TECHNISCHE UNIVERSITÄT WIEN

VIENNA UNIVERSITY OF TECHNOLOGY

DIPLOMARBEIT

Techniken und Werkzeuge für objekt-relationale Abbildungen

ausgeführt am Institut für
Informationssysteme
(Database and Artificial Intelligence Group)
der Technischen Universität Wien

unter der Anleitung von
Prof. Dr. Reinhard Pichler

durch

David Schmitt
Matrikelnummer: 9725491
Schmidg 4/9
A-1080 Wien

Wien, am 6. Juli 2006 _____

Kurzzusammenfassung: Jede Applikation, die Daten in einer relationalen Datenbank speichert, benötigt eine Abbildung von ihren internen Strukturen auf die rigiden Strukturen der Datenbank. Während manchen Applikationen eine einfache Identitätsabbildung von Tabellen auf Wertlisten genügt, benötigen objekt-orientierte Programmiermodelle reichhaltigere Abbildungen, um die flexiblen Objekt-Strukturen ausnützen zu können.

Nach einer Einführung in die Grundlagen der relationalen und objekt-orientierten Modellierung wird anhand von zwei Open Source Projekten die automatisierte Unterstützung objekt-relationaler Abbildungen mit der manuellen Programmierung verglichen.

Den Abschluß bildet ein Vergleich dieser Werkzeuge gegenüber den klassischen Definitionen von objekt-orientierten Datenbanken.

Eidesstattliche Erklärung: Ich erkläre an Eides statt, daß ich die vorliegende Arbeit selbständig und ohne fremde Hilfe verfaßt, andere als die angegebenen Quellen nicht benützt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

2007-07-07, Jülich

Diese Diplomarbeit und der erfolgreiche Abschluß meines Studiums wären mir ohne die folgenden Personen nicht möglich gewesen:

Meine Eltern, die mich immer unterstützt haben wenn es notwendig war.

Professor Pichler, der mir das freie Schaffen an dieser Arbeit ließ und trotzdem in kritischen Momenten wichtige Hinweise für die Fertigstellung gab.

Meine Frau, die auch in schwierigen Zeiten für mich da ist.

Dafür möchte ich mich herzlich bedanken.
David Schmitt, Wien, 2007

Inhaltsverzeichnis

Kapitel 1

Einleitung

Spätestens seit der in Ton verewigten Buchhaltung der Sumerer erleichtern sich Menschen mit Datenaufzeichnungen das Leben. Um immer neuen Anforderungen zu genügen, wurden immer komplexere Verfahren entwickelt. Diese Entwicklung hat bis zum heutigen Tag mit objekt-orientiertem Design und relationalen Datenmodellen zwei orthogonale Methoden hervorgebracht.

- Objekt-orientierte Architekturen stellen die *Beziehungen* von Objekten in den Vordergrund. Diese *vernetzte Darstellung* ermöglicht eine realitätsnahe Modellierung von Sachverhalten, die die Abbildung von *komplexen Zusammenhängen* ermöglicht.
- Bei relationalen Datenmodellen steht die *Effizienz* der Datenverwaltung im Mittelpunkt. Die dafür gewählte Einschränkung auf *starre, tabellarische Darstellungen* ermöglicht die Verwaltung *großer Datenmengen*.

Um komplexe Zusammenhänge und Abläufe auf großen Datenmengen umzusetzen, werden aber Techniken aus beiden Bereichen benötigt. Mittels objekt-relationaler Abbildungen versucht man die besten Teile aus beiden Welten zu vereinen. Die dabei verwendeten Techniken und Werkzeuge sind zentrales Thema dieser Diplomarbeit.

Kapitel 2 beschreibt die gängigen relationalen und objekt-orientierten Grundlagen, dazu wird eine Beispieldatenbank vorgestellt. In Kapitel 3 werden die verschiedenen Möglichkeiten der objekt-relationalen Abbildungen gegenübergestellt. Kapitel 4 stellt die ausgewählten Methoden – reines SQL und Java – und die Produkte – SimpleORM und Hibernate – vor. An-

schliessend werden diese Methoden und Produkte anhand der Beispieldatenbank in Kapitel 5 gegenübergestellt. Kapitel 6 beurteilt die Implementierungen in Hinblick auf die klassischen Definitionen von objekt-orientierten Datenbanksystemen. Kapitel 7 fasst die gewonnenen Erkenntnisse und Erfahrungen noch einmal zusammen. Ein kurzer Ausblick auf die aktuellen Entwicklungen und die Einbettung in größere Projekte schließt diese Arbeit ab.

Kapitel 2

Grundlagen

Objekt-relationale Abbildungen sind immer in eine größere Anwendung eingebettet. Jenseits der grundlegenden Anforderungen an jede Anwendung erzeugen die unterschiedlichen Ansätze von Datenbanksystemen und objekt-orientieren Modellen immer wieder Reibungsverluste. Diese Diplomarbeit demonstriert das an einer Bild- und Artikeldatenbank mit Schlagworten und einer darauf aufbauenden Ordner-ähnlichen Struktur.

Zuerst jedoch die Grundlagen von Datenbank- und Objektmodellierung, um eine Basis für dieses Beispiel zu schaffen. Nach einer kurzen Einführung in die Beispieldatenbank der folgen dann weitere Erläuterungen zu grundlegenden Aspekten der Datenhaltung.

2.1 Datenbankmodellierung

Jarosch bringt das Problem der Modellierung in [13] (S. 5 f) auf den Punkt. Menschen denken in Bedeutungen, zum Beispiel Namen oder Uhrzeiten. Computer können jedoch nur Buchstaben und Zahlen verarbeiten. Die für den Menschen darin offensichtlichen Zusammenhänge bleiben der Maschine verschlossen.

Die Aufgabe für den Datenbankprogrammierer ist es nun, im entworfenen Schema die Bedeutungen in einer, dem Computer zugänglichen, formalen Sprache festzulegen. Durch diese Festlegung können dann unterschiedliche Anwendungen auf einen gemeinsamen Datenbestand zugreifen und so ohne Missverständnisse Daten austauschen. Diese Festlegung bewegt sich in einem Spannungsfeld zwischen der einfachen Darstellung und der vollständigen Erfas-

sung aller Details.

Jarosch empfiehlt, die Entwicklung von Datenbanken in zwei separate Schritte zu trennen. Zuerst soll ein *fachspezifisches Wissensmodell* von den Experten des Zielfaches erstellt werden. Danach kann dieses Wissensmodell in ein Datenbankschema übersetzt werden.

2.1.1 Fachspezifisches Wissensmodell

Für die Erstellung des Wissensmodelles führt Jarosch vier elementare Schritte an:

Klassifizierung: Welche verschiedenen Arten von Objekten gibt es überhaupt? In der Beispieldatenbank sind dies Artikel, Bilder, Schlagworte und Ordner. Andere mögliche Objekte wie *Fotograph* oder *Autor* werden nicht berücksichtigt, da es sich um eine persönliche Datenbank handeln soll.

Abstraktion: Welche Gruppierungen und Eigenschaften sind signifikant für die Modellierung? Eine der signifikanten Unterscheidungen in der Beispieldatenbank ist zwischen Schlagworten und Artikeln: Ein Artikel ist ein Text mit Titel und Zusammenfassung, während ein Schlagwort hingegen einen (oder mehrere) Artikel einem Thema zuordnet. Ein interessanter Grenzfall ist die Unterscheidung zwischen Artikeln und Bildern: während sich Text und Bild auf der einen Seite deutlich unterschieden, handelt es sich doch bei beiden um Artefakte mit Titel, Erzeugungsdatum und Schlagworten. Die Einführung eines Oberbegriffes, der diese Gemeinsamkeiten zusammenfasst, eröffnet einen Ansatzpunkt für spätere Erweiterungen – zum Beispiel, wenn auch Musikstücke verwaltet werden sollen.

Identifizierung: Welche Eigenschaften unterscheiden Objekte der gleichen Art voneinander? Schlagworte sind offensichtlich ihre eigene Identität. Bei Artefakten hingegen wird die Identifizierung schon schwieriger. Soll zum Beispiel der Titel immer eindeutig sein? Sogar wenn ja, wird sich der Titel nie ändern? Um die Beantwortung solcher Fragen zu umgehen wird oft ein *künstlicher Schlüssel* eingeführt, mit dem Objekte fortlaufend durchnummeriert werden. Da dieser Schlüssel keinerlei Bedeutung trägt, ist er automatisch eindeutig und dauerhaft.

Sachlogische Zusammenhänge: Wie hängen die einzelnen Objekte (Objektarten) zusammen?
Artefakte haben eine beliebige Anzahl von Schlagworten. Schlagworte können naturgemäß mehreren Artefakten zugeordnet werden. Ordner schränken mit einem zusätzlichen Schlagwort die Auswahl des übergeordneten Ordners ein.

Ausgehend von diesen fachspezifischen Erkenntnissen kann ein logische Datenbankschema abgeleitet werden. Das logische Datenbankschema ist eine formale Aufbereitung der Datenstruktur in der gewählten Datenbeschreibungssprache des Zieldatenbank. Typischerweise handelt es sich dabei um einen SQL Dialekt.

2.1.2 Datenbankschema

Die im Wissensmodell beschriebenen Objekte und Attribute können nun als Tabellen in der Datenbank angelegt werden. Dabei bilden die identifizierenden Attribute die jeweiligen, eindeutigen Schlüssel. Mit zusätzlichen Fremdschlüsseln - das sind Verweise auf einen Eintrag in einer Tabelle - können die sachlogischen Zusammenhänge modelliert werden. Jarosch beschreibt in [13] 20 "Transformationsregeln", wie Beziehungen mit verschiedensten Einschränkungen in SQL implementiert werden.

2.1.3 Normalformen

Für die Genauigkeit und Benutzbarkeit der Datenbank ist die realistische Abbildung der relevanten Objekte in das fachspezifische Wissensmodell entscheident. Für die effiziente und sichere Programmierung der Datenbank selbst ist die korrekte Transformation dieses Wissensmodells in ein logisches Datenbankschema erforderlich.

Neben der Abbildung der betrachteten Daten ist das erste Ziel eines Datenbankschemas die Vermeidung von Mehrfachspeicherungen. Solche Redundanzen brauchen nicht nur mehr Speicherplatz sondern verlangen auch eine erhöhte Vorsicht bei Schreibvorgängen. Wird ein mehrfach abgelegtes Datum nicht überall geändert, kommt es zu internen Widersprüchen der Datenbasis.

Um Redundanzen und die dadurch ermöglichten Inkonsistenzen zu vermeiden, wurden in der Datenbanktheorie die sogenannten Normalformen eingeführt. Ein Schema in erster Normalform enthält nur Relationen unteilbarer Werte. Zweite und dritte Normalform vermeiden

direkte bzw. transitive Redundanzen in Bezug auf Schlüsselwerte. Vierte und fünfte Normalform vermeiden zusätzlich Redundanzen innerhalb der Datenattribute. Dabei bauen die Normalformen aufeinander auf: jede stärkere Normalform verlangt alle darunterliegenden.

Unteilbare Werte

In Kents Zusammenfassung der Normalformen [14] wird die erste Normalform definiert als Forderung, dass alle Datensätze eines Typs *die gleiche Anzahl an Feldern haben müssen*. Diese Einschränkung ist tief in der Struktur relationaler Datenbanken verankert, die Listen, Mengen und Graphen nicht als elementare Datentypen sondern nur als Beziehungen von Tabellen zueinander kennen.

In der Programmierpraxis wird unter diesem Titel auch gefordert, dass keine zusammengesetzten Felder vorkommen. Eine passende Definition von "'zusammengesetzt"' muss jedoch von Projekt zu Projekt und von Feld zu Feld neu gefunden werden, da dies stark von der erwarteten Anwendung abhängt. Ein typisches Beispiel für solche Abwägungen ist die postalische Adresse. Je nach Anwendung kann ein einfacher mehrzeiliger Text ausreichen oder ein komplexer Datensatz erforderlich sein. Relevante Fragen, um dieses Spannungsfeld auszuloten, sind vor allem die Homogenität der Daten – so unterscheiden sich US-amerikanische Adressen im Inhalt und der Anordnung der Felder deutlich von europäischen – und den zu erwartenden Anwendungen und Abfragen.

Redundanzen im Bezug auf Schlüsselwerte

Ein Wertefeld in einem Datensatz muss *den Schlüssel, den ganzen Schlüssel und nichts anderes als den Schlüssel* beschreiben.

Eine Verletzung der zweiten Normalform beschreibt nur einen Teil des Schlüssels. Ein Beispiel aus [14]:

$$\text{Inventar}(Teilenummer, Lager, \text{Anzahl}, \text{Lageradresse})$$

Die Tabelle "Inventar" hat die Spalten "Teilenummer", "Lager", "Anzahl" und "Lageradresse". Die ersten beiden Spalten bilden den identifizierenden Schlüssel. Durch die mehrfache Angabe der Lageradresse, nämlich für jedes gelagerte Teil einmal, ist diese Relation überbestimmt. Bei Datenänderungen wirft dies eine Reihe von Problemen auf. Gibt es in einem Lager

verschiedene Teile wird die Lageradresse mehrmals gespeichert. Kommt es zu einer Adressänderung oder werden Teile von einem Lager in ein anderes transportiert, kann es bei unvollständigen Datenaktualisierungen zu internen Widersprüchen kommen, die innerhalb des Systems nicht mehr reparierbar sind. Besonders bei langen Teilelisten ist auch der benötigte Speicherplatz nicht zu vernachlässigen, der durch die oftmals wiederholte Speicherung der Lageradressen verschwendet wird.

Um solche Widersprüche zu vermeiden, modelliert man solche Daten in zwei unabhängigen Relationen:

$$\text{Inventar}(\underline{Teilenummer}, \underline{Lager}, \text{Anzahl})$$

$$\text{Lager}(\underline{Lager}, \text{Lageradresse})$$

Dadurch wird die Lageradresse nur noch einmal pro Lager gespeichert. Bei Änderungen kann es zu keinen Missverständnissen mehr kommen.

Die dritte Normalform behandelt ähnlich gelagerte Situationen, nämlich wenn es sich bei dem überbestimmten Feld nicht um einen Schlüssel handelt. Als Beispiel die Liste der Lagerarbeiter: geht man davon aus, dass jeder Arbeiter nur in einem Lager arbeiten kann, ist dieses nicht mehr Teil des Schlüssels:

$$\text{Arbeiter}(\underline{Arbeiter}, \text{Lager}, \text{Gehalt}, \text{Lageradresse})$$

Probleme und Lösung sind die selben wie bei der zweiten Normalform. Bei unvollständigen Datenaktualisierungen kommt es zu internen Widersprüchen und die Modellierung in zwei unabhängigen Relationen vermeidet diese.

Interne Redundanzen

Interne Redundanzen tauchen auf, wenn sich mehrere Datenfelder in einer Relation korrekt auf den Schlüssel beziehen, aber untereinander unabhängig sind. Kent bringt in [14] Sprachen und Talente als Beispiel. Beide sind als Attribut einer Person in dritter Normalform. Werden die beiden Attribute gemeinsam in einer Relation gespeichert, wird damit auch ein Bedeutungszusammenhang – zum Beispiel, dass gewisse Talente nur in gewissen Sprachen ausgeübt werden können – suggeriert. Existiert dieser Zusammenhang im sachlogischen Wissensmodell

gar nicht, verletzt die Tabelle die vierte Normalfrom. Auch hier ist die Lösung wieder eine Zerlegung in die grundlegenden Relationen.

Zu beachten ist jedoch, dass das entscheidende Kriterium hier rein in der Bedeutung der Daten liegt und nicht mehr formalisiert werden kann. Bezieht sich das Sprachfeld tatsächlich darauf, in welcher Sprache das Talent ausgeübt werden kann – zum Beispiel "schreiben", "rechnen" – dann ist die Darstellung in einer Relation durchaus sinnvoll und zulässig.

Eine andere Art interner Redundanz wird von der fünften Normalform verboten. Diese verlangt, dass ein Datensatz sich nicht in Datensätze mit einem Schlüssel mit weniger Feldern zerlegen lässt. Ein solcherart unzerlegbarer Datensatz mit minimalem Schlüssel ist automatisch in zweiter bis vierter Normalform. Eine Datenbank in vierter Normalform kann diese Bedingung nur durch externe Einschränkungen auf den Daten verletzen. Kent gibt als klassisches Beispiel die Dreiecksbeziehung zwischen Vertretern, Herstellern und Produkten an. Im allgemeinen Fall ist die Relation, die diese Beziehung beschreibt, in vierter und fünfter Normalform:

$$\text{verkauft}(Vertreter, Hersteller, Produkt)$$

Verlangt jedoch das sachlogische Modell, dass Vertreter Produkte aus ihrem Sortiment von *allen* von ihm vertretenen Herstellern anbieten, so kann dies durch drei kleinere Relationen beschrieben werden:

$$\text{verkauft}(Vertreter, Produkt)$$

$$\text{vertritt}(Vertreter, Hersteller)$$

$$\text{erzeugt}(Hersteller, Produkt)$$

Praktische Anwendung

Aufgrund der mittlerweile weitverbreiteten Erfahrung mit Datenbankschemata haben die Normalenformen als Handlungsanweisungen an Bedeutung verloren. Gerade in Verbindung mit modernen Programmiersprachen wird die Datenbank oft aus dem Objektschema abgeleitet. Dort sind die Attribute schon in jener Granularität aufbereitet, die für die Anwendung relevant ist (erste Normalform). Ebenso treten Redundanzen zu Schlüsselwerten kaum auf, da

solche Attribute normalerweise als eigenständige Klassen – und damit auch als eigenständige Relationen – realisiert werden (zweite und dritte Normalform).

Die weiteren Normalformen haben in der Praxis geringere Bedeutung. Die vierte Normalform befasst sich mit einem Sachverhalt, der in der Implementierung offensichtliche Probleme aufwirft. Solche Konstruktionen werden daher intuitiv vermieden und nur in besonderen Spezialfällen – zum Beispiel zur Leistungsoptimierung – eingesetzt. Die fünfte Normalform hingegen verlangt sehr strikte Voraussetzungen für den Zusammenhang von Daten, der dem heutigen Drang nach flexibleren Geschäftsmodellen zuwiderläuft.

Denormalisierung

Die Redundanzfreiheit und die damit verbundenen Vermeidung von Inkonsistenzen führen vor allem bei größeren Abfragen zu Flaschenhälsen, da Relationenverknüpfungen mehr Laufzeitresourcen benötigen als einfache Indexabfragen. Um diesem Effekt entgegen zu wirken, kann man in der Entwicklung des Datenbankschemas gezielt Redundanzen einführen, um bestimmte Abfragen zu beschleunigen. Als Beispiel in einer Abrechnungsdatenbank könnte die Gesamtsumme einer Rechnung als eigenes Attribut gespeichert werden, anstatt sie bei jeder Abfrage aus den Einzelposten neu zu berechnen.

2.2 Objektmodellierung

Analog wie bei der Modellierung von Datenbanken liegt jedem Objektmodell ein fachspezifisches Wissenmodell zugrunde. Zusätzlich zu den Überlegungen bezüglich der gespeicherten Daten im vorigen Abschnitt, können informationstechnologische Objekte jedoch auch Verhaltensmuster enthalten. Diese werden in einem ersten Schritt oft als Anwendungsfalldiagramme (siehe zum Beispiel [17]) modelliert. Abbildung 2.1 zeigt die grundlegenden Anwendungsfälle für die Beispieldatenbank.

Zum besseren Überblick führen Anwendungsfalldiagramme auch Benutzergruppen ein. Hier – auf der linken Seite dargestellt – werden die Rollen "Autor" und "Gast" modelliert. Durch die Verbindungslinien werden die erlaubten Anwendungsfälle dargestellt. Strichlierte Pfeile stehen für Anwendungsfallinklusion. Zum Beispiel muss für den Anwendungfall "Ordner betrachten" der Anwendungsfall "Laden" abgearbeitet werden.

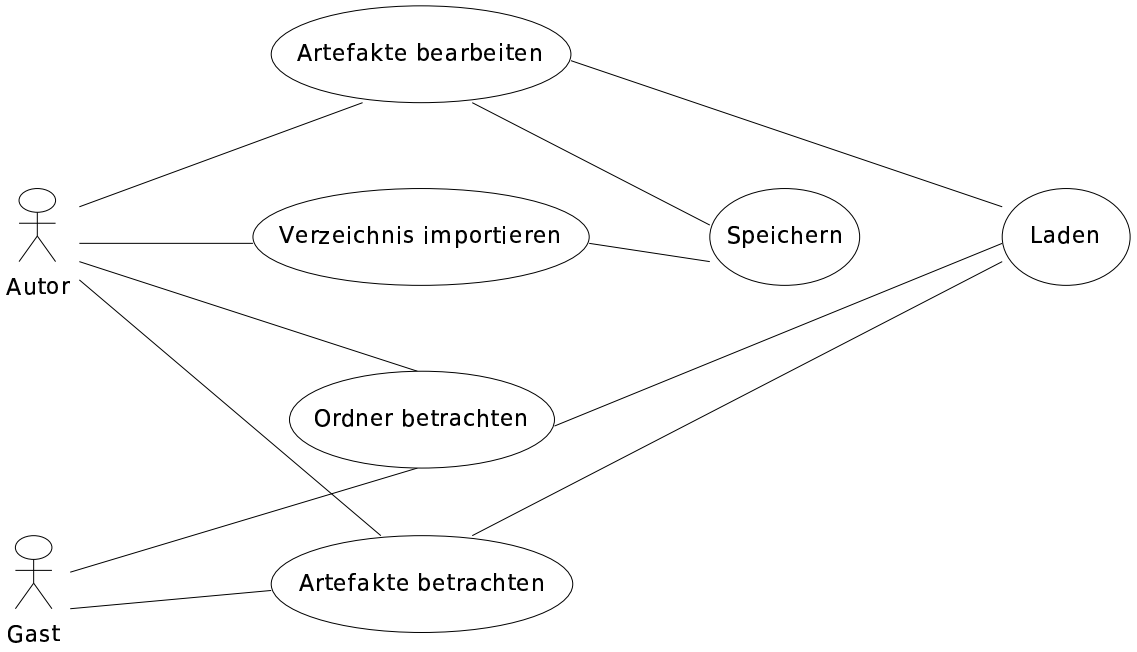


Abbildung 2.1: *webbook* Anwendungsfälle

Ebenso wie bei der Datenbankmodellierung können die Objekte und Attribute des fachspezifischen Wissensmodelles in die Objekte des Datenmodelles übernommen werden. Da Objekte an sich bereits unterscheidbar sind, benötigen sie keine besonderen Vorkehrungen für die Identifizierung. Diese Informationen sind erst später für allfällige Such- und Ablagesysteme von Bedeutung. Sachlogische Zusammenhänge werden daher auch ohne Umweg über Schlüsselwerte als Verweise auf andere Objekte modelliert.

2.3 Die *webbook* Beispieldatenbank

Um die in dieser Diplomarbeit behandelten Methoden und Werkzeuge vergleichbar zu machen, wurde eine kleine Bild- und Artikeldatenbank in Java implementiert. Ziel war es, eine überschaubare Schnittstelle unter Zuhilfenahme verschiedener Bibliotheken und Werkzeuge umzusetzen, um Erfahrungen mit den Werkzeugen zu sammeln und um daran die Effizienz und Effektivität beurteilen zu können.

Um die Funktionalität der Produkte zu demonstrieren implementiert die Beispielanwendung lediglich den Datenzugriff und keine Benutzeroberfläche. In einer umfassenden Archi-

tektur könnte aufbauend auf den hier implementierten Geschäftsobjekten die Geschäftslogik in einem Applikationsserver ablaufen. Alternativ kann die Benutzeroberfläche, zum Beispiel über einen Webserver, direkt mit dieser Bibliothek arbeiten.

Die grundlegenden Objekte der Datenbank sind Artikel und Bilder. Über Schlagworte können Gruppen gebildet werden. Diese Objekte und ihre Attribute werden im folgenden näher beschrieben.

Der Java-Quelltext ist in der Eclipse Entwicklungsumgebung nach verwendeten Werkzeugen in Projekte gegliedert; allen Implementierungen gemeinsam ist das *Common* Projekt, in dem gemeinsame Schnittstellendefinitionen im Paket *webbook* und die Modultests im Paket *webbook.tests* definiert sind. Abbildung 2.2 zeigt das *Common* Projekt und die darin enthaltenen Schnittstellen und Klassen.

2.3.1 Schnittstellen

Das *IDataObject* ist das Kernstück des gesamten Modells und Basisklasse für die anderen Objekte. Es enthält jene Attribute, die allen Artefakten gemein sind: eine eindeutige ID, Erzeugungs- und Modifikationsdatum, einen Titel und die Menge von Schlagworten, die diesem Objekt zugeordnet sind. Die Attribute sind als einfache *get-* und *set-*Zugriffsmethoden nach der Java Beans Konvention (Siehe [21], 8.3) implementiert:

```
public interface IDataObject {

    public abstract int getId();
    public abstract void setId(int id);

    public abstract Set<Tag> getTags();

    public abstract String getTitle();
    public abstract void setTitle(String title);

    /* ... */
```

Listing 2.1: IDataObject

IArticle und *IPicture* definieren die weiteren Attribute der Geschäftsobjekte: Zusammenfassung und Text für Artikel sowie Bilddaten, -unterschriften und Beschreibungstexte für Bilder.

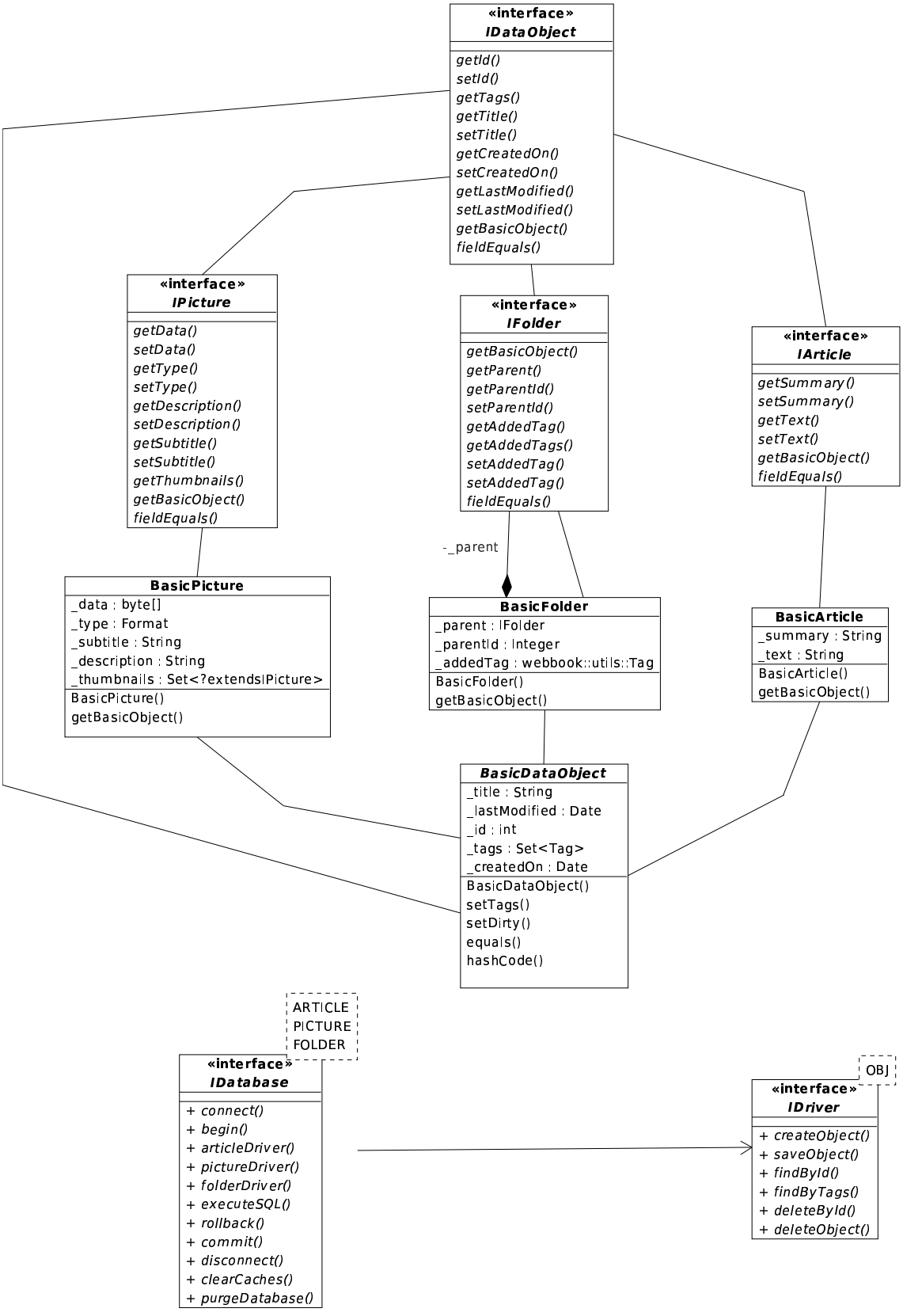


Abbildung 2.2: Überblick über die *webbook* Objekte

```
public interface IPicture extends IDataObject {

    public abstract byte[] getData();
    public abstract void setData(byte[] data);

    public abstract Set<? extends IPicture> getThumbnails();

    /* ... */
```

Listing 2.2: Datenobjekt am Beispiel von IPicture

Anmerkung: Eine Menge von Objekten kann in Java in einem *Set* gespeichert werden. Seit Java 1.5 gibt es mit der *Set<? **extends** IPicture>* Notation eine neue Syntax um "Menge von Objekten einer von *IPicture* abgeleiteten Klasse" zu definieren. Typparameter für Klassen werden später besonders bei den Tests verwendet.

Der *IFolder* ist eine Möglichkeit, aus der engen Folksonomie (siehe [3], 3.2.3) der Schlagworte Gruppierungen zu bilden, die die enthaltenen Strukturen abbilden. Sowohl die Mengenabfragen von Schlagwörtern, als auch die rekursiven Ordnerstrukturen sind von reinen relationalen Systemen nicht befriedigend gelöst worden und sind daher interessante Beispiele um die Grenzen von objekt-relationalen Abbildungen abzutasten.

```
public interface IFolder extends IDataObject {

    public abstract IFolder getParent();
    public abstract Tag getAddedTag();
    public abstract TagExpression getAddedTags();

    /* ... */
```

Listing 2.3: Attribute von IFolder

Die *IDatabase* Schnittstelle enthält alle Datenbank-relevanten Operationen für die Verbindungs- und Transaktionsverwaltung, sowie Methoden zum Laden und Speichern der Artefakte.

Alle Schnittstellen in diesem Projekt sind mit generischen Typparametern versehen. So haben alle konkreten Implementierungen die gleiche Aussenform und arbeiten doch mit ihren eigenen, konkreten Typen.

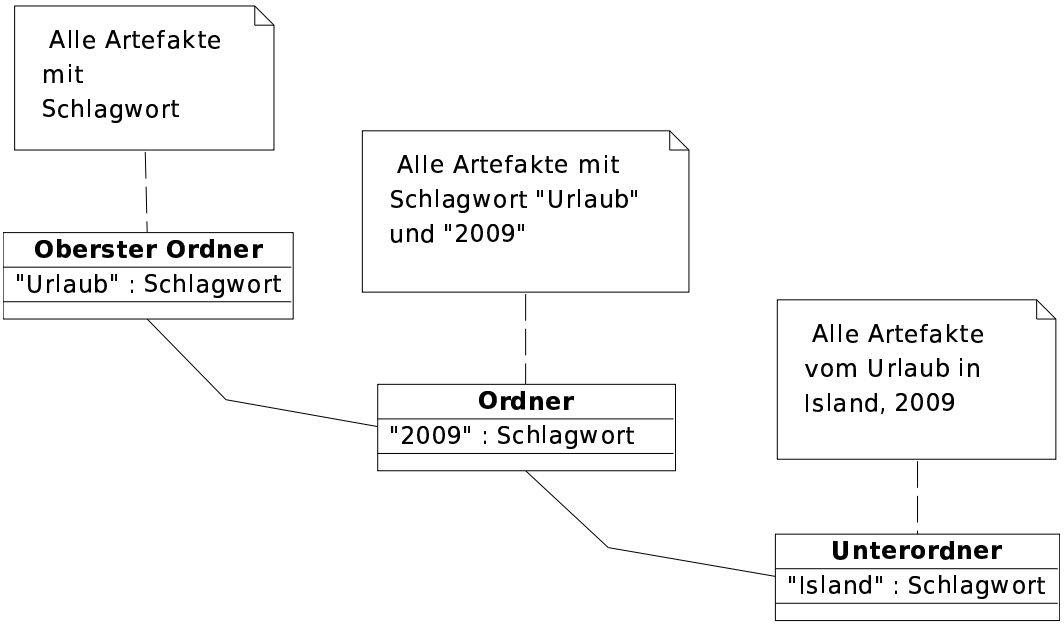


Abbildung 2.3: Folderstruktur

2.4 Datensicherheit

Um in einem Programmiersystem Daten sinnvoll verwalten zu können, muss das System zumindest die folgenden vier Anforderungen erfüllen:

Atomare Transaktion: Eine Transaktion heisst atomar, wenn alle Anweisungen aus denen die Transaktion besteht ganz oder gar nicht durchgeführt werden. Dazu wird zu Beginn der Ausführung ein Eintrag in eine Logdatei geschrieben. Alle Änderungen an der Datenbank versieht das DBMS mit einem Verweis auf den zugehörigen Eintrag in dieser Logdatei. Wurde die Anweisung erfolgreich abgeschlossen, wird dies in der Logdatei ebenfalls verzeichnet. Ist es notwendig den Zustand der Datenbank nach einem Absturz zu rekonstruieren, können anhand der Logdatei die offenen Transaktion aufgefunden und die potentiell fehlerhaften oder nur teilweise durchgeführten Anweisungen aufgeräumt werden.

Konsistenz: Um die Qualität des Datenbestandes jederzeit zu gewährleisten, muss ein DBMS dafür Sorge tragen, dass die angegebene Randbedingungen (Datentypen, Wertebereiche, gegenseitige Abhängigkeiten) für die Daten zu jedem Zeitpunkt eingehalten werden. Da-

zu werden diese Randbedingungen spätestens beim Abschluss einer Transaktion überprüft. Wird dabei eine potentielle Inkonsistenz entdeckt, bricht die Operation ab und die bisher getätigten Änderungen werden rückgängig gemacht.

Isolierung: Die Forderung nach Korrektheit im Mehrbenutzerbetrieb erweitert die Konsistenzanforderungen um einen wichtigen Aspekt: parallel ablaufende Anweisungen dürfen sich nicht gegenseitig stören. Das formale Kriterium hierfür – die Serialisierbarkeit der Abläufe – lautet, dass sich die Datenbank so verhält, als ob alle Anweisungen nacheinander statt gleichzeitig ausgeführt werden. Bei einzelnen Anweisungen ist das durch triviale Zeilen- oder Tabellensperren implementierbar. Isolierung über mehrere Anweisungen hinweg wird später im folgenden Abschnitt 2.5 Transaktionen noch detaillierter besprochen.

Dauerhaftigkeit: Meldet ein DBMS die erfolgreiche Durchführung einer Transaktion, so muss diese – im Rahmen der physikalischen Möglichkeiten – auch gesichert und dauerhaft gespeichert sein.

Die dafür oft verwendete Abkürzung lautet "ACID", aus dem Englischen von Atomicity, Consistency, Isolation und Durability – Atomare Operationen, Konsistenz, Isolierung und Dauerhaftigkeit.

2.5 Transaktionen

Um komplexere Operationen durchzuführen, kann man einzelne Basisoperationen in einer Transaktion zusammenfassen. Die Transaktion als Ganzes erfüllt dann wieder die ACID Anforderungen. Konsistenz und Dauerhaftigkeit folgen aus den entsprechenden Kriterien für Einzelanweisungen transitiv: Wenn jede einzelne Anweisung korrekt ist und die Ergebnisse dauerhaft gespeichert werden, dann müssen auch Aneinanderreihungen solcher Anweisung korrekt und dauerhaft sein.

Das Kriterium der Serialisierung – die virtuelle Nacheinanderausführung von Anweisungen – gilt selbstverständlich ebenfalls für Transaktionen. Während selbst bei komplexen Einzelanweisungen eine Isolierung zwischen gleichzeitig laufenden Anweisungen mittels Sperren einfach ist, führt ein unbedachter Einsatz von Sperren in länger laufenden Transaktionen zu

gravierenden Leistungsverlusten, da die meisten Benutzer darauf warten müssen, dass die gerade laufende, fremde Transaktion abgeschlossen wird.

Auf der Anwendungsseite kann dem durch die Vermeidung von lang laufenden Transaktionen entgegengewirkt werden.

Um Transaktionen atomar zu halten, müssen entweder alle oder keine der Anweisungen durchgeführt werden. Das klassische Beispiel dazu ist eine Überweisung von einem Konto auf ein anderes Konto. Um eine solche Überweisung durchzuführen, wird vom Ursprungskonto der Betrag abgezogen, während am Zielkonto der Betrag hinzuaddiert wird. Würde aufgrund einer Fehlersituation zum Beispiel nur der Betrag abgebucht aber nicht wieder gutgeschrieben, so verschwände der Betrag spurlos im Nichts.

Ein anderes Problem von Transaktionen ist die Möglichkeit zum sogenannten *Deadlock*. Dabei warten zwei Transaktionen jeweils auf die von der anderen Transaktion gesperrten Ressourcen. Da keine der beiden Transaktionen fortfahren kann bevor die andere nicht ihre Resourcen freigibt, bleiben sie – ohne Eingriff von aussen – ewig stecken. Die Methoden um solche Probleme zu entdecken und zu behandeln sind nicht Inhalt dieser Diplomarbeit.

2.6 Basisoperationen

Die Minimalanforderungen für die Datenverwaltung sind im Akronym "CRUD", vom englischen Create, Read, Update und Delete, zusammengefasst. Auf Deutsch: Datenobjekte müssen erzeugt, gelesen, verändert und gelöscht werden können.

In SQL werden die Basisoperationen mit diesen Befehlen durchgeführt:

−− Erzeugen einer Datenzeile
INSERT INTO tabelle (attr1, attr2) **VALUES** (wert1, wert2);

−− Abrufen
SELECT attr1, attr2 **FROM** tabelle **WHERE** bedingung;

−− Ändern
UPDATE tabelle **SET** attr1 = wert1 **WHERE** bedingung;

−− Löschen
DELETE FROM tabelle **WHERE** bedingung;

Listing 2.4: Grundlegende SQL Befehle

Dabei ist vor allem zu beachten, dass – bis auf *INSERT* – immer auf Mengen von Datenzeilen operiert wird, die die prädikatenlogische Formel *bedingung* erfüllen. Diese Mengen können natürlich auch kein oder nur ein Element enthalten.

In Java hingegen sind die Basisoperationen zwischen dem IDatabase Interface und den einzelnen Klassen aufgeteilt.

```
/* Erzeugen */
public abstract OBJ createObject();

/* Abrufen */
public abstract OBJ findById(int id) throws SQLException;
public abstract Set<? extends OBJ> findByTags(TagExpression tags) throws SQLException;

/* Abspeichern */
public abstract OBJ saveObject(OBJ o) throws SQLException;

/* Löschen */
public abstract OBJ deleteObject(OBJ o) throws SQLException;
```

Listing 2.5: IDatabase Operationen

Änderungen an Objekten werden direkt an den Objekten vorgenommen und über die IDatabase Schnittstelle als Ganzes gespeichert. Es bleibt dem Framework überlassen zu entscheiden wie die Daten gespeichert werden sollen. Im Gegensatz zu SQL sieht man sofort, dass die Abfragen wesentlich eingeschränkter sind, dafür aber auf einer wesentlich höheren Ebene formuliert werden.

2.7 Programmierung

Ein fundamentales Problem der Programmierung objekt-relationaler Anwendungen ist die Notwendigkeit, Programmtext und Datenbankschema synchron zu halten. Selbst im einfachsten Fall existieren fünf Listen aller Attribute eines Objektes: bei der Datenbank- und Objektdefinition sowie in den Datenbankanweisungen für Create, Read und Update, da hier ja immer Attributswerte von und zur Datenbank kopiert werden müssen. Bei diesen Kopiervorgängen

bringen objekt-relationale Abbildungen eine erhöhte Sicherheit, da sie durch die Werkzeuge weitestgehend automatisiert werden.

2.7.1 Datentypen

Mit geringerem Aufwand aber höherer Komplexität ist die Konversion zwischen programmiersprachlichen Datentypen und jenen der Datenbank zu beachten.

Zahlentypen: Da sich Wertebereiche und Genauigkeit oft an den Möglichkeiten der zugrunde liegenden Hardware und damit an weitverbreiteten Standards (zum Beispiel [9] für Gleitkommazahlen) orientieren, kann bei diesen Datentypen oft eine eindeutige Zuordnung getroffen werden.

Zeichenketten: Während bei Programmiersprachen Zeichenketten unbeschränkt sind, enthalten Datenbanken oft starke Optimierungen für längenbeschränkte Zeichenketten. Das Objektmodell muss die Einschränkungen des Datenbankschemas erzwingen, um Laufzeit- und Konsistenzfehler zu vermeiden. Verschärft werden diese Probleme noch durch eventuell erforderliche Zeichensatzkonversionen zwischen verschiedenen Anwendungsteilen und Zeichenkodierungen wie UTF-8, die einen variablem Platzbedarf bei konstanter Zeichenanzahl haben.

Zeit- und Datumsangaben: Wertebereich, Genauigkeit, Epoche, interne Repräsentation (Fixkomma oder Gleitkomma), externe Repräsentation (US-amerikanisch, europäisch, international), Zeitzonen, Kalender, Schalttage, -minuten und -sekunden, unterschiedliche Zeitquellen, Fehlerbehandlung. Subtile und nicht so subtile Unterschiede in diesen Parametern führen – zusammen mit komplexen und daher mangelhaft implementierten Standards – unweigerlich zu Problemen in der Kommunikation zwischen verschiedenen Produkten. Einige Probleme können durch eine gemeinsame, anwendungsweite, externe Repräsentation, die ohne Genauigkeitsverlust in alle beteiligten internen Repräsentation ein-eindeutig umgewandelt werden kann, behoben werden. Dadurch können Werte – die Korrektheit der Umwandlungen vorausgesetzt – beliebig oft umgewandelt werden ohne die gespeicherten Zeitpunkte zu verwischen.

Andere Probleme – zum Beispiel wie sich Systeme in den Umschaltstunden zum Sommerzeitwechsel zu verhalten haben – können nur durch eine präzise Definition des gewünschten Verhaltens umgangen werden.

Boolsche Werte: Die Unterstüzung von Bit-Werten schwankt zwischen verschiedenen Datenbanksystemen stark. Eine portable Alternative bildet Wahrheitswerte auf die Zeichen 'w' und 'f' – für wahr und falsch – ab und sorgt mittels Konsistenzbedingungen für die Einhaltung dieser Konvention.

Um umfangreichere Datenmengen – zum Beispiel Bilddateien oder unbeschränkte Texte – direkt in der Datenbank zu verwalten werden sogenannte BLOBs oder CLOBs[1] zur Verfügung gestellt. Um das Laufzeit-, Speicher- und Kommunikationsverhalten dieser Daten im Griff zu behalten, werden dafür meist zeichenstromorientierte Schnittstellen zur Verfügung gestellt, die wiederum seperat bedient werden müssen. Unterstützung für diese Datentypen ist weitverbreitet.

Neben diesen Standardtypen gilt es, für eine Vielzahl von komplexeren Datentypen effiziente Abbildungen und Konversionen zu finden. Um nur einige Beispiele zu nennen: Netzwerkadressen oder geometrische Daten für Geoinformationssysteme. Da es sich dabei um kaum oder gar nicht standardisierte Erweiterungen handelt, ist auch selten Unterstützung in Programmierwerkzeugen dafür vorhanden.

2.7.2 Abfragen

Abfragen auf einer Datenbank können um Größenordnungen schneller sein als gleichwertige Abfragen auf dem Objektmodell. Drei grundlegende Ursachen können dabei festgestellt werden:

Optimierung: Die Abfrageoptimierung ist ein intensiv erforschtes Gebiet der Datenbankprogrammierung. Sowohl auf der Seite der Programmierer als auch auf Seite der Datenbanken selbst existieren Rezepte um die bekannten Flaschenhälse zu umgehen.

Kommunikations- und Transformationsaufwand: Wird die Datenbank direkt abgefragt, so muß nur das gewünschte Abfrageergebnis zum Benutzer übertragen werden. Wird die

[1] Binary Large OBjects und Character Large OBjects, für große Binär- oder Textmengen.

gleiche Abfrage jedoch über ein zwischengeschaltetes Objektmodell moderiert, müssen potentiell alle Daten aller beteiligten Objete übertragen und interpretiert werden, auch wenn diese für die Berechnung irrelevant sind.

Abfragekomplexität: Abfragen die im Objektmodell sehr einfach zu formulieren sind, können bei der Ausführung auf der Datenbank nur schwer umsetzbar sein. Speziell polymorphe Abfragen und Abfragen nach komplexen berechneten Werden – zum Beispiel "Kreditwürdigkeit" – werden in objekt-orientierten Systemen direkt unterstützt, während sie in relationalen Systemen aufwändig nachprogrammiert werden müssen.

2.7.3 Portabilität

Um die Abhängigkeit von einem Datenbankhersteller zu minimieren ist die Portabilität zwischen unterschiedlichen Herstellern auch eine wichtige Anforderung an eine objekt-relationale Abbildung. Um diese Unabhängigkeit zu erreichen, werden die programmierten Abfragen in datenbankspezifische Anweisungen übersetzt.

2.8 Kommunikation

Sobald eine Anwendung aus mehr als einem (System-)Prozess besteht, wird die Kommunikation zwischen diesen Prozessen zu einem Kernproblem der gesamten Anwendung.

Zur Erhöhung des Transaktionsdurchsatzes werden mehrere, gleichzeitig arbeitende Systemprozesse eingesetzt. Um dabei die Korrektheit der Daten zu garantieren, müssen dabei auch die Anforderungen von CRUD und ACID über alle Prozesse hinweg erfüllt werden. Da diese bereits von allen üblichen Datenbankimplementierungen abgedeckt werden, bietet sich ein zentraler Datenbankserver an, als Angelpunkt für eine Mehrprozessarchitektur zu dienen.

Zur Einbettung in Dritt-Anwendungen müssen Kommunikationsschnittstellen nach außen zur Verfügung gestellt werden. Eine interessante Entwicklung in diesem Bereich ist die serviceorientierte Architektur (SOA). Hier werden Daten und Funktionalität über herstellerunabhängige Protokolle und Formate (zum Beispiel HTTP, SOAP und XML) im Netzwerk angeboten. Eine detaillierte Behandlung würde jedoch den Rahmen dieser Arbeit bei Weitem sprengen.

Kapitel 3

Konzepte

In der händischen Modellierung wird, ausgehend von dem fachspezifischen Wissensmodell, die technischen Modelle für Relationen und Objekte abgeleitet. Dabei müssen die beiden Modelle bezüglich Attributen und Wertebereichen aufeinander abgestimmt sein. Abbildung 3.1 illustriert diesen Vorgang.

Bei dieser getrennten Behandlung der beiden technischen Modelle kommt es jedoch zu Redundanzen zwischen diesen Modellen, die – besonders bei Änderungen am fachspezifischen Wissensmodell – zu unangenehmen Fehlerquellen werden. Hier setzt eine Implementierung mittels objekt-relationaler Abbildung an, um die technische Modellierung zu konzentrieren. Abbildung 3.2 zeigt diese Alternative. Da durch die Abbildung das Relationenmodell aus dem Objektmodell erzeugt werden kann, bleibt auch bei Änderungen die interne Konsistenz erhalten.

Für die Aufteilung der einzelnen Komponenten des Abbildungsapparates gibt es verschiedene Möglichkeiten. Je nach Anforderungen des Projektes und der Strukturierung der Entwicklung kann die Datenzugriffsschicht entweder direkt in die Datenklassen *integriert* oder für eine reduzierte Koppelung *getrennt* entwickelt werden. Zur Steigerung der Laufzeiteffizienz kann die Datenschicht während der Entwicklung erzeugt und *kompiliert* werden. Alternativ dazu kann eine erhöhte Flexibilität gewonnen werden, indem die Datenschicht zur Laufzeit *dynamisch konfiguriert* wird.

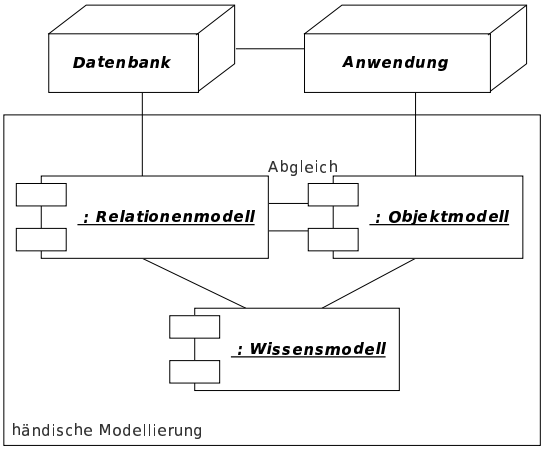


Abbildung 3.1: Klassische Architektur

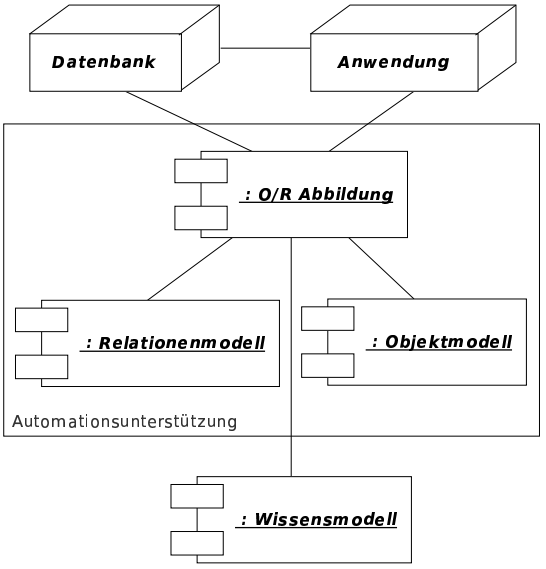


Abbildung 3.2: Architektur mit objekt-relationaler Abbildung

3.1 Funktionsumfang

Das Ideal des praktischen Programmierers in Sachen objekt-orientierter Persistenz wäre ein System mit der Einfachheit einer Datenbank und der Ausdruckskraft einer modernen Programmiersprache. Ein solches System müsste atomare Transaktionen und Isolation zwischen Prozessen und Benutzern bieten, sowie den vollen Funktionsumfang einer Programmiersprache. Außerdem braucht so ein System – unabhängig von der Programmierung des Systems – Möglichkeiten, um Änderungen am physikalischen Schema zur Leistungsoptimierung durchzuführen.

Die folgenden Abschnitte besprechen Möglichkeiten, wie mit heute verfügbaren Mitteln solche Systeme gestaltet werden können.

3.2 Integrierte Datenschicht

Mit einer *integrierten Datenschicht* wird der Datenbankzugriff fest in die Geschäftsobjekte eingebunden. Persistenzfunktionalität wird damit direkt in die Objekthierarchie integriert. Diese Vorgehensweise maximiert die Kohärenz zwischen Quelltext und Schema, da das Schema un-

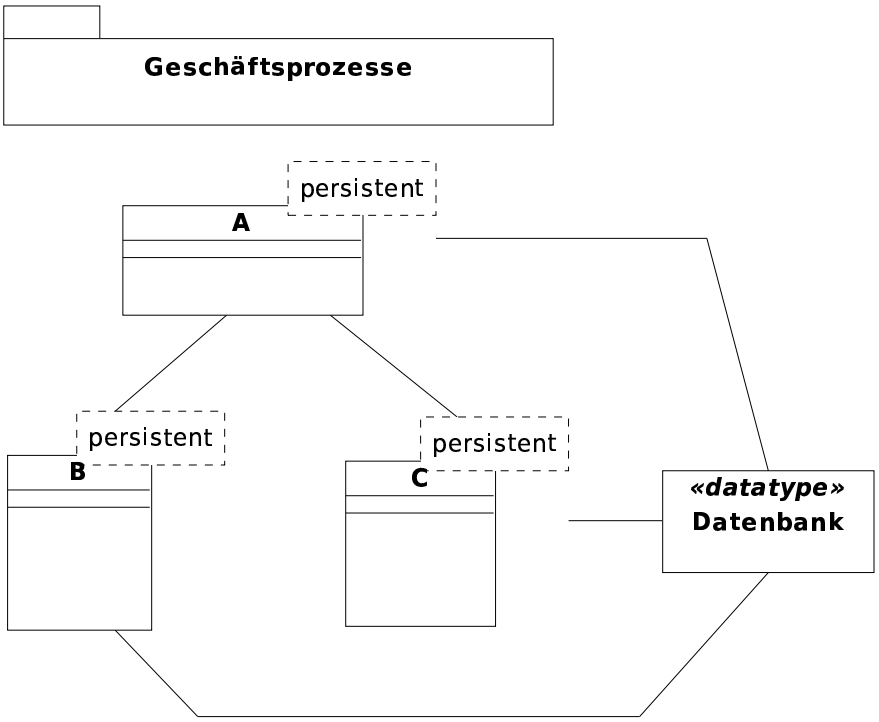


Abbildung 3.3: Integrierte Datenschicht

mittelbar aus den im Quelltext definierten Strukturen generiert wird. Abbildung 3.3 illustriert diese Architektur.

Aufgrund der hohen Koppelung zwischen Anwendung und Datenschicht kann auf generalisierende Schnittstellen verzichtet werden. Die Anwendung kann daher die Besonderheiten der verwendeten Datenschicht effizient nutzen. Wird die Datenschicht im gleichen Projekt implementiert, kann auch besonders auf die Anforderungen der Anwendung eingegangen werden. Durch die parallele Entwicklung bedarf es – jenseits der eigentlichen Programmierzeit – keiner weiteren Einarbeitung. Bei sehr kleinen Projekten kann das eine signifikante Ersparnis sein.

Auf der anderen Seite schlägt sich – ebenfalls aufgrund der hohen Koppelung – jede Änderung an der Datenschicht unmittelbar auf alle betroffenen Teile der Anwendung durch. Insbesondere ist ein Ersetzen der Datenschicht durch eine andere Implementierung mit hohem Aufwand verbunden.

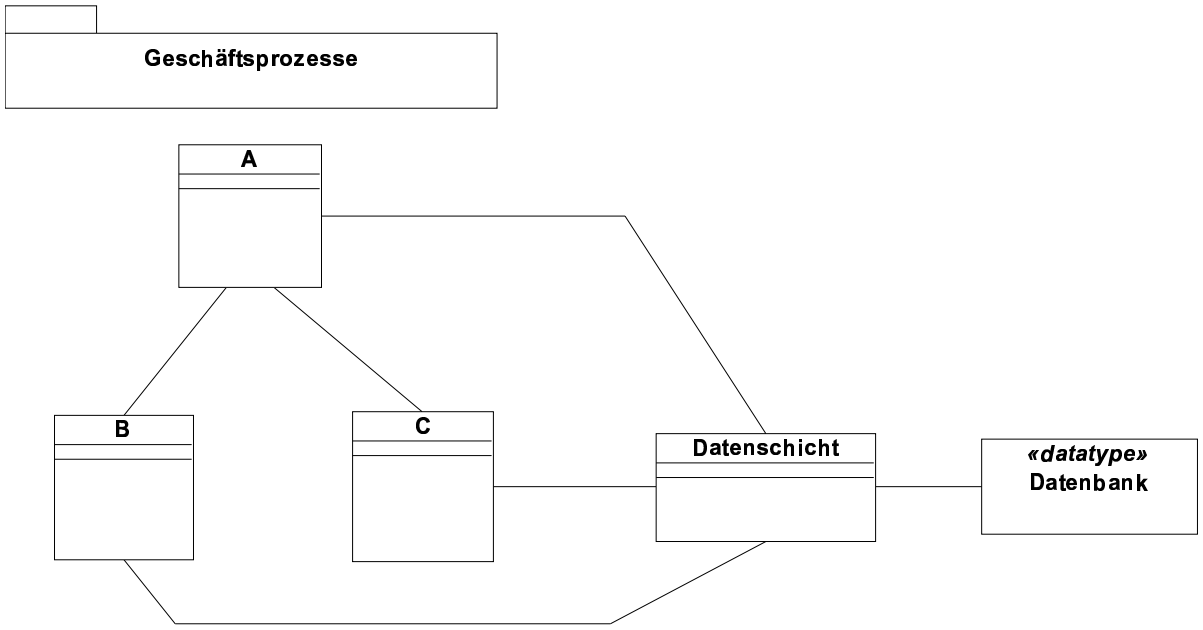


Abbildung 3.4: Getrennte Datenschicht

3.3 Getrennte Datenschicht

Je größer Anwendungen werden, desto stärker wird auch der Druck, einzelne Komponenten an Spezialisten (innerhalb wie außerhalb des Projektes) auszulagern. Da gerade Datenbankagenden bereits zu einem hohen Maß von Spezialisten bedient werden, ist die Datenschicht ein guter Kandidat für eine architekturelle Teilung. Abbildung 3.4 illustriert die Anordnung der Komponenten. Besonders zu beachten ist, dass die Geschäftsprozesse nun nichts mehr von der internen Mechanik der Datenschicht wissen müssen.

Aufgrund dieser reduzierten Koppelung kann die Datenschicht auch ohne Änderung der Geschäftsprozesse getauscht werden. Lediglich jene Teile der Geschäftslogik, die Daten abfragen, sind nun von solchen Änderungen betroffen. Durch eine passende *Fassadenschnittstelle* zur Datenschicht kann auch die Geschäftslogik von Änderungen dort geschützt werden. Diese Fassade bietet auch eine Plattform, um aus den daten-nahen Aufrufen der Datenschicht geschäfts-orientierte Methoden zu bauen.

3.4 Implementierung der Datenschicht

Unabhängig von der Beziehung der Datenschicht zu den Geschäftsobjekten, wird die Art der Implementierung der Datenschicht gewählt. Grundsätzlich besteht hier kein Zusammenhang. In der Regel werden jedoch integrierte Datenschichten eher statisch implementiert, während externe Datenschichten dynamische Methoden bevorzugen, da diese eine weitere Reduzierung der Koppelung versprechen.

3.4.1 Statische Implementierung

Vor allem bei der händischen Implementierung einer Datenschicht in kleineren Projekten kann sie *direkt* in den Klassen der Geschäftsobjekte implementiert werden. Dies stellt auch die stärkste Form der oben besprochenen integrierten Datenschicht dar: die Geschäftslogik manipuliert die Datenbank unmittelbar.

Eine andere Möglichkeit der statischen Implementierung der Datenschicht ist die *automatisierte Quelltexterzeugung*. Aus einer externen Beschreibung des Schemas werden Basisklassen erzeugt, die die reine Datenbankmanipulation implementieren. Um die Geschäftslogik unabhängig vom Datenschema und dem Erzeugungsprozess zu halten, wird sie in abgeleiteten Klassen implementiert. Dadurch erhält sie Zugriff auf die erzeugte Datenschnittstelle ohne mit dem Datenbankquelltext vermischt zu werden.

3.4.2 Dynamische Methoden

Die einfachste Möglichkeit einer dynamischen Datenschicht, ist die Implementierung eines *intelligenten Datensatzes* (engl.: "rich record"). Dabei wird nur das Schema statisch definiert, also Klassen, Attributsnamen und -typen. Während der Ausführung des Programmes werden aus diesen Informationen die Datenbankanweisungen erzeugt. Gegenüber der statischen Erzeugung der Datenschicht, wird bei diesem Verfahren weniger Bytecode[1] erzeugt, dafür leidet die Ausführungsgeschwindigkeit, da mehr Aufwand für die Erzeugung von Datenbankanweisungen und die Schemamanipulation notwendig ist.

Eine Stufe darüber befinden sich dynamische Methoden, die aus einer externen Schemadefinition zur Laufzeit die Geschäftsobjekte über *Reflection* manipulieren. Reflection ist eine

[1] Java-"Maschinensprache"

Methode um in Java die Struktur von Klassen erst zur Laufzeit zu erkunden. Damit können dann zum Beispiel Methoden aufgerufen werden, deren Existenz zur Übersetzungszeit gar nicht bekannt war. Die eigentliche Struktur der Datenschicht wird zur Laufzeit aus einer Konfigurationsdatei gelesen.

Ein wesentlich komplexere Möglichkeit ist die Manipulation des Bytecodes der Geschäftsobjekte nach der Übersetzung des Quelltextes oder zur Laufzeit. Dabei werden von einem speziellen Übersetzer oder einer Laufzeitkomponente der Datenschicht versteckte, abgeleitete Klassen der Geschäftsobjekte erzeugt, die die eigentliche Persistenz implementieren. Dieser Ansatz verbindet die Flexibilität einer zur Laufzeit konfigurierbaren Datenschicht mit der Effizienz einer statischen Implementierung. Auf der negativen Seite muss angemerkt werden, dass die Implementierung solcher Werkzeuge technisch aufwändig ist. Zusätzlich verursachen die Modifikationen am Bytecode eine Verschleierung der tatsächlich ablaufenden Vorgänge in der virtuellen Maschine, sodass es vor allem beim Debuggen schwer nachvollziehbar ist, was tatsächlich geschieht.

3.5 Objektidentität

Ein wesentlicher Punkt, in dem sich Objektmodelle von Relationen unterscheiden, ist die Feststellung der Identität. Während Relationentupel nur anhand ihres Primärschlüssels identifiziert werden, haben Instanzen eines Objektmodelles eine inhärente Identität, die sie von anderen Instanzen mit gleichen Werten unterscheidet. Um die striktere relationale Interpretierung in einer objekt-orientierten Sprache umzusetzen, muss die objekt-relationale Abbildung beim mehrfachen Laden eines relationalen Datensatzes diesen immer auf die *selbe* Objektinstanz abbilden.

Mit einem Objektpufferspeicher, in dem alle aktiven Instanzen referenziert werden, kann die Datenschicht die relationale Objektidentität wahren, indem bei wiederholten Anfragen zu dem gleichen Tupel auch die selbe Objektinstanz zurückgegeben wird. In der Datenbank erfordert das zumindest *REPEATABLE* ***READ*** Isolierung[2], damit diese Pufferung keine Inkonsistenzen verursachen kann. Daraus folgt auch unmittelbar, dass die Lebenszeit der Instanzen von der zugrundeliegenden Datenbanktransaktion abhängt. Wird eine neue Transaktion eröff-

[2]Details dazu in Abschnitte 5.7.1

net, können alle bereits geladenen Daten durch fremde Transaktionen modifiziert worden sein und müssen daher zumindest validiert werden.

3.6 Sperrmechanismen

Im Zusammenspiel von Datenschicht und Datenbank müssen auch die Sperrmechanismen den Gegebenheiten angepasst werden. Welche Möglichkeiten nutzbar sind, hängt dabei von der erwarteten Abarbeitungsdauer der Geschäftsfälle und der Möglichkeit, innerhalb der Anwendung die Datenbankverbindung aufrecht zu erhalten, ab.

3.6.1 Konservative Isolierung

Erlaubt die Systemarchitektur, jeden Geschäftsfall innerhalb einer eigenen Datenbanktransaktion durchzuführen, so ist dies die einfachste Art, parallel ablaufende Geschäftsfälle zu isolieren. Dafür muss jedoch die Datenbankverbindung während des gesamten Geschäftsfall bestehen bleiben und von diesem exklusiv genutzt werden. Geschäftsfälle, die auf Benutzereingaben warten, müssen dabei datenbankseitig sorgfältig gestaltet werden, um sich nicht gegenseitig durch zu großflächige Sperren zu behindern. Kommt es doch zu einem Konflikt, blockiert die Datenbank die Transaktion, bis die sperrende Transaktion beendet wird. Die dabei auftretenden Verzögerungen sind besonders für Benutzerschnittstellen nicht akzeptabel.

Abhängig von der Datenbankimplementierung können Transaktionen auch statt zu blokkieren abbrechen. Während automatisch ablaufende Geschäftsfälle dies meistens durch einfaches Wiederholen der Transaktion beheben können, ist das für Geschäftsfälle mit Benutzerinteraktion nicht in benutzerfreundlicher Art und Weise möglich.

3.6.2 Optimistische Sperren

Um die Probleme lange laufender Transaktionen zu umgehen, kann auch ein alternatives Sperrprotokoll innerhalb der Datenschicht implementiert werden. Die Isolierung von Datenbanktransaktionen wird dann nur noch zur konsistenten Kommunikation mit dem Datenspeicher genutzt.

Eine einfache und effiziente optimistische Sperre kann mit einem Zeitstempelattribut im-

plementiert werden. In diesem vermerkt man den Zeitpunkt der letzten Änderung. Ein Geschäftsfall kann dann beim Zurückschreiben überprüfen, ob sich der Zeitstempel seit dem Auslesen des Objektes geändert hat. In SQL kann das innerhalb der ***UPDATE*** Anweisung verglichen werden:

```
UPDATE tabelle
SET attribut=wert, ...
WHERE primärschlüsselbedingung
    AND letzte Änderung = gemerkter Wert
```

Wird durch diese Anweisung *kein* Tupel geändert, hat sich der Zeitstempel seit dem Lesevorgang geändert. Die Anwendung hat nun die Möglichkeit, die Änderungen festzustellen und dann eine intelligente Entscheidung über die weitere Vorgehensweise – zum Beispiel durch Anzeigen und Rückfragen beim Benutzer – zu treffen.

Wenn die Datenschicht eine Instanz nach dem Zurückschreiben weiterverwenden will, generiert sie den Zeitstempel zuerst selbst und schreibt ihn zusammen mit der Instanz in die Datenbank. Damit bleiben Objekt und Datenbank konsistent. Der Nachteil dieser Methode ist jedoch, dass jede Anwendung, die diese Datenbank benutzt auch den Zeitstempel korrekt verwalten muss. Besonders in heterogenen Umgebungen ist daher die Generierung des Zeitstempels mittels Triggermethoden in der Datenbank zu bevorzugen. Damit kann keine Anwendung einen Datensatz mehr verändern, ohne konformen Anwendungen diese Änderung zu signalisieren.

Mit etwas mehr Aufwand kann auch das Zeitstempelattribut vermieden werden. Dafür muss eine komplette Kopie der gelesenen Daten gespeichert werden. Beim Zurückschreiben wird dann nicht ein eigenes Attribut sondern alle gelesenen Attribute überprüft. Diese Methode hat gegenüber dem Zeitstempelattribut einen offensichtlichen Speicher-, Kommunikations- und Laufzeitmehraufwand. Dafür benötigt dieses Sperrprotokoll kein zusätzliches Attribut – besonders bei unmodifizierbaren Fremdschemata notwendig – und ist gegenüber unkooperativen Anwendungen robuster, da es unter keinen Umständen unbemerkt fremde Änderungen überschreibt.

3.6.3 Präventive Konfliktvermeidung

Um Schreibkonflikte überhaupt vollständig zu vermeiden, kann zu Beginn eines Bearbeitungsvorganges das Objekt als "in Arbeit" markiert werden. Andere Benutzer, die darauf zugreifen wollen, können nun erkennen, dass eine Bearbeitung zu einem Konflikt führen würde. Um erkennen zu können, ob so eine Markierung noch aktuell ist, oder ob der Bearbeitungsvorgang unerwartet unterbrochen worden ist, empfiehlt es sich, die Markierung mit einem Zeitstempel zu versehen, bis wann sie aufrecht erhalten werden soll. Ist diese Zeit abgelaufen, kann der nächste Bearbeiter die Markierung stillschweigend ignorieren. Braucht ein Bearbeitungsvorgang länger, muß die Markierung automatisch vor Ablauf verlängert werden.

Kapitel 4

Produkte

Im Rahmen dieser Diplomarbeit wurden einige Werkzeuge und Methoden getestet. Die gesamten Tests wurden in Java in der Eclipse Entwicklungsumgebung programmiert. Im diesem Kapitel werden die Komponenten und Bibliotheken vorgestellt.

4.1 SQL

SQL ist eine Abfrage- und Manipulationssprache für relationale Datenbanken. Diese Sprache wird von ISO in [10] standardisiert. Diesen Standard setzen die meisten Hersteller zu großen Teilen um. 1999 wurde eine neue Version des Standards [11] mit objekt-orientierten Merkmalen herausgebracht. Implementierungen dazu sind jedoch nicht weit verbreitet und vor allem im Hinblick auf die Objektorientierung unvollständig.

Als Abfragesprache genügt SQL nicht mehr modernen Anforderungen an Endnutzersoftware. Als Basis aller anderen Produkte bildet die Darstellung von Lösungsansätzen in SQL jedoch einen interessanten Hintergrund gegen den die anderen Methoden kontrastieren.

4.2 Reines Java

Diese objekt-orientierte Programmiersprache wurde von Sun Microsystems erfunden und wird zur Zeit im Rahmen des *Java Community Process* auf `http://jcp.org` weiterentwickelt. Für die meisten Datenbanken existieren Java Bibliotheken nach dem JDBC Standard (siehe [4]), die einen direkten SQL-Zugriff auf das DBMS über ein einheitliches Programmiermodell er-

lauben. Geschäftsobjekte und deren Verknüpfung mit der Datenbank müssen jedoch manuell implementiert werden.

4.3 SimpleORM

Berglas beschreibt in [6] diese Bibliothek als *Persistenzautomatisierung*. Der Schwerpunkt liegt also darauf, die repetetiven Aufgaben automatisch durchzuführen. SimpleORM ist jedoch kein Versuch, Persistenz transparent zu machen, da "für die meisten Informationssysteme Persistenz und Abfragen an die Datenbank alles durchdringende Aufgaben sind, die üblicherweise die 'Geschäftslogik' überschatten."[1] Die Bibliothek integriert daher die Datenschicht direkt in die Geschäftsobjekte und bietet nur die Kernfunktionalitäten einer objekt-relationalen Abbildung: Schemakonvergenz, typsichere Abfragekonstruktoren, einen Objektpuffer und Beibehaltung der Datenbankidentität.

SimpleORM benötigt keine zusätzlichen Übersetzungsschritte für Bytecode-Manipulationen und keine Laufzeitkonfiguration mit Reflection. Dadurch bleibt die Implementierung für den Anwender überschaubar, und bei der Fehlersuche "gilt" der unmodifizierte Quelltext der betrachteten Komponente.

4.4 Hibernate

Das erklärte Ziel der Entwickler von Hibernate ist es, dem Benutzer "95% der häufigsten persistenz-spezifischen Programmiertätigkeiten"[2] abzunehmen.

Hibernate positioniert sich als externe Datenschicht, die zwischen beliebig strukturierten Geschäftsobjekten und einer unabhängigen Datenbank übersetzt. In einer XML-Datei wird die gewünschte Abbildung zwischendem Daten- und dem Objektmodell definiert. Diese Datei wird zur Laufzeit von der Hibernatekomponente geladen und verarbeitet, um die notwendigen Übersetzungs- und Zugriffsoperationen zu erzeugen.

Hibernate wird als Open Source Projekt auf `http://www.hibernate.org` entwickelt.

[1] Aus [6], Abschnitt "Why SimpleORM?"; Übersetzung des Autors
[2] Aus [7], S. ix; Übersetzung des Autors

Kapitel 5

Gegenüberstellung

5.1 Schemadefinition

5.1.1 SQL

Tabellen werden mit ***CREATE TABLE*** erzeugt. Dabei werden Attribute und deren Datentyp festgelegt. Zusätzlich können Einschränkungen für die einzelnen Attribute angegeben werden, um die Randbedingungen für interne Konsistenz zu definieren.

Im folgenden Beispiel ist die Definition der Tabellen von Artikeln und ihren Schlagwörtern angeführt:

```
1  CREATE TABLE articles (
2      id              integer NOT NULL PRIMARY KEY, -- Primärschlüssel
3      created_on      timestamp without time zone DEFAULT now() NOT NULL,
4      last_modified   timestamp without time zone DEFAULT now() NOT NULL,
5      title           text  DEFAULT '' NOT NULL,
6      text            text  DEFAULT '' NOT NULL,
7      summary         text  DEFAULT '' NOT NULL,
8      CHECK ( created_on <= last_modified ) -- Letzte Änderung muss nach der Erzeugung sein
9  );
10
11 CREATE TABLE articles_tags (
12     id      integer NOT NULL REFERENCES articles ( id ), -- Artikelnummer muss existieren
13     tag     tag NOT NULL,
14     UNIQUE ( id, tag ) -- jedes Schlagwort darf bei jedem Artikel nur einmal auftauchen
15 );
```

In Zeile 2 wird mit dem Zusatz ***PRIMARY KEY*** das *id* Attribut als Identität der Artikel in der Datenbank definiert. Automatisch werden so gekennzeichnete Spalten mit einem Index hinterlegt, sodass Abfragen auf diese Werte besonders schnell abgewickelt werden können.

Normalerweise können alle Attribute in einer Tabelle den ***NULL***-Wert annehmen und so "kein Wert" aussagen. Mit der ***NOT NULL*** Einschränkung wird dies hier überall verboten.

In Zeile 8 wird definiert, dass das Erzeugungsdatum eines Artikels immer vor dem Datum seiner letzten Modifikation sein muss. Mit ***CHECK*** können beliebige zeilenweise Einschränkungen auf der Tabelle definiert werden.

Zeile 12 schränkt den Wertebereich des *id* Attributs auf Werte aus der Menge der Artikelnummern ein. Damit können nur Schlagwörter für existierende Artikel abgelegt werden.

Zuletzt wird in Zeile 14 noch festgelegt, dass jedes Schlagwort bei jedem Artikel nur einmal vorkommen darf.

Operationen, die eines dieser Kriterien verletzen würden – zum Beispiel doppeltes Einfügen von Schlagworten für den selben Artikel – brechen mit einer Fehlermeldung ab.

5.1.2 Reines Java

In einem Javaprogramm besteht das Schema aus zwei fundamental unterschiedlichen Teilen: auf der einen Seite die zur Kompilierung verwendete Schnittstellendefinition, aus der die Attributzugriffe erzeugt werden, und auf der anderen Seite das Datenformat, die "physikalische" Art und Weise wie die Attribute gepeichert werden.

Schnittstellendefinition

Die Schnittstelle des Datenobjektes gibt alle notwendigen Operationen der Klasse an. Das sind zumindest alle Operationen, um die Attribute zu setzen und wieder auszulesen:

```
public interface IArticle {

    public abstract int getId();
    public abstract void setId(int id);

    public abstract java.util.Set<Tag> getTags();

    public abstract String getTitle();
```

```
    public abstract void setTitle(String title);

    public abstract Date getCreatedOn();
    public abstract void setCreatedOn(Date created);

    public abstract Date getLastModified();
    public abstract void setLastModified(Date modified);

    public abstract String getSummary();
    public abstract void setSummary(String summary);

    public abstract String getText();
    public abstract void setText(String text);

}
```

Diese Schnittstellendefinition gibt – im Gegensatz zur SQL ***CREATE TABLE*** Anweisung – weder für die Laufzeit noch für die Persistenz ein Speicherschema vor.

Datenformat: Laufzeit

Für die Laufzeit werden im einfachsten Fall Attribute mit den passenden Javatypen als private Instanzvariablen abgelegt. Beispielhaft dafür hier die Implementation des *Summary*-Feldes des Artikels:

```
public class Article implements IArticle {
    private String _summary = "";

    public String getSummary()
    {
        return _summary;
    }

    public void setSummary(String summary)
    {
        _summary = summary;
    }
```

Diese Zugriffmethoden können auch zusätzliche Validierungsanweisungen enthalten. Das kann von einfachen Überprüfungen wie der Beschränkung der maximalen Länge von Zeichenketten bis zu komplexen Geschäftsregeln gehen.

Datenformat: Serialisierung

Der Standardpersistenzmechanismus von Java ist Serialisierung (siehe [22]). Wie der Name schon andeutet, handelt es sich dabei um eine Methode, Objektbäume in eine ein-eindeutige lineare Repräsentation zu transformieren und wieder daraus zu lesen. Diese lineare Repräsentation kann dann in einer Datei gespeichert und von einem anderen Prozess gelesen werden.

Für einfache Klassen, die ihrerseits nur aus serialisierbaren Teilen bestehen, wird das Schema von der virtuellen Maschine automatisch generiert und Instanzen können ohne weitere Programmierung in einen *ObjectOutputStream* geschrieben werden.

Darüber hinaus werden jedoch keine weiteren Hilfsmittel angeboten. Konkret fehlen Vorrichtungen zur Erfüllung der ACID Anforderungen (Siehe Abschnitt 2.4), Mechanismen, um einmal abgespeicherte Objekte wiederzufinden sowie Indizierungsmechanismen, um bei einer Suche nicht alle Objekte laden zu müssen. Das Speicherformat enthält Java Typinformationen zur korrekten Rekonstruktion, die speziell bei großen Mengen gleichartiger Objekte zu überflüssiger Redundanz führen.

Datenformat: SQL

Bei großen Datenmengen wird bevorzugt auf eine SQL-Datenbank zurückgegriffen. Über die JDBC Schnittstelle können dann Objektattribute manuell in der Datenbank gespeichert und gelesen werden. Über SQL Abfragen können einzelne Objekte und Objektmengen nach beliebigen Kriterien gesucht werden.

Ohne besondere Unterstützung durch Werkzeuge wird das SQL-Schema parallel zur Entwicklung der Java Schnittstellen und Klassen mitentwickelt und als SQL-Skript abgelegt. Zur Evolution des Schemas können dann einzelne SQL Befehle zur Transformation von einer Version zur nächsten abgelegt werden.

5.1.3 SimpleORM

Das Schema von SimpleORM Objekten wird direkt in der Java Klasse definiert. Die eigentliche Persistenzfunktionalität über eine Ableitung von der *SRecordInstance* Klasse eingebunden. Die Schemainformationen speichert ein Feld vom Typ *SRecordMeta*. Für jedes Attribut der Klasse

wird ein *SFieldMeta*-Deskriptor angelegt und mit dem *SRecordMeta* verknüpft. Hier eine vereinfachte Darstellung der *Article* Klasse:

```
public class Article extends SRecordInstance
{
    private static final SRecordMeta
            meta = new SRecordMeta(Article.class, "articles");
    public static final SFieldInteger
            ID = new SFieldInteger(meta, "id",
                SSimpleORMProperties.SFD_PRIMARY_KEY,
                SSimpleORMProperties.SGENERATED_KEY.pvalue(
                    new SGeneratorSequence(meta)));
    public static final SFieldString
            TITLE = new SFieldString(meta, "title", -1,
                SSimpleORMProperties.SMANDATORY.ptrue());
    public static final SFieldTimestamp
            CREATED_ON = new SFieldTimestamp(meta, "created_on",
                SSimpleORMProperties.SMANDATORY.ptrue(),
                SSimpleORMProperties.SEXTRA_COLUMN_DDL.pvalue("␣DEFAULT␣now()"));
    public static final SFieldTimestamp LAST_MODIFIED = // ...
    public static final SFieldString SUMMARY = // ...
    public static final SFieldString TEXT = // ...

}
```

Die Funktion *String Article.meta.createTableSQL()* erzeugt eine SQL Anweisung, die eine passende Tabelle für diese Klasse anlegt.

5.1.4 Hibernate

Javaklassen und das Datenbankschema können in Hibernate voneinander komplett unabhängig verwaltet werden, solange eine passende Abbildungsbeschreibung existiert. Es besteht auch die Möglichkeit, das Datenbankschema von Hibernate einmalig oder laufend aus der Abbildungsbeschreibung erzeugen oder erneuern zu lassen.

Die Beschreibung wird in XML notiert. Hier wieder das Beispiel für Artikel:

```
<hibernate-mapping package="webbook.hibernate">
    <class name="Article">
        <id name="id" column="id">
            <generator class="native"/>
        </id>
```

```
        <property name="title" type="text" not−null="true"/>
        <property name="createdOn" type="timestamp" not−null="true"/>
        <property name="lastModified" type="timestamp" not−null="true"/>

        <property name="text" type="text" not−null="true"/>
        <property name="summary" type="text" not−null="true"/>

        <set name="tags" table="articles_tags" cascade="all">
            <key column="id"/>
            <element column="tag" type="webbook.hibernate.utils.TagType" not−null="true"/>
        </set>
    </class>
</hibernate−mapping>
```

Dieses XML-Stück definiert die Abbildung der *Article* Klasse auf eine triviale – weil gleichförmige – Tabelle. Mittels weiterer Attribute können Tabellen (*table="table_name"*) und Spalten (*column="column_name"*) umbenannt werden. Am Ende ist die Menge der Schlagwörter mit dem <*set/*> Element definiert. Im Gegensatz zu den anderen Methoden können die Schlagworte hier direkt als Menge abgebildet werden. Um die Abbildung der Schlagworte direkt auf die korrekte Klasse *webbook.utils.Tag* zu machen, steht hier als *type* eine benutzerdefinierte Klasse *webbook.hibernate.utils.TagType*, die *Tag*-Instanzen auf *text* Spalten abbildet. Diese Klasse leitet von *org.hibernate.type.TextType* ab und muss nur eine kleine Anzahl von Funktionen implementieren, um die passenden Typumwandlungen durchzuführen.

Darüber hinaus bietet Hibernate Möglichkeiten, Ableitungsbäume auf unterschiedliche SQL Strukturen abzubilden, Attribute nach diversen Gesichtspunkten zu gruppieren sowie $1:1, 1:N$ und $N:M$ Abbildungen zu modellieren. Mehr dazu in den folgenden Abschnitten.

5.2 Objektmengen

Attribute, die mehrere Werte gleichzeitig annehmen können – vor allem Listen und Mengen – sind in der Datenmodellierung von besonderem Interesse, da sie die Grundlage für alle komplexeren Strukturen darstellen.

5.2.1 SQL

Relationale Systeme kennen nur einen Beziehungstyp: $[0,1] : [0,n]$, kurz $1 : N$ geschrieben: Jedes Element in der einen Menge kann, muss aber nicht, zu keinem, einem oder mehreren Elementen in Beziehung stehen. Durch Kombination und mittels Randbedingungen können alle anderen Beziehungsmannigfaltigkeiten abgebildet werden.

Einfache Beziehungen

Umgesetzt werden solche Beziehungen mittels Fremdschlüsseln. Als Beispiel hier die Beziehung zwischen Ordnern. Jeder Ordner kann – muss aber nicht – in genau einem übergeordneten Ordner – *parent* – enthalten sein. Umgekehrt kann jeder Ordner mehrere andere Ordneren enthalten.

In SQL wird dazu der Schlüsselwert des übergeordneten Tupels in einem weiteren Attribut gespeichert. Durch die *REFERENCES* Beschränkung wird festgelegt, welche Werte vorkommen dürfen. Hier die ganze Anweisung:

```
CREATE TABLE folders (
    id integer NOT NULL PRIMARY KEY,
    parent integer REFERENCES folders (id),
    -- weitere Attribute hier
);
```

Die großen Pluspunkte dieser Methode sind die Redundanzfreiheit dieser Darstellung und der daraus automatisch folgenden Symmetrie zwischen den Beziehungen "enthält" und "ist enthalten in". Die relevanten Abfragen zu diesen beiden Beziehungsrichtungen ist einerseits "Welche Ordner sind im Ordner *id=PARENT* enthalten?" und andererseits "Welcher Ordner enthält den Ordner *id=CHILD*?"

```
-- Unterordner
SELECT id FROM folder WHERE parent = PARENT;

-- Überordner
SELECT parent FROM folder WHERE id = CHILD;
```

Ein spezielles Problem solcher rekursiven Beziehungen ist die Forderung, dass ein Ordner sich nicht selbst enthalten darf. Durch einen Konsistenzprüfung der Form *parent IS* ***NULL OR*** *parent != id* lässt sich zwar verhindern, dass ein Ordner sich selbst *direkt* enthält, für eine allgemeine Lösung sind diese zeilenweisen Bedingungen jedoch nicht mächtig genug. Hier muss man sich entweder auf die korrekte Implementierung der Applikation verlassen oder mit Triggern komplexere – und damit in der Ausführung und Wartung teure – Überprüfungen implementieren.

Als flankierende Maßnahmen können auch externe Konsistenzprüfprogramme eingesetzt werden, um regelmässig solche Bedingungen "manuell" zu überprüfen.

Mengenbeziehungen

Komplexere Beziehungen mit $M : N$ Kardinalitäten implementiert man mit einer Hilfstabelle. Diese Konstruktion ist in Abbildung 5.1 dargestellt. In der Hilfstabelle können auch beschreibende Attribute der Beziehung gespeichert werden.

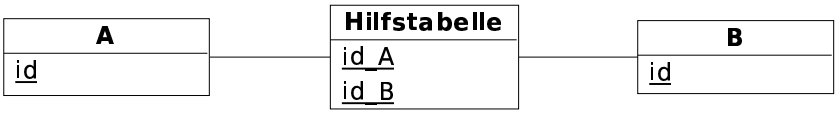


Abbildung 5.1: $M : N$ Beziehung

5.2.2 Reines Java

Modellierung

Java Objektmengen kommen in drei grundlegenden Varianten:

Set<*E*>: Menge von *E*-Objekten ohne Duplikate

List<*E*>: geordnete Liste von *E*-Objekten; Duplikate sind erlaubt

Map<*K,V*>: $1 : 1$ Abbildung von *K*-Objekten auf *V*-Objekte

Für jede dieser Schnittstellen gibt es in [20] ausführliche Garantien zu Laufzeit und Speicherverhalten der tatsächlichen Implementierungen. Java selbst liefert für die unterschiedli-

chen Anforderungen des Programmierers Implementierungen auf Vektor-, Listen- und Hashtabellenbasis, die an entscheidenden Stellen eben besser als die von der Schnittstelle geforderten Kriterien sind. Bei besonderen Anforderungen können diese Schnittstellen auch selbst oder von Drittanbietern implementiert werden.

Hier das Beispiel der Ordnerstruktur mit Hilfe eines *Set*s:

```
package webbook;
class BasicFolder implements IFolder {
    /* Die Menge der Unterordner dieses Ordners */
    java.util.Set<IFolder> subfolders
        = new java.util.HashSet<IFolder>();

    /* ... */
}
```

Diese Methode modelliert jedoch nicht die Symmetrie dieser Beziehung. In der Navigation durch dieses Modell kann nicht direkt von einem Unterorder zu seinem Überordner gesprungen werden. Dafür müsste die – ebenfalls händisch zu implementierende – Liste aller Ordner durchsucht werden.

Umgekehrt könnte das SQL-Modell übernommen werden, um immer im Modell "nach oben" navigieren zu können.

```
package webbook;
class BasicFolder implements IFolder {
    /* Der übergordnete Ordner */
    IFolder parent = null;

    /* ... */
}
```

Somit kann der übergeordnete Ordner festgestellt werden. Da Java keine Möglichkeit bietet, über alle Instanzen einer Klasse zu iterieren, kann mit dieser Methode – ohne zusätzlichen Programmieraufwand – nicht einmal die Menge aller enthaltenen Ordner festgestellt werden. Normalerweise werden daher beide Richtungen implementiert. Da es sich dabei um redundante Informationen handelt, muss bei Manipulationen dieser Struktur jedoch besondere Sorgfalt an den Tag gelegt werden. Durch die Kapselung der Attribute kann das Problem jedoch auf die betroffene Klasse beschränkt werden.

```
package webbook.manual;
class Folder implements IFolder {
    /* Die Menge der Unterordner dieses Ordners */
    private java.util.Set<Folder> subfolders
        = new java.util.HashSet<Folder>();
    /* Der übergordnete Ordner */
    private Folder parent = null;

    public void setParent(Folder f)
    {
        /* Konsistenzüberprüfung */
        if (f == this || (f != null && f.hasParent(this)) )
            throw new Exception("Konsistenzverletzung");

        /* entfernt diesen Ordner aus altem parent */
        if (parent != null)
            parent.removeFolder(this);

        /* fügt diesen Ordner zum neuen parent hinzu */
        parent = f;
        if (parent != null)
            parent.addFolder(this);
    }

    /* rekursiver Überprüfung der Überordnung */
    public bool hasParent(Folder p)
    {
        return parent == p || ( parent != null && parent.hasParent(p) );
    }

    /* ... */
}
```

Gleichzeitig können solche Methoden auch dazu benutzt, werden zusätzliche Konsistenzüberprüfungen durchzuführen, wie hier die *hasParent* Methode, deren Überprüfung die Baumstruktur garantiert, da kein Zyklus gebildet werden kann (***this*** müsste dazu bereits dem neuen Überordner übergeordnet sein).

Andere Beziehungstypen können mit entsprechendem Aufwand mit passenden Kombinationen von *Set*s und einfachen Attributen implementiert werden.

Implementierung mit Datenbank

Werden die betrachteten Objekte in einer Datenbank gespeichert, so müssen die Beziehungen ebenfalls gespeichert werden. Dazu eignen sich die im vorhergehenden Abschnitt besprochenen Fremdschlüssel.

Um beim Laden aus der Datenbank nicht den kompletten Objektbaum holen zu müssen, braucht man Beziehungen nicht automatisch mitladen, sondern lädt die verknüpften Daten erst bei Zugriffen auf die Mengenattribute in die Javaumgebung.

Je nach Beziehungstyp und Darstellung in der Datenbank wird beim Speichern einfach das betroffene Attribut überschrieben oder der gesamte betroffene Bereich der Relation wird gelöscht und neu geschrieben. Bei Beziehungen mit einer großen Anzahl von beteiligten Objekten lohnt es sich speziellen Mengenklassen zu implementieren, die nur inkrementell die durchgeführten Änderungen speichern.

Komplexere Abfragen auf Beziehungen werden direkt in SQL implementiert. Die Ergebnisse können dabei individuell an die Anforderungen angepasst werden.

5.2.3 SimpleORM

In [6], "Associations and Class Mappings", argumentiert Berglas, dass reale Anwendungen soviel Kontrolle – zum Beispiel für seitenweises Anzeigen – über die "zu N" Seite brauchen, dass besser die allgemeine Suchfunktionen verwendet werden sollen, als diese Funktionen nochmals für diesen Spezialfall zu implementieren. Daher unterstützt SimpleORM nur die "zu Eins" Seite von Beziehungen direkt. Durch eine *SFieldReference* werden die notwendigen Strukturen definiert. Hier die Definition der Eltern-Kind Beziehung der Ordner:

```
package webbook.simpleorm;
public class Folder implements IFolder {
    // Referenz auf den übergeordneten Ordner
    public final SFieldReference PARENT = new SFieldReference(meta, meta, "ref");

    public Folder getParent() {
        return (Folder)getObject(PARENT);
    }

    public void setParent(Folder parent) {
        setObject(PARENT, parent);
```

```
    }
```

Dank der höheren Abstraktionsebene von SimpleORM operiert die Klasse nun direkt auf *Folder*-Instanzen. Damit wird schon bei der Verarbeitung in der Javaapplikation die Konsistenz der Datenbank garantiert. Die Zyklusfreiheit des Graphen kann durch die oben vorgestellte *hasParent* Methode überprüft werden:

```
package webbook.simpleorm;
public class Folder implements IFolder {
    public void validateField(SFieldMeta field, Object newValue) {
        if (field == PARENT) {
            if (newValue == null)
                return;

            if (!newValue instanceof Folder)
                throw new SValidationException("unerwarteter␣Objekttyp");

            if (newValue == this || ((Folder)newValue).hasParent(this))
                throw new SValidationException("Konsistenzverletzung");

            // Alle Bedingungen erfüllt
            return;
        }
    }
```

Um alle Unterfolder zu erhalten, befragt man die Datenbank:

```
public class Folder implements IFolder {
    public List<Folder> getChildren() {
        SResultSet result = Folder.meta.newQuery().eq(Folder.PARENT, this)
            .descending(Folder.TITLE).execute();
        return Collection.unmodifiableList(result.getArrayList(1000));
    }
```

Bei der Modellierung der Schnittstelle nach außen stellen sich natürlich die gleichen Probleme wie bei reinem Java. Der minimalistische Ansatz von SimpleORM läßt bei der Implementierung jedoch keine Mißverständnisse zu. Die Beziehung ist über das *PARENT* Feld definiert. *add*- und *remove*-Methoden brauchen sich daher auch nur um dieses kümmern. Änderungen am Abfrageergebnis aus *getChildren()* können daher keine Auswirkung auf die Datenbank haben. Um das auch an Benutzer dieser Klassen zu kommunizieren, kann mit *Col-*

lection.unmodifiableList() eine nur-Lesen Liste erzeugt werden, die bei Schreibversuchen eine Ausnahmebedingung erzeugt.

Eine Spezialität von SimpleORM ist die Notwendigkeit einer oberen Schranke für die Anzahl der Ergebnisobjekte bei der *getArrayList* Methode. Im Quelltext wird dies mit der Früherkennung unbeschränkter Abfragen begründet. Darüber hinaus ist es aber auch sinnvoller, große Ergebnismengen zeilenweise abzuarbeiten, da bereits bearbeitete Objekte wieder freigegeben werden können.

5.2.4 Hibernate

In der XML-Abbildungsbeschreibung werden $1 : N$ und Mengenbeziehungen direkt mit eigenen Elementen unterstützt. Dadurch ergibt sich eine sehr komfortable Modellierung dieser Sachverhalte.

Die Menge der Schlagworte wird mit einem <*set*/> beschrieben:

```
<set name="tags" cascade="all">
    <key column="id"/>
    <element column="tag" type="webbook.hibernate.utils.TagType" not−null="true"/>
</set>
```

Hibernate erzeugt aus dieser Definition automatisch die Tabelle *tags* mit den Spalten *id* und *tag*. Da die Schlagwortmenge kein eigenständiges Objekt ist, sondern immer nur im Kontext des beschlagworteten Artefakts existiert, wird durch *cascade="all"* angezeigt, dass alle Operationen auf dem übergeordneten Element – vor allem Änderungen am Schlüsselwert und Löschung – auch auf diesem <*set*/> durchgeführt werden sollen.

Hibernate kennt auch <*list*/>, <*array*/> und <*primitive−array*/> für angeordnete Daten mit unterschiedlichen Javarepräsentationen, <*map*/> für 2-Tupel und <*bag*/> für ungeordnete Multimengen.

Beziehungen zwischen Objekten werden mit den Elementen <*one−to−many*/>, <*many−to−one*/> und <*many−to−many*/> beschrieben. Die $: N$ Varianten müssen dabei in einer Mengendefinition eingebettet werden, um die Semantik der Beziehung zu spezifizieren. Von den eingeschränkten Möglichkeiten der Modellierung in reinem Java erbt auch Hibernate die Assymetrie von Beziehungen. Die beiden Enden müssen seperat modelliert werden. Hier wieder am Beispiel der Ordner:

```
<many−to−one name="parent" class="Folder"/>
<set name="children" inverse="true">
    <key column="id"/>
    <one−to−many class="Folder"/>
</set>
```

Dabei verhindert die *inverse="true"* Deklaration, dass über die *children* Menge die Beziehung verändert werden kann. Die *children* Menge muss auf der Javaseite wie schon bei der Implementierung mit reinem JDBC beschrieben gepflegt werden.

5.3 Datenbankverbindung und -abfragen

5.3.1 SQL

Der direkte Zugriff auf die Datenbank erfolgt entweder mit SQL-Befehlen auf einem eigenen kommandozeilenorientierten Klienten oder mit einer Verwaltungsapplikation, die auch Unterstützung in der Verwaltung und Wartung des Datenbankschemas sowie der Erstellung von Abfragen bietet.

5.3.2 Reines Java

Die JDBC API unter Java ist eine gemeinsame Schnittstelle für den Datenbankzugriff. Um die verschiedenen Kommunikationsprotokolle zu unterstützen, wird von den jeweiligen Datenbankherstellern auch ein JDBC-Treiber zur Verfügung gestellt.

Ein typischer Verbindungsablauf sieht so aus:

```
1  Class.forName(TREIBER); // registrieren
2  Connection connection = DriverManager.getConnection(URL); // verbinden
3  PreparedStatement stmt = connection.prepareStatement(SQL);
4  stmt.set<<Type>>(INDEX, VALUE); // Parameter setzen
5  stmt.execute(); // Abfrage durchführen
6  ResultSet rs = stmt.getResultSet();
7  while(rs.next()) { // Cursor bewegen
8      rs.get<<Type>>(INDEX); // Werte abfragen
9  }
10 // Resourcen explizit freigeben
11 rs.close();
12 stmt.close();
```

```
13 connection.close();
```

Zuerst lädt *Class.forName("JDBC−Treiber")* den JDBC-Treiber in die virtuelle Maschine (Zeile 1). Danach öffnet *DriverManager.getConnection("URL")* die Verbindung *conn* zur Datenbank (Zeile 2). Optional werden über diesen Aufruf weitere Parameter an den Treiber übergeben. Die Anweisung *conn.prepareStatement(SQL)* erzeugt aus der SQL-Abfrage ein parameterisiertes, wiederverwertbares Kontainerobjekt (Zeile 3) in dem *set*-Aufrufe Parameter verschiedener Typen setzen (Zeile 4). *execute()* überträgt die Abfrage und die Parameter an die Datenbank, wo sie direkt interpretiert und ausgeführt wird (Zeile 5). Als Ergebnis erhält man ein *ResultSet*, eine Repräsentation eines Datenbankcursors (Zeile 6). Parallel zu den *set*-Methoden des *PreparedStatement*s hat das *ResultSet get*-Methoden, die Attribute in verschiedenen Datentypen auslesen.

5.3.3 SimpleORM

Diese Bibliothek übernimmt die Verwaltung, jedoch nicht die Erstellung der JDBC-Verbindung. Auch hier lädt zuerst *Class.forName("JDBC−Treiber")* den JDBC-Treiber in die virtuelle Maschine (Zeile 1). *SConnection* übernimmt nun die direkte Verwaltung der JDBC-Verbindung (Zeile 2-4).

```
1 Class.forName(TREIBER); // registrieren
2 SDataSource ds = new SDataSource(URL, new Properties());
3 SConnection.attach(ds, "default"); // verbinden
4 SConnection.begin();
```

Wenn die Verbindung steht, können Standardabfragen nach dem Primärschlüssel automatisiert mit *mustFind(PRIMARY_KEY)* erzeugt werden. Über die *SQuery* Schnittstelle können komplexere Anfragen programmatisch gebaut werden.

5.3.4 Hibernate

Die Datenbankverbindung von Hibernate wird über eine XML-Datei konfiguriert.

```
1 <hibernate−configuration>
2     <session−factory>
3         <!−− Databankverbindungsparameter −−>
4         <property name="connection.driver_class">JDBC−TREIBER</property>
```

```
5       <property name="connection.url">URL</property>
6       <!-- SQL Dialekt -->
7       <property name="dialect">org.hibernate.dialect.PostgreSQLDialect</property>
8       <!-- ... -->
9       <!-- Schema Definition einbinden -->
10      <mapping resource="webbook/hibernate/DataObject.hbm.xml"/>
```

Wie bei den anderen Bibilotheken auch, liegt hier eine JDBC-Verbindung zu Grunde. Der Treiber und die URL dafür werden mit *connection.driver_class* und *connection.url* angegeben (Zeile 4,5). Damit Hibernate korrekt optimierte SQL-Definitionen und Abfragen erzeugt, gibt der *dialect* auf Zeile 7 den – für die Tests verwendeten – *PostgreSQLDialect* an. Zuletzt werden über *mapping*-Elemente die einzelnen Abbildungsdefinitionen eingebunden (Zeile 10). Über weitere *property*-Elemente können zusätzliche Parameter von Hibernate konfiguriert werden.

Da die Konfiguration von Hibernate über die XML-Datei läuft, enthält der Quelltext keine Parameter mehr, Objekte können direkt über Javakonstrukte angesprochen werden:

```
1  // hibernate.cfg.xml einlesen
2  SessionFactory sessionFactory = new Configuration().configure().buildSessionFactory();
3
4  // Abfrage nach Primärschlüssel
5  Object result = sessionFactory.getCurrentSession().get(KLASSE, ID);
6
7  // beliebige Abfragen
8  List results = sessionFactory.getCurrentSession().createQuery(
9      "from␣NAME␣join␣NAME.tags␣where␣CONDITION").list();
```

Hibernate bringt eine eigene, SQL-ähnliche Abfragesprache mit, die in der letzten Anweisung gezeigt wird.

5.4 Datensicherheit

Die in Abschnitt 2.4 aufgelisteten Forderungen sollten natürlich auch im Objektmodell halten. Welche Vorkehrungen die einzelnen Systeme dafür haben, wird im folgenden Abschnitt besprochen.

5.4.1 SQL

In SQL können mehrere Anweisungen zu einer Transaktion zusammengefasst werden. Die Anweisungen einer Transaktion werden entweder alle gemeinsam oder überhaupt nicht ausgeführt. Eine Transaktion wird mit dem Befehl *BEGIN* geöffnet und mit ***COMMIT*** *WORK* abgeschlossen. Das Datenbanksystem garantiert mit dem erfolgreichen Abschluss der Transaktion deren Dauerhaftigkeit. Kommt es zu einem Fehler, kann die Transaktion mit dem Befehl ***ROLLBACK*** *WORK* abgebrochen werden.

Das sogenannte Isolationslevel beschreibt die (Un-)Abhängigkeit von Transaktionen untereinander. Starke Isolierung von parallel ablaufenden Anweisungen erfordert einen gewissen Aufwand auf Seiten des Datenbanksystems. Durch die Angabe eines niedrigeren Isolationslevels verzichtet man auf nicht benötigte Mechanismen. Das sollte natürlich nur gemacht werden, wenn die Anwendung auch ohne diese Zusicherungen korrekt funktioniert.

Die beiden wichtigsten Isolationslevels sind ***READ*** *COMMITTED* und *SERIALIZABLE*. Mit ***READ*** *COMMITTED* Isolation sieht jede Anweisung nur Datansätze aus bereits abgeschlossenen Transaktionen. Innerhalb einer Transaktion kann es jedoch vorkommen, dass beim neuerlichen Lesen eines Datensatzes auch neue, geänderte Daten gelesen werden, wenn von einer parallelen Transaktion dieser Datensatz geändert wurde. Damit sichert ***READ*** *COMMITTED* einen konsistenten Blick auf die Datenbank *innerhalb einer einzelnen Anweisung* zu. Mit *SERIALIZABLE* Isolation verhält sich die Datenbank so, als würden alle Transaktionen strikt nacheinander ausgeführt werden. Je nach Implementierung in der Datenbank kommt es dadurch zu verstärkten Wartezeiten auf den benutzten Tabellen- und Zeilensperren oder zu erzwungenen Transaktionsabbrüchen bei Schreibkonflikten (wie bei MVCC in Abschnitt 5.7.1 beschrieben).

5.4.2 Reines Java

In der Standardkonfiguration befindet sich JDBC im *autocommit*-Modus. Dabei wird nach jeder SQL-Anweisung automatisch die aktuelle Transaktion abgeschlossen und eine neue eröffnet. Muss eine Applikation mehrere Anweisungen gemeinsam in einer Transaktion abwickeln, so kann der autocommit-Modus mit *conn.setAutoCommit(**false**);* deaktiviert werden. Die gesamte Struktur einer Transaktion sieht dann so aus:

```
1 public static void machWas(java.sql.Connection conn)
```

```
2       throws java.sql.SQLException
3   {
4       conn.setAutoCommit(false);
5
6       try {
7
8           /* Datenbankanweisungen hier */
9
10          conn.commit();
11      } catch (Throwable t) {
12          conn.rollback();
13          throw t;
14      }
15  }
```

Das Deaktivieren des automatischen Transaktionsabschlusses in Zeile 4 eröffnet gleichzeitig eine neue Transaktion. Danach kann im ***try***-Block die Datenbank über JDBC-Aufrufe abgefragt und modifiziert werden (Zeile 8). Ist alles korrekt abgelaufen, schliesst *conn.commit()* die Transaktion ab (Zeile 10). Kommt es zu einem Fehler, bricht *conn.rollback()* die Transaktion ab (Zeile 12), bevor ***throw*** *t* den Fehler weiterreicht (Zeile 13). Situationsabhängig kann hier auch versucht werden, den Fehler lokal zu behandeln, zum Beispiel durch das Neustarten der Transaktion.

Es ist dabei auf die Synchronisation zwischen den verschiedenen Ausführungssträngen der Java-Applikation zu achten, da diese die Möglichkeit haben sich gegenseitig auf der JDBC-Verbindung zu stören. Normalerweise wird dieses Problem umgangen, indem für mehrere Threads jeweils eigene Verbindungen geöffnet werden.

5.4.3 SimpleORM

Die JDBC-Verbindung wird in eine *SConnection* Instanz gehüllt. Diese Instanz ist Thread-lokal. Damit wird die gesamte Synchronisation zwischen Threads an das DBMS delegiert. Transaktionen können über die Methoden *begin()*, *rollback()* und *commit()* gesteuert werden:

```
SConnection.begin();

try {
    /* Datenbankmodifikationen */
```

```
    SConnection.commit();
} catch (Throwable t) {
    SConnection.rollback();
    throw t;
}
```

*SRecordInstance*s sind jedoch an die Lebenszeit einer Datenbanktransaktion gebunden, da außerhalb einer Transaktion die Konsistenz zur Datenbank nicht gewährleistet werden kann. Will man solche Objekte ausserhalb einer Transaktion verwenden, zum Beispiel um sie via RMI zu transportieren, kann man mit *INSTANZ.detach()* den Record von der Transaktion lösen. Dabei aktiviert SimpleORM automatisch optimistische Sperrmechanismen, um auch über die Datenbanktransaktion hinaus Inkonsistenzen feststellen zu können. *INSTANZ.attach()* fügt einen Record in die aktuelle Transaktion wieder ein.

5.4.4 Hibernate

Das Hibernate Gegenstück zur JDBC-Verbindung, ist die *Session*. In ihr wird die Verbindung zur Datenbank aufgebaut und die einzelnen Transaktionen abgehandelt. Die Hibernate Dokumentation empfiehlt entweder eine Session pro Anfrage in einem Client/Server System zu verwenden oder eine Session für die Abwicklung eines ganzen Geschäftsfalls zu verwenden. Im zweiten Fall werden die Objekte automatisch versioniert und beim Abschluss einer Transaktion optimistisch in die Datenbank zurückgeschrieben. Dadurch werden keine Sperren in der Datenbank gebraucht, es kann jedoch bei Schreibkonflikten zu einem erzwungenen Transaktionsabbruch kommen. Um diese zu vermeiden, kann man an einer bekannten Stelle in der Datenbank vermerken, welche Objekte gerade "in Arbeit" sind und so bereits beim ersten Zugriff verhindern, dass es zu solchen erzwungenen Abbrüchen kommt.

```
Transaction trans = sessionFactory.getCurrentSession().getTransaction();
trans.begin();

try {
    /* Datenbankmodifikationen */

    trans.commit();
} catch (Throwable t) {
    trans.rollback();
    throw t;
```

}

5.5 Objektlebenszyklus

Der typische Lebenszyklus eines Objektes besteht aus seiner Erzeugung, einer beliebigen Abfolge von Lese- und Modifikationsoperationen und endet mit der Löschung des Objektes. Das wird in der Literatur im Akronym CRUD, engl. für Create, Read, Update, Delete, zusammengefasst. Hier zeigt sich der erste große Vorteil der Bibliotheken gegenüber der manuellen Implementierung, da diese Operationen aus den bisher definierten Strukturen automatisch abgeleitet werden können.

5.5.1 SQL

Die SQL Anweisung ***INSERT INTO*** *Tabelle (Spalten)* ***VALUES*** *(Werte)* ist die Grundform der Objekterzeugung. Dabei wird eine neue Zeile in *Tabelle* mit den angegeben Werten geschrieben.

Um Objekte aus der Datenbank zu lesen, können mit ***SELECT***-Abfragen beliebig komplexe Relationen und Bedingungen spezifiziert werden.

UPDATE *Tabelle* ***SET*** *Spalte1=Wert1, Spalte2=Wert2* ***WHERE*** *Bedingung* ändert die angegebenen Attribute in *Tabelle* bei allen Tupeln, die *Bedingung* erfüllen.

DELETE FROM *Tabelle* ***WHERE*** *Bedingung* löscht alle Zeilen, die *Bedingung* erfüllen.

Ergebnisse werden in tabellarischer Form zurückgegeben. Die genau Darstellung hängt von der benutzten Software ab.

5.5.2 Reines Java

Über *java.sql.Statement*s oder *java.sql.PreparedStatement*s können SQL-Anweisungen direkt an die Datenbank abgesetzt werden. Das Ergebnis wird durch ein *java.sql.ResultSet* repräsentiert.

Schnittstelle

Will man die Koppelung zwischen Geschäftslogik und Persistenz gering halten, empfiehlt sich eine strikte Trennung der beiden Teile durch die Einrichtung einer eigenen Datenzugriff-

schicht[1]. Dabei ist zwischen den Datenbank- und den Objektoperationen zu unterscheiden:

```
1  public interface IDatabase
2  {
3      public abstract void connect();
4      public abstract void begin();
5      public abstract void executeSQL(String sql) throws SQLException;
6      public abstract void rollback();
7      public abstract void commit();
8      public abstract void disconnect();
9  }
10
11 public interface IDriver<OBJ extends IDataObject>
12 {
13
14     /** Objekt erzeugen */
15     public abstract OBJ createObject();
16     /** Objekt abspeichern */
17     public abstract OBJ saveObject(OBJ o) throws SQLException;
18
19     /** Objekt nach Primärschlüssel suchen */
20     public abstract OBJ findById(int id) throws SQLException;
21     /** Objekt nach Schlüsselwörtern suchen */
22     public abstract Set<? extends OBJ> findByTags(TagExpression tags) throws SQLException;
23
24     /** Objekt löschen */
25     public abstract void deleteById(int id) throws SQLException;
26     public abstract void deleteObject(OBJ o) throws SQLException;
27
28 }
```

Suchen und Laden

Der einfachste Fall ist das Laden eines *Article*s nach seinem Primärschlüssel:

```
1  public class ArticleDAL extends BaseDAL implements IDAL<Article> {
2
3      public Article findById(int id) throws SQLException
4      {
5          PreparedStatement stmt = _conn.prepareStatement(
6              "SELECT␣*␣FROM␣articles␣WHERE␣id␣=␣?");
7          stmt.setInt(1, id);
```

[1] engl.: DAL, Data Access Layer

```
8           ResultSet result = stmt.executeQuery();
9
10          if (result.next())
11              return loadFrom(result);
12          else
13              return null;
14
15      }
```

In Zeile 5-7 wird die SQL Anweisung vorbereitet und ausgeführt. Wenn ein Artikel mit dieser ID existiert (Zeile 9), dann lädt *BaseDAL.loadFrom(ResultSet)* die Attribute aus dem *ResultSet* in eine neue Instanz. *loadFrom* ist eine virtuelle Methode, bei der die Basisimplementierung in *BaseDAL* die *IDataObject*-Attribute lädt, während *ArticleDAL.loadFrom* die zusätzlichen Attribute des Artikels ausliest. Das *PreparedStatement* kann auch wiederverwendet werden, wie das nächste Beispiel zeigt.

Schreiben

Hier die Methode, um einen neuen *Article* in die Datenbank zu speichern:

```
1       private PreparedStatement _getNewId = null;
2       private PreparedStatement _insertArticle = null;
3
4       protected void insertObject(Article obj) throws SQLException
5       {
6           if (_getNewId == null) {
7               _getNewId = getConnection().prepareStatement(
8                   "select␣nextval('articles_id_seq')");
9           }
10          _getNewId.execute();
11          ResultSet rs = _getNewId.getResultSet();
12          rs.next();
13          obj.setId(rs.getInt(1));
14
15          if (_insertArticle == null) {
16              _insertArticle = getConnection().prepareStatement(
17                  "INSERT␣INTO␣articles␣(created_on,␣last_modified,␣title,␣text,␣summary,␣id)\n"
18                  + "VALUES␣(?,␣?,␣?,␣?,␣?,␣?)");
19          }
20          // prepareSave(obj, _insertArticle):
21          {
22              _insertArticle.setTimestamp(1, new Timestamp(obj.getCreatedOn().getTime()));
```

```
23              _insertArticle.setTimestamp(2, new Timestamp(obj.getLastModified().getTime()));
24              _insertArticle.setString(3, obj.getTitle());
25              _insertArticle.setString(4, obj.getText());
26              _insertArticle.setString(5, obj.getSummary());
27              _insertArticle.setInt(6, obj.getId());
28          }
29          _insertArticle.execute();
30      }
```

Das zentrale Stück hier ist die ***INSERT***-Anweisung (Zeile 17,18) und das Befüllen der Parameter (Zeile 20-28). Durch geschicktes Anordnen der Parameter in der SQL-Anweisung können die *set*-Anweisungen bei ***INSERT*** und ***SELECT*** gemeinsam genutzt werden, wie hier mit dem *prepareSave* Kommentar angedeutet ist (Zeile 20). Aufgrund der fehlenden Unterstützung für benannte Parameter von *Statement*s ist dabei besonders auf die korrekte Reihenfolge der Platzhalter und Attribute zu achten.

Die benötigten SQL-Anweisungen werden in Instanzvariablen als *PreparedStatement* gespeichert und erst bei der ersten Benutzung initialisiert (Zeile 1,2,7,16). Das ermöglicht dem JDBC-Treiber, die SQL-Anweisung einmalig im Datebankserver in optimierter Form abzulegen. Nachfolgende Aufrufe übertragen nur noch die Parameter und ersparen sich somit das wiederholte Parsen und Optimieren der Anweisung.

Am Anfang der Methode ist ein Block um einen eindeutigen ID für den neuen Artikel aus der Datenbank abzufragen (Zeile 6-13). Dadurch kann sichergestellt werden, dass ein eindeutiger ID benutzt wird, ohne die ganze Tabelle sperren zu müssen. Alternativ können auch UUIDs[2] eingesetzt werden. Diese sind sogar über unabhängige System hinweg eindeutig, bestehen aber aus 128 Bit und sind daher in der Bearbeitung teurer. UUIDs sind in [1] definiert.

Löschen

Objekte werden mit einer einfachen ***DELETE*** Anweisung gelöscht. Die Java-Methode dazu gleicht in der Struktur den bisher vorgestellten Methoden.

[2] Universally Unique IDentifier

5.5.3 SimpleORM

Um die Werkzeuge vergleichbar zu halten, implementiert auch der SimpleORM Test die gleichen Schnittstellen wie zuvor vorgestellt. Aufgrund der direkten Koppelung zur Persistenzschicht durch die Ableitung von *SRecordInstance* sind die meisten Methoden der *IDAL* Schnittstelle nur Weiterreichungen an *SRecordInstance* Methoden des betroffenen Objektes. Diese werden hier vorgestellt.

Suchen und Laden

Zuerst wieder das Auffinden eines Objektes nach seinem Primärschlüssel:

```
public Article findById(int id)
{
    // Das ist wirklich so einfach:
    return (Article) Article.meta.find(new Integer(id));
}
```

Damit erstellt die SimpleORM Bibliothek automatisch die notwendigen SQL-Abfragen und erzeugt ein befülltes Java Objekt. Um auf die Attribute zugreifen zu können, ist es notwendig, auf die *SRecordInstance* zuzugreifen. Für den Titel sieht das so aus:

```
public class DataObject extends SRecordInstance {
    public static final SFieldString TITLE
        = new SFieldString(meta, "title", −1, SSimpleORMProperties.SMANDATORY.ptrue());

    public String getTitle()
    {
        return getString(TITLE);
    }

    public void setTitle(String title)
    {
        setString(TITLE, title);
    }
```

getString und *setString* sind dabei Funktionen der *SRecordInstance*, die das angegebene Feld bearbeiten. Aufgrund der Metainformationen im *SFieldString* Objekt können bereits beim Setzen des Feldes Randbedingungen wie Typ, Wertebereich oder benutzerspezifische Validierungsregeln überprüft werden.

Schreiben

SConnection.commit() überträgt alle ausstehenden Änderungen an die Datenbank und schliesst die Transaktion ab. Soll ein Objekt frühzeitig in die Datenbank geschrieben werden – zum Beispiel, um in einer nachfolgenden Abfragen innerhalb der gleichen Transaktion aufzuscheinen, kann die Methode *flush* der *SRecordInstance* verwendet werden.

Löschen

Die *SRecordInstance.deleteRecord()* Methode markiert ein Objekt als gelöscht. Mit dem Abschluss der Transaktion wird es auch in der Datenbank gelöscht.

5.5.4 Hibernate

Um die Werkzeuge vergleichbar zu halten, implementiert auch der Hibernate Test die gleichen Schnittstellen wie zuvor vorgestellt.

Suchen und Laden

Wie auch bei SimpleORM ist die Abfrage nach dem Primärschlüssel sehr einfach und erzeugt ebenfalls gleich die notwendigen Java Objekte.

```
public Article findById(int id)
{
    // Das ist wirklich so einfach:
    return (Article)sessionFactory.getCurrentSession().get(Article.class, id);
}
```

Schreiben und Löschen

Mit den *saveOrUpdate(OBJEKT)* und *delete(OBJEKT)* Methoden der *Session* können Objekte in die Datenbank geschrieben oder gelöscht werden.

5.6 Ableitung

Selbst in so einfachen Systemen wie der *webbook* Beispieldatenbank kann mit einer passenden Klassenhierarchie doppelter Programmtext vermieden werden. Alle Klassen der *webbook*

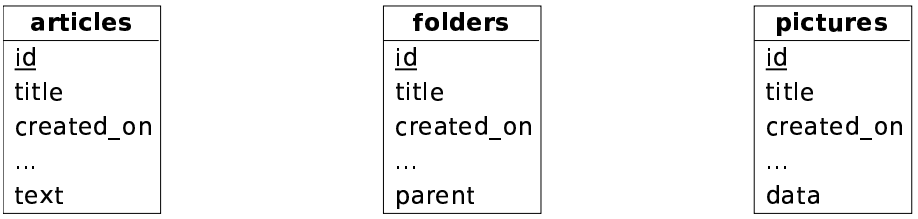


Abbildung 5.2: Tabelle pro konkreter Klasse

Beispieldatenbank haben die Attribute der *IDataObjekt* Schnittstelle gemein. Diese Attribute in einer gemeinsamen Basisklasse zu verwalten reduziert den Programmieraufwand und verhindert Inkonsistenzen in der Definition zwischen den Klassen.

5.6.1 SQL

Für verschiedene Modellierungs- und Leistungsanforderungen gibt es unterschiedliche Ansätze, um Ableitungshierarchien abzubilden. Da die korrekte Auswahl des Ansatzes wichtig ist, um die für die Applikation notwendigen Abfragen effizient durchführen zu können, hat sich hier kein einzelner "bester" Ansatz herauskristallisiert.

Tabelle pro konkreter Klasse

Ein sehr direkter Ansatz zur Abbildung einer Klassenhierarchie ist der Einsatz einer eigenen Tabelle für jede Klasse, von der direkt Instanzen[3] erzeugt werden. Abbildung 5.2 illustriert das.

Vorteile: Durch die direkte Abbildung ist dieser Ansatz der einfachste der hier vorgestellten und erzwingt keine Koordination der Objektklassen auf der Applikationsseite. So erfordert zum Beispiel das Hinzufügen neuer Unterklassen keine Änderungen an vorhandenen Strukturen und bei Änderungen an bestehenden Attributen ist der Implementierungsaufwand auf die direkt betroffenen Tabellen und Klassen beschränkt.

Abfragen auf einzelne Klassen/Tabellen werden ebenfalls unmittelbar umgesetzt und können mit minimalem Aufwand in der Datenbank auch tabellenspezifisch optimiert werden, oh-

[3] typischerweise nur Blätter im Ableitungsbaum

ne Interferenzen auszulösen. Aufgrund der separaten Tabellen bleiben Transaktionen auf die von ihnen direkt betroffenen Hierarchieteile eingeschränkt.

Nachteile: Bei jeder Abfrage müssen alle beteiligten konkreten Klassen bekannt sein, da danach die Tabellen ausgewählt werden.

Datenbankanweisungen, die zentrale Attribute betreffen – Abfragen und Datenänderungen – müssen auf mehrere Tabellen zugreifen. Das wiederum erfordert einen Mehraufwand pro Tabelle in Form von längeren Anweisungen und mehrfachen Indexzugriffen.

Da es keinen gemeinsamen Primärschlüssel gibt, muss bei polymorphen Abfragen zur Klassenunterscheidung eine künstliche Typspalte erzeugt werden. Dabei können Attribute je nach Position in der Ableitungshierarchie nicht oder nur schwer polymorph abgefragt werden. Ebenfalls aufgrund des fehlenden gemeinsamen Primärschlüssels können Fremdschlüssel nur als Verweis auf eine konkrete Klasse definiert werden. Polymorphe Relationen können so entweder als unüberprüfte Konvention über ein Hilfskonstrukt wie das oben erwähnte künstliche Typattribut angelegt werden, oder indem für jede Unterklasse wiederum eine eigene, verweisende Tabelle angelegt wird.

Bei Änderungen an gemeinsamen Attributen müssen alle Tabellen einzeln modfiziert werden. Es gibt dabei keine Absicherung gegen unbeabsichtigte Inkonsistenzen in der Struktur der einzelnen Tabellen.

Einsatzgebiet: Dieses Schema eignet sich besonders für kleine Objektmodelle wie die *webbook*-Datenbank, in der polymorphe Abfragen oder Relationen nur eine geringe Rolle spielen aber die einfachen Strukturen dem Verständnis förderlich sind.

Hier einige beispielhafte Abfragen:

−− einfache Abfrage nach Primärschlüssel
SELECT ∗ **FROM** articles **WHERE** id = ?

−− polymorphe Abfrage mit künstlicher Typspalte
−− spezielle Attribute müssen separat nachgeladen werden
SELECT 'articles' **AS** class, id, created_on **FROM** articles **WHERE** −− ...
UNION ALL
SELECT 'pictures' **AS** class, id, created_on **FROM** pictures **WHERE** −− ...
UNION ALL
SELECT 'folders' **AS** class, id, created_on **FROM** folders **WHERE** −− ...

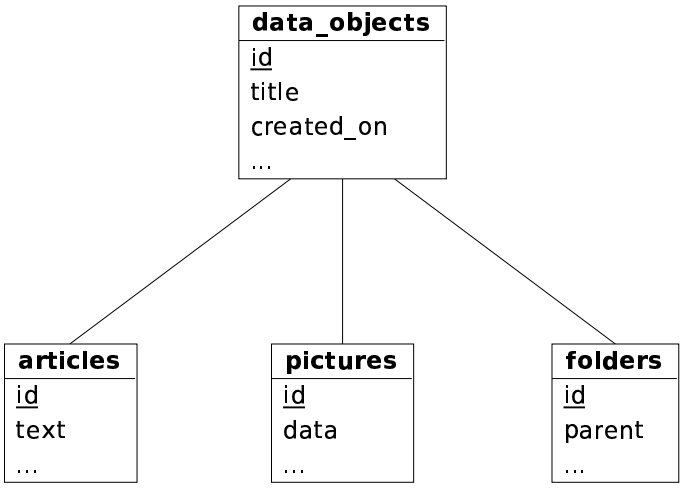


Abbildung 5.3: Tabelle pro Klasse

Tabelle pro Klasse

Mit $1 : 1$ Beziehungen wird der Ableitungsbaum direkt auf Tabellen abgebildet. Attribute werden jeweils in der "allgemeinsten" Tabelle abgelegt.

Im Gegensatz zum "Tabelle pro konkreter Klasse" Schema sind hier die Primärschlüssel in der gesamten Hierarchie eindeutig, und Attribute von Knoten des Ableitungsbaumes werden nur einmal definiert.

Vorteile: Dieser Ansatz bildet die Objekte nicht nur als Attributslisten ab, sondern erfasst auch die Struktur der Ableitungshierarchie. Dadurch sind polymorphe Abfragen auf den inneren Knoten des Ableitungsbaumes möglich. Eine typische polymorphe Abfrage in der Beispieldatenbank könnte die Suche von Artefakten nach Worten aus dem Titel sein. Bei einer solchen Abfrage ist der tatsächliche Typ des gefundenen Objekts zweitrangig. Da "Titel" ein Attribut der Basistabelle *data_objects* ist, genügt eine einfache Abfrage auf das *title* Attribut.

Durch die Abbildung der Ableitungshierarchie gibt es nun auch einen gemeinsamen Primärschlüssel: *data_objects.id*. Dadurch können nun Fremdschlüssel auf beliebige Teilbäume der Hierarchie referenzieren, indem die *id* Spalte aus der entsprechenden Tabelle angegeben wird.

Bei globalen Modifikationen in und an der Datenbank gibt es aufgrund der geringen Strukturredundanz weniger Möglichkeiten, inkonsistente Manipulationen durchzuführen. Speziell bei größeren Schemata und dem Hinzufügen oder Entfernen von Attributen kann hier Aufwand gespart werden. Wie beim Tabelle-pro-konkreter-Klasse Ansatz bleiben die Auswirkungen von solchen Änderungen auf die direkt betroffenen Klassen und Tabellen beschränkt und neue Unterklassen verursachen keine Änderungen an vorhandenen Strukturen.

Nachteile: Da die Attribute eines einzelnen Objektes über mehrere Tabellen verteilt sind, multipliziert sich damit auch der Aufand an diese Daten heranzukommen. Bei jeder Lese- oder Schreiboperation müssen die verschiedenen Tabellen extra verknüpft werden. Das bedeutet auch, dass der Benutzer der Datenbank diese Strukturen kennen muss.

Ein anderer gravierender Nachteil ergibt sich in einer Schwäche der Datenbankbeschränkungen von SQL. Dort können Eindeutigkeitsbeschränkungen nur auf einzelne Tabellen gelegt werden. Ohne zusätzliche Attribute oder Triggermethoden kann daher nicht verhindert werden, dass eine fehlerhafte Anwendung ein Objekt mit Attributen mehrerer Klassen erzeugt, zum Beispiel, indem Einträge in der *pictures* und der *articles* Tabelle auf ein gemeinsames *data_object* verweisen.

Durch die gemeinsamen Basistabellen blockieren sich Tabellensperren für spezifische Unterklassen gegenseitig. Wie stark dieser Nachteil sich auf die Anwendung auswirkt, hängt stark von den Abfrageprofilen ab.

Einsatzgebiet: Dieser Ansatz eignet sich besonders, wenn polymorphe Suchanfragen auf alle Objekte gestellt werden sollen aber nur wenige Resultate die kompletten Attribute benötigen.

Hier einige beispielhafte Abfragen:

−− einfache Abfrage nach Primärschlüssel
SELECT ∗ **FROM** data_objects **JOIN** articles **ON** (id) **WHERE** id = ?

−− polymorphe Abfrage, Klassenattribute müssen einzeln nachgeladen werden,
−− keine Unterklasseninformation
SELECT ∗ **FROM** data_objects **WHERE** bedingung

−− polymorphe Abfrage, Klassenattribute werden mitgeladen
SELECT
*,

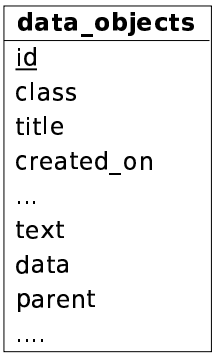


Abbildung 5.4: Tabelle pro Hierarchie

```
    CASE
        WHEN articles.id IS NOT NULL THEN 'articles'
        WHEN pictures.id IS NOT NULL THEN 'pictures'
        WHEN folders.id  IS NOT NULL THEN 'folders'
    END AS class
FROM
    data_objects
        LEFT JOIN articles ON (data_objects.id = articles.id)
        LEFT JOIN pictures ON (data_objects.id = pictures.id)
        LEFT JOIN folders ON (data_objects.id = folders.id)
```

Tabelle pro Hierarchie

Bei diesem Ansatz werden alle Attribute der gesamten Ableitungshierarchie in einer Tabelle vereinigt. Nicht benötigte Spalten werden einfach mit ***NULL*** Werten aufgefüllt. Um unterscheiden zu können, um welche Klasse es sich bei einer gegebenen Zeile handelt, ist ein zusätzliches Attribut notwendig.

Fremdschlüssel können – ohne zusätzlichen Aufwand – nur auf die Wurzel der Hierarchie verweisen. Werden Fremdschlüssel auf Teilbäume benötigt, kann dies über einen Trick mit dem Unterscheidungsattribut erreicht werden. Dazu nimmt man das Unterscheidungsattribut in den Primärschlüssel auf und schränkt im Fremdschlüssel das verweisende Attribut über eine Datenbankregel auf die gewünschten Werte ein. Das folgende Quelltextfragement demonstriert das für eine Tabelle, die nur auf Artikel verweisen kann, obwohl in der *data_objects* Tabelle alle Artefakte gespeichert werden.

```
1  CREATE TABLE data_objects (
2      id INTEGER NOT NULL UNIQUE,
3      -- a für Artikel, f für Ordner und p für Bilder
4      class CHAR NOT NULL CHECK ( class IN ('a', 'f', 'p') ),
5      PRIMARY KEY (id, class),
6      -- weitere Attribute hier
7
8  CREATE TABLE something_has_articles (
9      id INTEGER NOT NULL,
10     -- nur Verweise auf Artikel zulassen
11     class CHAR NOT NULL CHECK ( class = 'a' ),
12     FOREIGN KEY (id, class) REFERENCES data_objects (id, class)),
13     -- weitere Attribute hier
```

Dabei ist zu beachten, dass *data_objects.id* als ***UNIQUE*** definiert ist, um diese Werte alleine zur Identifikation nutzen zu können. Die ***CHECK*** Regel in Zeile 4 definiert die zulässigen Klassen von Artefakten. In Zeile 11 wird dann die Menge der zulässigen Klassen – und damit die Verweismöglichkeiten – auf Artikel eingeschränkt.

Vorteile: Dieser Ansatz zeichnet sich besonders durch seine einfache und äusserst effiziente Möglichkeit zur polymorphen Abfrage aus. Es sind keine ***JOIN***s oder ***UNION***s notwendig, und verschiedene Kriterien für verschiedene Unterklassen können direkt in einer Anweisung kombiniert werden.

Indizes werden damit auch nur einmal verwaltet. Dies ist bei Anfragen positiv, da Abfragen über den Index nur einen Zugriff benötigen. Bei Schreibanweisungen hingegen müssen die Indizes auch für jene Attribute mitverwaltet werden, die gar nicht zu dieser Klasse gehören.

Wenn einmal die Infrastruktur implementiert ist, um Attribute für manche Zeilen "auszublenden", erlaubt das auch bei Spezialanforderungen Attribute nach anderen Kriterien als der Klasse zuzuordnen.

Nachteile: Dieser Ansatz verletzt alle strukturellen Normalformen. Um die Integrität der Datenbank aufrechtzuerhalten, sind daher besonders aufwändige Vorkehrungen notwendig. Eine mögliche ***CHECK*** Regel für die in Abbildung 5.4 gezeigten Felder sieht so aus:

```
CHECK (
    (class = 'a' AND text IS NOT NULL AND data IS NULL AND parent IS NULL)
```

```
    OR (class = 'p' AND text IS NULL AND data IS NOT NULL AND parent IS NULL)
    OR (class = 'f' AND text IS NULL AND data IS NULL AND parent IS NOT NULL)
)
```

Nimmt man eine gewisse Datenbankabhängigkeit in Kauf, kann man mit einer SQL-Funktion, die die Klassenzugehörigkeit prüft, den Ausdruck vereinfachen.

```
CREATE OR REPLACE FUNCTION inclass (which_class char, actual_class char, attr_value TEXT)
    RETURNS boolean
    LANGUAGE sql
    IMMUTABLE
    CALLED ON NULL INPUT
    AS 'SELECT ($1 = $2 AND $3 IS NOT NULL) OR ($1 <> $2 AND $3 IS NULL)';

-- Vereinfachte CHECK Regel
CHECK ( inclass('a', class, text) AND inclass('p', class, data) AND inclass('f', class, parent) )
```

Die Abhängigkeiten der Attribute von dem Unterscheidungsattribut werden zwar in der ***CHECK*** Regel überprüft, sind dort aber nicht mehr automtisiert verarbeitbar. Alle Anwendungen, die auf diese Tabelle zugreifen, benötigen daher externe Informationen über ihre Struktur.

Da dieser Ansatz alle Information in einer einzigen Tabelle zusammenfasst, ergeben sich einzigartige Einschränkungen, die zu beachten sind. Jedes Objekt reserviert Platz für alle Attribute der Hierarchie. Entsprechend benötigt dieser Ansatz auch mehr Speicherplatz als alle anderen. Beim Hinzufügen neuer Unterklassen oder Änderungen an Attributen ändert sich das Schema für alle eingetragenen Objekte. Bei der Benennung von Feldern müssen über die ganze Hierarchie eindeutige Namen gewählt werden. Standardwertregeln können im Datenbanksystem nur für Felder, die allen Klassen gemein sind, vergeben werden.

Einsatzgebiet: Bei besonders komplexen Objekthierarchien, und wenn viel öfter gesucht als geschrieben wird, spielt dieser Ansatz seine Stärken aus. Bei Klassenhierarchien mit vielen Unterklassen, die sich nur durch wenige Attribute unterscheiden, fällt der zusätzliche Platzverbrauch weniger ins Gewicht.

Hier einige beispielhafte Abfragen:

```
-- einfache Abfrage nach Primärschlüssel
SELECT * FROM data_objects WHERE id = ?

-- polymorphe Abfrage, Klassenattribute werden mitgeladen
```

```
SELECT * FROM data_objects
```

5.6.2 Reines Java

Auf Klassenebene unterstützt Java einfache Vererbung von Attributen und Methoden. Solche Ableitungsbäume können je nach Anforderungen auf entsprechende SQL Strukturen abgebildet werden.

Suchen und Laden

Um Programmtextduplikation zu vermeiden, implementiert *BaseDAL* eine Methode, um die *IDataObject* Attribute aus einem *RecordSet* zu laden:

```
public abstract class BaseDAL<OBJ extends DataObject> {
    protected void loadFieldsFrom(OBJ obj, RecordSet rs)
    {
        obj.setId(rs.getInt("id"));
        obj.setCreatedOn(new Date(rs.getTimestamp("created_on").getTime()));
        obj.setLastModified(new Date(rs.getTimestamp("last_modified").getTime()));
        obj.setTitle(rs.getString("title"));
    }
```

Die Klassen-spezifische Implementierung erweitert diese Funktionalität um die eigenen Attribute:

```
public class ArticleDAL extends BaseDAL<Article> implements IDAL<Article> {
    @Override
    protected void loadFieldsFrom(Article obj, RecordSet rs)
    {
        super.loadFieldsFrom(obj, rs)
        obj.setSummary(rs.getInt("summary"));
        /* weitere Article Attribute hier */
```

Diese Methode erfordert nur die passende Formulierung der SQL-Abfrage und ist bei allen Tabellenstrukturen anwendbar.

Bei der Tabelle-pro-Klasse Struktur kann mit einer zusätzlichen SQL-Abfrage pro Ableitungsebene die Koppelung innerhalb der Datenzugriffschicht reduziert werden. Dazu werden die einzelnen Tabellen in den jeweiligen DAL Klassen separat abgefragt und das Objekt von der Ableitungswurzel aus aufgebaut. Für einen *Article* sieht das so aus:

```
public class ArticleDAL extends BaseDAL<Article> implements IDAL<Article> {
    // Der obj Parameter hat bereits alle IDataObject Attribute ausgefüllt
    public Article fillObject(Article obj) throws SQLException
    {
        // Abfrage vorbereiten
        PreparedStatement stmt = _conn.prepareStatement(
            "SELECT␣*␣FROM␣articles␣WHERE␣id␣=␣?");
        stmt.setInt(1, obj.getId());

        ResultSet rs = stmt.executeQuery(); // ausführen

        obj.setSummary(rs.getInt("summary"));
        /* weitere Article Attribute hier */

        return obj;
    }
```

Schreiben

Da JDBC nur positionelle und keine benannten Parameter unterstützt, müssen bei ***INSERT***s und ***UPDATE***s spezielle Vorkehrungen getroffen werden, um die Feld- und Parameterlisten synchron zu halten.

Für den Tabelle-pro-Klasse Fall ist die Umsetzung sehr einfach: jedes Objekt schreibt seine Attribute in die jeweilige Tabelle, die entsprechenden Methoden müssen nur in der korrekten Reihenfolge aufgerufen werden, um die Datenbankkonsistenzbedingungen zu erfüllen, also von der Ableitungswurzel beginnend.

Für die beiden Schemata mit Tabellen für jede konkrete Klasse oder einer Tabelle für die gesamte Hierarchie kann das Schreiben ebenfalls nach Klassen getrennt werden. Im ersteren Fall muss dem Datenobjekt noch mitgeteilt werden, in welche Tabelle geschrieben werden soll. In beiden Fällen müssen Unterklassen dann nur noch ***UPDATE***s durchführen, da die entsprechende Zeile ja schon existiert. Transaktionsisolierung verhindert, dass halb-geschriebene Objekte von anderen Prozessen gesehen werden.

Durch zusätzlichen Aufwand auf der Javaseite kann ein Objekt in einer einzigen SQL-Anweisung in die Datenbank geschrieben werden.

5.6.3 SimpleORM

SimpleORM unterstützt keine komplexen Abbildungen zwischen Unterklassen und SQL-Tabellen. Speziell die Konvention, die eigentliche Struktur in statischen Konstanten abzulegen, läuft einer Ableitungshierarchie zuwider, da *SField*-Beschreibungen von gemeinsamen Attributen nicht mehr einer einzigen *SRecordMeta*-Instanz zugeordnet werden können.

Tabelle pro konkreter Klasse

Um das Tabelle-pro-konkreter-Klasse-Schema zu erhalten, kann die eigentliche Strukturdefinition in Singletons[4] ausgelagert werden. Diese bilden dann wieder die benötigte Ableitungshierarchie und haben damit jeweils eigene Definitionen der gemeinsamen Felder, wie das SimpleORM erfordert.

Im *DataObjectMeta* werden die gemeinsamen Attribute definiert. Der Konstruktor initialisiert die ***final*** *SField* Konstanten. Durch die Definition als ***protected*** können nur Instanzen abgeleiteter Klassen erzeugt werden. SimpleORM benötigt die konkrete Klasse der *SRecordInstance* zur Laufzeit, um neue Instanzen zu erzeugen. Über den Klassenparameter *MAIN* kann im Konstruktor eine passende Typbeschränkung für den *cls* Parameter formuliert werden, um Fehleingaben schon bei der Übersetzung zu verhindern.

```
public class DataObjectMeta<MAIN extends SRecordInstance>
{

    public final SRecordMeta meta;
    public final SFieldInteger ID;
    public final SFieldString TITLE;
    /* weitere gemeinsame Attribute */

    protected DataObjectMeta(Class<MAIN> cls, String tablename)
    {
        meta = new SRecordMeta(cls, tablename);
        ID = new SFieldInteger(meta, "id",
                SSimpleORMProperties.SFD_PRIMARY_KEY,
                SSimpleORMProperties.SGENERATED_KEY
                        .pvalue(new SGeneratorSequence(meta)));
        TITLE = // ...
```

[4] also Klassen mit genau einer Instanz pro Laufzeitumgebung

```
    /* weitere SFieldMeta Objekte erzeugen */
```

Als nächstes werden die spezifischen Attribute der Unterklassen definiert. Dazu wird eine Ableitung des *DataObjectMeta* angelegt, die die fehlenden Teile spezifiziert. Die Metaklassen für die anderen Artefakte werden analog definiert.

```
public class ArticleMeta extends DataObjectMeta<Article>
{
    public final SFieldString TEXT;
    public final SFieldString SUMMARY;

    public ArticleMeta()
    {
        super(Article.class, "articles");
        TEXT = // ...
        SUMMARY = // ...
```

Beim Einsatz in den *SRecordInstance*s bekommt die Elterninstanz *DataObject* die notwendige Metadefinition wieder im Konstruktor übergeben. Da hier ja nur die Artefaktattribute benötigt werden, braucht es hier auch keinen Klassenparameter.

```
public abstract class DataObject extends SRecordInstance implements IDataObject
{
    private final DataObjectMeta meta;

    protected DataObject(DataObjectMeta meta)
    {
        this.meta = meta;
        /* ab jetzt: Zugriff auf IDataObject Attribute via meta.meta */
```

Zuletzt wird im eigentlichen Geschäftsobjekt alles zusammengeführt. Die globale Instanz der Strukturdefinition *ArticleMeta* wird erzeugt und gespeichert. Wird eine neue Instanz erzeugt, so wird die übergeordnete Klasse mit dieser Definition initialisiert.

```
public class Article extends DataObject implements IArticle
{
    public Article()
    {
        super(meta);
    }

    static final ArticleMeta meta = new ArticleMeta();
```

```
/* ... */
```

Mit dieser Methode können weitere Artefaktarten ohne großen Aufwand hinzugefügt werden. Bereits implementierte Artefaktarten weiter ableiten erfordert jedoch Vorbereitungen in der Elternklasse, um die Strukturdefinition – wie beim *DataObject* – für jede Instanz getrennt zu setzen.

Tabelle pro Klasse

Ableitung kann auch mit $1 : 1$ Abbildungen simuliert werden. Die Artikelklasse von Abschnitt 5.1.3 mit einem Tabelle-pro-Klasse-Schema benötigt dafür zuerst eine Definition der gemeinsamen Attribute im *DataObject*. Diese folgt der Empfehlung von SimpleORM, wie das schon oben gezeigt wurde.

Die *Article* Klasse definiert anschließend nur noch die Artikel-spezifischen Attribute und verweist auf das zugrundeliegende *DataObject* mit einer *SFieldReference*. Die *getDataObject()* zeigt, wie auf die Instanz zugegriffen werden kann.

```
public class Article extends SRecordInstance implements IArticle
{

    static final SRecordMeta meta = new SRecordMeta(Article.class, "article_details");

    private static final SFieldReference DATA_OBJECT = new SFieldReference(meta,
            DataObject.meta, "do", SSimpleORMProperties.SFD_PRIMARY_KEY);

    private DataObject getDataObject() {
        return (DataObject)getReference(DATA_OBJECT);
    }

    public static final SFieldString TEXT = // ...
    public static final SFieldString SUMMARY = // ...
```

Über die in der *IArticle* Schnittstelle definierten *get-* und *set*-Methoden werden die Attributzugriffe den Gegebenheiten entsprechend implementiert. Dadurch bleibt die einheitliche Schnittstelle für die Benutzer erhalten.

5.6.4 Hibernate

```
1  <hibernate−mapping package="webbook.hibernate">
2      <class name="webbook.BasicDataObject" abstract="true">
3          <id name="id" column="id">
4              <generator class="native"/>
5          </id>
6
7          <property name="title" type="text" not−null="true"/>
8          <property name="createdOn" type="timestamp" not−null="true"/>
9          <property name="lastModified" type="timestamp" not−null="true"/>
10
11         <union−subclass name="Article" table="articles">
12             <property name="text" type="text" not−null="true"/>
13             <property name="summary" type="text" not−null="true"/>
14             <set name="tags" table="articles_tags">
15                 <key column="id"/>
16                 <element column="tag" type="webbook.hibernate.utils.TagType" not−null="true"/>
17             </set>
18         </union−subclass>
19
20         <!−− ... −−>
```

In Zeile 2 wird die abstrakte Basisklasse *BasicDataObject* abgebildet. Zuerst wird ein Schlüsselattribut (Zeile 3-5) deklariert, das über einen Datenbank-abhängigen Standardmechanismus *"native"* erzeugt wird. Anschließend werden die einzelnen Attribute, die allen Objekten gemein sind, angeschrieben (Zeile 7-9). Jede Deklaration enthält dabei den Namen der Java-Property[5] *name*, den Datenbanktyp *type* und eventuelle Einschränkungen in der Datenbank, hier *not−null*. Zusätzlich könnten hier auch von der Java-Property unabhängige Spaltenbezeichnung angeschrieben werden.

Ab Zeile 11 beginnt die Definition der *Article* Klasse. *union−subclass* erzeugt eine eigene Tabelle für jede konkrete Unterklasse. Die Attribute von *Picture* und *Folder* werden in eigenen *union−subclass* Elementen notiert.

Eine Besonderheit gegenüber den anderen Methoden ist die Möglichkeit, die Schlagworte mit *set* direkt als Menge abzubilden. Um die Abbildung der Schlagworte direkt auf die korrekte Klasse *webbook.utils.Tag* zu machen, steht hier als *type* eine benutzerdefinierte Klasse *webbook.hibernate.utils.TagType*, die *Tag*-Instanzen auf *text* Spalten abbildet.

[5] nach der JavaBean-Konvention mit *get*- und *set*-Methoden

5.7 Prozesskoordination

Da die meisten Applikationen mehrere Geschäftsfälle gleichzeitig abarbeiten, ist auch die Koordination zwischen Prozessen ein wichtiger Aspekt in der Implementierung eines Persistenzschemas, um nicht Anomalien beim Datenzugriff zu erzeugen.

5.7.1 SQL

Der SQL Standard schreibt nur eine implizite Koordination über die Isolationslevel der Transaktionen vor. Zur Unterstützung von alten Applikationen und um in komplexen Situationen Deadlocks zu vermeiden, unterstützen DBMS normalerweise auch explizite Koordination mit Tabellen- und Zeilensperren. Diese werden jedoch nicht vom Standard verlangt.

Transaktionen werden mit einer *BEGIN*-Anweisung gestartet. Mit ***COMMIT*** werden sie abgeschlossen und mit ***ROLLBACK*** verworfen.

Isolationslevels

Je nach Anforderung der Applikation und an eine gegebene Transaktion können mit ***SET TRANSACTION ISOLATION LEVEL*** unterschiedliche Isolationslevels angefordert werden sein.

READ UNCOMMITED: Der Zustand geringster Isolation. Parallele Transaktionen sehen alle bereits geänderten Daten unabhäging von dem Zustand der umgebenden Transaktion.

READ COMMITTED: Um zumindest grundlegende Zusicherungen an die gelesenen Daten einer Transaktion zu geben, sehen Transaktionen in diesem Isolationslevel nur noch Daten, die von bereits erfolgreich abgeschlossenen Transaktionen stammen.

REPEATABLE READ: Zusätzlich wird zugesichert, dass sich innerhalb einer Transaktion bereits gelesene Daten nicht mehr verändern. Das erlaubt immer noch, dass neue Datensätze bei Wiederholgen der gleichen Abfrage hinzukommen.

SERIALIZABLE: Jede Transaktion sieht nur Ergebnisse jener Transaktionen, die zu ihrer Eröffnung bereits abgeschlossen waren. Andere gerade laufende Transaktionen sehen Er-

gebnisse dieser Transaktion wiederum erst nach ihrem Abschluß. Kommt es zu Zugriffsüberschneidungen, muss eine der Transaktionen auf den Abschluß der anderen warten.

Der Standard erlaubt einer Implementierung auch strenger als das angeforderte Isolationslevel zu handeln. PostgreSQL zum Beispiel kennt nur ***READ*** *COMMITTED* und *SERIALIZABLE*. Die beiden anderen Levels werden jeweils wie die nächststrengere Stufe behandelt.

Explizite Sperren

Zusätzlich zu den Isolationslevels kennt SQL noch die *FOR* ***UPDATE*** Klausel, mit der die Ergebnisse einer ***SELECT***-Anweisung gesperrt werden können. ***UPDATE***, ***DELETE*** oder ***SELECT*** *FOR* ***UPDATE*** aus anderen Transaktionen blockieren bis zum Abschluß dieser Transaktion.

Ausserhalb des Standards implementieren manche DBMS noch explizite Tabellensperren. Damit können bei komplexen Transaktionen Deadlocks durch eine determinierte Sperrreihenfolge vermieden werden. In PostgreSQL wird dies mit dem *LOCK* ***TABLE*** Befehl durchgeführt.

Transaktionen

Auf Seite des DBMS kann durch intelligentere Transaktionskontrolle die Auswirkungen langer Transaktionen durch feinere Sperren – zum Beispiel zeilenweise statt auf Tabellenebene – verbessert werden. Ein anderes Verfahren ist die Versionierung der gesamten Datenbank. Dieses Verfahren wird zum Beispiel von PostgreSQL implementiert. [16] beschreibt das MVCC[6] Verfahren. Damit bleibt der aktuelle Zustand der Datenbank für die Dauer der Transaktion in einem virtuellen Schnappschuß erhalten. Dadurch kommt es zu keinen Behinderungen von parallelen Lese- und Schreibvorgängen auf den selben Daten. Allerdings muss dafür in Kauf genommen werden, dass bei möglicherweise auftretenden Schreibkonflikten Transaktionen abgebrochen und von neuem gestartet werden müssen.

5.7.2 Reines Java

Die Programmiersprache Java enthält einen Mechanismus zur Synchronisierung zwischen Ausführungssträngen einer einzelnen virtuellen Maschine. Das *synchronised* Schlüsselwort

[6] Multiversion Concurrency Control

markiert Methoden, die nicht gleichzeitig aus verschiedenen Strängen aufgerufen werden dürfen. Soll ein Objekt über mehrere Aufrufe hinweg gesperrt sein, kann ein Ausschlußbereich definiert werden.

```
void doSomething(Article a) {
    synchronized(a) {
        a.setTitle("Neuer␣Titel");
        a.setText("Neuer␣Text");
    }
}
```

Bei zwei gleichzeitigen Aufrufem von *doSomething* mit der selben *Article* Instanz sorgt die virtuelle Maschine dafür, dass einer der Stränge blockiert, bis der andere die Abarbeitung des Ausschlußbereichs beendet hat.

In *java.util.concurrent* befinden sich Werkzeuge für komplexere Ausschlußmechanismen und Datenstrukturen, die auch bei gleichzeitigem Zugriff von mehreren Ausführungssträngen konsistent bleiben. Eine detaillierte Beschreibung findet sich in der offiziellen Dokumentation bei [23].

JDBC

JDBC Verbindungen sind normalerweise im sogenannten *autocommit*-Modus. Dabei wird jede Anweisung in einer eigenen Transaktion ausgeführt, die sofort abgeschlossen wird. Wird dieser Modus mit *connection.setAutoCommit(**false**)* verlassen, kann die Transaktion manuell mit *connection.commit()* oder *connection.rollback()* abgeschlossen oder abgebrochen werden. Die JDBC Bibliothek sorgt dabei dafür, dass immer eine Transaktion offen ist. Daher wird keine *begin()* Methode benötigt.

5.7.3 SimpleORM

Die Transaktionen werden bei dieser Bibliothek mit den statischen Methoden *begin()*, *commit()* und *rollback()* der *SConnection* Klasse gesteuert. Dabei wird über interne Mechanismen sichergestellt, dass immer die richtige Verbindung für den gerade aktiven Kontext bearbeitet wird.

5.7.4 Hibernate

Hibernate kapselt die Transaktion in eine eigene Klasse, um unterschiedliche Implementierungen zu unterstützen. Eine Instanz, die die *Transaction* Schnittstelle implementiert, erhält man von einer *Session* mit *getTransaction()*. Die Schnittstelle bietet wiederum die schon bekannten *begin()*, *commit()* und *rollback()* Methoden an.

5.8 Leistungsorientiertes Programmieren

Die *Leistung* eines Systems ist eine heikle Angelegenheit. Zu lange Antwortzeiten bremsen die Produktivität der Benutzer und schrecken Kunden ab. Zu frühe oder zu intensive Konzentration auf die Optimierung der Anwendung bremst die Entwicklungsgeschwindigkeit bei immer geringer werdenden Renditen. Für den optimalen Einsatz von Resourcen bei der Optimierung ist es daher notwendig. die speziellen Anforderungen der Anwendung zu analysieren und dort zielgerichtet zu investieren.

5.8.1 SQL

Die Abfrage ***SELECT*** $*$ ***FROM*** *articles* ***WHERE*** *id* = *42* liest bei einer trivialen Implementierung die gesamten Daten von den Platten und durchsucht sie nach dem gewünschten Objekt mit der Nummer 42. Dieser Aufwand fällt zum Beispiel auch bei allen Operationen an, die die Eindeutigkeit des Primärschlüssels überprüfen müssen. Um diese großen Datentransfers zu vermeiden kann für häufig benutzte Spalten ein Index eingerichtet werden.

Indizes sind automatisch gepflegte Datenstrukturen, die von Attributwerten direkt auf Tupel(-adressen) abbilden. Je nach Implementierung werden verschiedene Datenstrukturen mit verschiedenen Fähigkeiten und Leistungskriterien angeboten. PostgreSQL unterstützt *B-tree* Indizes für Gleichheits- und Bereichsabfragen sowie *GiST*[7] Indizes für Benutzer-programmierte Erweiterungen. Im Gegensatz zum $O(n)$ Aufwand der trivialen Implementierung braucht die Abfrage eines *B-tree* Indexes typischerweise nur $O(\log n)$. Ist das Tupel im Index gefunden, können die Daten ohne weitere Suche geladen werden. Wird der gesuchte Wert im Index nicht gefunden, existiert auch kein passendes Tupel in der Tabelle.

[7] Generalized indexed Search Tree

Indizes kosten dafür in anderen Bereichen: zuerst einmal wird zusätzlicher Speicherplatz auf der Platte benötigt, um die Verwaltungsdaten zu speichern. Diese Daten brauchen dann auch Platz in den diversen Zwischenspeichern des Betriebssystems. Je nach Plattform ist die kleinste im Betriebssystem verwaltete Einheit zwischen zwei und acht Kilobyte groß. Für kleine Tabellen fressen diese zusätzlichen Daten jeden Leistungsgewinn des Indexes wieder auf.

Weiters muss der Index bei jeder Schreibanweisung gepflegt werden. Besonders beim Laden großer Mengen von Daten – Zurückspielen von Backups zum Beispiel – ist es daher empfehlenswert, Indizes zu deaktivieren und erst nach dem Laden auf einen Sitz neuzuberechnen. Bei Tabellen, in die laufend geschrieben wird, aber nur selten gelesen – Logbücher zum Beispiel – sind Indizes generell eher schädlich.

Denormalisierung

Abfragen auf redundanzfreien Schemata erfordern oft Verknüpfungen zwei oder mehrerer Tabellen, um die gewünschten Antworten zu erhalten. Zur Beschleunigung solcher Abfragen können oft angeforderte Daten über Tabellengrenzen hinweg verschoben werden, um Verknüpfungen zu vermeiden. Dabei ist darauf zu achten, dass auftretende Redundanzen durch Maßnahmen wie Triggermethoden oder Konsistenzprüfer möglichst automatisch und nahe der Datenbank gepflegt oder überwacht werden.

Eine andere Möglichkeit in diesem Bereich ist die Bildung und Pflege von oft benötigten Summen, zum Beispiel bei Rechnungen.

5.8.2 Reines Java

Die Leistung einer JDBC-Anwendung hängt stark davon ab, wie gut die zugrundeliegende Datenbank ausgenützt werden kann. Neben der schon angesprochenen Optimierung des Schemas und der Einführung passender Indizes, bewegt sich bei der Implementierung in Java das vor allem in der Reduktion der Kommunikation mit der Datenbank in Anzahl und Menge.

Vermeidung unnötiger Schreibvorgänge: Wird ein Attribut geändert, muss dieses nicht sofort in die Datenbank geschrieben werden. Änderungen müssen nur vor dem Abschluß der Transaktion zurückgeschrieben werden. Dabei können dann mehrere Änderungen in einer

Anweisung übertragen werden. Wird eine Transaktion abgebrochen, brauchen sie überhaupt nicht zurückgeschrieben werden.

Vermeidung unnötiger Attribute: Besonders bei Objekten mit vielen oder großen Attributen kann es sich lohnen, selten benutzte Attribute nicht mit dem ersten Zugriff auf das Objekt zu laden, sondern erst, wenn sie tatsächlich von der Anwendung benötigt werden.

Zwischenspeichern: Oft benötigte Objekte können in einem lokalen Zwischenspeicher abgelegt werden. Abfragen, die direkt aus dem Zwischenspeicher beantwortet werden können, benötigen so keine Kommunikation mit der Datenbank. Die Einhaltung der Transaktionsisolierung erfordert jedoch einigen Aufwand bei der Implementierung und beschränkt die Wirksamkeit des Zwischenspeichers.

Stapelverarbeitung: Einer der größten Vorteile der Implementierung ohne zusätzliche Bibliotheken ist die direkte Einsetzbarkeit des gesamten SQL-Umfangs. So ist es zum Beispiel viel effizienter, große Datenmengen direkt in der Datenbank zu kopieren oder umzuformen, als die selben Daten in die Javaanwendung zu laden und dann wieder zurückzuschreiben. Dabei wird jedoch die gesamte Geschäftslogik im Java-Objektmodell umgangen, besondere Vorsicht ist daher geboten.

Mit JDBC 1.2 wurde die Möglichkeit geschaffen, mehrere Datenbankanweisungen gleichzeitig an die Datenbank zu übertragen. Das reduziert zwar den Kommunikationsaufwand, erfordert jedoch anwendungsseitig Anpassungen.

5.8.3 SimpleORM

Vermeidung unnötiger Schreibvorgänge: SimpleORM schreibt Objekte erst beim *commit()* in die Datenbank. Da SimpleORM immer direkt die Datenbank abfragt, ist es manchmal notwendig, innerhalb einer Transaktion Objekte vorzeitig zurückzuschreiben, damit es zu keinen Anomalien kommt. Um das auszulösen hat jede *SRecordInstance* eine *flush()* Methode.

Vermeidung unnötiger Attribute: SimpleORM bietet drei Stufen der Wichtigkeit von Feldern an. *SFD_DESCRIPTIVE* markiert Felder, die eine – für den Benutzer signifikante – Be-

schreibung der Instanz darstellen. Werden für eine Überblicksliste oder ähnliches nur diese Felder benötigt, kann eine Abfrage für nur diese Felder mit *SQY_DESCRIPTIVE* abgesetzt werden.

Für große oder selten benutzte Felder ist die *SFD_UNQUERIED* Markierung gedacht. So markierte Felder werden nur dann geladen, wenn die Abfrage auch die entsprechende *SQY_UNQUERIED* Markierung trägt.

Zwischenspeichern: Diese Bibliothek implementiert einen Zwischenspeicher für Objekte. Um nicht die Isolationszusicherungen der Datenbank zu verletzen, werden Instanzen nur für die Dauer einer Transaktion gespeichert.

Der Zwischenspeicher wird auch genutzt, um die Objektidentät an die Datenbankidentät zu knüpfen: Wird das selbe Tupel innerhalb einer Transaktion mehrmals abgefragt, wird in Java immer die selbe Objektinstanz zurückgegeben.

Stapelverarbeitung: SimpleORM nutzt nicht die Möglichkeiten von JDBC zur Stapelverarbeitung.

5.8.4 Hibernate

Vermeidung unnötiger Schreibvorgänge: Wie SimpleORM schreibt Hibernate so spät wie möglich zurück in die Datenbank und muss in Grenzfällen manuell ausgelöst werden.

Vermeidung unnötiger Attribute: Einzelne Attribute können in Hibernate zu *Komponenten* (von engl. "components") gruppiert werden. Diese Komponenten werden dann auf eigene Objekte abgebildet. Über HQL Abfragen ist es dann möglich, diese Komponenten einzeln abzurufen. Dabei ist zu beachten, dass in der Abbildung der Komponente ein <*parent*/> Attribut eingefügt wird, um von der Komponente auf das Gesamtobjekt zu kommen. Für Details siehe [7], Kapitel 8.1.

Zwischenspeichern: Die *Session* implementiert einen Zwischenspeicher auf Transaktionslevel. Zusätzlich kann ein anwendungsweiter Zwischenspeicher (engl.: second-level cache) eingerichtet werden, in dem Objekte über Transaktionen und Ausführungsstränge hinweg gepuf-

fert werden. Dieser Zwischenspeicher sieht jedoch nur Änderungen, die von Hibernate selbst durchgeführt werden. Im Einsatz kann die Korrektheit daher nur durch externe Maßnahmen sichergestellt werden. Die verschiedenen Implementierungen und Beschränkungen werden in [7], Kapitel 19.2 aufgelistet.

5.9 Leistungsvergleich Beispieldatenbank

Für einen empirischen Vergleich der Bibliotheken mit der manuellen Implementierung wurden vier synthetische Benchmarks implementiert, die die verschiedenen Komponenten testen. Alle Tests wurden auf einem Samsung M40 Laptopp mit einem Intel Perntium M 1.80GHz mit 1GB RAM und einer FUJITSU MHT2080AT 80GB Platte durchgeführt. Für die Implementierung der Benchmarks wurde das Japex (siehe [19]) Framework eingesetzt.

Verbindungsaufbau: Der erste Benchmark öffnet nur eine Verbindung zur Datenbank und schliesst diese wieder. Damit werden die grundlegenden Kosten einer Implementierung gemessen. Alle Projekte brauchen hier circa 4 ms. Während die Implementierung in reinem Java und Hibernate jeweils rund drei MB Speicher brauchen, kommt SimpleORM mit nur 2.5 MB aus.

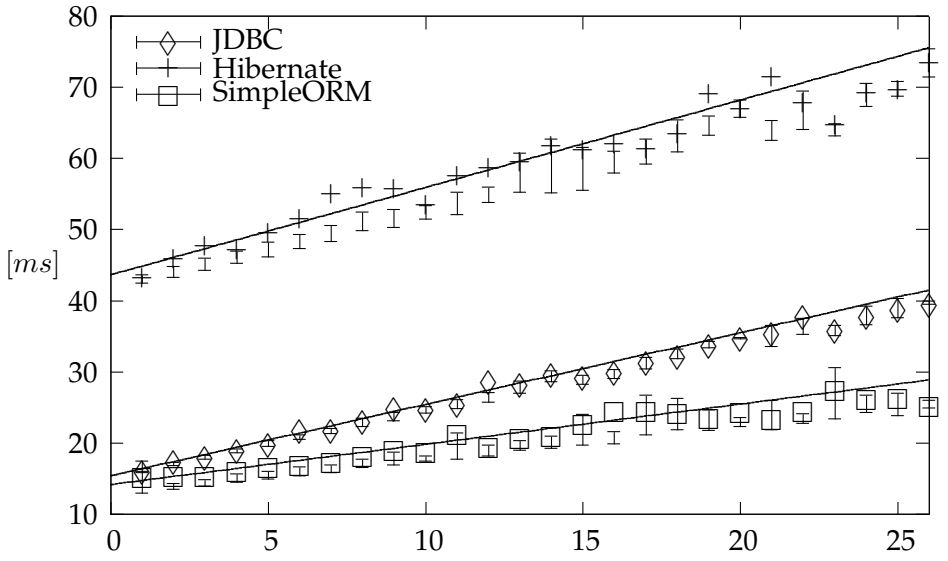
Verbindungsaufbau	ms / Verbindung	σ	max. Speicher
Manuell	3.8 ms	0.2%	2965.0 kB
SimpleORM	3.7 ms	2.1%	2450.6 kB
Hibernate	4.0 ms	1.7%	2912.0 kB

Schlüsselzugriff: Hier wird ein kleines Objekt nach seinem Primärschlüssel geladen. Die angegebenen Zeiten sind die Differenz zwischen dem letzten Benchmark und diesem.

Schlüsselzugriff	ms / Objekt	σ	max. Speicher
Manuell	3.0 ms	0.1%	5188.6 kB
SimpleORM	2.9 ms	1.4%	4484.9 kB
Hibernate	2.6 ms	2.6%	5252.3 kB

Bei diesen beiden Tests gibt es keine großen Überaschungen. Beide sind großteils von der Geschwindigkeit des JDBC Treibers und der Datenbank abhängig. Auffällig ist die geringere Varianz der manuellen Implementierung und die Tatsache, dass SimpleORM bei den Speicherspitzen rund 500 Kilobyte weniger verbraucht.

Artikel laden: Eine Auswahl von Artikeln wird über eine Oder-Verknüpfung zweier zufälliger Tags geladen. Die Artikel sind Emails aus dem Archiv einer Mailingliste. Als Tags wurden die einzelnen Worte der Betreffzeile gesetzt. Die Graphik zeigt Dauer der Transaktion über der Anzahl gefundener Artikel. Ebenfalls eingezeichnet ist ein lineares Modell. Die gefunden Parameter sind anschliessend aufgelistet.

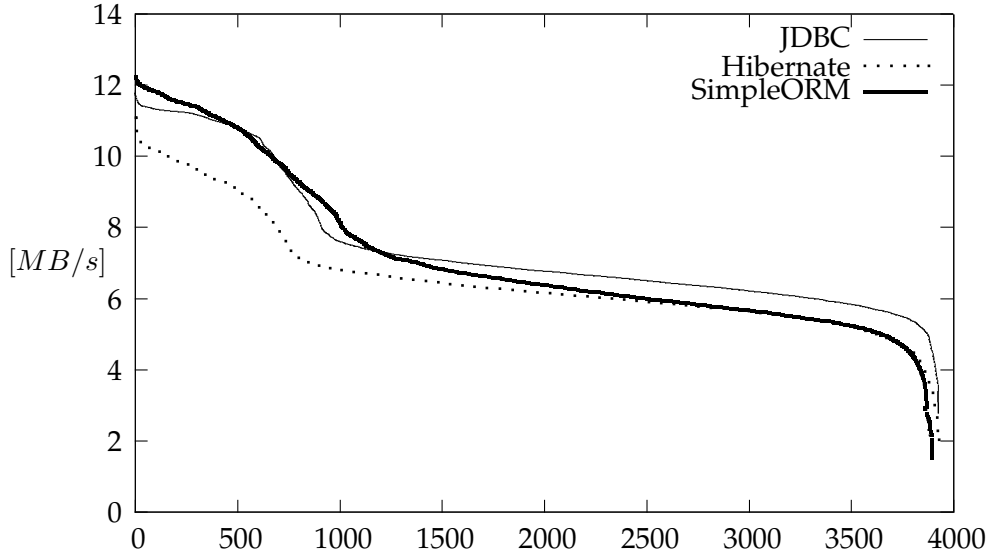


Artikelabfrage	ms / Verbindung	ms / Artikel
Manuell	15.4 ms	1.0 ms
SimpleORM	14.2 ms	0.5 ms
Hibernate	43.6 ms	1.2 ms

Bilder laden: Eine Auswahl von Bildern wird über Tags geladen. Um die Ergebnisse vergleichbar zu halten, haben alle Bilder den gleichen Inhalt. Die Struktur der Daten ist der Ordnerstruktur eines privaten Bilderalbums entnommen. Zusätzlich wurde jedes Bild mit einem – zufällig aus elf weiteren Tags gewählten – Tag markiert. In der Verknüpfung wird jeweils ein Tag aus der Ordnerstruktur und eines der zusätzlichen Tags Und-verknüpft.

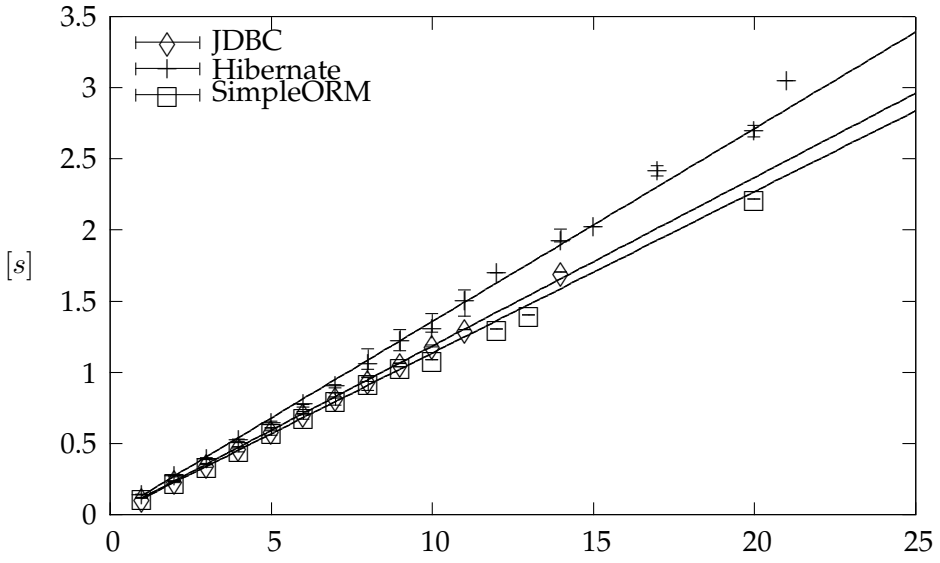
Die gesamte Bilddatenbank umfasste rund 1.4 GB an Bilddaten, passte daher nicht vollständig in den Arbeitsspeicher des Testgeräts. Daher verursachten die Pufferspeicher der Datenbank und des Betriebssystems ausgeprägte Leistungsunterschiede, je nachdem ob diese Puffer für eine Abfrage genutzt werden konnten oder nicht.

Zuerst eine Darstellung dieser Puffereffekte. Die folgende Graphik zeigt die einzelnen Durchgänge sortiert nach der Übertragungsgeschwindigkeit.

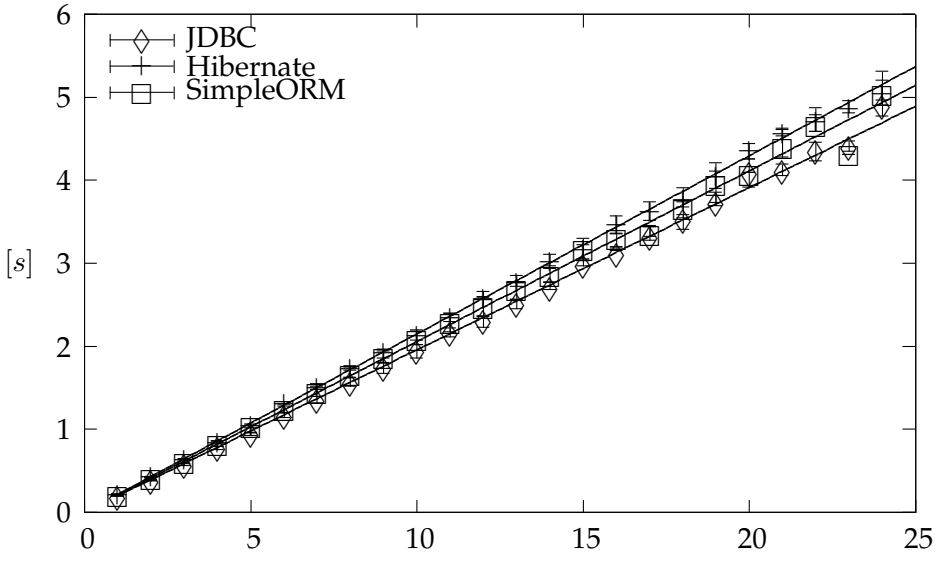


Deutlich sichtbar sind die Spitzeneffekte der Pufferspeicher auf der linken Seite der Gra-

phik. In der Mitte folgt ein breiter Bereich von Abfragen, die ausserhalb des Pufferspeichers arbeiten. Die folgende Graphik zeigt die Antwortzeit über die Anzahl der gefundenen Bilder, allerdings beschränkt auf die 500 Transaktionen mit der höchsten Übertragungsgeschwindigkeit. Damit wird die Leistung des Pufferspeichers des Betriebssystems gezeigt und wie gut dieser von der Implementierung ausgenutzt werden kann.



Die durchschnittliche Leistung der Implementierungen zeigt die folgende Graphik, mit dem zentralen Drittel der Transaktionen.



Abschliessend eine tabellarische Übersicht der Puffereffekte:

	ms / Bild	ms / Bild		
Bilderabfrage	sonst	Puffer	% Gewinn	max. Speicher
Manuell	195.6 ms	118.4 ms	39.6%	128.2 MB
SimpleORM	205.7 ms	113.5 ms	45.0%	96.2 MB
Hibernate	214.7 ms	135.6 ms	36.9%	290.1 MB

5.10 Programmieraufwand

Ein konsistentes Maß für Programmieraufwand ist die Anzahl von Zeilen im Quelltext. Hier die Daten[8] für die drei Projekte – ohne die zusätzlichen Tests.

SLOC	Projekt	nach Programmiersprache
970	JDBC	java=763,sql=207
895	SimpleORM	java=895
519	Hibernate	java=453,xml=66

Dabei wird natürlich der nicht-meßbare Aufwand des Einlernens in die Bibliotheken nicht mitgerechnet. Dabei handelt es sich aber um konstante Aufwendungen, die nur bei Projekten mit geringen Anforderungen ins Gewicht fallen.

Bei der Architektur der JDBC-Implementierung ist kein solcher Basisaufwand zu verzeichnen. Dafür muss jedoch bei steigenden Anforderungen in diesem Bereich weiter programmiert werden. Zum Beispiel unterstützt die Pufferimplementierung keine parallelen Transaktionen.

Darüber hinaus existieren für Hibernatae noch eine Reihe an Tools für die Integration in den Entwicklungsablauf. Zum Beispiel enthält *Hibernate Tools* (siehe [8]) ein Eclipse Plugin zur automatischen Erzeugung der Datenklassen aus der Hibernate XML Definition.

[8] erzeugt mit David A. Wheelers "SLOCCount"

Kapitel 6

Objekt-orientierte Datenbanksysteme

1990 – in den Hochtagen der OODBMS Entwicklung – hat sich eine Gruppe führender Forscher dieses Gebietes zusammengetan, um das "Object Oriented Database System Manifesto"[5] zu schreiben um dem kommerziell getriebenen Hype um OODBMS' mit einem "Lakmustest" der Objektorientierung entgegenzutreten. Auch sollte damit ein einheitliches Vokabular für die weitere Diskussion geschaffen werden. Als kommerzielles Gegengewicht wurde mit [2] das "Third-Generation Database System Manifesto" geschrieben, um den Notwendigkeiten des Einsatzes solcher Systeme außerhalb der akademischen Welt festzuhalten.

Um dieses Spannungsfeld zu beleuchten, zuerst eine Klassifizierung der drei Implementierungen nach [5], sowie eine Diskussion der zusätzlichen Anforderungen von [2].

6.1 Die Goldenen Regeln

Die folgenden dreizehn Kriterien wurden von den Autoren als unbedingt notwendig identifiziert, um ein objektorientiertes Datenbanksystem zu charakterisieren.

6.1.1 Thou shalt support complex objects

Dieses Gebot fordert neben den üblichen atomaren Typen (Zahlen, Zeichenketten, Wahrheitswerte) auch zusammengesetzte Typen, zumindest Mengen, Listen, und Tupel. Diese werden ja bereits von Java in Form von Collections, Arrays und Klassen bereitgestellt und sind daher für alle Implementierungen gleichermaßen verfügbar.

6.1.2 Thou shalt support object identity

Alle drei Systeme unterstützen Primärschlüssel als Objektidentitätsträger.

SimpleORM beschränkt das auf eine Transaktion, um die Isolierung auch zwischen Transaktionen in einer virtuellen Maschine zu gewährleisten. Hibernate implementiert in der *Session* ähnliche Mechanismen, um das gleiche zu erreichen. In der reinen Javaversion habe ich mich auf eine globale *Registry* beschränkt, um wiederholbare Lesevorgänge zu ermöglichen. Mehrere parallele Transaktionen in einer VM werden damit zwar nicht unterstützt, dem Gebot ist jedoch Genüge getan.

6.1.3 Thou shalt encapsulate thine objects

Wie bei der Unterstützung komplexer Objekte kann dieses Gebot durch die korrekte Anwendung der Javamechanismen erfüllt werden. Alle drei gezeigten Beispiele implementieren die selbe Schnittstelle, die keinen Zugriff auf die Interna zulässt und sind also gleichauf.

6.1.4 Thou shalt support types or classes

Auch dieses Gebot wird durch den Einsatz von Java ohne Probleme erfüllt.

6.1.5 Thine classes or types shalt inherit from their ancestors

Die reinen Javaversion nutzt zwar Vererbung, um die Objektseite des Beispieles zu modellieren. Auf der *DAL* Seite ist die Verteilung der Verantwortlichkeiten des Datenbankzugriffes jedoch zu komplex, um sich für so ein kleines Projekt zu lohnen. Im Falle einer Weiterentwicklung könnten entsprechende Mechanismen nachgereicht werden. Eine Möglichkeit dazu wäre das Weiterreichen von SQL-Teilen und *ResultSet*s durch die Ableitungshierarchie. Damit werden die einzelnen Klassen natürlich stark gekoppelt, die Performance bleibt jedoch hoch, da nur eine minimale Anzahl an Anweisungen an die Datenbank abgesetzt wird. Eine andere Möglichkeit ist die autonome Verwaltung von Attributen in den jeweiligen Klassen. Dadurch wird eine starke Koppelung zwischen den Klassen vermieden, jede Ebene in der Ableitungshierarchie muss jedoch eigene Anweisungen an die Datenbank stellen.

Hibernate unterstützt die Abbildung von beliebigen Ableitungshierarchien auf fast beliebige Datenbankstrukturen. Die Implementierung auf der SQL Seite kann damit flexibel an die

Anforderungen der Applikation angepasst werden.

Entsprechend der SimpleORM Philosophie wird keine Ableitungsart speziell unterstützt. Trotzdem können – wie in Abschnitt 5.6 beschrieben – auch mit SimpleORM Hierarchien flexibel abgebildet werden.

6.1.6 Thou shalt not bind prematurely

Gemeint ist dabei Methodenüberladung und späte Bindung. Wiederum ein Gebot, das schon aufgrund der Nutzung von Java erfüllt wird.

6.1.7 Thou shalt be computationally complete

Von einer Programmiersprache wie Java offensichtlich erfülltes Gebot. Die Autoren selbst weisen schon darauf hin, dass dieses Gebot sinnvollerweise durch die Anbindung an eine Programmiersprache erfolgen kann und nicht unbedingt eine neue Datenbanksprache erfordert.

6.1.8 Thou shalt be extensible

Dieses Gebot fordert, dass es keine Unterscheidung zwischen system- und benutzerdefinierter Typen gibt. Speziell sollten eventuelle Unterschiede in der Unterstützung benutzerdefinierter Typen durch das System vor der Applikation und dem Programmierer versteckt sein.

Da innerhalb eines Werkzeugs alle Objekte gleichberechtigt und benutzerdefiniert sind, kann dieses Gebot auch als erfüllt betrachtet werden.

6.1.9 Thou shalt remember thy data

Dieses, auch für heutige Programmiersprachen, ungewöhnliches Gebot fordert, dass beliebige Objekte unabhängig von ihrem Typ und ohne explizite Übersetzung gespeichert werden können.

Java alleine erfüllt dies natürlich nicht. Beschränkt man die Betrachtung jedoch wie beim letzten Gebot auf die implementierten persistenten Typen, so kann auch dieses Gebot als erfüllt betrachtet werden. Die zugrundeliegende Forderung nach automatischer globaler Persistenz wird jedoch von keiner der Implementierungen erfüllt.

6.1.10 Thou shalt manage very large databases

Um große Datenmengen effizient verwalten zu können, werden hier analog zu den relationalen Datenbanken transparentes Indexmanagement, Datengruppierungen (Clustering), Pufferspeicher, Zugriffspfadauswahl und Abfrageoptimierungen verlangt.

Dieses Gebot ist auf mehreren Ebenen zu beurteilen. Am einfachsten nach den Buchstaben des Manifestes. Dort wird – 1990, einige Jahre vor dem ersten PC – nur die komplette Kapselung der physikalischen Ebene (Speicherverwaltung, Indexstrukturen, Plattenzugriffe) von der logischen Ebene der Programmierer getrennt. Durch die Verwendung einer SQL Datenbank als Basis ist das offensichtlich und mit einer hohen Implementierungsqualität gewährleistet. Es kann jedoch auch nicht geleugnet werden, dass die Menge an individuellem und vor allem stark leistungsbeeinflussenden Design und Programmierung innerhalb der Datenobjekte nicht dem Geiste dieses Gebots entspricht. Die Flexibilität auf der Javaseite ist zwar notwendig um auch in Extremfällen akzeptable Leistung zu erhalten, sie ist aber nicht, wie gefordert, komplett unsichtbar.

Es liegt jedoch nahe, dass gerade die korrekte Auswahl eines der in 5.6 beschriebenen Implementierungsschematas – ebenso wie die korrekte Auswahl von Indexarten – aufgrund der Abhängigkeit von der Abfragelast auf lange Zeit hinweg nicht automatisierbar sein wird.

6.1.11 Thou shalt support concurrent users

Auf Datenbankebene unterstützen dies alle Methoden. Innerhalb einer virtuellen Maschine kann das jedoch nur von den Bibliotheken mit einem Session-beschränkten Pufferspeicher gewährleistet werden.

6.1.12 Thou shalt recover from hardware and software failures

Transaktionen auf dem Objektmodell werden in allen Implementierungen direkt auf Datenbanktransaktionen abgebildet. Damit können dort alle tradionellen und etablierten Sicherungsmassnahmen greifen.

6.1.13 Thou shalt have a simple way of querying data

Einzig Hibernate bringt seine eigene Abfragesprache mit. Für sie existiert jedoch zur Zeit kein allgemein verfügbares Benutzerinterface. Alternativ kann in allen Fällen direkt mit SQL auf die zugrundeliegende Datenbank zugegriffen werden. Da dies eine Verletzung der Kapselung darstellt, sollte man sich hier aber auf Lesezugriffe beschränken.

6.2 Details

Die folgenden Eigenschaften sind signifikante Verbesserungen der Implementierungsqualität, aber – wie zum Beispiel mehrfache Vererbung – schon ausserhalb des unbedingt Notwendigen.

Die ersten beiden Forderungen, mehrfache Vererbung sowie Typüberprüfungen und -inferenz, werden von Java nicht oder nur teilweise zur Verfügung gestellt und sind daher von keiner der Implementierungen erfüllt.

6.2.1 Verteilung

Schon die grundlegende Architektur erlaubt die Trennung von virtueller Maschine und Datenbank. Weitere Verteilung kann durch die parallele Ausführung mehrerer virtueller Maschinen erreicht werden. Diese können sich ja weiterhin über die Datenbank synchronisieren. Bei der Verteilung der Datenbank kann es je nach eingesetzter Technologie jedoch zu zusätzlichen Programmieranforderungen auf der Javaseite kommen (zB Trennung von Schreib- und Lesepfaden).

6.2.2 Langlaufende Transaktionen

Die reine Javaimplementierung hat keine besondere Unterstützung für lang laufende Transaktionen. Lediglich die vom verwendeten Datenbanksystem zur Verfügung gestellten Mechanismen (zB *CHECKPOINT* von Postgres) könnten verwendet werden. Sollte dies nicht ausreichen, können zusätzlich optimistische Sperrstrategien (siehe Abschnitt 3.6.2) eingesetzt werden, um Objekte von Datenbanktransaktionen zu entkoppeln.

Hibernate und SimpleORM unterstützen beide das Auskoppeln von Objekten aus einer Transaktion. Dadurch werden die Objekte serialisierbar – sie haben ja keine Isolierungsanfor-

derungen mehr – und können zum Beispiel in der *Session* eines Webservers gelagert werden. Werden die Objekte wieder in eine neue Transaktion eingekoppelt und Änderungen in die Datenbank zurückgeschrieben, so wenden beide Bibliotheken Methoden aus dem Bereich der optimistischen Sperren an, um Schreibkonflikte zu bemerken und Datenschäden zu vermeiden.

6.2.3 Versionskontrolle

Keine der vorgestellten Produkte unterstützt Versionskontrolle der Objekte oder des Datenschemas.

6.3 Offene Parameter

Im letzten Abschnitt stellen die Autoren jene Eigenschaften dar, über die damals noch kein Konsens vorhanden war. Sie bilden die Freiheitsgrade für Implementierungen. Es handelt sich dabei um

Programmierparadigma logische (zB Prolog), funktionale (zB Haskell) oder imperative (zB Java) Programmierung

eingebaute Typen Es wird zwar in Abschnitt 6.1.1 eine minimale Menge an eingebauten Typen vorgestellt. Einer Implementierung steht es jedoch frei, diese beliebig zu erweitern.

Typsystem Generische Typen, Templates, Typeinschränkungen und vieles mehr.

Vereinheitlichung In wie weit Methoden und Typen Objekte sind.

Alle dieser Eigenschaften werden durch die Wahl von Java als Implementierungssprache festgelegt. Ein Konsens ausserhalb einzelner Programmiersprachen ist jedoch nicht abzusehen.

6.4 Datenbanken der "'Dritten Generation"'

In [2] berichten Autoren mit starkem industriellen Hintergrund[1] über ihre Erfahrungen mit der Weiterentwicklung relationaler Systeme und stellen die These auf, dass die wichtigsten

[1] IBM, DEC, Oracle

objektorientierten Eigenschaften dadurch genausogut oder besser erreicht werden können.

Speziell im Bereich des Datenbankzugriffes und der Einbettung in Programmiersysteme, wo [5] nur einen einfachen Weg zur Datenabfrage fordern, wird [2] wesentlich detaillierter.

6.4.1 Randbedingungen und Trigger

[2] prophezeit einen Wachstumsschub bei der Anwendung von Datenbank-basierter Überprüfung von Klasseninvarianten und Geschäftsregeln. Aktuelle DBMS inkludieren dafür auch weitgehende Unterstützung. Zum Beispiel Oracle mit integrierter Java Umgebung, der MS-SQL Server mit .NET Triggermethoden oder PostgreSQL mit der Möglichkeit zur Einbettung von Java, R, Ruby und Shellskripten. Besonders in kleineren Projekten ist jedoch die höhere Komplexität der Verteilung von Quelltext auf mehrere Komponenten den erwarteten Gewinnen gegenüberzustellen.

In vielen Fällen haben auch Applikationsserver die Integration der Geschäftslogik übernommen. Die SQL-Datenbank wird in diesem Modell nur noch als verlässlicher, leistungsfähiger Datenspeicher benutzt.

Kapitel 7

Zusammenfassung

Reines SQL ist immer noch unübertroffen in Kompaktheit und reiner Abfrageleistung. Im Gegensatz zur ursprünglichen Absicht, SQL als Benutzerschnittstelle anzubieten, ist die Text- und Anweisungs-orientierte Arbeitsweise nicht mehr Endnutzer-tauglich. Moderne Anwendungen – sei es auf Web- oder Desktopbasis – bauen jedoch auf objekt-orientierten Sprachen und Bibliotheken auf. Die besprochene Kluft zwischen den leistungsfähigen – aber inflexiblen – relationalen Systemen und den Objektmodellen kann sowohl durch selbstprogrammierte Übersetzungsfunktionen als auch durch Fremdprodukte überbrückt werden.

Die vorgestellten Bibliotheken – SimpleORM und Hibernate – zeigen dabei ganz unterschiedliche Ansätze, wodurch sie in unterschiedlichen Situationen ihre Stärken ausspielen können.

SimpleORM ist eine leistungsstarke Bibliothek, die dem Programmierer die grundlegende Kontrolle über die Datenbank aus der Hand nehmen. In der Abwägung zwischen Funktionsumfang und Benutzbarkeit zielt SimpleORM auf den erfahrenen JDBC-Programmierer, der sich die mechanische Arbeit des Datenbankzugriffes erleichten möchte ohne die Möglichkeiten der Stapelverarbeitung zu verlieren. Die knappe, präzise Dokumentation in [6] beschreibt in rund 17 Seiten die gesamte Programmierschnittstelle und die wichtigsten Eckpfeiler der zugrundeliegenden Designphilosophie.

SimpleORM eignet sich besonders für Projekte mit einfachen Datenstrukturen und hohen Anforderungen an den Datendurchsatz. Komplexere Beziehungen zwischen Daten – wie die

Hierarchie in der Beispieldatenbank – müssen selbst entworfen werden, stellen aber aufgrund der hohen Flexibilität der Bibliothek keine besonderen Hindernisse dar.

Hibernate baut zwar auch auf JDBC auf, um sich zur Datenbank zu verbinden, bietet aber die gesamte Datenzugriffsfunktionalität selbst an. Durch diese Kapselung kann Hibernate einen wesentlich intensivere Unterstützung der Datenbankkommunikation anbieten, erfordert aber ebenso eine intensivere Beschäftigung mit Hibernate selbst. Die Entkoppelung der Datenbank von den Geschäftsobjekten durch die XML-Abbildungsbeschreibung erlaubt auch die getrennte Weiterentwicklung von Anwendung und Datenbank. Die Hibernate Referenz [7] beschreibt auf über 200 Seiten die gesamte Programmierschnittstelle und das XML Schema der Abbildungsbeschreibung.

Hibernate eignet sich besonders für Projekte mit komplexen Datenstrukturen oder nichttrivialen Datenbanksituationen. Hier wiederum im Speziellen ist der Zugriff auf Alt-Systeme hervorzuheben, da der Datenzugriff auf beliebige Schemaformen erfolgen kann. Der Datendurchsatz von Hibernate blieb zwar hinter den anderen Bibliotheken zurück, dafür gibt es allerdings eine breite Palette an Optimierungsparametern und Zusatzprodukten zur Leistungssteigerung. Deren Konfiguration und Einsatz lohnt sich ebenfalls nur für Projekte mit entsprechenden Anforderungen.

SimpleORM als auch Hibernate erfüllen im Zusammenspiel mit der Implementierungssprache die meisten Anforderungen, die an objekt-orientierte Datenbanksysteme gestellt werden. Gegenüber der händischen Implementierung mit JDBC werden damit respektable Einsparungen im Bereich des Programmieraufwandes und der Komplexität geboten, dem gegenüber steht eine Investition in das Erlernen der Bibliothek.

Literaturverzeichnis

[1] *DCE 1.1: Remote Procedure Call*. Berkshire, UK, 1997.

[2] ADVANCED DBMS FUNCTION, THE COMMITTEE FOR: *Third-generation database system manifesto*. SIGMOD Rec., 19(3):31–44, 1990.

[3] ALBRECHT, CHRISTINE: *Folksonomy*. Diplomarbeit, Technische Universität Wien, März 2006.

[4] ANDERSEN, LANCE: *JDBC 4.0 Specification — JSR 221*. Sanda Clara, California, 2006.

[5] ATKINSON, MALCOM, DAVID DE WITT, DAVID MAIER, FRANÇOIS BANICILHON, KLAUS DITTRICH und STANLEY ZDONIK: *The Object-Oriented Database System Manifesto*. In: KIM, WON, JEAN-MARIE NICOLAS und SHOJIRO NISHIO (Herausgeber): *Building an Object-Oriented Database System*, Seiten 223–240, North-Holland, 1990. Elsevier Sience Publishers B.V.

[6] BERGLAS, DR. ANTHONY: *SimpleORM White Paper*, 2005.

[7] HIBERNATE.ORG: *Hibernate Reference Documentation*. Version 3.2.4.

[8] HIBERNATE.ORG: *Hibernate Tools*.

[9] IEEE STANDARDS COMMITTEE 754, Institute of Electrical and Electronics Engineers, New York: *IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985*, 1985. Reprinted in SIGPLAN Notices, 22(2):9-25, 1987.

[10] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION: *ISO/IEC 9075:1992: Title: Information technology — Database languages — SQL*. International Organization for Standardization, Geneva, Switzerland, 1992.

[11] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION: *ISO/IEC 9075-1:1999: Title: Information technology — Database languages — SQL — Part 1: Framework*. International Organization for Standardization, Geneva, Switzerland, 1999.

[12] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION: *ISO/IEC 9075-1:2003: Title: Information technology — Database languages — SQL — Part 1: Framework*. International Organization for Standardization, Geneva, Switzerland, 2003.

[13] JAROSCH, HELMUT: *Datenbanken: Eine beispielhafte Einführung für Studenten und Praktiker*. Vieweg, Braunschweig/Wiesbaden, 2002.

[14] KENT, WILLIAM: *A simple guide to five normal forms in relational database theory*. Commun. ACM, 26(2):120–125, 1983.

[15] NIEMANN, CHRISTOPH: *Datenbankfähige Client/Server Anwendungen: Generierung aus OOA-Modellen*. Forschung in der Softwaretechnik. Spektrum Akademischer Verlag, Heidelberg; Berlin, 2000.

[16] THE POSTGRESQL GLOBAL DEVELOPMENT GROUP: *PostgreSQL 8.1.8 Documentation*.

[17] ROFF, JASON T.: *UML: Das mitp Standardwerk zur professionellen Softwareentwicklung*. IT Studienausgabe. mitp-Verlag, Bonn, 2004.

[18] SAAKE, GUNTER und KAI-UWE SATTLER: *Datenbanken & Java: JDBC, SQLJ und ODMG*. iX Edition. dpunkt Verlag, Heidelberg, 2000.

[19] <SANTIAGO.PERICASGEERTSEN@SUN.COM>, SANTIAGO PERICAS-GEERTSEN: *Japex – Version 1.0.26*, 2006.

[20] SUN MICROSYSTEMS, Mountain View: *Java 2 Platform Standard Ed. 5.0*.

[21] SUN MICROSYSTEMS, Mountain View: *JavaBeans API Specification*, August 1997. Version 1.01-A.

[22] SUN MICROSYSTEMS, Palo Alto, California: *Java Object Serialization Specification*, 2003. Version 1.5.0.

[23] SUN MICROSYSTEMS, INC.: *Concurrency Utilities*, 2004. Abgerufen am 16. 5. 2007.

[24] VETTER, MAX: *Objektmodellierung: eine Einführung in die objektorientierte Analyse und das objektorientierte Design*. Leitfäden der Informatik. Teubner, Stuttgart, 1995.

Abbildungsverzeichnis