verändern, dass es in der praktischen Anwendung überhaupt keine reinen relationalen Datenbanken mehr gibt. Deshalb gibt es genug Szenarien in denen der IM, oder Teile des IM nach wie vor eine große Rolle spielen und eine universellere Methode zur Lösung gefunden werden muss.

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 modellieren 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 entwickeln 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. Ein Mapping (welches meist auch sehr simpel sein kann) wandelt dabei den Typen des Objektattributes in den Typen der Tabellenspalte um. Eine Instanz eines Objektes wird dann durch ein Tupel in der Klassenrelation 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 Objektidentität 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 objektorientier-

ten Programmiersprachen eine eigene eindeutige Identität. Diese ist meist nicht als `Shadow information` verwendbar, da beim Laden eines Objektes aus der Datenbank 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 Abfragen 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 aller Objekte in der Applikation. Jede Instanz in der Tabelle *Instanzen* hat eine eigene OID (`Shadow`

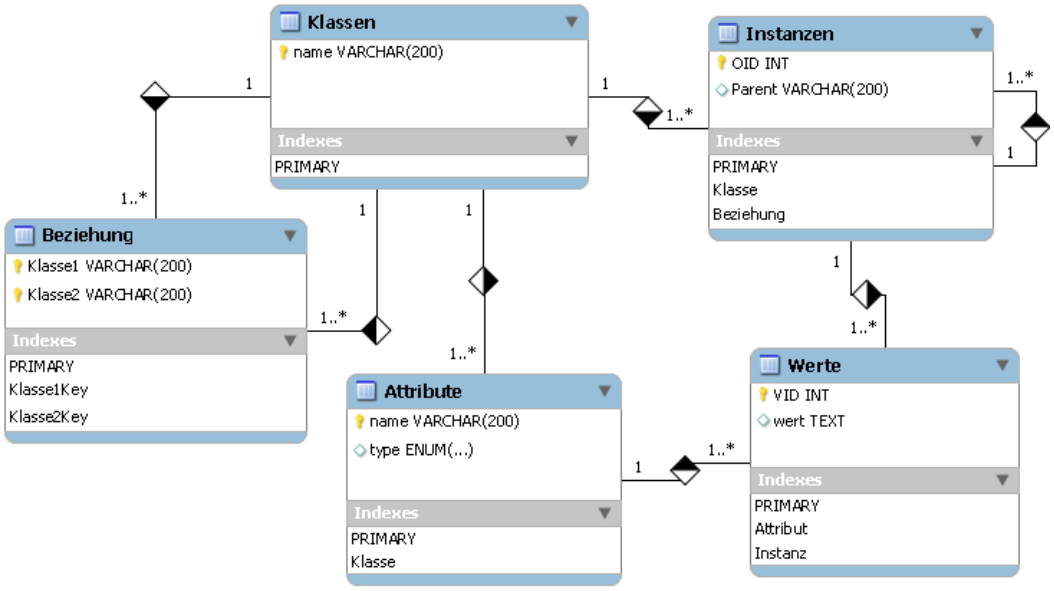

Abbildung 4.1: Beispiel einer Struktur für generelles Mapping

`Information`). Eine Instanz kann von einer anderen Instanz abgeleitet sein, oder mit einer anderen Instanz eine Beziehung haben (z. B. Vererbung).

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 Beziehungsrelation eingefügt werden die von Relation A und Relation B jeweils einen Fremdschlüssel speichert (Zwischentabelle).
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 Beziehungsrelation eingefügt (Zwischentabelle wie bei n:m im klassischen Fall). 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 für die zusätzlichen Tabellen im Schema ist, dass z. B. bei einer Beziehung zwischen Relation A und Relation B mit Multiplizität 1:n der Fremdschlüssel beim klassischen Schema in Relation B eingefügt werden muss. Betrachtet man die Beziehung aus der Sicht von Relation A, ist nicht mehr zu erkennen, dass eine Beziehung zu Relation B überhaupt besteht. Es ist auch nur Relation B für die Pflege der Beziehung verantwortlich. Dies entspricht nicht dem Prinzip der Kapselung von Informationen im Objekt der Relation A. Außerdem kann die Liste der Fremdschlüssel in einer Relation, die an besonders vielen Beziehungen beteiligt ist, schnell sehr lang wer-

den. Durch eine empirische Untersuchung kommt [LG07] zu dem Entschluss, dass die Lösung mit vielen Beziehungsrelationen ungefähr gleich schnell im Vergleich zu der klassischen ist.

4.4 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 simple 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: Jedes Objekt möchte seine eigenen Abfragen an die Datenbank verwalten (Kapselung), aber bei relationalen Abfragen werden Objekte in großen Mengen zurückgegeben (Performanz).
Ein weiterer wichtiger Aspekt ist die Entscheidung wie die Anfragen an das System gestellt werden. Am häufigsten werden diese drei Möglichkeiten benutzt:

1. Alle Abfragen werden in reinem SQL geschrieben. Der Entwickler muss dafür Sorge tragen, dass alle Informationen für jedes Objekt vollständig vorhanden sind. (reines SQL)
2. Die Abfragen werden in einem Abfrageobjekt mit einer objektorientierten API gekapselt. Diese Struktur wird später in reines SQL umgewandelt (objektorientierte SQL API)
3. Es wird eine eigene Abfragesprache definiert, die große Ähnlichkeiten mit SQL hat. Die Sprache vereinfacht den Umgang mit Objekten und wird ebenfalls später zu reinem SQL transformiert. (eigene Abfragesprache)

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 (2.). 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 gegen das Prinzip der Kapselung angesehen. Wenn in der Applikation SQL direkt benutzt wird, bindet man die Applikation dauerhaft 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. Wird das `DMBS` gewechselt muss auch die Applikation aufwendig geändert werden. Das Design der Abfragesprache wird als besonders schwierig betrachtet. Die Sprache soll in der Domain des Objekt-Models designed werden, aber muss mächtig genug sein um alle komplexen SQL-Abfragen, die sonst geschrieben werden würden, behandeln zu können ([Rus08, S. 24]). Die Entscheidung SQL, eine objektorientierte SQL API oder eine eigene Abfragesprache zu entwerfen, ist deshalb sehr komplex und viele Lösungen kann man daran unterscheiden.

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 die Basisklasse 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 `Project`-Objekt und ein `Aggregation`-Objekt und ruft auf beiden jeweils die `get()` Methode auf. Dadurch werden dann die beiden Unterobjekte korrekt geladen und

das `Aggregation2Project`-Objekt ist insgesamt geladen. Da zuerst das Hauptobjekt und dann erst die Unterobjekte geladen werden, bezeichnet man den Vorgang die Unterobjekte später zu laden als `Lazy Loading`.
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 das `Project`-Objekt und für das `Aggregation`-Objekt `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 Queries scheint verhältnismäßig groß zu sein. Hinzu kommt, dass viele der Queries mehrfach ausgeführt werden. In der Tabelle *projects* befinden sich nur ungefähr 180 Tupel. Die Beziehungen die in *aggregations_projects* gespeichert sind referenzieren also Tupel aus *aggregations* und *projects* mehrmals (Wir nehmen an, dass jedes Projekt mindestens einmal referenziert wird). Mit dem naiven Ansatz gibt es eine Mehrzahl von Objekten die aus *aggregations_projects* geladen werden, die dieselbe OID 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 Objektidentität.
Das Problem ist mit einem zusätzlichem positivem Nebeneffekt einfach zu lösen. Aber zunächst ein anderer Ansatz die Objekte zu laden:
Die Struktur bleibt weitesgehend gleich, es werden nur die `init()`- und `get()`-Methode der Klasse `Aggregation2Project` modifziert. Die `get()`-Methode wird aus der `BaseClass` überschrieben, so 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 Unterobjekte weitergegeben. Dann wird nur `init()` auf beiden aufgerufen, sodass diese ihre Attribute nun aus dem Resultset des großen SELECT erhalten können.
Der Vorteil dieses Vorgehens ist, dass die Anzahl der Queries um ein Vielfaches reduziert wird. Der Nachteil ist natürlich, dass nun die Klasse `Aggregation2Project` die Daten für `Aggregation` und `Project` aus der Datenbank geladen hat, denn das bedeutet, dass `Aggregation2Project` mindestens den Namen der Tabellen von `Aggregation` und `Project` wissen muss. Die Unterobjekte haben nun nicht mehr das Laden der Daten gekapselt wie in der ersten Version. Zwar wurden jetzt einige doppelte Abfrage eliminiert, aber auch hier entsteht 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 bereitzustellen. 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 in beiden Versionen 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 implementieren, müssen wir noch eine weitere Struktur einfügen: Einen `Object-Cache`. Der `Object-Cache` speichert eine Referenz auf die Objektinstanz 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 nun gesparten Abfragen und werden performanter. Trotzdem ist die erste Version immer noch langsamer als die zweite. Erst wenn man zuerst alle Objekte für *projects* und alle Objekte von *aggregations* aus der Datenbank lädt, bevor man mit dem Laden für die `Aggregation2Project`-Objekte von *aggregations_projects* beginnt, laufen beide Versionen in ungefähr derselben Zeit. Bevor also alle `Aggregation2Project`-Objekte erzeugt werden, wird nur einmal für `Project` und einmal für `Aggregation` folgende Abfrage ausgeführt:

```
SELECT * FROM :tabelle
```

Wir erhalten also mit einer Abfrage alle Datensätze aus *projects* bzw *aggregations*, erzeugen die Objekte und referenzieren diese im Cache. Diesen Vorgang nennt man `Prefetching`, da Informationen zu einer Menge gebündelt werden und mit einer großen Abfrage geladen werden, statt mit vielen kleinen.
Dies ist einfach zu verstehen, dass dies in diesem Fall die Performanz stark verbessert: Durch das Laden aller Objekte z. B. von *projects* wird nur ein einziger `Roundtrip` benötigt. Die Abfrage erstellt alle Instanzen der Objekte, die im weiteren Verlauf sowieso geladen werden würden und legt diese im `Object-Cache` ab. Wenn die `init()`-Funktion von `Aggregation2Project` nun die Objektinstanzen seiner Unterobjekte abfragt, muss keine neue Anfrage an die Datenbank gestellt werden. Es wird also kein Datensatz für ein Objekt mehr doppelt geladen.
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 (`Lazy Loading`).
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 anfangs 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. Ich verschaffe zuerst einen kleinen Überblick über die bestehenden Standards und ein paar der bestehenden `ORM`-Frameworks und Lösungen. Im zweiten Teil werden ein paar davon detailierter untersucht. Die verschiedenen Methoden wie diese Frameworks die Probleme des `IM` gelöst haben, liefern einen guten Eindruck über die Komplexität solch einer Implementierung.

5.1 Übersicht

Zur Entwicklung von `ORM`-Software wurden viele Spezifikationen entwickelt. Einen Überblick über allein die Java-spezifischen 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 objektrelationale 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. Man findet meistes auch noch eine passende 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 wird deshalb hauptsächlich für Java genutzt. Wird meistens als Datenbank unabhäniger Persistenz-Service 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 unter der GPL Lizenz veröffentlicht. Es unterstützt LINQ und kann auch ohne Datenbank benutzt werden. Die Internationalisierung von Objekten ist möglich und vieles mehr.
Cayenne	Entwickelt von der Apache Software Foundation, ist Cayenne ist Open Source `ORM`-Framework unter der Apache Lizenz. Außer dem Verbinden von Datenbankschemata mit 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) oder aber auch durch eine objektorientierte API. Es werden jeweils nur die Schnittstellen angesprochen und dieselben internen Datenbankbefehle genutzt. Mit der Jalapeño -Technologie (**JA**va **LA**nguage **PE**rsistence with **NO** mapping) ist es möglich `POJOs` mit einem `Object-Manager` persistent zu machen [Int06]. Mit dieser Technologie entfällt das Mapping von Klassen und die Objekte werden durch automatische generierte Methoden, die unabhängig von der Javaklasse sind, in der Caché Datenbank (oder sogar in einer relationalen Datenbank) gespeichert.

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, an Datenstrukturen innerhalb des Programmes. Die Spracherweiterung kann so von Herstellern erweitert werden, dass die Transformierung in andere Anfragesprachen möglich ist. Deshalb löst `LINQ to SQL` den `IM` [Mei06].
LinqConnect	Eine Implementierung die nah an `LINQ to SQL` von Microsoft angelehnt ist. Es ist eine einfach zu benutzende `ORM`-Lösung die als Server Oracle, SQL Server, MySQL, PostgreSQL und SQLLite unterstützt. Zusätzlich wird ein Modell-Designertool für Visual Studio mitgeliefert.
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 benutzt ähnliche Konzepte wie Hibernate und hat somit auch eine eigene Query Language (DQL).
Propel	Als Teil vom Symphonie Framework für PHP5, ist Propel für den `ORM`-Bereich zuständig. Propel generiert automatisch PHP 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	Ist eine Lösung für Ruby on Rails (ein Ruby Framework), die das ActiveRecord Pattern implementiert.
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. Sie ermöglicht durch ihre dynamischen Features interessante und neue Methoden alte Probleme zu lösen. GORM gilt deshalb als innovative Lösung für den `IM` [Ric08].

Einen Eindruck in die Performanzleistungen ein paar dieser Frameworks, die für .NET entwickelt wurden bietet [ORM09]. Dort wird auch beleuchtet inwiefern diese Frameworks die 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. Dies bedeutet natürlich nicht, dass nicht genannte besonders unrelevant in den anderen Bereichen sind. Am meisten konzentriere ich mich also auf die Frameworks, in die ich den meisten Einblick erhalten konnte:

- 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* Alternative für kommerzielle Produkte durchgesetzt.
Die Konfiguration eines Mappings wird in Hibernate durch den Entwickler 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, benötigt man bei Verwendung von horizontalem Mapping zu viele Joins um ein Objekt laden zu können. Filter-Mapping kann besonders effektiv sein, wenn die Basisklasse sehr groß ist und die Kindklassen nur wenige Attribute haben. Der schlechteste Lösunge wäre in diesem Fall ein vertikales Mapping zu benutzen, weil es viele, sehr kleine Tabellen erzeugen würde.
Anders als andere Frameworks entscheidet sich Hibernate nicht für nur für ein 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]
Der Entwickler muss sich nicht unbedingt für eine der verschiedenen Strategien entscheiden, sondern kann sogar mehrere miteinander kombinieren. Somit ist es z. B. möglich einen Teil der Hierarchie als `Filter Mapping` und den anderen Teil als `vertikales Mapping` abzubilden. Dies bietet höchste Flexibilität und lässt die Benutzung der performantesten Strategie zu.

5.3 Abfragesprachen

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 werden beide in vielen Projekten benutzt. Kohana überzeugt durch einen geringen und übersichtlichen Code, während Doctrine eine konsequente objektorientierte Umsetzung und viele Konzepte von Hibernate besitzt.
Wie in den Kapiteln zuvor betont, ist das Design der `Query-Language`, mit der Objekte aus der Datenbank geladen werden können, eins der anspruchsvollsten Aufgaben des `ORM`. Hibernate bietet eine eigene Implemetierung einer Abfragesprache genannt HQL (`Hibernate Query Language`), ermöglicht aber auch die Verwendung von purem SQL oder einer objektorientierten API. Kohana bietet nur diese API an (Beispiel 2).
Doctrine nennt seine Abfragesprache - analog zu Hibernate - DQL (`Doctrine Que-`

```
SELECT `timeslice`.`id` AS `timeslice:id`,
       `timeslice`.`user_id` AS `timeslice:user_id`,
       `timeslice`.`aggregation_id` AS `timeslice:aggregation_id`,
       `timeslice`.`start` AS `timeslice:start`,
       `timeslice`.`end` AS `timeslice:end`,
       `aggregations`.*,
       `users`.*,
FROM `aggregations`
LEFT JOIN `timeslices` AS `timeslice` ON (`timeslice`.`
    aggregation_id` = `aggregations`.`id`)
LEFT JOIN `users` AS `user` ON (`timeslice`.`user_id` = `users`.`
    id`)
WHERE `aggregations`.`closed` = '1'
AND `timeslice`.`user_id` = 10
ORDER BY `timeslice:start` DESC
```

Beispiel 1: Abfrage auf der Beispiel-Datenbank in reinem SQL

`ry Language`). Die DQL ähnelt SQL, allerdings wird diese Sprache erweitert, sodass es möglich ist z. B. Klassennamen als Tabellen und `JOINs` ohne `ON`-Bedingung zu benutzen. Da Doctrine einen eigenen Parser benutzt, um die DQL zu bearbeiten, sind auch eigene Erweiterungen dieser Sprache möglich und die Kapselung der unterliegenen Datenbank bleibt erhalten. Die DQL-Abfrage 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,

```
$res = ORM::factory('aggregation')
        ->with('timeslice')
        ->with('timeslice:user')
        ->with('aggregations_project')
        ->with('aggregations_project:project')
        ->where(Array(
            'aggregations.closed'=>'1',
            'timeslice.user_id'=>10,
         ))
       ->orderby(array('timeslice:start'=>'DESC'))
       ->find_all();
```

Beispiel 2: Abfrage auf der Beispiel-Datenbank in Kohana (objektorientierte SQL API)

das Datenbanksystem zu wechseln.
Es ist aber nicht nur möglich in DQL Abfragen zu schreiben. Doctrine stellt einen `QueryBuilder` zur Verfügung, der ähnlich wie in Kohana, eine objektorientierte SQL API zur Verfügung stellt.
Die HQL von Hibernate ermöglicht ebenfalls eine einfache Schreibweise für die Abfrage von Objekten: Im FROM-Teil von Abfragen können Objektnamen genutzt werden, `JOINs` können ohne `ON`-Bedingung benutzt werden und die erweiterte Punkt-Notation wie z. B. `aggregation.timeslice.start` erlaubt implizit auf Unterobjekte des Objektes zuzugreifen [1].
Es können - von der unterliegenden Datenbank unterstützte - skalare Funktionen

```
select Timeslice,
       Aggregation,
       User
from Aggregation agg
  left join fetch agg.timeslices as timeslices
  left join fetch timeslices.user as user
where agg.closed = 1
    and user.id = 10   # dasselbe wie agg.timeslices.user.id
```

Beispiel 3: Abfrage auf der Beispiel-Datenbank in Hibernate (HQL)

und Join-Typen benutzt werden. Damit werden nahezu alle Features von SQL durch HQL abgebildet. Eine eigene Abfragesprache zu entwickeln ist durchaus sinnvoll. 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 [Beispiel 4]. Die Vorteile sind also ersichtlich: Es ist viel einfacher große und komplexe Anfragen in einem ähnlichen Stil wie einem bereits bekannten Standard zu schreiben, als eine neue, objektori-

[1] in dem Falle hier ist `timeslice` ein Unterobjekt von `aggregation` und hat ein Attribut `start`

```
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
     )



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)
```

Beispiel 4: Die gleiche Abfrage in SQL (oben) und HQL (unten)

entierte SQL API zu lernen, die möglicherweise nicht vollständig alle Konstrukte von SQL übernommen hat[1]. Trotzdem wird mit den `HQL`-Abfragen die Applikation nicht an das Schema der relationalen Datenbank gekoppelt, da die Anfrage zuerst geparst wird und dann an die Datenbankverwaltung 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[2]. 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. 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

[1] Das ist leider bei Kohana der Fall

[2] Dieses Problem lässt sich mit Caches für PHP relativ gut umgehen

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 Vorgehen für ein bidirektionales Mapping für die Tabellen *timeslices* und *users* im Beispiel-Datenbankschema (Anhang 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 benutzt eine ähnliche Methode, allerdings werden Annotations oder XML Files als Konfigurationsmöglichkeiten genutzt. 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 im relationalen Schema abgebildet werden soll. 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[1]. 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.

[1] Dies ist das Prinzip der Zwischentabelle wie im Kapitel 4.3 erklärt

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, Vector, 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 `Query-Builder` (objektorientierte SQL API) 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 objektrelationale 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

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. Vor allem waren viele nicht unter dem Stichwort `Object-relational Impedance Mismatch` zu finden. Es war dennoch schwer Publikationen zu finden, die das Offensichtliche weiter vertiefen, viele Arbeiten stellten zwar die Probleme und eine eigene Lösung vor, aber wurden dabei nicht sehr spezifisch. 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.

6.1 Gibt es eine perfekte Lösung?

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 Per-

formanz. 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 (flexibel, einfacher) oder eine objektorientierte SQL API (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.
- Kapselt man die Datenverwaltung eines Objektes streng in den Methoden des Objektes selbst, oder lässt man zu, dass andere Objekte Daten für andere Objekte laden dürfen, sodass Synergie Effekte genutzt werden können (`Prefetching` von Attributen beim `Object Loading`).

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 nicht für jeden Anwender pauschal zu beantworten und somit wird es auch nie eine einzige, perfekte Lösung für den `IM` geben.

Teil II

Lösungsansätze am Beispiel-Framework PSCORM

KAPITEL

SIEBEN

Einführung

In diesem Teil der Arbeit möchte ich *PSCORM* (gesprochen: Pe-Skorm) vorstellen. *PSCORM* steht für: **P**HP **S**mart **C**ompiling **O**bject-**r**elational **M**apper und bezeichnet das Framework, welches ich im Rahmen dieser Arbeit konzipiert habe.
Im ersten Teil wurde eine Vielzahl von existieren Frameworks vorgestellt. Ein paar davon habe ich detailierter untersucht und festgestellt, dass es schon gute und konkrete Lösungen für den `IM` gibt. Wie ist also eine Neuentwicklung eines Frameworks in PHP motiviert? Anders als bei Java und .NET ist die Anzahl der Lösungen für ORM Probleme in PHP eher klein. Es gibt nicht viele innovative Lösungen, die sich durchsetzen konnten. Vielmehr ist es so, dass größere Frameworks, die nicht nur den `IM` als Problem addressieren, eine eigene `ORM`-Lösung mitlieferten. Die meisten Entwickler wählten zuerst ihr allgemeines Framework und danach die `ORM`-Lösung. Viele der `ORM`-Frameworks die es bereits gab, hatten zwar einige gute Ideen, wurden aber nicht weiterentwickelt und stetig verbessert.
Systeme die sich durchzusetzen scheinen, sind entweder viel zu umfangreich oder zu unflexibel und unvollständig. Vorallem die Spezialisierung auf Webentwicklung mit PHP, ist in den meisten Frameworks nicht vorhanden. So ist z. B. Doctrine stark an Hibernate angelehnt, welches ursprünglich für Java geschrieben wurde. Es leuchtet also ein, dass Doctrine nicht speziell und ausschließlich für PHP entwickelt wurde.
Auch wenn ich nicht behaupten will, dass meine Idee für *PSCORM* eine völlig Neue ist **??**, so habe ich zumindest versucht, die besten Ideen der bestehenden Frameworks auf einen Nenner zu bringen. Darüber hinaus habe ich versucht, den aktuellen Forschungsstand über den `IM` in das Konzept für *PSCORM* einfließen zu lassen.

7.1 Aufbau dieses Teils

Zuerst gehe ich auf die Besonderheiten von PHP in der Webentwicklung ein, die ein wenig Umdenken beim Programmieren verlangen. Anders als in Programmiersprachen wie Java, die durch die Virtual Machine ausgeführt werden und vorher kompiliert wurden, gibt es in PHP keine vergleichbare Struktur.
Danach werden die Konzepte von *PSCORM*, die aus den Analysen des ersten Teils entstanden vorgestellt. Viele Konzepte wurden von bekannten Frameworks übernommen, auf diese werde ich dann nicht detalierter eingehen, da sie bereits im ersten

Teil ausführlich behandelt wurden. Besonders die Programmierung von dynamischen Code wird immer wieder eine Rolle spielen. Dabei benutzt *PSCORM* einen eigenen Codeerzeuger, um einen Compiler zu simulieren. Dieses Konzept stellen wir im Kapitel 8.1 vor.
In den weiteren Kapiteln werden die Implikationen dieses Konzept auf die anderen schon bekannten Konzepte aus dem ersten Teil der Arbeit übertragen. Dabei wird oft mit den schon bekannten Frameworks (insbesondere Doctrine) verglichen.
Im letzten Kapitel werden die zwei Testfälle aus dem Kapitel 4.5 um die Performanz von Doctrine, Kohana und *PSCORM* zu verglichen, die Ergebnisse werden vorgestellt und analysiert.

7.2 Besonderheiten von PHP

Bei einer objektorientierten Applikation in PHP gibt es ein paar Besonderheiten, die man beachten muss. Insbesondere ist PHP eine Scriptsprache für Webanwendungen. Webanwendungen haben eine Besonderheit in Bezug auf den flüchtigen Speicher: Wird eine Webseite aufgerufen, sendet der Browser des Clients einen `HTTP-Request` an den Server. Der Server löst den Request auf und gibt diesen an den Interpreter (in dem Falle PHP) weiter. Der Interpreter liefert die HTML Ausgabe des Ergebnisses und der Request ist beendet. Danach sind alle Informationen dieses Requests, die nicht persistent gemacht wurden, wieder verloren. Es gibt auch keine Daten, die zur Laufzeit in einem Arbeitsspeicher gehalten werden können.
Was bedeutet das für die Programmierung in PHP? Alle Informationen die man zur Ausführung der Applikation braucht, müssen bei jedem Request in den Speicher geladen werden. Das bedeutet, dass z. B. Konfigurationsdateien jedes Mal neu ausgelesen werden müssen, Datenbankverbindungen neu erstellt werden müssen, oft genutzte Klassen müssen neu instanziiert werden und alle Informationen müssen persistent gemacht werden, die im Speicher lagen und nicht verloren gehen dürfen. Es ist nicht möglich die Applikation einmal zu initialisieren und dabei kostenspielige Operationen auszuführen, die im weiteren Verlauf gespart werden können. In jedem Request werden alle Initialisierungen der Applikation wieder neu ausgeführt.
Zusätzlich muss der Zugriff auf eine Webapplikation leseoptimiert sein. Stellt man sich eine durchschnittliche Webseite vor, die von einem Content Management System (CMS) gepflegt wird, sieht man, dass der Anteil der Aktionen, die Daten persistent machen, viel geringer ist als der Anteil der Aktionen die Daten lesen: Ein Redakteur speichert eine neue Seite im CMS ab. Eine Seite hat z. B. 200.000 Requests pro Tag; Das bedeutet, dass die Seite einmal gepeichert, aber 200.000 mal geladen wurde.
Im Allgemeinen gibt es noch etwas anderes zu beachten: Eine PHP Applikation mit geringem Codeumfang und einfach gehaltenem Code, läuft schneller als eine PHP Applikation mit großem Codeumfang, mit starker Abstraktion und vielen Klassen. Das liegt daran, dass der Interpreter bei jedem Request alle Klassen der Applikation neu parsen muss (Syntaxüberprüfung, Transformation, Optimierung, etc). Da PHP ursprünglich keine objektorientierte Sprache war, sind auch manche Funktionen eher für Arrays (Collection und Universaldatenstruktur in PHP) als für Objekte optimiert.

Wenn man also versucht ein Projekt für PHP zu optimieren, sollte man folgende Dinge berücksichtigen:

- Man sollte immer den Lesezugriff auf Daten, auf Kosten der Performanz vom Schreibvorgang optimieren.
- Die Applikation sollte sich so schnell wie möglich initialisieren
- Die Applikation sollte versuchen in einem Request nur die wichtigsten Daten zur Ausführung des Programms in den Arbeitsspeicher zu laden
- Der Codeumfang sollte so gering wie möglich gehalten werden
- Der Code sollte einfach gehalten werden

KAPITEL

ACHT

Konzepte und Implementierung

Anmerkung (Funktionen und Methoden)**:**
In PHP sind Variablen immer mit $ gekennzeichnet. Wenn ich:
`call($p1, $p2, $p3)`
schreibe, meine ich, dass die Funktion `call` *die Parameter* `p1`*,* `p2` *und* `p3` *hat.*

8.1 Dynamischer Code

Betrachtet man die Besonderheiten von einer Webapplikation in PHP, stellt sich schnell die Frage, wie man es überhaupt realisieren kann, eine sowohl sauber abstrahierte und komplexe, aber gleichzeitig performante Lösung zu schreiben. Umfangreicher Code bedeutet viel Aufwand für den Interpreter. Umfangreicher Code entsteht aber dadurch, dass die Applikation eine hohe Abstraktion besitzt. Eine hohe Abstraktion ermöglicht dem Entwickler einfacheres Arbeiten mit den Methoden der Applikation. Er ist flexibler in allen Entscheidungen, die er treffen muss, kann effizienter Aufgaben implementieren und kann sich somit auf die Businesslogik der Applikation konzentrieren. Der Code wird einfacher zu warten und ist deklarativer. In keinem Fall, kann man also auf umfangreichen Code in einer objektorierten Applikation aufgrund der Performanz verzichten. Es möglich in PHP einen sogenannten Bytecode-Cache zu benutzen, der zumindest das wiederholte Parsen des Codes überflüssig macht. Leider ist es oft so, dass die benötigte Architektur für so einen Bytecode-Cache in normalen Webprojekten nicht zur Verfügung steht. Nur auf eigenen selbstbetreuten Hostinglösungen, ist es möglich Bytecode-Caches zu konfigurieren und zu warten. Da die meisten Webseiten auf kommerziellen, einfachen Hostinglösungen liegen, ist dies nicht immer der Fall.
Die Lösung des Problems ist in *PSCORM* die Möglichkeit PHP Code dynamisch zu erzeugen. Durch die Ideen von Groovy (GORM) und Propel [Pro10], [Ric08] verleitet, kam ich auf die Idee die `ORM`-Lösung in zwei Ebenen zu organisieren.
Die erste Ebene ist eine komplett abstrahierte, umfangreiche und objektorientierte Applikation. Sie soll die Anforderungen für ein modernes `ORM`-Framework erfüllen und flexibles und schnelles Arbeiten ermöglichen. Hier sollen die Mappings erstellt und konfiguriert, Abfragen geschrieben und Modelle definiert werden.
Die zweite Ebene wird von der ersten Ebene dynamisch erzeugt. Der PHP Code der

ersten Ebene schreibt PHP Code für die zweite Ebene. Damit ist die erste Ebene mit einem Compiler vergleichbar, deshalb das **C** in *PSCORM*. Die Strukturen der ersten Ebene sollen in einfachen Code für die zweite Ebene transformiert werden. Die zweite Ebene wird dann auf den Webserver geladen und funktioniert ohne den umfangreichen Code. Jediglich der schlanke Core wird von beiden Ebenen benutzt.

Anmerkung (Laufzeit und Entwicklungszeitpunkt)**:**
Aktionen die in der ersten Ebene ausgeführt werden, bezeichne ich auch als „Aktionen zum Entwicklungszeitpunkt“ oder „Aktionen beim Kompilieren“.
Wenn ich von Algorithmen spreche, die „zur Laufzeit“ aufgerufen werden, meine ich Algorithmen, die durch Code in der zweiten Ebene umgesetzt werden.

8.2 Metainformationen

Zu jeder Klasse muss eine Menge an zusätzlichen Informationen bereitgestellt werden. Der Entwickler muss diese Metainformationen ergänzen können, sodass er das Verhalten von *PSCORM* kontrollieren kann. In den Metainformationen befinden sich z. B.:

- der Name der Relation in dem sich die Datensätze befinden
- die Spalten der Relation und ihre Typen
- die Mappings zu den einzelnen Spalten
- Beziehungen zu anderen Objekten. Inklusive `Lazy Loading`-Einstellungen und Constraints für Updates und Deletes
- Informationen über die Vererbung und die Klassenhierarchie der Klasse
- Name und Eigenschaften des Primärschlüssels

So wie in Doctrine könnte das Format indem der Entwickler diese Informationen an das System übergibt variabel sein. In Doctrine sind XML, YAML und Annotations (formatierte Kommentare in der Klasse selbst) möglich. Da in der zweiten Ebene diese Informationen in einfach ausführbaren Code umgewandelt werden können, sind die Voraussetzungen für verschiedene Metadatenformate gegeben. Bis jetzt gibt es nur einen Prototypen einer PHP API. Die Metainformationen sollten aber nicht in der Entity-Klasse selbst untergebracht werden, da sie sonst mit den Businessinformationen des Objektes kollidieren könnten. In der Demo Applikation für die Evaluation ist dies noch für den Namen der Tabelle der Fall. Im Moment wird dieser im Objekt selbst gespeichert, da er die einzige gebrauchte Metainformation in der Demo ist. Dies soll in Zukunft geändert werden.

8.3 Cache

Der `Object-Cache` kann in einer `ORM`-Software nicht fehlen. Er bringt nicht nur einen enormen Performanzvorteil sondern löst auch automatisch das Problem der Objektidentifizierung (Kapitel 4.5). Durch das Cache-Kriterium, welches entscheidet ob ein

Objekt bereits geladen wurde, ist notwendigerweise jedes Objekt eindeutig identifizierbar. Die Applikation muss also darauf achten, dass jedes Objekt eine OID besitzt. Dies ist einfach zu gewährleisten, indem immer wenn es keine eindeutige OID gibt, ein künstlicher Integer-Schlüssel in die Tabelle eingefügt wird. Dieser wird dann für jede neue Zeile automatisch erhöht und ist immer eindeutig für alle Objekte dieser Klasse. Wenn eine Abfrage oder eine `Lazy Loading`-Methode nun versucht ein Objekt zu instanziieren, befragt es immer erst den Cache. Die Frage ist ein Methodenaufruf mit 2 Parametern: Der erste Parameter ist die Klasse des gewünschten Objektes und der zweite Parameter die OID. Der `Object-Cache` verwaltet somit alle Referenzen auf alle Objekte in der Applikation und sollte überall erreichbar sein. Denn immer wenn ein neues Objekt aus der Datenbank geladen wird, sollte dies dem Cache mitgeteilt werden, damit alle Teile der Applikation vom Cache profitieren können. Vielmehr darf es nicht erlaubt sein, ein Objekt zu erzeugen, ohne, dass der Cache darüber informiert wird, denn sonst wäre es möglich, dass zwei nicht identische Objekte die selbe OID in der Applikation besitzen.
In *PSCORM* wird der Cache als Singleton implementiert. Er kann überall aus der Applikation heraus mit `Cache::getInstance()` geladen werden. Objekte werden mit `populate()` dem Cache bekannt gemacht und können mit `get($class, $oid)` abgefragt werden.
Die Methode um von überall ein einzelnes Objekt zu laden, fragt immer zuerst beim Cache nach, ob das Objekt existiert, ansonsten setzt es eine Abfrage an die Datenbank ab. Die Funktion die also überall benutzt wird, um eine Instanz einer Klasse zu erhalten lautet `ORMObject::load($class, $oid)`.

8.4 Datenbankabstraktion

Über die Notwendigkeit eines DBAL (`Database Abstraction Layer`) wurde schon genug diskutiert [Amb98]. Dennoch sollte man die Vorteile nutzen, die man hat, wenn man das unterliegende Datenbanksystem gut kennt. Natürlich ist es praktisch jederzeit eine Datenbank gegen eine andere (möglicherweise schnellere oder günstigere) auszutauschen, ohne die Applikation zu verändern. Ich denke aber, dass dies eher ein weniger genutzter Sonderfall ist. (In Analogie dazu, dass `RDBMSs` auch nicht durch `ODBMSs` ersetzt wurden, eben weil große Firmen nur ungern ihr `DMBS` wechseln). Man sollte diesen Punkt also verinnerlichen und berücksichtigen, während man seine Applikation entwirft, aber dies sollte sich nicht auf Kosten der Performanz geschehen.
In PHP ist in den neueren Versionen (ab PHP5) der richtige Schritt im PHP-Core gemacht worden: PDO ist eine PHP-Erweiterung die es ermöglicht mit demselben Interface verschiedene Datenbanktypen anzusprechen (Postgres, Oracle, MySQL, MS SQL, DB2, ODBC, SQLLite, Informix). Die Basisklasse der Extension kann mit einer eigenen Klasse abgeleitet werden, sodass die API sogar gekapselt werden kann. PDO soll genauso schnell wie die nativen Treiber der Datenbanken für PHP sein. D. h. es ist für die Datenbankabstraktion völlig ausreichend seine Abfragen über PDO abzusetzen.
In *PSCORM* übernimmt die Klasse DB die Ableitung von PDO. Dabei wurde nur

der Konstruktor überschrieben, der die Zugangsdaten zur Datenbank aus einer Konfigurationsdatei liest und die allgemeine Query-Funktion um das Loggen von SQL Befehlen zu ermöglichen.

8.5 Abbildung von Klassen und Attributen

Am besten haben mir die Lösungen von TopLink und Doctrine für die Zuordnung von Attributen zu Objekteigenschaften gefallen. Natürlich werden in auch *PSCORM* Klassen als Relationen, und Spalten von Tabellen als Objekteigenschaften abgebildet. Jede Klasse, die ein abstraktes Interface `ObjectMapping` implementiert, soll dafür genutzt werden ein Mapping zu modellieren. Das Interface benötigt dafür nur zwei Hauptfunktionen: convertToPHPValue() und convertToDatabaseValue(). In Doctrine ist ein Mapping in dieser Form umgesetzt worden. Ein Standardset von Basismappings für Integer, String, Datetime, Floats, Decimals usw wird direkt mitgeliefert. So soll es auch in *PSCORM* sein. Der einzige Unterschied ist, dass beim Kompilieren (Exportieren) des Projektes nicht die komplette abstrake Mappingklasse in die Applikation eingefügt wird, sondern allein der Code aus den beiden Funktionen zur Umwandlung direkt in die betreffenden Objekte kopiert wird. In der Praxis gibt es dabei ein paar Besonderheiten zu beachten, die ich zum Zeitpunkt dieser Arbeit noch nicht ganz lösen konnte. So war es z. B. ein Problem, dass der Code nicht auf Teile des Objektes zugreifen darf, weil es in der exportierten Applikation nicht im Kontext eines `ObjectMapping` sondern direkt im betreffenden Datenobjekt aufgerufen wird. Mit PHP 5.3 werden aber Closures (anonyme Lambda Funktionen) eingeführt, die höchstwahrscheinlich das Problem mit nativen PHP Sprachelementen lösen können.

8.6 Abbildung von Vererbung

Bei der Abbildung der Vererbung musste zuerst entschieden werden, welche Vererbungstypen in *PSCORM* umgesetzt werden sollten. Ich habe zuerst nur vertikales und horizontales Mapping sowie Filter-Mapping implementiert. Ich hatte überlegt generelles Mapping zusätzlich zu implementieren. Das Problem beim generellen Mapping ist aber, dass es sehr schnell unperformant wird, da so viele JOINs zum laden eines Objektes in der Abfrage nötig sind. Dieses Verfahren hätte aber gut genutzt werden können um sehr kleine Projekte mit geringem Umfang on-the-fly persistent machen zu können. Ein paar kleine Tests haben aber gezeigt, dass es schneller ist, das Objekt serialisiert (bei PHP für alle Objekte möglich) in eine Tabelle mit BLOBs zu speichern, mit den Primärschlüsseln OID und Klasse. Die Struktur ähnelt etwas einem `Object-Cache` ist aber persistent in der Datenbank. Zur Abfrage eines Objektes wird nur eine Abfrage benötigt. Zur Performanceoptimierung könnte man noch für jede Klasse so eine Tabelle anlegen, sodass der zweite Primärschlüssel Klasse wegfallen würde.
Die einfachste Form bei der Implementierung der Vererbung ist nicht das horizontale Mapping, sondern das Filter-Mapping. Beim Filter-Mapping gibt es nicht viele Probleme zu lösen. In den Metainformationen zur Klasse wird definieren der Name

der Klassentabelle und die Filter-Spalte (Discriminator) definiert. Durch die dynamische Code Generierung können wir die `save()` und `get()` der Entity-Klasse schon so modifizieren, dass diese die Abfrage an die Klassentabelle stellt und den Discriminator immer als Filter im WHERE benutzt. Beim Speichern (und beim Laden) des Objektes rufen wir auch `save()` vom Elternobjekt auf, sodass Änderungen von Informationen für Spalten, die nur im Elternobjekt existieren, ebenfalls gespeichert werden. Dieser rekursive Aufruf setzt sich dann bis zur ersten Klasse der Hierarchie fort, da das Elternobjekt wiederum die Methode seines Elternobjekts aufruft. Bei einer tiefen Klassenhierarchie könnte dies ein Performanzproblem beim Speichern geben. Dann müsste man im rekursiven Aufruf die Updates der einzelnen Klassen sammeln und diese in einem Update an die Datenbank schicken. Dazu wäre eine größere Modifikation der `save()`-Methoden in allen Klassen nötig, aber es wäre nicht unmöglich.
Der Vorteil beim Filter-Mapping ist, dass alle OIDs einer Klassenhierarchie für ein Objekt gleich sind. Da ein Datensatz für ein Objekt nur eine Zeile in einer Relation ist, sind alle Datensätze der Unterklassen von diesem Primärschlüssel abhängig. Jedes Objekt kennt also beim Speichern seine OID und kann somit einfach ein

```
UPDATE :filter-tabelle SET attribute1 = attribute1wert, ...
  WHERE id = :OID
```

ausführen, und so seine Spalteninformationen aktualisieren. Das ist bei horizontalem und vertikalem Mapping nicht so. Dort ist es nämlich so, dass der Datensatz für ein Objekt immer auf mehrere Tabellen verteilt ist, sofern das Objekt mindestens ein Elternobjekt besitzt. Beim Aufrufen der `get()`-Funktion wird auch die `get()`-Funktion des Elternobjektes aufgerufen. Doch wann soll dies in der `get()`-Funktion des Kindes passieren? Passiert dies vor dem Initialisieren des Kindobjektes, überschreibt die Elternklasse zumindest das Attribut OID in der Kindklasse, d. h. die Kindklasse kennt seine eigene OID nicht mehr. Werden die Funktionsaufrufe umgedreht (dies ist in Java z. B. gar nicht möglich) wird das Attribut für die OID der Elternklasse überschrieben. D. h. entweder muss garantiert werden, dass jede OID der Klassenhierarchie einen eindeutig unterscheidbaren Namen hat, oder aber die OIDs müssen noch separat abgesichert werden. Ersteres wäre natürlich für die Applikation das Eleganteste, aber für den Datenbankdesigner ein großes Hinderniss, wenn man bedenkt, dass eine Klasse auch in mehreren verschiedenen Klassenhierarchien existieren kann. Der beste Ansatz ist also in einem Stack (Array) die OIDs separat beim Aufrufen der jeweiligen `init()`-Funktion abzuspeichern. Von diesem Stack kann dann jede Klasse bei `save()` seine OID herunternehmen und so seinen UPDATE-Befehl ausführen.
Das Vorgehen ist für horizontales Mapping und vertikales Mapping gleich, da beim horizontalen Mapping zwar Attribute doppelt aktualisiert werden, dies aber nicht weiter schlimm ist (höchstens bei einer tiefen Klassenhierarchie, aber dieses Problem könnte man auch Analog wie das Problem für das Filter-Mapping beheben).
Jediglich die Beziehungen zwischen den Objekten müssen noch gepflegt werden. Wir markieren dieses Vererbungsbeziehung als 1:1 Beziehung und speichern sie als gewöhnliche Beziehung ab. Die Konzepte dazu werden im nächsten Kapitel erläutert.

8.7 Beziehungen zwischen Objekten

Es soll in *PSCORM* zum Entwicklungszeitpunkt festgelegt werden können, welche Abhängigkeiten eine spezielle Klasse zu anderen Klassen hat. Dabei soll entschieden werden, ob *PSCORM* die Funktionen zur Verwaltung einer Collection selbst hinzufügen soll. Dies ist möglich, da die erste Code Ebene auch neue Eigenschaften zu Objekten hinzufügen kann. Wenn gewünscht, übernimmt auch der automatisch erzeugte Code die Pflege der Beziehung beim Speichern des Objektes (Delete, Update, Insert). Eine Beziehung zwischen Objekten kann somit unidirektional und bidirektional spezifiziert werden, indem entweder beide Seiten oder nur eine Seite die Metainformationen für die Beziehung angeben.
Der dynamische Code ermöglicht hier, eine eigentlich komplexe Implementierung einer transparenten Collection als PHP-Basisstruktur (Array). Normalerweise müsste ein Array, durch ein Objekt gekapselt werden. Denn immer wenn Objekte der Collection hinzugefügt werden, oder entfernt werden, muss auch die Beziehung in der Datenbank gepflegt werden. Durch *PSCORM* kann aller Code der zum Verwalten der Collection benötigt wird, direkt in die Entity-Klasse geschrieben werden. In Doctrine muss dafür eine komplexe Datenstruktur benutzt werden, was den Nachteil hat, dass die sehr performanten, in PHP eingebauten Array-Funktionen nicht mehr benutzt werden können. Gerade bei großen Datenmengen ist in PHP nichts performanter als ein einfacher Array. Durch die dynamisch angelegten Methoden wird die Kapselung des Arrays direkt in der Klasse erreicht.
Für jede Beziehung soll es möglich sein, den Fremdschlüssel im relationalen Schema auf der Many-Seite der Beziehung einzufügen oder eine Zwischentabelle zu benutzen, wie in [LG07] beschrieben.

8.8 Abfragesprache

Die Entscheidung, wie Abfragen für Mengen von Objekten an das System gestellt werden (siehe Kapitel 4.4), war nicht besonders leicht. Eine der Anforderungen, die ich beim Arbeiten mit Kohana am wichtigsten hielt, war, dass die Anfragesprache vollständig sein muss. Es ist ein Problem, wenn der Entwickler seine Anfrage an die Datenbank in SQL formulieren kann, aber keine Möglichkeit hat diese Anfrage an die `ORM`-Software zu stellen. Eine eigene Anfragesprache zu schreiben, war für den ersten Entwurf von *PSCORM* eine viel zu große Aufgabe. Die Entscheidung fiel dann letzendlich auf eine eigene Abfragesprache, jedoch sollte das Schreiben von SQL möglich sein. Die größte Kritik SQL zu erlauben ist neben der Bindung ans Schema, dass es sehr fehleranfällig wäre. Denkt man insbesondere an `SQL-Injections`, ist meist ein Entwicklerfehler für unsicheren oder fehlerhaften Code verantwortlich. Mein Focus bei der Entwicklung des Abfragesystems von *PSCORM* lag deshalb darauf, alles zu ermöglichen, aber alle Standardarbeiten vom System übernehmen zu lassen:

- JOINs können mit einer Funktion erstellt werden, sodass nicht die komplette ON-Bedingung geschrieben werden muss.
- Das Escaping (auch Schutz vor `SQL-Injection` von Datentypen kann durch

eine Spracherweiterung des SQLs automatisch erledigt werden.

- Die Spalten eines Objektes müssen nicht explizit referenziert werden

Der Vorteil von *PSCORM* ist, dass jede noch so aufwendige Funktion, die für das Parsen von handgeschriebenem SQL oder das Überprüfen vieler zusätzlicher Bedingungen beim Erstellen eines SQL-Befehls sich nicht auf die Laufzeit der Applikation auswirken kann. So könnten z. B. in Zukunft handgeschriebene SQL-Befehle gegen das Datenmodell geprüft werden, Sicherheitsaspekte überprüft oder Optimierungstipps gegeben werden. Bei der Kompilierung der Applikation wird dann reines SQL exportiert, welches sicherer und fehlerunfällliger sein wird.
Zusätzlich zu der Unterstützung für das Schreiben von eigenem SQL gibt es Query-Klassen die zur Laufzeit genutzt werden können. Diese Query-Klassen sind erweiterbar und bieten einen Grundstock an Standardfunktionen. Somit gibt es eine Funktion für jedes Business-Objekt der Klasse, welche die ganze Relation als Objekte hydriert. (z. B. für `Project Query::getALLProjects()`). Diese Query-Klassen können dann von mehreren Stellen der Applikation auch außerhalb eines Objekt-Kontext benutzt werden. In den Tests für die Evaluation (9) werden zwei Beispiele für Query-Klassen gezeigt.

8.9 Object Loading

Wie das generelle Object-Loading einzelner Objekte in *PSCORM* funktioniert, wurde zum Teil schon im Abschnitt über den Cache beschrieben (Kapitel 8.3).
Im Kapitel 4.5 über das `Object Loading` wurde ein Ansatz vorgestellt für die Optimierung von Ladestrategien. Eine Idee für ein Statistiksystem, welches Tipps geben kann, welche Strategie gerade die sinnvollste wäre, möchte ich hier vorstellen. Es soll sich um das Problem handeln, wie man sich entscheidet eine Menge von Unterobjekten in einer Applikation zu laden. Wie zuvor schon erwähnt, gibt es eigentlich zwei Strategien Objekte zu laden:

1. Alle Datensätze der Tabelle werden zuvor mit einer Abfrage geladen und als Objekte im Cache referenziert (`Prefetching`)

2. Die Datensätze werden beim Zugriff auf die Unterobjekte durch die Applikation einzeln, durch simple Abfragen geladen und zu Objekten transformiert (`Lazy Loading`)

Bei der 1. Mögichkeit werden also immer alle Datensätze der gesamten Relation geladen. Bei der 2. Möglichkeit werden nur die Datensätze geladen, auf die durch die Applikation tatsächlich zugegriffen wird.
Zur Veranschaulichung noch ein kleines Beispiel: Es ist bekannt, dass in einer Applikation auf 140 Objekte vom Typ `Project` aus der Tabelle *projects* zugegriffen wird. Die Tabelle *projects* hat eine Kardinalität von 182. Für welche Möglichkeit sollte sich der Entwickler entscheiden?
Die `Prefetching`-Methode würde bedeuten zuerst die Abfrage `SELECT * FROM projects` auszuführen und die daraus enstehenden Objekte mit populate() dem Cache

bekannt zu machen. Werden dann im weiteren Verlauf in der Applikation die 140 `Project`-Objekte referenziert, werden diese aus dem Cache geladen. Weitere Anfragen an die Datenbank entfallen.
Die `Lazy Loading`-Methode wäre einfach keine Abfrage auszuführen. Die Objekte würden in der Applikation mit `ORMObject::load()` geladen werden. Sofern sich die Objekte dann noch nicht im Cache befinden, wird eine einzelne Abfrage an die Datenbank gestellt [8.3]. Dann werden natürlich 140 Abfragen der Form `SELECT * FROM projects WHERE id = :id` an die Datenbank gestellt.
In diesem speziellen Beispiel müsste sich der Entwickler für `Prefetching` entscheiden, wenn er die schnellere Methode wählen will. Wie gelangt man aber zu so einer Entscheidung? Exakter formuliert müsste man fragen: ab wann lohnt es sich `Prefetching` statt `Lazy Loading` zu benutzen? Oder: Wann ist `Lazy Loading` langsamer als `Prefetching`? Ich kann zu diesem Zeitpunkt noch keine endgültige, mathematische Formel angeben, dennoch lassen sich die Variablen dieser Formel gut bestimmen.
Die Kardinalität der Domain aus der die Objekte kommen ist sehr ausschlaggebend. In diesem Beispiel hat *projects* 182 Datensätze, die ein Objekt repräsentieren. Würden in der Applikation auf alle `Project`-Objekte zugegriffen, ist klar, dass `Prefetching` die beste Methode ist, denn die `Roundtrips` zur Datenbank werden minimiert [9]. Wird nur eine sehr geringe Anzahl von `Project`-Objekten geladen (Anzahl der Objekte $\ll$ Kardinalität der Tabelle) leuchtet ein, dass `Lazy Loading` benutzt werden sollte. Ich bezeichne die gesamte Anzahl der Datensätze in der Datenbank als x.
Ein weiterer Faktor ist die Größe der einzelnen Datensätze. Bei der kleinen Größe eines Datensatzes der Relation *projects* im Beispieldatenbankschema (Anhang A), welcher aus 3 Integers und einem Varchar besteht, war der Unterschied zwischen den beiden Methoden in einem Benchmark schwer zu erkennen. Deshalb wurde die Tabelle für *projects* mit einer weiteren Spalte `data` ergänzt. `data` ist ein Largeblob und wurde mit verschieden großen, Binäredaten befüllt, sodass jeder Datensatz nun mindestens die Größe des Largeblob hat. Diese Veränderung der Datengröße hat zur Folge, dass nun nicht mehr die `Roundtrips` and die Datenbank der dominierende Zeitfaktor sind, sondern eine Abfrage eines einzelnen Datensatzes viel Zeit kostet.
Es wurde dann mit unterschiedlichen Größen des Largeblobs (256 KB, 512KB, 4MB, 16MB) der folgende Test durchgeführt: Es wurde die Laufzeit für die `Prefetching`-Methode ausgewertet, wenn in der Applikation alle Objekte, die in der Datenbank existierien, angefragt werden (x Objekte). Danach wurde die `Lazy Loading` Methode mit einer variablen Anzahl y von Objekten solange laufen gelassen, bis die vorher ermittelte Laufzeit der `Prefetching`-Methode erreicht wurde. Das Ergebnis dieses Versuches ist also, dass in derselben Laufzeit mit der `Prefetching`-Methode x Objekte und mit der `Lazy Loading`-Methode y Objekte geladen werden können.
Das Verhältnis zwischen x und y veränderte sich, wenn die Menge der Daten im Feld `data` vergrößert oder verkleinert wurde. Natürlich ist dies nur ein experimentelles Ergebnis, aber es lässt sich eine Tendenz ablesen: Desto größer der Datensatz wurde, desto mehr Objekte konnten mit `Lazy Loading` in der selben Zeit geladen werden. Das bedeutet, dass es sich bei größeren Datensätzen länger lohnt `Lazy Loading` zu benutzen, als die kompletten Daten mit `Prefetching` zu laden. Dies ist auch das

logisch erwartete Ergebnis: Sind die Datensätze sehr klein, dominiert der Overhead der `Roundtrips` an die Datenbank die Laufzeit und es ist sinnvoller direkt alle Daten auf einmal zu laden. Sind die Datensätze groß, möchte man möglichst wenig nicht genutzte Daten übertragen.
Für *PSCORM* könnte dann eine Strategie zur Optimierung sein, dass dieses Verhältnis zwischen y und der Kardinalität der Tabelle dieser Objekte dynamisch errechnet wird. Dieses Verhältnis muss dann zusätzlich von der Datensatzgröße beeinflusst werden. Erhalten wir z. B. ein Verhältnis von 73% so bedeutet das, dass in der Applikation mindestens 73% der Datensätze der Tabelle als Objekte angefordert werden müssen, damit sich `Prefetching` lohnt. Werden die Datensätze größer, so muss das Verhältnis weiter ansteigen, denn für große Datensätze ist der dominierende Zeitfaktor die Übertragung zum Client, also wird die Menge der übertragenen Daten minimiert (wir fragen nur die Daten ab, die wir der Applikation liefern müssen). Werden die Datensätze kleiner, so fällt das Verhältnis. Jetzt ist es schneller die Anzahl der Anfragen zu minimieren, da der Overhead eines `Roundtrips` die Laufzeit dominiert. Es ist nicht mehr so relevant, wieviele Daten gelesen werden. Es können also Daten zusätzlich geladen werden, die in der Applikation nicht benötigt werden. Diese Berechnungen sollen bei der Entwicklung der Applikation bereits geschehen, sodass der Entwickler - sofern er mit guten Testdaten arbeitet - früh Tipps erhalten kann (z. B. über eine Entwicklerkonsole), wie er das Object-Loading verbessern kann.
Ein praktisches Problem ist dabei die Berechnung von y, da natürlich keine Laufzeittests bei der Entwicklung von Hand durchgeführt werden können. Mein naiver Ansatz war ein Basisverhältnis von 60% (geschätzt durch Experimente) anzunehmen und dieses durch Modifikationen wie eine übergroße Datensatzgröße oder die Kardinalität der Tabelle im Verhältnis zur Anzahl der zu ladenen Objekte in der Applikation zu beeinflussen.
Es ist auch noch nicht klar, ob noch weitere wichtige Faktoren das Laden von Objekten stark beeinflussen können, die ich hier noch nicht überblicken konnte.

KAPITEL
NEUN

Evaluation

In diesem Kapitel werden *PSCORM*, Kohana und Doctrine in einem Benchmark verglichen. Dabei wird der automatisch generierte Code der zweiten Ebene von *PSCORM* genutzt. Teile dieser Demo-Applikation sind im Anhang B zu finden. Kohana und Doctrine wurden so konfiguriert, wie in ihren Dokumentationen angegeben. Im ersten Kapitel stelle ich die Konfiguration des Testsystems vor. Danach werden beide Tests mit einigen Implementierungsdetails vorgestellt. Im letzten Kapitel befindet sich die Auswertung des Benchmarks.

9.1 Konfiguration

Das Beispiel aus dem Kapitel 4.5 wurde als erster Test genommen. Es werden alle Tests in allen Applikationen (Kohana, Doctrine, *PSCORM*) durchgeführt und ihre Laufzeit gemessen. Dabei wird die Laufzeit als arithmetisches Mittel über mindestens 5 voneinander unabhängige Aufrufe des Tests errechnet (um Schwankungen auf dem Hostsystem auszugleichen). Die Ergebnisse der Tests sind in 9.4 zu sehen. Die Tests wurden auf einem Hostsystem mit folgender Konfiguration ausgeführt:

- AMD Athlon 64 X2 Dual Core Prozessor 4800+ (2 x 2,41 GHz)
- 2GB RAM
- Microsoft Windows XP Professional 32 Bit Service Pack 3
- PHP 5.3.3 (VC6 Thread Safe)
- MySQL 5.0.41-community-nt
- APC 3.1.5-dev
- Xdebug v2.1.0

Alle nicht notwendigen Prozesse wurden während der Tests ausgeschaltet. Die Laufzeiten wurden mit dem XDebug-Profiler aufgezeichnet und mit Wincachegrind (1.0.0.12) ausgewertet. Da der Schwerpunkt der Analyse auf die Performanz der PHP Applikation gelegt werden soll, wurde der Anfragencache von MySQL deaktiviert. Der

Opcode-Cache `APC` wurde für alle Skripte aktiviert und so benutzt, dass die erste Auswertung bereits mit einem warmen Opcode-Cache startete.

9.2 Test1

In Test1 und Test2 sollen alle `Aggregation2Project` inklusive der Unterobjekte `Project` und `Aggregation` instanziiert werden. In Test1 werden wir die 3 Objekte jeweils einzeln konstruieren:

1. Lade alle Objekte der Relation *projects*.
2. Lade alle Objekte der Relation *aggregations*.
3. Lade alle Objekte der Relation *aggregations_projects*. Lade zusätzlich die Informationen für die Unterobjekte `Project` und `Aggregation`
4. Zeige die Informationen aller `Aggregation2Project`-Objekte mit deren Unterobjekten an

Damit das Anzeigen der Daten im letzten Schritt in jedem Test und jedem Aufruf gleich ist, wird ein Anzeigeskript am Ende jedes Tests aufgerufen (Quelltext 1). Die Applikation selbst muss einen Array (`$objects`) von allen `Aggregation2Project`-Objekten zurückgeben.
Wenn die Applikation von einem Cache gebrauch macht, ist die Objektidentität

```
foreach ($objects as $test) {
  print $test->getProject()->getName();
  print '('.$test->getProject()->getListvisible().')  ';
  print $test->getSeconds().' '.$test->getShare().'  ';
  print $test->getAggregation()->getClosed();
  print '<br />';
}
```

Quelltext 1: print_objects.php: Die Daten jedes Tests werden ausgegeben

gewährleistet und die Unterobjekte des Objektes `Aggregation2Project` werden aus dem Cache geladen, sodass bei der Ausgabe keine weiteren Anfragen an die Datenbank nötig sind. Im Anhang B ist der entscheidene Teil des Quelltextes der Applikationen abgebildet. Ich fasse die Vorgehensweisen der einzelnen Applikationen für die jeweiligen Tests kurz zusammen:

Kohana In Kohana ist es nicht möglich einen Cache zu benutzen. Der Entwickler müsste selbst dafür sorgen, dass die Objekte die in 1. und 2. geladen werden, nicht erneut beim Instanziieren von `Aggregation2Project`-Objekten nachgeladen werden. Dieses Problem wurde in diesem Test nicht berücksichtigt, da ein Entwickler ohne die Kentnisse über die Interna von Kohana diesen Fehler ebenfalls begehen

würde. Schritt 1 und Schritt 2 wurden dann weggelassen, da beide Schritte auf die Ausführung von Schritt 3 keinen Einfluss haben. In Kohana wird also nur Schritt 3 ausgeführt und im Anzeigeskript werden dann die `Project`-Objekte und `Aggregation`-Objekte dynamisch nachgeladen.

PSCORM Aus der `Query-Collection` für den Test1 wird getAllAggregation2Project() aufgerufen. Es werden zuerst alle Objekte aus *projects* dann aus tabelleaggregations hydratet. Der Cache von *PSCORM* nimmt alle Objekte auf, die in diesen Funktionen mit populate() übergeben werden. Danach werden die Unterobjekte beim Instanzieren von den `Aggregation2Project`-Objekten jeweils aus dem Cache referenziert.

Doctrine In Doctrine ist der Ablauf ähnlich wie in *PSCORM*. Durch sogenannte `Entity-Repositories` können direkt Objekte von einer ganzen Tabelle geladen werden. Es werden zuerst alle Objekte aus *projects* geladen dann von *aggregations* ohne das Ergebnis dieser Abfragen zu untersuchen. Danach werden alle Einträge aus *aggregations_projects* in Objekte umgewandelt. Da man die SQL Befehle die Doctrine absetzt ausgeben kann, sieht man, dass nach einmaligen Laden von z. B. *projects* keine weiteren Abfragen auf diese Tabelle gemacht werden. Da die Unterobjekte in dem Ergebnis der dritten Abfrage von *aggregations_projects* aber korrekt instanziiert sind, muss Doctrine diese Objekte aus einem eigenen Cache geladen haben.

9.3 Test2

Der zweite Test unterscheidet sich dadurch, dass nun nicht mehr die Unterobjekte von *projects* und *aggregations* vorher geladen werden sollen. Der Applikation soll es erlaubt sein in einem großen Query alle Informationen auf einmal abzufragen. Die Ausgabe soll aber dieselbe sein wie für Test1.

1. Lade alle Objekte der Relation *aggregations_projects*. Lade zusätzlich die Informationen für die Unterobjekte `Project` und `Aggregation`
2. Zeige die Informationen aller `Aggregation2Project`-Objekte mit deren Unterobjekten an

Kohana In Kohana ist es möglich das `Lazy Loading` wie in Test1 manuell zu unterbinden. Beim Erstellen der Abfrage für die `Aggregation2Project`-Objekte kann man angeben, welche Unterobjekte durch einen `JOIN` mitgeladen werden sollen. Dies geschieht durch die Funktion with() 2. Dass die Objekte nicht mehr mit eigenen Abfragen nachgeladen werden, kann man am SQL-Log der Applikation sehen.

PSCORM Zu Demonstrationszwecken für den Test2 wurde eine ähnliche Funktionalität hinzugefügt. Die Klasse QueryTest2 führt in getAllAggregation2Project() einen `JOIN` über alle 3 Tabellen aus und gibt dann jeweils den gesamten Datensatz an alle Unterobjekte weiter, um init() ausführen zu können. Da bevor init() von `Aggregation2Project` aufgerufen wird die Unterobjekte dem Cache bekannt gemacht wurden, kann die init()-Funktion von `Aggregation2Project` auf die Objekte

im Cache zurückgreifen. Wie in 4.5 gezeigt, ist dies aber die langsamere Methode in *PSCORM* Objekte zu laden, deshalb wurde Sie nur zum Vergleich zu den anderen Frameworks hier aufgenommen.

Doctrine In Doctrine wird die Anfrage in DQL geschrieben. Dabei kann man mit 2 `fetch Joins` bestimmen, dass die Daten für die Unterobjekte für `Aggregation2Project` direkt mitgeladen werden. Im SQL-Log sieht man, dass nur eine Abfrage mit 2 `JOINS` ausgeführt wird und keine weiteren Abfragen abgesetzt werden.

9.4 Ergebnisse

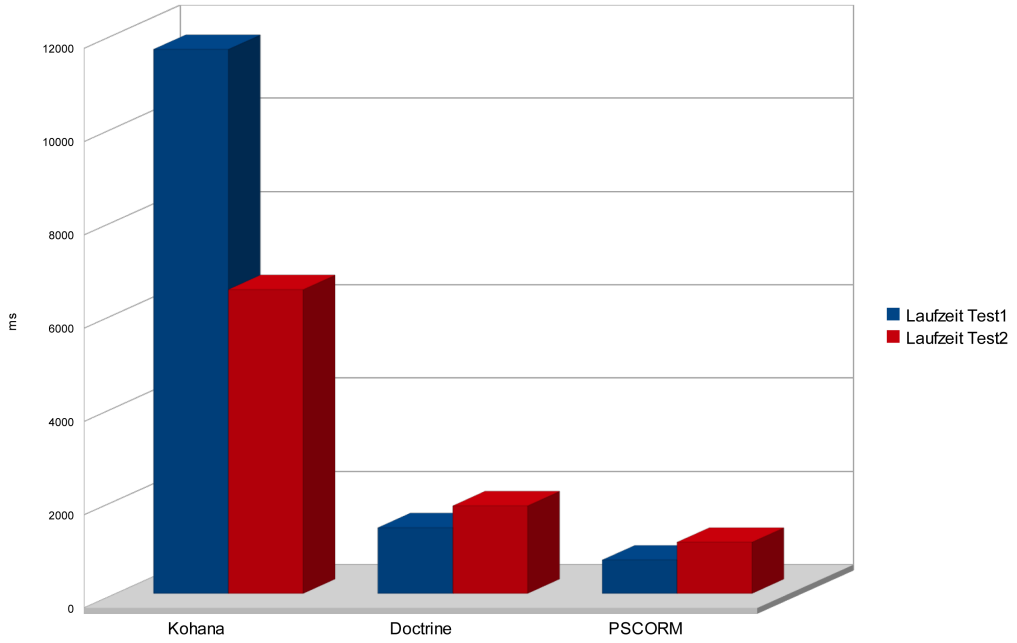

Abbildung 9.1: Auswertung der Laufzeiten für die Tests

Auf der Y-Achse des Diagrammes ist die Laufzeit der verschiedenen Tests in Millisekunden aufgetragen. Kohana braucht für Test1 ungefähr 11 Sekunden. Doctrine und *PSCORM* lösen Test1 im Sekundenbereich (1,4s und 0,7s).
Test2 verbraucht bei Doctrine und *PSCORM* mehr Resourcen und Zeit als Test1. Kohana verbessert sich und benötigt ungefähr 6,5s, Doctrine 1,8s und PSCORM 1,1s.
Zuerst fällt also auf, dass Kohana fast das 10-fache an Zeit für den ersten Test braucht und sich beim zweiten Test verbessern kann, während *PSCORM* und Doctrine sich leicht verschlechtern. Wie schon beschrieben, ist es nicht möglich einen Cache für die Objekte in Kohana (Version 2.4) zu benutzen. Es gibt zwar die Möglichkeit einen allgemeinen APC Cache anzusprechen, man müsste aber die komplette Logik für die Objekte selbst schreiben. Außerdem ist es nicht möglich in den Prozess des

`Object-Marshalling` einzugreifen, sodass die Funktionen von Kohana gar nicht von einem `Object-Cache` profitieren könnten. Zur Erinnerung: in *aggregations_projects* gibt es 11.862 Tupel. Diese müssen in beiden Tests als Objekte geladen werden. Jedes Objekt besitzt zwei weitere Unterobjekte. Da Kohana keinen Cache benutzen kann, muss das Framework 11.862 * 2 Einzelanfragen der Form `SELECT * FROM aggregation|project WHERE id = :id` an die Datenbank stellen und daraus 11.862 * 3 Objekte herstellen. Die schlechte Laufzeit ist die Konsequenz von dem, was im Kapitel über das Object-Loading (4.5) festgestellt wurde; Zu viele `Roundtrips` an die Datenbank verschlechtern die Performanz erheblich. *PSCORM* und Doctrine führen nur 1 + [Kardinalität von *projects*)] + [Kardinalität von *aggregations*] Anfragen an die Datenbank aus. Davon hydrieren sie nur so viele Objekte wie es Tupel in der Datenbank gibt: 11.862 + 193 (`Projects`) + 3.742 (`Aggregations`) = 15797 Objekte. Kohana hydriert dagegen die mehr als doppelte Anzahl von 35.586 Objekten.
Dies erklärt auch direkt, warum Kohana im zweiten Test besser wird. Da es die Daten direkt in der Abfrage für die `Aggregation2Project`-Objekte mit einem JOIN mitlädt, entfallen die mehr als 22.000 simplen Einzelabfragen an die Datenbank. Trotzdem muss es noch die gleiche Anzahl an Objekten aus diesem Query erstellen.

Warum ist *PSCORM* noch ein bißchen schneller als Doctrine in beiden Tests? Wie im einleitenden Kapitel über PHP-Webapplikationen (7.2) beschrieben, gewinnt man Performanz, wenn man den Code möglichst einfach hält und besonders wenig mit Klassen oder Funktionen abstrahiert. Die Klassen von *PSCORM* auf denen die Tests ausgeführt wurden, sind eigentlich automatisch generierte (kompilierte) Codeteile. Während Doctrine die DQL-Anfrage noch in SQL umwandeln muss, benutzt *PSCORM* eine als String hartcodierte Anfrage, die vorher durch den Export des Projektes erstellt wurde. Diese muss zur Laufzeit nicht erst dynamisch von PHP-Funktionen oder Klassen zusammengebaut werden. Doctrine opimiert dieses Parsen der DQL-Anfrage, indem es den APC Cache benutzt. Der Vorteil durch das Benutzen von möglichst simplen Code von *PSCORM* zeigt sich aber auch beim Hydrieren von Objekten: Doctrine muss hier Klassen instanziieren, um die speziellen- und Basismappings für die Umwandlung von einem Wert einer Spalte der Datenbankrelation in einen Wert der objektorientierten Applikation vornehmen zu können. In *PSCORM* wird der Code direkt verwendet, weil er dynamisch in die init()-Funktion eingebunden wurde. *PSCORM* führt also zur Umwandlung eines Strings aus der Datenbank in einen Integer in PHP nur einen Typcast aus, während Doctrine die (gecachten) Metadaten für die Spalte auslesen muss, die Klasse für das Mapping instanziieren muss und die Funktion zur Umwandlung aufruft. Auch wenn der Performanzunterschied mit Caching und Wiederverwendung des Mappings zwar sehr reduziert werden kann, muss man bedenken, dass dieses Mapping im Test1, wenn es im `Aggregation2Project`-Objekt benutzt wird, 11.000 mal ausgeführt wird.

KAPITEL

ZEHN

Zusammenfassung

Die Probleme es IM ergeben sich dadurch, dass sich die zwei Paradigma unabhängig voneinander entwickelten und somit andere Schwerpunkte bei der Modellierung der realen Welt setzten. Die Probleme erstrecken sich über die Struktur der Klassen, die Objektzustände, die Objektidentität und dem Prinzip der Kapselung bis zum Abfragen und Verarbeiten von Objekten.
Diese Probleme können entweder auf Softwareebene gelöst werden (durch eine ORM-Lösung, oder man ersetzt das relationale Datenbankmanagementsystem mit einem alternativen. Es wurde erklärt, warum sich ODBMSs nicht durchsetzen konnten und dass eine Kategorisierung der vorhandenen DMBS als ORDMBS schwer ist. Die relationalen DMBS entwicklen sich in Richtung der ODBMSs weiter, während diese die Nischenmärkte versorgen. Da aber nicht für jedes der Kernprobleme bereits eine Lösung auf Datenbankebene existiert, muss ein Object-relational Mapping definiert werden.
Der Zustand eines Objektes wird dadurch gespeichert, dass seine Attribute in elementare oder erweiterte Datentypen zerlegt werden und als Datensätze in Tabellen der Datenbank gespeichert werden, die nach der Klasse des Objektes benannt sind. Dabei ist es möglich, dass ein Objekt auf mehrere Relationen verteilt wird. Besonders bei Klassenhierarchien ist die Struktur der Modellierung sorgfältig zu wählen, da sie unmittelbar Einfluss auf die Performanz oder Flexiblität der Anwendung haben kann. Beziehungen von Objekten werden so wie im relationalen Datenmodell gespeichert. Entweder fügt man Fremdschlüssel in Relationen ein oder verbindet alle Klassentabellen durch Zwischentabellen. Beim Laden und Hydrieren von Objekten, muss sich die ORM-Lösung für eine odere mehrer Möglichkeiten entscheiden:
Sie benutzt reines SQL und überlässt dem Entwickler alle möglicherweise verteilte Daten eines Objektes selbst auszuwählen und dem Objekt zur Verfügung zu stellen. Dabei muss der Entwickler der objektorientierten Applikation Kentnisse über das Schema der Datenbank, über das Paradigma der relationalen Datenbanken und über den Datenbanktyp haben, um effizient arbeiten zu können. Man verliert den Vorteil den Datenbanktyp ändern zu können, oder gar das relationale Schema zu ändern, ohne den Quelltext der objektorientierten Applikation anzupassen. Die Applikationen wird durch menschliche Fehler fehleranfälliger und unsicherer. Andererseits ist es dem Entwickler möglich die beste Performanz zu erreichen und es ist gewährleis-

tet, dass er alle Optimierungen des relationalen Datenbanksystems nutzen kann. Die Abfragesprache (meistens ein SQL-Dialekt, welcher sehr gut bekannt ist) kann als Schnittstelle zur Datenbank uneingeschränkt genutzt werden, sodass dem Entwickler bekannte, Methoden und Tools verwendet werden können.
Die zweite Mpglichkeit ist, dass sich die `ORM`-Lösung dazu entscheidet eine eigene API für den Massenzugriff auf Objekte bereitzustellen. Diese von mir genannte objektorientierte SQL API ist fehlerunfälliger und sicherer, aber sollte den selben Funktionsumfang wie die Abfragesprache der Datenbank besitzen, was in der Praxis nicht immer gelingt (Kohana). Die Abfragen werden in einer abstrakten Struktur modelliert, die unabhängig vom Datenbanktyp ist und in Abfragen der nativen Anfragesprache der Datenbank umgewandelt werden kann. Man gewinnt an Flexiblität, Effizienz und Sicherheit, der Entwickler muss jedoch die objektorientierte SQL API so gut kennen, dass er seine Abfragen auch effizient formulieren kann und die Performanz der Abfragen nicht verschlechtert.
Die dritte und von Hibernate genutzte Möglichkeit, ist eine bekannte Abfragesprache wie SQL mit eigenen Sprachelementen abzuleiten und dem Entwickler zur Verfügung zu stellen. Der Benutzer der `ORM`-Lösung kann nun auf gewohnte Art und Weise Abfragen an die Datenbank erstellen, ihm wird jedoch ermöglicht viele fehleranfällige Aufgaben, die er mit der nativen Anfragesprache selbst erledigen müsste, dem System zu übergeben. Die Abfragen werden dadurch kürzer, übersichtlicher, für menschliche Fehler unanfälliger und sicherer. Der Nachteil ist, dass die Abfrage zuerst geparst und ausgewertet werden muss, erst dann kann sie in eine native Abfrage an die Datenbank umgewandelt werden. Dies kostet zusätzliche Resourcen und das System muss dafür Caching einsetzen.
Viele der großen `ORM`-Lösungen bieten deshalb alle drei Möglichkeiten an, um dem Entwickler seine preferierte Methode selbst wählen zu lassen. Dabei wird davon ausgegangen, dass die Vor- und Nachteile verstanden und akzeptiert werden.
Das Laden von Objekten wurde als spezielles Thema in dieser Arbeit behandelt. Die naive Lösung, in der jedes Objekt seine eigenen Anfragen schreibt und seine Attribute und `Mappings` selbst verwaltet, lässt sich mit einem `Object-Cache` und der richtigen Abfragestrategie zu einer konkurrenzfähigen Lösung erweitern. Wird ein Objekt geladen, welches Unterobjekte agreggiert, lässt es seine Unterobjekte eigene Abfragen an die Datenbank stellen und somit seine Daten kapseln.
Die alternative Lösung ist, dass das Hauptobjekt die Daten für die Unterobjekte aus der Datenbank anfragt, und die Unterobjekte aus diesem Ergebnis ihre Daten erhalten. Beide Lösungen sind ungefähr gleich schnell, wenn beide einen `Object-Cache` benutzen, der bereits instanziierte Objekte referenziert, sodass ein bereits geladenes Objekt nie noch einmal geladen werden muss.
Die richtige Strategie zum Laden der Objekte hat enorme Auswirkung auf die Performanz der Applikation. Dabei wird zwischen zwei Hauptstrategien unterschieden, die sich aber auch vermischen lassen: `Lazy Loading` und `Prefetching`. Beim `Lazy Loading` werden Unterobjekte eines Hauptobjektes nicht direkt geladen, wenn das Hauptobjekt aus der Datenbank geladen wird. In viele `ORM`-Lösungen werden die Unterobjekte durch ein Proxy-Objekt ersetzt, welches erst dann zu einem richtigen Objekt transformiert wird, wenn die Applikation auf dieses Proxy-Objekt zugreift. Mit dieser Strategie werden nur die Daten aus der Datenbank geladen, die tatsächlich in

der Applikation benötigt werden. Der Nachteil gegenüber der Prefetching-Strategie ist, dass zu viele Unterabfragen an die Datenbank gestellt werden, wenn die Anzahl der geforderten Objekte sehr groß wird. Es werden zuviele Roundtrips an die Datenbank benötigt, die die Laufzeit dominieren und die Performanz verschlechtern. Überschreitet das Verhältnis zwischen der Anzahl der angeforderten Objekte in der Applikation und der Anzahl der Objekte in der Datenbank insgesamt eine bestimmte Schwelle, ist Prefetching dem Lazy Loading vorzuziehen. Desto größer die Datensätze in der Datenbank für ein einzelnes Objekt sind, desto lukrativer ist es Lazy Loading zu benutzen. Desto kleiner die Datensätze, desto performanter ist es Prefetching zu benutzen.
Prefetching und Lazy Loading können auch auf weitere Bereiche beim Laden von Objekten ausgeweitet werden. So ist es z. B. möglich einzelne Attribute von Objekten erst dann zu laden, wenn sie benötigt werden. Man könnte sich eine Übersicht von Artikeln vorstellen, die alle einen 4 seitigen Text als BLOB in der Datenbank gespeichert haben. Würde eine Liste dieser Artikel-Objekte angezeigt, in denen nur der Titel und der Author angezeigt werden, wäre das Laden des großen BLOB sicherlich fatal für die Performanz. Man könnte dann den Text des Artikel-Objektes erst laden, wenn wirklich darauf zugegriffen wird. Mit dem Konzept des Prefetching würde man dann alle Text von allen Artikeln laden, weil man davon ausgehen könnte, dass noch weitere Texte angefragt werden würden (Dies ist das Ergebnis aus [BPS99]).
Die Frameworks die im ersten Teil untersucht wurden, zeigen wie flexibel ORM-Lösungen sein müssen. In Hibernate sind alle Möglichkeiten, die als allgemeine Lösungen für die Kernprobleme verstanden werden, implementiert. Vererbung kann mit allen drei Strategien umgesetzt werden, diese können sogar miteinander gemischt werden. Hibernate hat eine eigene Anfragesprache (HQL) und es ist möglich Abfragen in SQL zu formulieren. Doctrine übernimmt viele Ideen von Hibernate für PHP. TopLink von Oracle konzentriert sich auf ein eigenes ORM-Tool, welches auch ohne viel Programmierkentnisse benutzt werden kann um objektorientierte Modelle mit relationalen Schemata zu verbinden. Hibernate und TopLink erscheinen auf den ersten Blick sehr komplex, sind aber sehr gut durchdacht und lassen sich hervorragend konfigurieren und an alle Bedürfnisse anpassen.
Die Frage, ob es die perfekte Lösung für den Object-relational Impedance Mismatch gibt, kann man vermutlich verneinen. Sei es ein ODBMSs für einen Nischenmarkt oder Hibernate für eine Java-Umgebung oder Doctrine für ein PHP Webprojekt. ORM-Lösungen müssen so vielen unterschiedlichen Anforderungen gerecht werden und so viele Tradeoffs zwischen Abstraktion und Performanz lösen, so dass immer Schwerpunkte von der einen oder anderen Lösung besser zu einem speziellen Projekt passen. Auf lange Sicht würde ich sagen, dass die flexibelste Lösung, die von den Entwicklern meist benutzte werden wird. So ist es aus meiner Sicht schon mit Hibernate. Hibernate bietet alle Konfigurationen und alle Strategien an ein Kernproblem des IM zu lösen, sodass man immer die für seinen speziellen Problemfall beste Lösung aussuchen kann. Nicht ohne Grund gibt es NHibernate für .NET und Doctrine für PHP, die dieselben Ideen wie Hibernate benutzen.

Die Entwicklung von *PSCORM* war hauptsächlich dadurch motiviert, dass es noch

kein speziell angepasstes Framework für PHP gab. Es gibt genug gute `ORM`-Lösungen für PHP. So halte ich z. B. Doctrine für eine sehr gutes `ORM`-Framework, welches ich jederzeit in einem neuen Projekt einsetzen würde. Doctrine basiert aber größtenteils auf Konzepten von Hibernate, welches wiederrum für Java geschrieben wurde. Natürlich funktioniert diese Lösung für eine objektorientierte Programmiersprache für eine andere - zumindest größtenteils - objektorientierte Programmiersprache wie PHP. Durch die Evaluation und die Vorüberlegungen in den Kapiteln über die Konzepte von *PSCORM*, wurde aber auch gezeigt, dass eine Spezialisierung sowohl Vorteile für die Performanz als auch für das Design der Applikation mit sich bringt. Viele Konzepte konnten für *PSCORM* aus schon vorhandenen Frameworks übernommen werden. Dabei konnte bei jedem System vom Konzept des dynamischen Code gebrauch gemacht werden: Bei den Metadaten für die Konfiguration der Mappings, beim Erstellen der eigenen Abfragesprache, beim Verwalten von Beziehungen und natürlich auch beim `Object Loading`. Die Entwicklung für all diese Komponenten dynamischen Code zu erzeugen, ist fast abgeschlossen und könnte vermutlich ein Thema für eine weitere Arbeit sein. Jedoch reichte die Zeit nicht, eine vorzeigbare Version mit dieser Arbeit zu veröffentlichen.
Ich hätte gerne noch viel mehr Kernprobleme des `Object-relational Impedance Mismatch` bearbeitet, als es in dieser Arbeit möglich war. Da ich mich mit diesem Problem seit über 4 Jahren in der täglichen Arbeit beschäftige und es immer wieder unter neuen Gesichtspunkten von der praktischen Seite kennengelernt habe, war ich sehr erstaunt, wieviele neue Aspekte ich durch die theoretische und experimentelle Auseinandersetzung mit dem Problem gewinnen konnte. Es ist sicherlich möglich in weiteren Arbeiten die Konzepte für alle Kernprobleme des `IM` zu verfeinern und weiter auszuarbeiten, so wie ich es für das `Object Loading` gemacht habe. Ich war etwas überrascht, als ich die Evaluation durchführte und mit meinem einfachen Ansatz bessere Laufzeiten als bereits etablierte Systeme erreichen konnte. Sicherlich ist dies nicht repräsentativ für die gesamte Skalierbarkeit der bestehenden `ORM`-Lösungen, aber dennoch zeigt es, dass noch viel Potential zur Optimierung vorhanden ist. Ich hoffe, dass in Zukunft nicht nur die großen Software- und Datenbankfirmen interne Forschungen für ihre eigenen `IM`-Lösungen anstellen, sondern viele weitere Arbeiten zum Thema enstehen, die vielleicht auch in einer Suchmaschine mit dem Schlüsselwort `Object-relational Impedance Mismatch` gefunden werden.

ANHANG

A

Beispiel Datenbank Schema

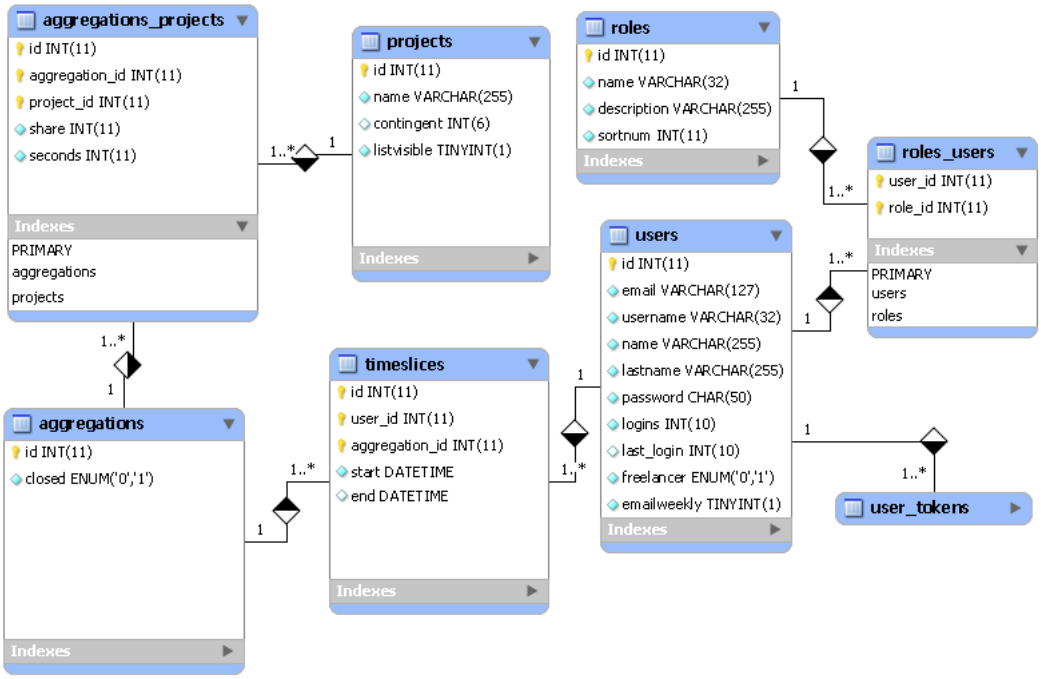

Abbildung A.1: Die Beispieldatenbank

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

ANHANG

B

Quelltext der Evaluation

```
// [...] bootstrap

$em->getRepository('Entities\Project')->findAll();
$em->getRepository('Entities\Aggregation')->findAll();
$objects = $em->getRepository('Entities\Aggregation2Project')->
    findAll();

?>
<html><head><title>Doctrine Test1</title></head>

<body>
<?php include 'print_objects.php'; ?>
</body>

</html>
```

Quelltext 2: Doctrine Test1

```
// [...] bootstrap

$query = $em->createQuery("SELECT a2p, p, a FROM
    Aggregation2Project a2p JOIN a2p.project p JOIN a2p.aggregation
     a ORDER BY a2p.id");
$objects = $query->getResult();

?>
<html><head><title>Doctrine Test2</title></head>

<body>
<?php include 'print_objects.php'; ?>
</body>

</html>
```

Quelltext 3: Doctrine Test1

```
// Code im Controller

$objects = ORM::factory('Aggregations_Project')->find_all();

?>
<html><head><title>Kohana Test1</title></head>

<body>
<?php include 'print_objects.php'; ?>
</body>

</html>
```

Quelltext 4: Kohana Test1

```
// Code im Controller

$objects = ORM::factory('Aggregations_Project')
    ->with('project')
    ->with('aggregation')
    ->find_all();
?>
<html><head><title>Kohana Test2</title></head>

<body>
<?php include 'print_objects.php'; ?>
</body>

</html>
```

Quelltext 5: Kohana Test2

```
// Code im Controller

$q = new PScorm_QueryTest1($db);
$objects = $q->getAllAggregation2Project();


?><html><head><title>PSCORM Test1</title></head>

<body>
<?php include 'print_objects.php'; ?>
</body>

</html>

// QueryTest1 Class
public function getAllAggregation2Project() {
  $this->getAll('Project');
  $this->getAll('Aggregation');

  return $this->getAll('Aggregation2Project',' ORDER BY
      aggregations_projects.id');
}

public function getAll($class, $orderbySQL = NULL) {
  $class = 'PScorm_'.$class;
  $o = new $class;

  $sql = "SELECT * FROM ".$o->getTable().$orderbySQL;

  $ret = array();
  $cache = PScorm_Cache::instance();
  foreach ($this->fetchResult($sql) as $row) {
    $o = new $class();
    $o->result = $row;
    $o->init();

    $cache->populate($o);
    $ret[] = $o;
  }
  return $ret;
}
```

Quelltext 6: PSCORM Test1

```
// Code im Controller

$q = new PScorm_QueryTest2($db);
$objects = $q->getAllAggregation2Project();

?><html><head><title>PSCORM Test2</title></head>

<body>
<?php include 'print_objects.php'; ?>
</body>

</html>
```

Quelltext 7: PSCORM Test2

```
public function getAllAggregation2Project() {
  $o = new PScorm_Aggregation2Project;

  // SQL fuer die JOIN Abfrage
  $sql = "SELECT aggregations_projects.*,
              aggregations.id as aggregations_id,
              aggregations.closed as aggregations_closed,
              projects.id as projects_id,
              projects.name as projects_name,
              projects.contingent as projects_contingent,
              projects.listvisible as projects_listvisible ";
  $sql .= "FROM ".$o->getTable()." ";
  $sql .= SQL::LEFTJOIN('aggregations',array($o->getTable(),'id','
     aggregation_id'),'1:n');
  $sql .= SQL::LEFTJOIN('projects',array($o->getTable(),'id','
     project_id'),'1:n');

  $ret = array();
  $cache = PScorm_Cache::instance();
  foreach ($this->fetchResult($sql) as $row) {

    //Aggregation-Objekt laden und Ergebnis aus der Datenbankzeile
        uebergeben
    $a = new PScorm_Aggregation;
    $a->result = $row;
    $a->prefix = 'aggregations_';
    $a->init();
    $cache->populate($a);

    //Project-Objekt laden und Ergebnis aus der Datenbankzeile
       uebergeben
    $p = new PScorm_Project;
    $p->result = $row;
    $p->prefix = 'projects_';
    $p->init();
    $cache->populate($p);

    //Gesamtobjektladen
    $o = new PScorm_Aggregation2Project;
    $o->result = $row;
    $o->init();
    $cache->populate($o);

    // dem Ergebnis hinzufuegen
    $ret[] = $o;
  }
  return $ret;
}
```

Quelltext 8: PSCORM Test2: QueryTest2 Klasse

ANHANG

C

Implementierungsübersicht Demoapplikation PSCORM

Hier wird die Struktur der *PSCORM*-Demoapplikation vorgestellt um eine Übersicht zu erhalten. Die Quelltexte der Demoapplikation befinden sich auf Github:

`https://github.com/pscheit/Diplomarbeit`

Die wichtigen Auszüge dieser Quelltexte wurden schon in Anhang B aufgelistet.
In `htdocs/pscorm` sind alle Klassen für die Demoapplikation. Diese Klassenstruktur sieht genauso aus, wie ein Export von *PSCORM* in Zukunft aussehen soll. Es lassen sich keine komplexen Klassen, oder große Ordnerstrukturen finden.
Zusätzlich liegen in `htdocs/doctrine` und `htdocs/kohana` die Quelltexte für die Tests in Kapitel 9. In `htdocs/kohana` sind die Klassen und Libraries von Kohana direkt enthalten. In `htdocs/doctrine` befinden sich nur die Controller, die Sourcen befinden sich in `../inc/doctrine/-orm/Doctrine`. Die Abhängigkeiten von *PSCORM* zum Psc-Framework wurden nicht aufgelöst.

Alle Pfade die ich jetzt nenne sind relativ zu `htdocs/pscorm`.

inc.config.php

Für das übergeordnete Psc-Framework werden hier die Zugangsdaten für die Datenbank und die Datenbank angegeben. Das Schema der Datenbank befindet sich in `../../schema.sql`

class

Im Verzeichnis befinden sich alle Klassen der Demoapplikation. Alle Klassen haben den Prefix `PScorm_`, damit sie unterscheidbar bleiben. Diese sollten mit einem PHP-Autoloader dynamisch geladen werden. Fast alle Klassen leiten `Object` ab. Diese kann als Basisklasse des Psc-Frameworks verstanden werden.

DB.php

Die Klasse mit der native Anfragen an die konfigurierte Datenbank abgesetzt werden können. Leitet PDO ab, welches eine in die Pecl-Extension PDO (`http://www.php.`

`net/PDO`) eingebaute Klasse ist. Die funktion `query()` wurde dabei überschrieben, sodass alle SQL Befehle aus der Applikation in das Logfile gesichert werden können. Das Logfile kann dann zum Debug gebraucht werden (siehe `__toString()`).

Cache.php

Die Klasse für den Cache. Ist als Singleton implementiert und kann von überall mit `PScorm_Cache::getInstance()` geladen werden.

ORMObject.php

Die abstrakte Basisklasse für alle Entity-Klassen (Project, Aggregation2Project, Aggregation). Sie stellt die wichtige Funktion `PScorm_ORMObject::load`, die immer in der Applikation aufgerufen wird, wenn ein Objekt aus der Datenbank geladen werden soll.
Die Abstrakte Funktion `init()` muss von allen Entity-Klassen implementiert werden. Der Datensatz aus der Datenbank wird im Property `result` gespeichert.

Query.php

Die Basisklasse für alle Queries. Stellt `getAll()` zur Verfügung, welche z. B. in Query1 benutzt wird.

Query[0-9].php

Die Query-Klassen für die verschiedenen Tests. Leiten die Klasse Query ab.

Project.php Aggregation.php Aggregation2Project.php

Entity-Klassen für Project Aggregation und Aggregation2Project. Aggregation2Project benutzt die `load()`-Funktion von ORMObject um seine Unterobjekte zu initialisieren.

LITERATURVERZEICHNIS

[ABD+89] Malcolm Atkinson, François Bancilhon, David DeWitt, Klaus Dittrich, David Maier, and Stanley Zdonik. The object-oriented database system manifesto, 1989. 9

[Amb98] S W Ambler. The design of a robust persistence layer for relational databases. *AmbySoft Inc. White Paper*, 1998. 19, 49

[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. 22, 39, 65

[CI05] William R. Cook and Ali H. Ibrahim. Integrating programming languages and databases: What is the problem. In *ODBMS.ORG, Expert Article*, 2005. 8

[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. 40

[Dev01] Ramakanth Subrahmanya Devarakonda. Object-relational database systems — the road ahead. *Crossroads*, 7(3):15–18, 2001. 10

[Doc10a] Doctrine Project. *Doctrine Documentation*, 2010. 28

[Doc10b] Doctrine Project. *Doctrine: Object Relational Mapper*, 2010. 29

[IBM01] IBM. *DB2's object-relational highlights: User-defined structured types and object views in DB2*, 2001. 12

[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. 8, 9

[Int06] Intersystems. *Intersystems Plain Old Java Persistence With Caché (Jalapeño)*, 2006. 26

[Int10] Intersystems. *Intersystems Caché*, 2010. 12

[JBo10a] JBoss Community. *Hibernate*, 2010. 28, 29

[JBo10b] JBoss Community. *Hibernate: Relational Persistence for Java & .NET*, 2010. 28

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

[Koh10a] Kohana Team. *Kohana*, 2010. 29

[Koh10b] Kohana Team. *Kohana Documentation*, 2010. 28

[Lea00] Neal Leavitt. Whatever happened to object-oriented databases? *Computer*, 33:16–19, 2000. 10

[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. 18, 19, 52

[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. 12, 27

[Mic09] Microsoft. *What's New in SQL Server 2008*, 2009. 12

[New06] Ted Neward. The vietnam of computer science. In *ODBMS.ORG, News*, 2006. 4

[Ora08] Oracle. *Oracle Database Object-Relational Developer's Guide 11g Release 1 (11.1)*, 2008. 12

[Ora09] Oracle. *Oracle Fusion Middleware Developer's Guide for Oracle TopLink 11g Release 1 (11.1.1)*, 2009. 28, 32

[ORM09] ORMBattle.NET Team. *ORM Comparison and Benchmarks on ORMBattle.NET*, 2009. 28

[Pro10] The Propel project. *Propel: Smart, easy object persistence*, 2006-2010. 47

[Ric08] Chris Richardson. Orm in dynamic languages. *Queue*, 6:28–37, May 2008. 27, 47

[Rus08] Craig Russell. Bridging the object-relational divide. *Queue*, 6(3):18–28, 2008. 20

[SM95] Michael Stonebraker and Dorothy Moore. *Object Relational DBMSs: The Next Great Wave*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1995. 11