Universität Leipzig
Fakultät für Informatik und Mathematik
Institut für Informatik

Entwurf und Implementierung einer Java-basierten Middleware zur Abstraktion von Backendsystemen

Diplomarbeit

vorgelegt von Christoph Wagner

Lehrstuhl Rechnernetze und Verteilte Systeme
Betreuer: Dipl.-Inf. H. Schulze
Hochschullehrer: Prof. Dr. K. Irmscher

Leipzig, November 2001

Danksagung

Die Gewandtheit im Formulieren der Danksagungen charakterisiert den Wissenschaftler von Rang. Es kann vorkommen, dass ein Wissenschaftler am Ende seiner Arbeit entdeckt, dass er niemandem Dank schuldet. Macht nichts, dann muss er Dankschulden erfinden. Eine Forschung ohne Dankschulden ist suspekt...

Umberto Eco "Wie man ein Vorwort schreibt"

Da diese Arbeit nicht im stillen Kämmerlein entstand, sondern ich auf die Unterstützung vieler bauen konnte, danke ich ganz herzlich:

meinem Betreuer Dipl.-Inf. Hendrik Schulze, der durch seine unermüdlichen Kommentare und Ideen zum Gelingen dieser Arbeit wesentlich beigetragen hat,

meinem Hochschullehrer Prof. Dr. Klaus Irmscher, der mir eine höchst interessante und komplexe Diplomarbeit ermöglichte,

meinen Eltern, deren fortwährende moralische und nicht zuletzt auch finanzielle Unterstützung dazu beitrug, daß ich das Studium aufnehmen und auch erfolgreich abschließen konnte,

allen Freunden und Kommilitonen, die durch ihr großes Interesse an dieser Arbeit mich immer wieder neu motiviert und ermutigt haben.

Zusammenfassung

In dieser Arbeit wird der Entwurf und die Realisierung einer Software beschrieben, die vorhandene Serversysteme, die strukturiert Daten bereitstellen, abstrahiert und damit die vorhandenen Differenzen solcher Backends für eine Client-Anwendung eliminiert, um einen standardisierten Zugriff auf beliebige, und damit austauschbare, Datenquellen für eine Anwendung bereitzustellen. Dazu werden verschiedene Abstraktionsebenen eingeführt und es wird eine einheitliche Datenstruktur und Schnittstelle definiert.

Das entwickelte System ist als Middleware konzipiert, das heißt, es dient als Grundlage für andere Software. Um die Stabilität und Praktikabilität der Middleware zu verifizieren, wurden zusätzlich mehrere Anwendungen implementiert. Ein graphischer Swing-basierter Konfigurationsclient ermöglicht die komplette Verwaltung der entwickelten Middleware auf eine einfache und benutzerfreundliche Weise. Ein weiterer Client implementiert einen Benchmark, um durch die Erzeugung von hohen Lasten neben der Speicher- und Durchsatzperformance die Systemstabilität zu verifizieren.

Um mittels einer installierten Middleware verschiedene Clients zu ermöglichen und um die Verteilung von Anwendungen im Sinne des 3-tier-Modells zu unterstützen, basiert die Lösung auf einer Verteilungsplattform, wobei konkret die Java Remote Method Invocation benutzt wurde.

Eine weitere wichtige Funktion ist die Möglichkeit, vorhandenen Client-Anwendungen zur Laufzeit den Zugriff auf weitere Backendsysteme zu ermöglichen. Dieses wurde durch die Entwicklung einer Treiberarchitektur realisiert. Diese Treiberarchitektur wurde dabei so strukturiert, daß neben den reinen Datenbanken auch lokale Dateien oder XML-Strukturen als Datenquellen eingebunden werden können. Dadurch ist es beispielsweise möglich, bei der konkreten Softwareenwicklung mit einer Oracle-Datenbank zu arbeiten und erst bei der Installation des fertigen Produkts bei einem Kunden das System auf eine IBM DB2 Datenbank zu konfigurieren, ohne daß dazu eine Änderung des Programms notwendig wäre.

Abstract

This diploma thesis describes the concept and the creation of software that abstracts existing server systems delivering structured data. Thus, it eliminates the existing differences of such backend machines for client applications. In this context, various abstraction levels will be provided, and both a consistent data structure and an integrative interface will be defined.

The system was created following the middleware concept so that it can establish a base system for other software. In order to verify both the dependability and the usability of the middleware, various additional applications were also implemented. A graphical and Swing-based Configuration Client enables an easy and user-friendly middleware management. An additional client implements a benchmark and induces high traffic in order to verify both the dependability and the storage and capacity performance of the system.

Using the Java Remote Method Invocation, the solution is based on a distribution platform in order to enable both different clients and an application distribution in terms of the 3-tier model by means of an installed middleware.

Another important feature is the enabling option for existing client applications in order to access additional backend systems. This feature was implemented by creating a driver architecture. The structure of the driver architecture was realized in a way it will allow the integration of pure databases on the one hand side, but also of local files and XML structures on the other hand side. Thus, the system allows the use of an Oracle database for specific software development processes, and it must only be configured to an IBM DB2 database during the final product installation at the customers´ site. Moreover, this configuration can be done without changing the system.

Inhaltsverzeichnis

1. Einleitung **12**
1.1. Motivation . . . 12
1.2. Ziel . . . 14

2. Voruntersuchungen **16**
2.1. Einführung . . . 16
2.2. Backends . . . 16
2.2.1. Datenbank-Management-Systeme . . . 16
2.2.2. Enterprise Resource Planning-Systeme . . . 19
2.2.3. Archivsysteme . . . 19
2.2.4. Weitere Systeme . . . 20
2.2.5. Zusammenfassung . . . 20
2.3. Verteilungsplattform . . . 20
2.3.1. Corba . . . 20
2.3.2. Remote Method Invocation . . . 21
2.3.3. Enterprise Java Beans . . . 21
2.3.4. Zusammenfassung . . . 22

3. Prototypische Architektur **23**
3.1. Einführung . . . 23
3.2. Strukturen . . . 23
3.2.1. Backend . . . 24
3.2.2. Attribut . . . 24
3.2.3. Datasources . . . 24
3.3. Funktionen . . . 25
3.3.1. Strukturzugriff . . . 25
3.3.2. Datenzugriff . . . 25
3.3.3. Sichern von Datensätzen . . . 26
3.3.4. Transaktionen . . . 26
3.3.5. Benutzerverwaltung . . . 27
3.3.6. Datentypen . . . 28
3.3.7. Large Objects . . . 29
3.4. Schnittstellen . . . 30
3.4.1. Einleitung . . . 30
3.4.2. Frontend-Schnittstelle . . . 30

3.4.3. Backend-Schnittstelle 31

4. Realisierung **32**
4.1. Einleitung 32
4.2. Klassenstruktur der Middleware 33
4.3. Datenzugriff 33
4.3.1. Treiber 35
4.3.2. Datentupel-Kapselung 38
4.3.3. Selection-Chains 39
4.4. Interner Aufbau der Meta-Repository 40
4.4.1. Tables 40
4.4.2. Columns 41
4.4.3. Checkouts 41
4.4.4. Backends 42
4.4.5. Backend-Aliase 42
4.4.6. User 42
4.4.7. Roles 43
4.5. Meta-Informationen 44
4.6. Provider 45
4.6.1. Generischer SQL-Treiber 45
4.6.2. Mysql-Treiber 47
4.6.3. Oracle-Treiber 47
4.7. Sicherung von Datensätzen 48
4.8. Verteilung 48

5. Client-Anwendungen **51**
5.1. Administrationsclient 51
5.2. Backendtester 53
5.3. Beispielhafter Kommandozeilenclient 53

6. Performance-Untersuchungen **56**
6.1. Einleitung 56
6.2. Auswahl des Benchmarks 56
6.3. TPC-B Benchmark 57
6.4. Durchführung 58
6.5. Ergebnisse 58
6.6. Auswertung 63
6.7. Fazit 64

7. Abschluss und Ausblick **65**
7.1. Abschluss 65
7.1.1. Lösungansatz für dokumentbasierte Systeme 67
7.1.2. Lösungsansatz für XML-Formate 67
7.2. Ausblick 68

A. Datentypen **70**

B. TPC-B Benchmark **71**
B.1. JDBC-Benchmark 71
B.2. Middleware-Benchmark 72

C. Glossar **74**

Abbildungsverzeichnis

1.1. Anwendungsstruktur beim 3-tier Modell 13

2.1. Strukturierung der verschiedenen JDBC-Treibertypen 17

3.1. Sichten der Middleware für die verschiedenen Beteiligten. Die Backend-Komponente läuft als Singleton Instanz und verwaltet unbegrenzt viele Datasource-Instanzen, welche den konkreten Datenzugriff realisieren. . . . 23
3.2. Aufbau der Benutzerverwaltung 28
3.3. Einleitung einer Lob-Transaktion 29
3.4. Beginn einer Lob-Transaktion 30

4.1. Interner Aufbau der Middleware 32
4.2. Klassendiagramm des org.abla.server.datasource-Package 33
4.3. Aufbau eines Middleware-Treibers 37
4.4. Klassendiagramm der Klassen, die Datentupel enthalten 39
4.5. Interne Verknüpfungen der Meta-Repository 40
4.6. Aufrufpartitionierung des generischen SQL-Treibers. Deutlich sind die internen Queraufrufe zu erkennen, die den eigentlichen SQL String erzeugen und daher bei abgeleiteten Klassen lediglich partiell überschrieben werden müssen. 46
4.7. Wrapperfunktion der Hilfsklassen für Streams 49

5.1. Screenshot des Administrationsclients mit geöffnetem Dialog für eine Datasource mit den sichtbaren Attributen und den definierten ACL-Rechten 51
5.2. Suchdialog für bereits in einem Backend existierende Datasources, dessen angezeigte Ergebnisse unter Verwendung der Wildcards gebildet wurden . 52

6.1. Performance-Übersicht der Middleware mit der Darstellung aller Benchmarkläufe die über die Middleware durchgeführt wurden 59
6.2. Performance-Übersicht der direkten JDBC-Implementierung mit der Darstellung aller Benchmarkläufe 59
6.3. Vergleich der Performance zwischen der Middleware und der direkten JDBC-Verbindung bei einem Client 60
6.4. Vergleich der Performance zwischen der Middleware und der direkten JDBC-Verbindung bei fünf parallelen Clients 60

6.5. Performance bei zehn parallelen Clients mit dem sichtbaren Einfluss des ConnectionPools der Middleware bei wenigen Transaktionen 61
6.6. Performance bei fünfzehn parallelen Clients mit dem erkennbaren Einfluss des ConnectionPools der Middleware bei wenigen Transaktionen 61
6.7. Performance bei zwanzig parallelen Clients mit dem erkennbaren Einfluss des ConnectionPools der Middleware bei wenigen Transaktionen 62
6.8. Performance bei fünfzig parallelen Clients mit dem erkennbaren Einfluss des ConnectionPools der Middleware bei wenigen Transaktionen 62

7.1. Vorschlag für den Aufbau von dokumentenbasierten Treibern 67

Tabellenverzeichnis

4.1. Struktur der internen Datasource Tables 41
4.2. Struktur der internen Datasource Columns 41
4.3. Struktur der internen Datasource Checkouts 42
4.4. Struktur der internen Datasource Backends 42
4.5. Struktur der internen Datasource Backend-Aliase 43
4.6. Struktur der internen Datasource User 43
4.7. Struktur der internen Datasource Roles 43

A.1. Datentypspezifikation 70

Sourceverzeichnis

2.1. JDBC-Beispiel 18
4.1. BackendMetaData.java 44
4.2. AttributeMetaData.java 44
4.3. DatasourceMetaData.java 44
4.4. SQLDatasource Implementierung 45
5.1. Test-Programm 54
6.1. TPC-B Transaktion 57
B.1. JDBC-Implementierung 71
B.2. Middleware-Implementierung 72

1. Einleitung

1.1. Motivation

Aufgrund der Öffnung der Weltmärkte und der weltweiten Verfügbarkeit von Dienstleistungen sind schnelle Reaktionen entscheidend für den Erfolg eines Unternehmens. Um ebensolche Aktivitäten effektiv zu ermöglichen, müssen viele Informationen in den entscheidenden Gremien möglichst effizient zur Verfügung gestellt werden. Dabei wird unterschiedlichste Hard- und Software eingesetzt. Da Firmen ihre Rechnersysteme in gewissen Abständen erweitern, ausbauen oder durch neue ersetzen, besteht die Notwendigkeit, vorhandene Daten auch auf den neuen Systemen zur Verfügung zu stellen. Grundsätzlich gibt es dabei mehrere Möglichkeiten, die aber alle mit großen Nachteilen behaftet sind:

- Migration: Bei der Migration werden die Daten von einem Altsystem auf das neue System kopiert. Die Migration stellt erhebliche technische Anforderungen, da die kopierten Daten in ihrer Integrität geprüft werden müssen, bei einem Wechsel der Softwareplattform eventuell eine Konvertierung erforderlich ist und bei Fehlern manuell nachgearbeitet werden muss. In der Zeit einer Migration stehen die Daten desweiteren auch nicht zur Verfügung, weil sie zur Sicherung gesperrt werden. All diese Probleme bei der Umstellung auf neue Rechnersysteme sorgen für hohe Kosten und eine lange Dauer bei der Migrierung.
- Parallelisierung: Eine weitere Möglichkeit ist die Parallelisierung der Daten, das heißt, die alten Daten werden im alten System weitergepflegt, neue Daten werden im neuen System gespeichert. Dazu müssen umfangreiche und komplizierte Replikations- und Synchronisationsmechanismen eingesetzt werden, um die Integrität sicherzustellen. Große Schwierigkeiten gibt es auch, wenn Auswertungen der Daten erfolgen sollen und zu diesem Zwecke eine Zusammenführung der Daten in eine gemeinsame Datenbasis erfolgen muss. Bei diesem Verfahren erschwert gerade die Heterogenität der Systeme die praktische Handhabung.

In den letzten Jahren drängte sich ein neues Akronym immer mehr in den Vordergrund, die TCO. Diese "Total Cost of Ownership" soll angeben, welche echten Kosten durch einen Rechner entstehen. Darunter versteht man neben dem reinen Anschaffungspreis auch die Kosten, die mit der Software- und Hardwarewartung anfallen. Viele Anwendungen erfordern auf jedem Rechner die Installation eines Clientpakets, das durch einen Administrator installiert, gewartet und mit den entsprechenden Updates versehen werden

muss. Solche Architekturen steigern die TCO in erheblichem Maße, da ein Systemadministrator jeden Rechner physikalisch vor sich haben muss, um die jeweiligen notwendigen Änderungen durchzuführen. Dieses Problem lässt sich zwar durch den Einsatz von Systemmanagementsoftware, wie etwa "Tivoli" von der IBM, minimieren, aber solche Programme sind sehr teuer und auftretende Problemfälle sind aufgrund der innewohnenden Komplexität nur schwer aufklärbar. Aus diesem Grunde traten mit dem Siegeszug des Internets auch mehr und mehr Software-Systeme in den Vordergrund, die keine eigenen proprietären Client-Anwendungen mehr besitzen, sondern zu diesem Zwecke den auf jedem System vorhandenen Internet-Browser einsetzen. Diese Thin-Client-Architektur,

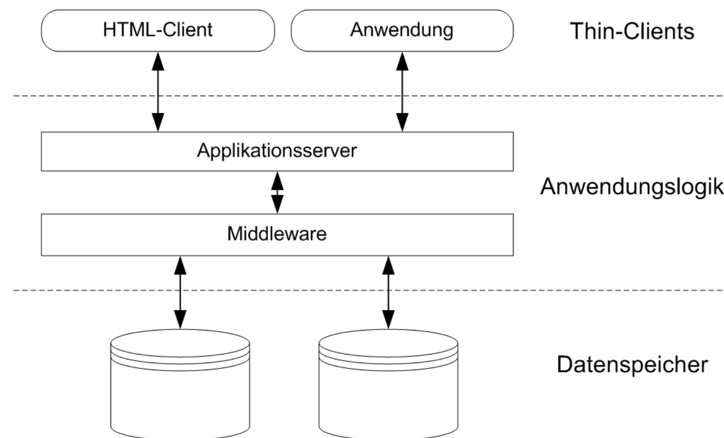

Abbildung 1.1.: Anwendungsstruktur beim 3-tier Modell

bei der die Anwendung nur noch auf einem Server läuft und die Anwender mit einem Browser auf diesen zugreifen, senkt die TCO, da heute auf jedem Betriebssystem ein Internet-Browser vorhanden ist, und wird daher zurecht auch positiv bewertet. Ein prominentes Beispiel für eine solche Multi-Tier-Architektur ist etwa die ERP-Software von SAP, die es neben der klassischen Variante auch als mySAP gibt, bei der nur noch der Browser als Client dient. Das Referenzmodell ist dabei das 3-Tier-Modell (siehe Abbildung 1.1), bei dem die unterste Schicht aus den Rechnern besteht, die die Daten speichern, also in aller Regel Datenbanken, die mittlere Schicht aus der Anwendung die die Daten in der gewünschten Strukturierung für die Anwender bereitstellt und der oberen Schicht, dem Browser oder einem proprietären Client, der die Daten darstellt und Datenmanipulationen ermöglicht.

Solche mehrstufigen Software-Architekturen basieren auf der Einsicht, daß durch die Kapselung der verschiedenen Ebenen einer Anwendung die Entwicklung vereinfacht und die Stabilität erhöht wird. Die Vereinfachung wird durch die von Beginn an definierten Schnittstellen zwischen den verschiedenen Komponenten erreicht. So kann die Präsentationsschicht bereits entwickelt werden, wenn die darunterliegende Schicht noch nicht vollständig implementiert wurde, da die Zugriffsschnittstellen zwischen beiden bereits feststehen. Durch diese Form der Architektur kann man an Stabilität gewinnen, weil beliebige Ebenen jederzeit durch qualitativ bessere ersetzt werden können, ohne

daß eine Anpassung der anderen Komponenten erfolgen muß. Die in dieser Arbeit zu entwerfende Middleware stellt eine solche Komponente mit klar definierten Schnittstellen dar. Zukünftige, auf diese Middleware aufsetzende Anwendungen profitieren daher in der oben beschriebenen Art und Weise. Im Endergebnis wird die Entwicklungsdauer einer Anwendung stark reduziert, wodurch die "Time-To-Market" beziehungsweise die "Time-To-Customer", also die Zeit bis ein Produkt Marktreife erlangt und beim Kunden eingeführt werden kann, sinkt. Diese Zeitverkürzung zieht automatisch auch eine Kostenersparnis nach sich, durch die sich ein Produkt wiederum attraktiver am Markt plazieren lässt. All diese Faktoren sind starke Beweggründe, eine Lösung für die dargestellten Probleme zu finden.

1.2. Ziel

Aus den eben genannten Gründen soll die zu entwickelnde Middleware den Applikationen einen transparenten Zugriff auf die sich im Einsatz befindenden Rechnersysteme ermöglichen. Dies geschieht durch ein System, bei dem eine einheitliche Sicht eines Client-Programms auf die Daten dann innerhalb der Middleware auf die physikalisch vorhandenen Systeme aufgeteilt wird. Diese Aufteilung sollte dynamisch geschehen, das heißt, die Übersetzung der Middleware-Datenstrukturen in die in den Endsystemen eingesetzten Datenstrukturen sollte auf einer Treiberebene erfolgen. Bei einer nachträglichen Erweiterung der Rechnerstruktur wäre es wünschenswert, bestünde die Möglichkeit, innerhalb der Middleware einen neuen Treiber zu installieren, der dann die notwendigen Funktionen bereitstellt, ohne dass eine Client-Anwendung eine solche Änderung mitbekommt oder gar zu diesem Zweck beendet und neu gestartet werden muss.

Bei den Zugriffen auf Daten müssen Strukturen bereitgestellt werden, die die Daten von den proprietären Formaten der jeweiligen Serversysteme abstrahieren. Diese Strukturen sollten durchgängig sowohl bei der Datenabfrage, als auch bei der Datenmanipulation genutzt werden. Bei der Vielfalt der Rechnersysteme am Markt ist es auch wünschenswert, daß die Middleware nicht an ein bestimmtes Basissystem oder Betriebssystem gebunden ist. Zusammenfassend lassen sich die folgenden Aufgabenstellungen und Ziele spezifizieren:

1. Entwicklung einer Middlewarelösung mit klar definierten Schnittstellen für Client-Anwendungen und Treiber für den konkreten Serverzugriff.
2. Die Middleware soll plattformunabhängig sein.
3. Die verwandten Datenstrukturen zur Speicherung von Inhalten sollen unabhängig von der Herkunft der Daten sein.
4. Transparenter Zugriff für Client-Anwendungen auf die Serversysteme.
5. Eine Treiberarchitektur soll es ermöglichen, zur Laufzeit der Middleware den Zugriff auf neue Serversysteme bereitzustellen, ohne daß die Dienste der Middleware während dieser Systemerweiterung in irgendeiner Weise eingeschränkt werden.

6. Multi-Tier-Architekturen sollen durch den Einsatz einer Verteilungsplattform unterstützt werden.
7. Die Treiber-Architektur soll durch eine variable Strukturierung flexibel genug für die Anpassung an heterogene Systeme sein.
8. Transaktionen innerhalb eines Treibers sollen optional verfügbar sein.
9. Ein Ressourcenpooling soll für Treiber möglich sein.
10. Eine Benutzerverwaltung soll komplexe Rechtesysteme ermöglichen und für Client-Anwendungen erweiterbar sein.
11. Zur Unterstützung von n-tier-Architekturen sollen Datensätze gesperrt werden können.
12. Die Anpassung von Client-Anwendungen an die installierten Systeme soll zur Installation erfolgen können, ohne daß Änderungen am Quellcode eines Clients notwendig werden.
13. Die Middleware soll durch mindestens eine benutzbare Anwendung auf Praxistauglichkeit getestet werden.
14. Ein Stabilitäts- und Geschwindigkeitstest soll abschließend durchgeführt werden.

2. Voruntersuchungen

2.1. Einführung

Die Konzeption und Entwicklung einer integrierenden Middleware erfordert einen exakten Überblick über die verfügbaren Systeme und Lösungen, um die mit diesen verbundenen Probleme zu umgehen. Desweiteren sollen einige Begriffe geklärt werden, die im Zusammenhang mit der Middleware eine Rolle spielen.

2.2. Backends

Der in dieser Arbeit benutzte Begriff des Backends steht ganz allgemein für Systeme, die Daten speichern und bereitstellen. In diesem Abschnitt soll zunächst erläutert werden, welche Varianten dieser Serversysteme derzeit verbreitet sind und welche Möglichkeiten des Zugriffs es bereits gibt und welche Probleme dabei auftreten können.

2.2.1. Datenbank-Management-Systeme

Die heute typisch eingesetzten Datenspeicher sind Datenbanken. “Eine Datenbank ist eine Sammlung gespeicherter operationaler Daten, die von den Anwendungssystemen eines bestimmten Unternehmens benötigt werden.” [RA99]. In der Praxis werden zu diesem Zwecke Datenbankmanagement-Systeme (DBMS) eingesetzt, die als standardisierte Software die Definition, Verwaltung, Verarbeitung und Auswertung der in den Datenbanken enthaltenen Daten sicherstellen. Datenbanken werden seit den fünfziger Jahren eingesetzt, seit Mitte der siebziger Jahre haben sich die sogenannten relationalen Datenbanken durchgesetzt. Nachfolger der relationalen Datenbanken sollten die objektorientierten Datenbanken werden, die sich jedoch nicht auf breiter Basis durchsetzen konnten, da sie die Generik der relationalen Datenbanken, bei denen sich im Zweifelsfall die Daten auch ohne die auf die Datenbank aufsetzende Software nutzen lassen, vermissen ließen. Derzeit laufen die Entwicklungen eher auf die zusätzliche Implementierung objektorientierter Ideen in die bereits bestehenden relationalen Datenbanksysteme hinaus. Der Zugriff auf ein Datenbank-Management-System geschieht über proprietäre Protokolle, Sprachen und Client-Anwendungen, kann aber in der Regel auch mittels der Sprache SQL erfolgen: Bereits in den siebziger Jahren begann IBM mit der Entwicklung der Sprache SQL (damals noch *Structured English Query Language*), die im Jahre 1986 von der International Standards Organization (ISO) unter dem Namen SQL1 beziehungsweise SQL86 (nun unter der Bezeichnung *Standard Query Language*) verabschiedet

wurde. Der Standard wurde in der Vergangenheit mehrfach erweitert, im Jahre 1992 wurde SQL2 (bzw. SQL92) verabschiedet und SQL3 ist auf dem Weg. SQL ist dabei mehrschichtig aufgebaut, wobei es jeweils Befehlsgruppen für die Definition von Strukturen, also der Tabellen, für die Abfrage und Manipulation der darin enthaltenen Daten sowie für die Einbindung von SQL in andere Programmiersprachen gibt. Inzwischen hat praktisch jeder Datenbankanbieter die Sprache SQL implementiert.

Um aus einem Programm heraus auf Datenbanken zugreifen zu können, waren bis vor einiger Zeit datenbankspezifische Programmteile notwendig, da jede Datenbank nur über ein proprietäres Kommunikationsprotokoll angesprochen werden konnte. Beschränkt auf die Windows-Plattform entwickelte Microsoft daher die Open Database Connectivity, kurz ODBC. Dieses Application Programming Interface (kurz API) soll es Applikationsentwicklern ermöglichen, Datenbanken einheitlich ansprechen zu können und dabei SQL zu verwenden. Dabei werden neue Datenbanken einfach per Installation eines Treibers hinzugefügt. Obwohl von Microsoft entwickelt, gibt es ODBC-Implementierungen inzwischen auch für andere Plattformen, etwa Solaris oder den Apple Macintosh. Der große Nachteil der ODBC-Architektur liegt in den nativen Treibern selbst, denn die Existenz eines Windows-Treibers für Oracle ermöglicht nicht den Einsatz dieses Treibers unter Linux etwa. Dies erhöht nicht nur den Aufwand der Entwicklung eines ODBC Treibers für verschiedene Plattformen, sondern bedingt auch, daß die pure Implementierung der Schnittstelle ODBC auf einer Plattform noch nichts über die Verfügbarkeit eines Treibers für diese Plattform aussagt.

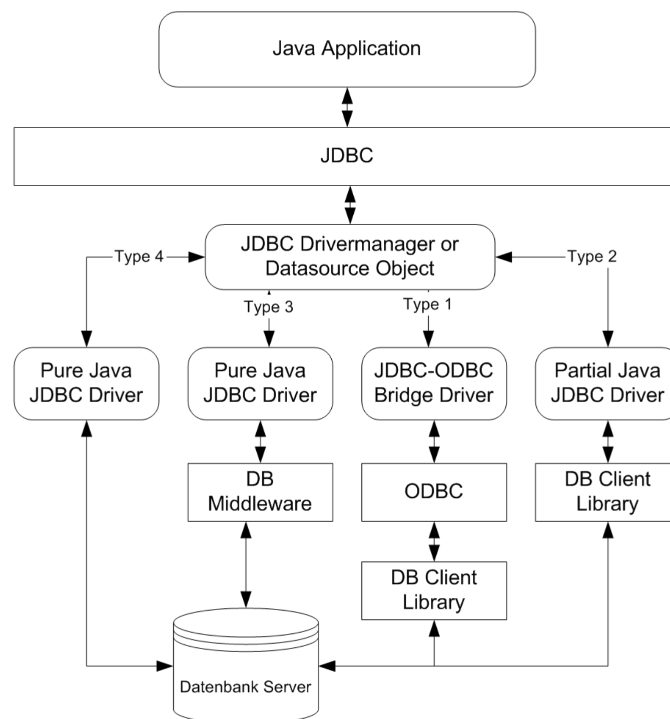

Abbildung 2.1.: Strukturierung der verschiedenen JDBC-Treibertypen

Mit der Einführung der Programmiersprache Java veröffentlichte Sun auch eine eigene Datenbankschnittstelle, die stark an ODBC erinnert, Java Database Connectivity (JDBC) genannt. Sun orientierte sich am bewährten ODBC-Modell und erweiterte es um Details, die durch Java als Programmierplattform notwendig wurden. Die mit JDBC eingeführten Treiber sind in verschiedene Kategorien eingeteilt. Hauptsächlich wird dabei die Plattformunabhängigkeit des Treibers unterschieden. Inzwischen gibt es praktisch für jede Datenbank auch einen JDBC-Treiber. Die Strukturierung der JDBC-Treiber ist in Abbildung 2.1 auf der vorherigen Seite dargestellt.

Mit ODBC und JDBC wurde eine gewisse Abstraktion von den spezifischen Datenbankdetails erreicht, allerdings bergen sie immer noch das Problem, daß sich Datenbanken in der SQL-Implementierung unterscheiden. Dies betrifft beispielsweise die Syntax einiger SQL-Kommandos, aber auch Details wie die verwendeten Datentypen. Unter Oracle werden beispielsweise alle in SQL definierten numerischen Datentypen auf den Typ NUMBER umgesetzt. Viele Datenbanken unterstützen auch nur eine Teilmenge, der in den Abstraktionsschichten definierten Datentypen.

Beispielhaft soll der Zugriff auf eine per JDBC verfügbare Datenbank gezeigt werden. Dazu wird ein Connection-Objekt erzeugt, das die physikalische Verbindung in eine Datenbank darstellt und welches dann ein Statement-Objekt erzeugt, das SQL-Befehle in die Datenbank schickt und die Ergebnisse entweder als Zahlwert oder ResultSet zurückliefert.

Program 2.1: JDBC-Beispiel

```
1  import java.sql.Connection;
2  import java.sql.DriverManager;
3  import java.sql.ResultSet;
4  import java.sql.SQLException;
5  import java.sql.Statement;
6
7  public class JDBCExample
8  {
9
10     private Connection con = null;
11     public static void main (String[] Args)
12     {
```

Festlegung der Treiberklasse und der URL unter der die Datenbank angesprochen werden soll.

```
13         String drivername = "org.gjt.mm.mysql.Driver";
14         String dburl = "jdbc:mysql://localhost/database?user=username&password=pass";
15         try
16         {
```

Dynamisches Nachladen der Treiberklasse und Erzeugen des entsprechenden Verbindungsobjekts.

```
17             Class.forName (drivername);
18             con = DriverManager.getConnection (dburl, "username", "password");
19             Statement stmt = con.createStatement ();
```

Bilden des Abfragestrings und auslesen der Ergebniswerte.

```
20             String query = "SELECT balance FROM accounts WHERE id = 123";
```

```
21              ResultSet rs = stmt.executeQuery (query);
22              if (rs.next ())
23              {
24                  int oldAccountBalance = rs.getInt (1);
25              }
26              query = "UPDATE accounts SET balance = 234 WHERE id = 123";
27              stmt.executeUpdate (query);
```

Nach der Verarbeitung der Daten wird die Verbindung in die Datenbank geschlossen.

```
28              stmt.close();
29              con.close();
30          }
31          catch (SQLException e)
32          {
33              e.printStackTrace ();
34          }
35      }
36  }
```

2.2.2. Enterprise Resource Planning-Systeme

Enterprise Resource Planning-Systeme ermöglichen die Verarbeitung fast aller firmeninternen Daten, von der Kundenakquise, über Vertriebsprozesse und Auftragsabwicklung bis hin zum Kundenservice. Ab gewissen Firmengrößen wird der Einsatz eines solchen ERP-Systems, wie SAP R/3 oder Siebel, unumgänglich, da sich nur so die komplexen Prozesse kontrolliert steuern und verifizieren lassen. Diese Systeme verfügen über eine sehr vielschichtige und schwer zu durchschauende interne Struktur. R/3 etwa ist rollenbasiert, das heißt, Mitarbeiter können von vielen Orten auf die benötigten Daten zugreifen, was natürlich besonders für große Firmen mit dezentralisierten Strukturen ein nicht zu unterschätzender Vorteil ist. Zwar lassen sich ERP-Systeme intern mit Datenbanken vergleichen, allerdings werden diese um erhebliche Funktionalitäten erweitert, so daß es für den Benutzer nicht ersichtlich ist, in welcher Form die Daten in der Datenbank gespeichert werden.

Für den Zugriff auf ERP-Systeme gibt es bisher keinen einheitlichen Weg. Zwar haben Anbieter wie SAP inzwischen Funktionsbibliotheken veröffentlicht, die den Zugriff auf das System erlauben, dies sind aber höchst kompliziert einzusetzende und proprietäre Schnittstellen. Es ist daher ebenso wünschenswert, in der Middleware auch den Zugriff auf diese Systeme nicht auszuschließen.

2.2.3. Archivsysteme

Während vieler Geschäftsprozesse entstehen Dokumente in nicht-digitaler Form, seien es Rechnungen, Verträge oder Photos. Solche Papiere werden derzeit digitalisiert in sogenannten Archivsystemen, in der Regel auf Basis optischer WORM-Medien (Write Once Read Many), gespeichert. Dort sind sie dann jederzeit im Zugriff und die Archivsysteme ermöglichen durch eine Attributierung das leichte Wiederauffinden der Dokumente. Verschiedenste Hersteller bieten auch noch zusätzliche Funktionalitäten, wie etwa eine Versionierung, mit ihren Produkten an. Aufgrund der dokumentenorientierten Daten

sollen die Archivsysteme in dieser Arbeit nicht mit einbezogen werden. Die in der Arbeit erstellten Konzepte lassen sich aber ebenso auf Archivsysteme anwenden und portieren.

2.2.4. Weitere Systeme

Neben den klassischen Datenbanken und Archivsystemen gibt es weitere Strukturen, in denen Daten gehalten werden. Das reicht von applikationspezifischen Formaten bis hin zu XML-Dateien. Auch diese Datenmengen sollen in der Middleware genutzt werden können, im ungünstigsten Fall durch die separate Entwicklung eines Treibers. Die jeweiligen Implementierungsdetails sind stark vom zu unterstützenden Format abhängig und bedürfen einer getrennten Untersuchung als Teil einer weiterführenden Arbeit.

2.2.5. Zusammenfassung

Als Datenquellen unverzichtbar sind Datenbanken und ERP-Systeme. Proprietäre Formate sollen unterstützt werden, von der Implementierung eines Treibers wird in dieser Arbeit aber abgesehen. Die Archivsysteme werden nicht weiter betrachtet, sind aber in der Konzeption auf ähnliche Weise anschließbar.

2.3. Verteilungsplattform

Um die Middleware portabel zu halten und Anwendungen verteilt entwickelt zu können, ist der Einsatz einer Verteilungsplattform sinnvoll. Der Markt hat sich dabei in den letzten Jahren auf einige wenige Systeme und Architekturen konzentriert, die im folgenden vorgestellt werden sollen.

2.3.1. Corba

Die Common Object Request Broker Architecture (CORBA) wurde spezifiziert durch die Object Management Group (OMG). Einfach gesprochen, erlaubt Corba verschiedensten Anwendungen, miteinander zu kommunizieren, völlig unabhängig vom Standort und der konkreten technischen Umgebung. Corba 1.1 wurde 1991 eingeführt und definiert die Interface Definition Language (IDL) und das Application Programming Interfaces (API), das Client/Server-Objekten die Kommunikation über einen Object Request Broker (ORB) erlaubt. Im Dezember 1994 wurde Corba 2.0 verabschiedet, das eine echte Interoperabilität zwischen den ORBs verschiedener Hersteller spezifizierte.

Der ORB ist die Middleware, die die Verbindung zwischen Client-Server Objekten etabliert. Durch die Nutzung eines ORBs kann ein Client-Objekt Methoden an einem Server-Objekt aufrufen, das sich auf der gleichen Maschine oder irgendwo im Netzwerk befindet. Der ORB ist dabei für den Aufruf zuständig, er findet das Objekt, übergibt die Methodenparameter und liefert das Ergebnis zurück. Das Client-Objekt muss dazu nicht wissen, wo sich das Server-Objekt befindet, in welcher Sprache es entwickelt wurde oder auf welchem Betriebssystem es läuft. Damit ermöglicht der ORB eine echte Interoperabilität zwischen Objekten in heterogenen Netzwerken.

2.3.2. Remote Method Invocation

Die Remote Method Invocation (RMI) ist eine Java-beschränkte Verteilungsplattform. Sie wurde kurz nach der Einführung von Java durch Sun in den Markt eingeführt und ermöglicht die einfache Distribution von Java-Objekten im Netzwerk. Als Verteilungsplattform ist RMI sehr mächtig, wenn man sich auf Java beschränkt. Es bietet viele der Features, die auch Corba bereitstellt, sowie einige weitere Funktionen. Dazu gehören etwa ein Naming-Service, das transparente Handling von verteilten Objekten, die verteilte Garbage Collection oder die Aktivierung von virtuellen Maschinen erst bei Aufruf einer dazugehörigen Funktion eines enthaltenen Objekts.

Grundsätzlich funktioniert RMI mittels eines weiteren Compilers. Dazu werden die Interfaces, die das java.rmi.Remote Interface, also das Interface, das ein Server-Objekt markiert, ableiten, durch einen eigenen Compiler, den rmic, in sogenannte Stub- und Skeleton-Klassen umgewandelt. Die Stubs liegen dann beim Client und sorgen bei Aufrufen an Server-Objekten für die Umwandlung der Parameter in einen Bytestream, die über einen Socket zum Server-Objekt getunnelt werden können, wo sie dann als Kopien der Original-Objekte wieder erzeugt werden, um in den eigentlichen Server-Objektmethoden verarbeitet zu werden. Äquivalent funktioniert die Rückmeldung der Methodenresultate. Objekte, die als RMI-Objekte mittels des Namingservice in einem Netzwerk bekannt gemacht worden sind, erlauben die transparente Einbindung in lokale Objekte. Sie lassen sich in den eigenen Software Modulen dergestalt einsetzen, daß es für die konkrete Nutzung keinerlei verteilungsspezifischen Aufwand erfordert, wurde das Objekt erst einmal per Namingservice lokalisiert. Zur Kommunikation wird das in der Java2 Standard Edition (J2SE) enthaltene Serialization Package genutzt, mit dessen Hilfe Objekte in einen Bytestream transformiert werden können. Ein einfaches Beispiel für die Kommunikation per RMI findet sich im Quellcode des Abschnitts 5.3 auf Seite 53.

Da RMI sehr flexibel aufgebaut ist, lassen sich mit wenig Aufwand auch im Nachhinein die RMI-Objekte in Corba-Objekte umwandeln. Dies geschieht, indem das Standard-Protokoll mit dem die Objekte miteinander kommunizieren, durch IIOP ersetzt wird. Das Paket, das dieses ermöglicht, namens “RMI over IIOP”, ist ab dem Release 1.3 in die J2SE integriert.

2.3.3. Enterprise Java Beans

In den letzten Monaten hat die komponentenorientierte Softwareentwicklung das Bild in den Fachpublikationen bestimmt. Sie soll das baukastenähnliche Zusammenbauen von Applikationen durch Verwendung standardisierter Komponenten ermöglichen. Für die Java-Plattform hat Sun die Enterprise Java Beans (EJB) spezifiziert. EJBs sind als Komponenten spezifiziert, die als Laufzeitumgebung einen Container voraussetzen, der auch im Nachhinein durch einen anderen, der Spezifikation genügenden Container, ausgetauscht werden können soll.

In der Spezifikation unterscheidet man Entity Beans, Session Beans und Message-Driven Beans, wobei letztere die Verarbeitung asynchroner Nachrichten ermöglichen. Erstere sind Containerobjekte für Daten, die in aller Regel aus Datenbanken stammen.

Session Beans stellen Funktionsbibliotheken bereit, unter anderem auch auf Entity Beans. Sucht man ein Entity Bean, ruft man am sogenannten Home-Objekt eine find-Methode auf, die ein Entity Bean zurückliefert, das mit den Daten aus der Datenquelle, die den Suchkriterien entsprachen, gefüllt wurde. Die Einschränkung der EJB-Plattform liegt in der Beschränkung auf Datenquellen, die einen JDBC-Treiber zur Verfügung stehen oder mittels Java Message Service (JMS) bereitstehen. Dabei werden im ersten Fall die Implementierungen des javax.sql.Datasource-Interfaces genutzt. Dieses Interface generiert java.sql.Connection-Objekte, die wiederum die vom JDBC bekannten Statements bereitstellen. Damit sind auch hier die Probleme in Bezug auf die Backend-spezifischen Eigenheiten gegeben.

Neben diesen Problemen ist es einem Bean ebenfalls nicht erlaubt, mittels java.io Package auf das lokale Dateisystem zuzugreifen, Verbindungen auf einem Socket zu empfangen oder native Bibliotheken anzusprechen. Aus all diesen genannten Gründen sind EJBs keine geeignete Verteilungsplattform, sondern eher als Zielplattform geeignet. Dazu würden Entity Beans in einem bean-managed Persistence Kontext laufen und dann auf die zu entwickelnde Middleware zugreifen.

2.3.4. Zusammenfassung

Die geforderte Plattformunabhängigkeit der Middleware stellt bei der Wahl der Programmiersprache hohe Abstraktionsforderungen an eigentlich nicht portable Sprachen wie etwa C++. Daher fiel die Wahl auf Java. Bei der Entscheidung bezüglich der Verteilungsplattform gab die Kompatibilität von RMI im Vergleich zu Corba und die gute Einfügung von RMI in die Java-Plattform den Ausschlag für RMI. Insbesondere die Generalisierung der Datentypen unabhängig vom verwandten Backend ist mittels Java einfacher und problemloser zu realisieren als mit C++.

Nach der Wahl der zugrundeliegenden Systeme folgt nun die Beschreibung der Systemarchitektur, mit der die oben beschriebenen Anforderungen realisiert werden sollen.

3. Prototypische Architektur

3.1. Einführung

Für das Verständnis der Architektur ist das Verständnis der benutzten Begriffe notwendig, die daher zuerst vorgestellt werden sollen. Anschließend werden ausgehend von den benötigten Funktionen die Konsequenzen auf die konkrete Realisierung beschrieben.

3.2. Strukturen

Die den Clients durch die Middleware zur Verfügung zu stellenden Funktionen erfordern eine Generalisierung der Datenstrukturen, derer sich die Clients dann bedienen können. Dies umfasst in erster Linie die Definition der zur Verfügung stehenden Backends und Datenquellen. Zu diesem Zwecke werden virtuelle Strukturen eingeführt, die in diesem Abschnitt vorgestellt werden sollen. Einen Überblick über die Systemarchitektur zeigt die Abbildung 3.1.

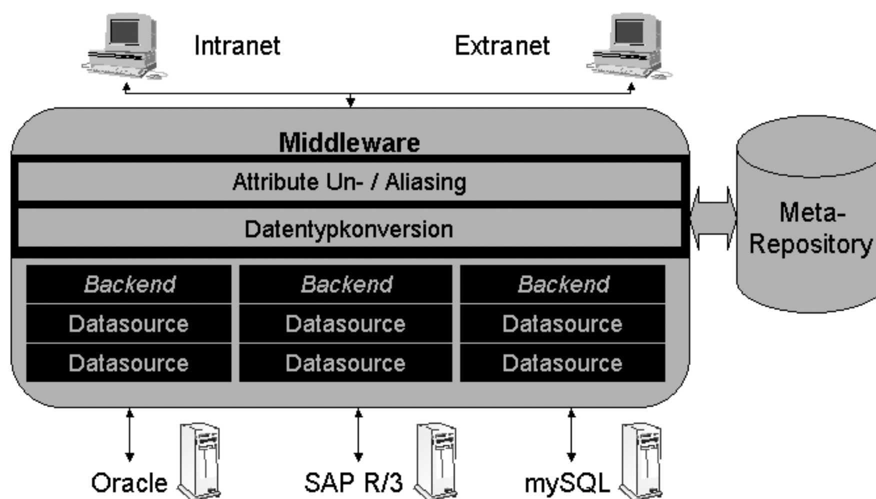

Abbildung 3.1.: Sichten der Middleware für die verschiedenen Beteiligten. Die Backend-Komponente läuft als Singleton Instanz und verwaltet unbegrenzt viele Datasource-Instanzen, welche den konkreten Datenzugriff realisieren.

3.2.1. Backend

Aufgrund der großen Vielfalt der möglichen Backends (siehe dazu 2.2 auf Seite 16) ist es notwendig, die entsprechenden Informationen an einer zentralen Stelle zu spezifizieren. Dazu zählen neben den allgemeinen Daten, also etwa dem Namen (Bezeichner) des Backends, auch Informationen, mittels welcher Zugangsdaten der Zugriff auf das Backend erfolgen soll. Desweiteren muß ein Backend jede enthaltene Datenquelle in ihrer Struktur beschreiben können, sowie eine Liste von enthaltenen Datenquellen, die einer spezifizierten Suchmaske entsprechen, liefern.

Backend: Ein Backend besteht aus einem Bezeichner, den Zugangsdaten über die der Zugriff auf die Datenquellen erfolgt, die in dem Backend liegen, der Festlegung, welcher Treiber dieses Backend verwaltet und einer Liste von Aliasbezeichnern, die alternativ von Clients genutzt werden können.

3.2.2. Attribut

Betrachtet man die Struktur von Attributsystemen genauer, bestehen sie in der Regel aus Mengen von Attributen - mit einem festgelegten Datentyp pro Attribut - und damit einem gewissen Wertebereich, einem Namen, sowie zwei boolsche Aussagen über die Füllbarkeit mit Nullwerten und die Einmaligkeit der Varianten eines Attributs. Eine Untermenge einer solchen Attributmenge wird in der Regel als Schlüsselliste definiert, um ein Datentupel aus der Attributstruktur genau identifizieren und um eindeutig darauf zugreifen zu können. Bei Datenbanken sind solche eindeutigen Schlüssellisten als Primary Keys bekannt. Zusätzlich zu diesen Eigenschaften sind für die Middleware weitere Informationen nötig. Das ist etwa ein weiterer Bezeichner eines Attributs, der unabhängig vom Namen des Attributs ist. Motivation für diesen Aliasnamen ist die Tatsache, dass eine Applikation intern eine Datenstruktur implementiert, ohne zu wissen, ob das Zielsystem die identischen Namen verwendet. Würde man diese Information nicht verwenden, würde jedes neue Zielsystem die Neuübersetzung des Programmcodes mit den geänderten Namen erfordern, was natürlich nicht praktikabel ist. Bei einem Einsatz einer Client-Software, die auf die Middleware aufbaut, ist lediglich das Mapping der Aliasattributnamen einzurichten, und das Mapping auf die jeweiligen echten Attributnamen geschieht dann innerhalb der Middleware.

Attribut: Ein Attribut definiert sich durch einen eindeutigen Namen, einen eindeutigen Aliasbezeichner, einen Datentyp mit zugehöriger Längenangabe, die Aussage, ob die Inhalte des Attributs null sein können, sowie die Aussage, ob das Attribut zur Liste der Schlüsselattribute gehört.

3.2.3. Datasources

Eine Menge von solchen Attributen definiert eine Datenquelle, im folgenden Datasource genannt. In einer Datenbank abstrahiert eine Datasource etwa eine Tabelle. Zusätzlich zu der Menge der Attribute wird eine Datasource durch einen eigenen Bezeichner, das

Backend, in dem die Daten wirklich vorliegen, sowie einen echten Namen im Backend definiert. Auch hier gilt wieder, daß die Abstraktion des Datasourcebezeichners vom echten Datenquellennamen, also etwa dem Tabellennamen in einer Datenbank, vonnöten ist, um jede feste Zuweisung zu änderbaren Parametern zu verhindern.

Datasource: Eine Datasource definiert sich durch einen eindeutigen Bezeichner, einen eindeutigen Namen, der im ebenfalls spezifizierten Backend vorliegt, sowie eine Menge von Attributen.

3.3. Funktionen

Eine Middleware, die als Abstraktionsebene entwickelt werden soll, muß umfangreiche Funktionen bereitstellen. Im folgenden werden diese Operationen nicht nur weitestgehend spezifiziert, sondern es werden auch Lösungsansätze für die erkannten Probleme beschrieben.

3.3.1. Strukturzugriff

Alle virtuellen Strukturen, die durch die Middleware verwaltet werden, müssen auch durch direkte Client-Schnittstellen manipulierbar sein. Dadurch wird sichergestellt, daß eine Anwendung auch zur Laufzeit neue Strukturen in der Middleware erstellen kann. Soll etwa ein Client eine Datasource aus einem Backend in ein anderes kopieren, muß dazu im Zielbackend die Datasource erst angelegt werden, was auch per Middleware ermöglicht werden soll. Erweitert man diese Regel auf alle Metastrukturen, erlaubt das die Entwicklung eines Verwaltungsclients für die Middleware, der nicht an der Middleware vorbei die Informationen in ein Backend schreibt, sondern selbst die Middleware als regulärer Client nutzt. Manipulierbar, also anlegbar, änderbar, abfragbar und löschbar, müssen die Backends, die Datasources, deren Attribute und die Elemente des Benutzersystem sein.

3.3.2. Datenzugriff

Zu den als selbstverständlich vorauszusetzenden Funktionen der Middleware gehören die aus dem Umfeld der Datenbanken bereits bekannten Selektionsvarianten. Dazu werden entweder alle Daten, die in der Datasource vorliegen, geliefert, oder eine Teilmenge, die einem zu spezifizierenden Selektionskriterium entspricht. Dieses Selektionskriterium sollte als generische, also Backend-unabhängige, Objektstruktur definiert werden können. Jeder Backend-Treiber muß dann die Umwandlung in ein kompatibles Format vornehmen, also etwa die Wandlung in einen SQL-String bei einer Datenbank. Zusätzlich sollte sich die Menge der gewünschten Attribute einschränken lassen, sowie die Ergebnisse um vorhandene Duplikate reduzieren. Neben diesen Selektionsfunktionen sollen auch etwas spezifischere Funktionen ermöglicht werden. Dazu zählen mathematische Funktionen, wie etwa der Durchschnitt oder die Summe eines Attributs, aber auch das Berechnen der Größe der Ergebnismenge bei einer bestimmten Anfrage.

Beim Lesen und Schreiben von Daten in und aus einer Datasource werden zwei Klassen benutzt. Die Record-Klasse ist dabei der kleinste gemeinsame Nenner. Sie erlaubt einen Hashtable-artigen Zugriff auf einen Datensatz. Das bedeutet, ein SELECT liefert eine Collection von Record-Objekten, in denen jeweils über den Attributnamen der Datasource der Inhalt des Attributs verfügbar ist. Desweiteren verfügt die Klasse über Methoden, die einen einfacheren Zugriff auf standardisierte Typen ermöglichen, etwa ein getString() oder getInt(). Eben diese Record-Klasse wird auch für Einfügeoperationen genutzt. Dazu wird ein Record mit den Attributnamen-Wert-Kombinationen gefüllt, die in einer Datasource eingefügt werden sollen.

Eine Besonderheit ist die CRecord-Klasse, die von Record abgeleitet ist, wobei das C für Complete steht. Ein CRecord muß von einem Treiber immer dann returniert werden, wenn eine Anfrage alle Attribute einer Datasource umfasst hat. Damit ist dann sichergestellt, daß alle Primärschlüsselfelder des Datensatzes ebenfalls enthalten sind. Ein CRecord ermöglicht dann einige Funktionen mehr, er kann den enthaltenen Datensatz in der Middleware sperren (siehe 3.3.3), kann LobTransaktionen, also Transaktionen auf großen Datenmengen, etwa binären Dateien oder Textdateien, (siehe 3.3.7 auf Seite 29) durchführen, sowie sich selbst in der Datenbank updaten, ohne daß die ursprüngliche Datasource gebraucht wird.

3.3.3. Sichern von Datensätzen

Bei n-tier-Anwendungen ist es ein häufig notwendig, daß Datensätze, die im Client- beziehungsweise im 2-tier bearbeitet werden, in der Datenquelle für einen manipulativen Zugriff gesperrt werden. Daher muß die Middleware auch Mechanismen anbieten, Datensätze, die sich gerade in einem zu bearbeitendem Zustand befinden, für konkurrierende Zugriffe zu sperren. Hierbei sind Locks, die Datensätze nur bis zum Beenden der Middleware sperren, und Checkouts, die auch nach einem Neustart der Middleware noch vorhanden sind, zu unterscheiden. Letztere sind für Langzeitbearbeitungen, etwa durch Außendienstmitarbeiter, die ihre Daten nur unregelmäßig synchronisieren, gedacht. Keine Möglichkeiten der Sicherung bestehen natürlich für den Zugriff durch Programme, die nicht über die Middleware gehen, sondern ein Backend direkt ansprechen.

3.3.4. Transaktionen

Beim Einsatz von komplexen Programmen im Umfeld verteilter Anwendungen sind Mechanismen der Transaktionskontrolle vorzusehen. Dazu müssen vier Merkmale erfüllt sein, die allgemein unter dem Begriff “ACID” bekanntgeworden sind. Diese Merkmale sind:

- Atomarität (atomicy): ein Programm wird entweder komplett oder gar nicht durchgeführt. Werden also mehrere Funktionen auf einer Datasource durchgeführt, werden diese nach außen erst dann sichtbar, wenn dabei kein Fehler durch Dateninkonsistenzen oder durch andere Programme bzw Ursachen aufgetreten ist.

- Konsistenz (consistency): nach dem Ablauf einer Transaktion befindet sich die Datasource immer in einem konsistentem Zustand. Wird also etwa bei einer Einfügeoperation ein doppelter Schlüssel erkannt, wird die Datasource wieder in den Zustand versetzt, der vor dem Aufruf bestand.
- Isolation (isolation): jedes Programm läuft isoliert von anderen Programmen, bzw. Transaktionen und arbeitet dadurch nur auf konsistenten Daten der Datasource
- Persistenz (durability): wenn ein Aufruf erfolgreich beendet wird, überleben diese Daten auch einen Software- oder Hardwaredefekt

Jeder einfache Zugriff, wie eine einzelne Datenabfrage, sind für sich genommen schon eine Transaktion, weil sie den genannten Kriterien entsprechen. In der Middleware wird der Begriff Transaktion allerdings als eine Liste von solchen atomaren Aufrufen verstanden, die durch einen einleitenden Aufruf als Transaktion gekennzeichnet werden, und wo dann bei einem lokalisierten Fehler durch einen Client selbständig ein rollback, also die Rücksetzung der vorher bereits durchgeführten Operationen veranlasst wird.

Diese Transaktionen sollen allerdings nicht als über mehrere Datasources verteilte Transaktionen implementiert werden, weil das die Transaktionslogik in die Middleware verlagern würde. Vielmehr sollen die bereits in aller Regel in den Backends vorhandenen Transaktionsmechanismen genutzt werden. Die Implementierung von Transaktionen sollte daher auch zu den optionalen Funktionen eines Backentreibers gehören.

3.3.5. Benutzerverwaltung

Viele Backendsysteme benutzen ein eigenes proprietäres Nutzersystem, was es erforderlich macht, ein eigenes Benutzersystem einzuführen. Diese Notwendigkeit wird insbesondere dann deutlich, wenn man sich nochmals vor Augen führt, daß alle Datasources, die aus einem Backend stammen, über den selben Benutzeraccount in diesem Backendsystem laufen, weil jedes Backend alle Datasources über den identischen Zugang laufen lässt. Aus diesem Grund wird ein völlig auf die Middleware zugeschnittenes Benutzersystem entworfen, das es ermöglicht, Clients als Inkarnation eines bestimmten Nutzers zu sehen.

Jeder Nutzeraccount besteht aus dem Nutzernamen, einem dazugehörigen Passwort, einem Datum des letzten Logins, einem Datum zu dem der Account abläuft, einem binären Rechteobjekt, das anwendungsspezifische Rechte hält, sowie einer Liste von Rollen, die auch leer sein kann. Eine Rolle ist lediglich eine mit einem Bezeichner versehene Sammlung von Rechten. Diese Rechte regeln die erlaubten Zugriffe der Mitglieder dieser Rolle, wobei dadurch, daß ein Nutzer auch Mitglied mehrerer Rollen sein kann, sich Rechte auch gegenseitig widersprechen können und es dann aufgrund der kumulativen Addition der Rechte zur Gewährung des Rechtes für den betroffenden Nutzer kommt. Desweiteren enthält eine Rolle, ebenso wie ein Benutzer, ein binäres Rechteobjekt, in welchem jede Anwendung eigene Rechte speichern kann. Diese Rechte werden auch kumulativ jedem Nutzer zugeordnet.

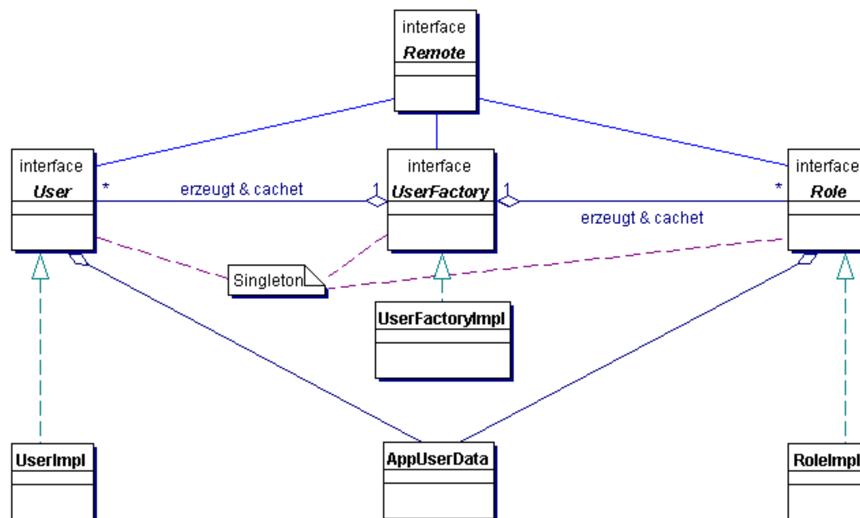

Abbildung 3.2.: Aufbau der Benutzerverwaltung

Die Idee hinter diesem Objekt ist die Tatsache, daß eine Client-Anwendung auf die Benutzerverwaltung aufbauen will, aber weitere Rechte abprüfen möchte. Diese können dann über dieses binäre Objekt verwaltet werden, ohne daß sich der Client dazu um die technische Speicherung dieser Rechte kümmern muß. Zusätzlich können so auch Eigenschaften eines Nutzers, etwa die letzte Fensterposition des Client-Fensters gespeichert werden.

Die einem Benutzer zugeordneten Rechte werden bei jedem Zugriff auf die Middleware überprüft. Deswegen ist es notwendig, daß sich jeder Nutzer zunächst identifiziert, woraufhin der Client ein Nutzerkontextobjekt erhält, daß dann allen anderen Middlewarefunktionen übergeben werden kann.

3.3.6. Datentypen

Jeder Zugriff auf ein Backendsystem produziert oder verändert vorhandene Daten. Jedes Datum ist dabei einem Wertebereich zugeordnet, dem Datentyp. Viele Backends unterstützen eine große Fülle an Datentypen, von denen aber nur die wenigsten miteinander direkt vergleichbar sind. Mit wachsender Komplexität des abgebildeten Wertebereichs differieren auch die Datentypen der Backends. Aus diesem Grund ist es notwendig, eine Menge an Datentypen mit ihren Wertebereichen zu spezifizieren, um dem Entwickler eines Clients ein standardisiertes Umfeld zu garantieren. Als Grundlage dient die Menge der Datentypen wie sie von Sun für die Datenbankschnittstelle JDBC definiert wurden. Die genaue Liste der Datentypen mit ihrer Spezifikation in der Middleware findet sich im Anhang A auf Seite 70.

3.3.7. Large Objects

Einige Datentypen sind für die Speicherung großer Datenmengen gedacht. Dabei handelt es sich um die sogenannten Large Objects, auch kurz LOBs genannt. Man unterscheidet dabei noch zwischen binären und textbasierten Daten, dann passenderweise *Binary Large Object* (BLOB) bzw *Character Large Object* (CLOB) genannt. Da solche Attribute mit erheblichen Datenmengen, die sich durchaus in der Größe von Gigabytes bewegen können, gefüllt werden, muß eine Lösung gefunden werden, die nicht mit im Speicher gehaltenen Objekten arbeitet, sondern einen lesbaren, bzw. schreibbaren Stream zur Verfügung stellt. Dazu wurde ein Verfahren entwickelt, das einen solchen Zugriff ermöglicht.

Grundsätzlich muß zwischen der Schreib- und der Leseoperation unterschieden werden. Zunächst das Lesen von Lob-Elementen: ein Treiber ist angehalten, Lob-Attribute in spezielle Objekte zu kapseln, bei BLOBs handelt es sich um das RMIBlobImpl-Objekt, bei CLOBs um das RMIClobImpl-Objekt. Diese stellen einen Adapter ([GOF01, Seite 171]) auf die in einem Blob zur Verfügung gestellten Streams (bei BLOBs ein InputStream, bei CLOBs ein Reader) dar. Dies ist notwendig, weil Streams nicht serialisiert werden können, da es sich dabei um an physikalische Ressourcen, etwa einen Socket, gebundene Objekte handelt. Die mit der Middleware bereitgestellten Adapterklassen erlauben es, Streams als Referenzobjekte über ein Netzwerk zu verschicken (siehe dazu auch die Abbildung 4.7 auf Seite 49). Will ein Client also auf ein Lob-Feld lesend zugreifen, führt er einen normalen SELECT durch, erhält dann aber in dem Ergebnisrecord für das Lob-Attribut eben entweder ein RMIBlob- oder ein RMIClob-Objekt, über das dann der Stream angesprochen werden kann.

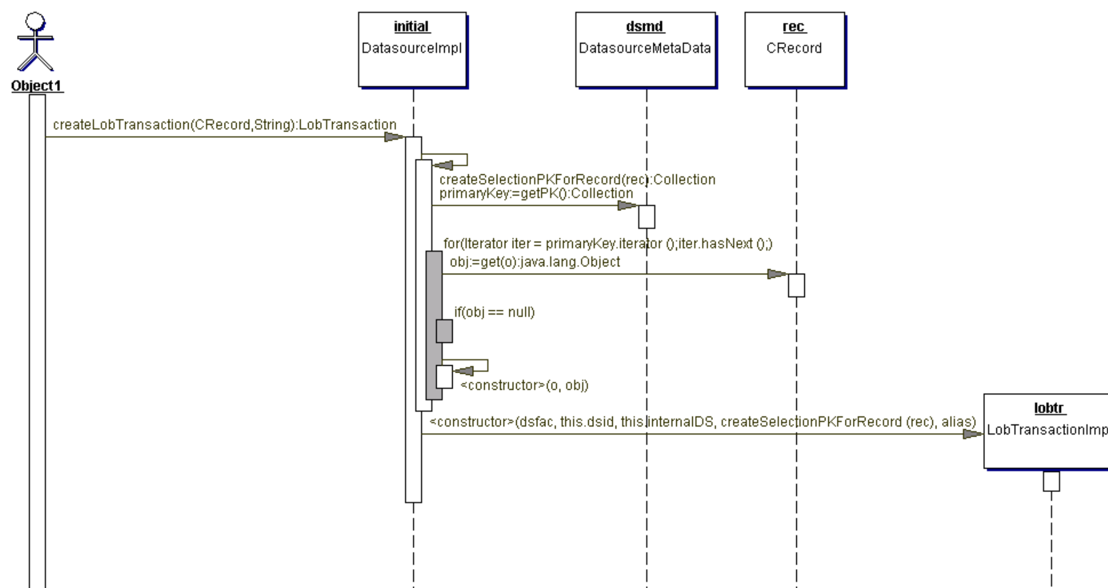

Abbildung 3.3.: Einleitung einer Lob-Transaktion

Das Schreiben von Lobs gestaltet sich etwas schwieriger, da Streams nur auf bereits bestehende Datensätze generiert werden können. Daher muß bei einem neu zu schreibenden Datentupel zunächst ein INSERT mit den restlichen Tupelinhalten in einer Datasource durchgeführt werden. Im nächsten Schritt wird der zuvor geschriebene Datensatz wieder per SELECT gelesen. Am nun erhaltenen CRecord-Objekt wird die Funktion createLobTransaction (String alias) aufgerufen, wobei der Parameter der Attributbezeichner der Lob-Spalte ist. Die Methode liefert dann ein LobTransaction-Objekt, das die Methoden zur Erzeugung eines OutputStreams (zum Schreiben in einen BLOB) oder eines Writers (für CLOBs) bereitstellt. Nach dem Schreiben in den Stream wird die Transaktion mit einem abschließenden endTransaction (boolean doCommit) entweder finalisiert oder abgebrochen. Während einer LobTransaktion kann sich die Datasource in einer echten Transaktion befinden und darf daher nicht parallel verwendet werden.

Mit diesem Verfahren sind also auch für verteilte Objekte effektive Streamzugriffe möglich. Der einzige Nachteil dieser Lösung liegt in der Reduzierung auf einen Stream per Datasource-Instanz, da mehrere Streamzugriffe nicht parallelisiert werden können.

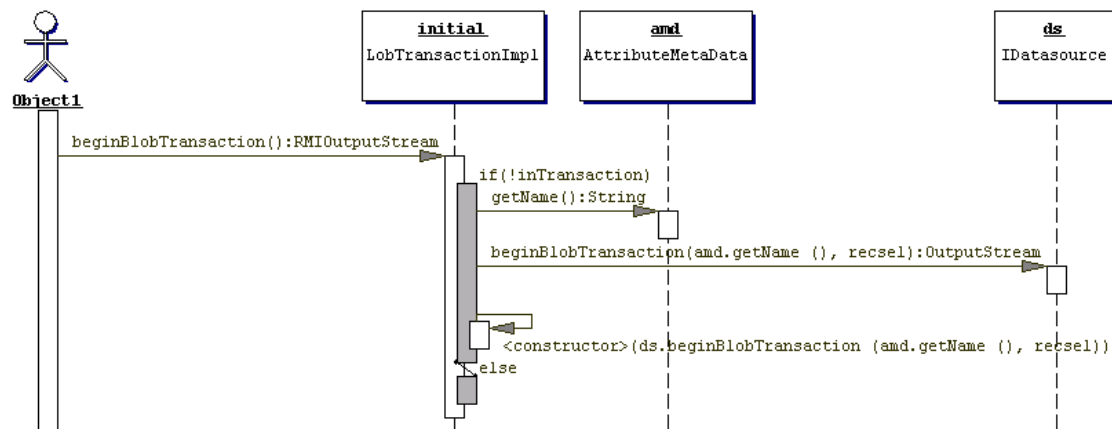

Abbildung 3.4.: Beginn einer Lob-Transaktion

3.4. Schnittstellen

3.4.1. Einleitung

Eine Middleware muß Schnittstellen nach oben und unten bereitstellen. Die Schnittstellen nach oben spezifizieren die Funktionen, die Clients in der Middleware aufrufen können. Die Schnittstellen nach unten sind die Funktionsanforderungen, die an die Treiber gestellt werden, und von diesen implementiert werden müssen.

3.4.2. Frontend-Schnittstelle

Das Frontend wird aufgeteilt in eine Schicht der Datenmanipulation, in der die Funktionen für den Zugriff auf eine Datasource festgeschrieben sind, und eine Schicht der Date-

nadministration, mittels derer Hilfe die virtuellen Strukturen von einem Client verwaltet werden können. Dazu wurde eine Trennung vorgenommen, indem die entsprechenden Bereiche in unterschiedlichen Interfaces definiert wurden. Die Datenmanipulation erfolgt mittels des Datasource-Interface, das im Rahmen der bereits beschriebenen gewünschten Operationen mehr als 30 Funktionen bereitstellt. Davon getrennt wurde ein weiteres Interface, die DatasourceFactory, dazu genutzt, die Datenadministrationsfunktionen zu bündeln, wobei über 40 Funktionen entworfen wurden. Als Ausnahme sind die Methoden der Benutzerverwaltung zu sehen, die in ein eigenes Interface, die UserFactory (weit über 20 Funktionen), ausgelagert wurden.

Die Implementierungen der Interfaces, welche die Funktionen zur Datenadministration spezifizieren, sind als Middlewarefunktionen komplett bereitzustellen. Der Umfang der Implementierung des Datasource-Interfaces ist abhängig vom verwendeten Backend.

3.4.3. Backend-Schnittstelle

Ebenso wie der Client-Entwickler hat auch der Entwickler eines Backend-Treibers exakt festgelegte Schnittstellen mit Leben zu füllen. Das ist zum einen das Backend-Interface, das alle Methoden definiert, über die ein Backend über sich selbst Auskunft geben muß. Diese Methoden sind immer zu implementieren, weil sie fundamental innerhalb der Middleware benötigt werden. Die zweite Schnittstelle, das IDatasource-Interface, ist wesentlich komplexer und braucht nur partiell implementiert zu werden. Sie beinhaltet in komprimierter Form alle Aufrufe, die ein Client zur Datenmanipulation abschicken könnte. Ein Beispiel: Das Datasource-Interface stellt drei verschiedene Varianten für ein SELECT zur Verfügung, auf der Backend-Schnittstelle existiert nur die komplexeste Ausprägung, eventuell fehlende Parameter werden innerhalb der Middleware durch Nullwerte ersetzt. Desweiteren existieren in diesem IDatasource-Interface Methoden, die der Entwickler eines Treibers füllen muß, etwa die Funktion, die darstellt, welche Funktionalitäten die Datasource mittels des dahinterliegenden Backends bietet.

4. Realisierung

4.1. Einleitung

In diesem Kapitel werden die spezifischen Lösungen innerhalb der Middleware vorgestellt. Dabei sind insbesondere die Lösungen für die oben angeschnittenen Probleme in ihrer Umgehung aufgeführt.

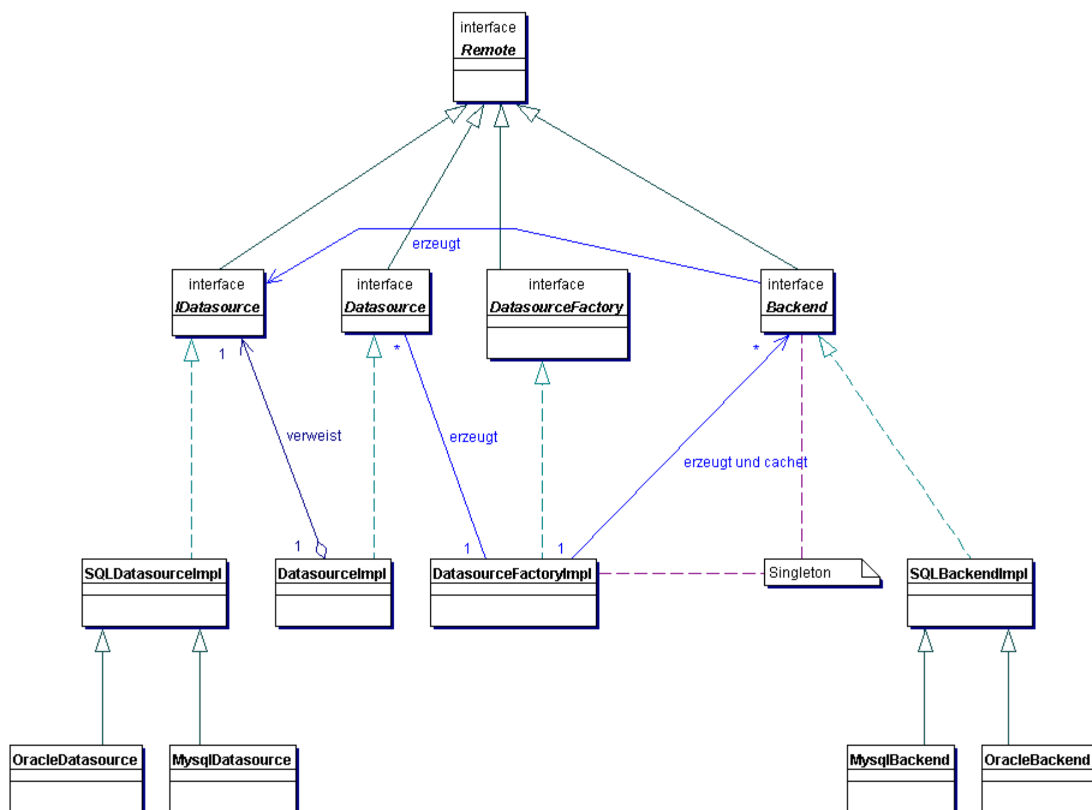

Abbildung 4.1.: Interner Aufbau der Middleware

4.2. Klassenstruktur der Middleware

Zur Strukturierung der Middleware wurden die Klassen in verschiedene Packages verschoben. Zunächst wurden die Klassen in die Server- und die Client-Klassen aufgeteilt, wobei der Client unter org.abla.client.* und die Middleware, die die eigentlichen Serverfunktionen bereitstellt, unter org.abla.server.* zu finden ist. Alle Klassen, die sich in irgendeiner Form mit dem Datenzugriff beschäftigen, stehen im org.abla.server.datasource Package, das sich noch weiter unterteilt in die Packages für die Fehlerstrukturierung (org.abla.server.datasource.exceptions), ein Package für die Informationen über der virtuellen Datenstrukturen (org.abla.server.datasource.meta) und ein Package, das die Treiber enthält (org.abla.server.datasource.provider). Die Benutzerverwaltung befindet sich im org.abla.server.usermgmt Package.

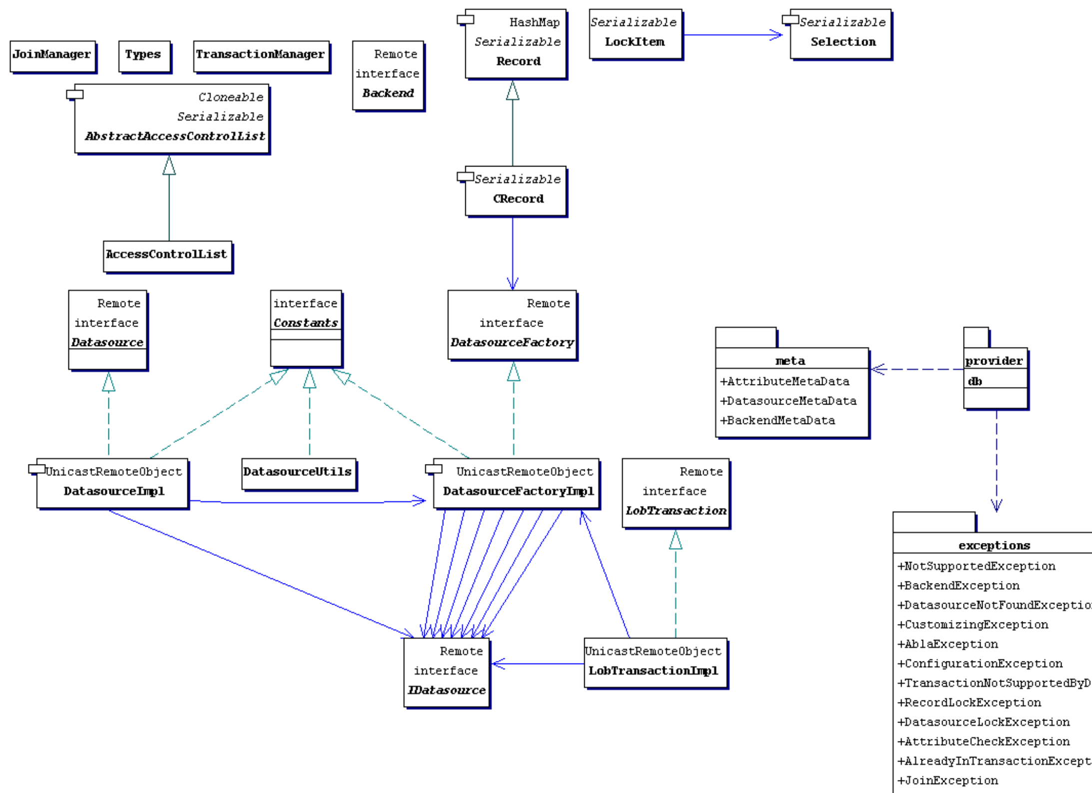

Abbildung 4.2.: Klassendiagramm des org.abla.server.datasource-Package

4.3. Datenzugriff

Das Design der Datenzugriffsfunktionen orientiert sich an SQL und bietet den Clients viele Möglichkeiten, Daten auszulesen und zu schreiben. Die Funktionen sind in zwei Interfaces unterteilt, wobei das eine die Methoden zur Abfrage und Manipulation der

konkreten Daten fasst, und das zweite die Funktionen zur Definition der Datenstrukturen. Das erste Interface liegt in einer komprimierten Form auch auf der Treiberebene als ein eigenständiges Interface vor. Siehe dazu die Erklärung unter 4.3.1 auf der nächsten Seite. Das zweite Interface kapselt zusätzlich zu den oben angesprochenen Funktionen auch die Factory-Methoden zur Erzeugung von Objekten zum Datenzugriff. Die beiden Interfaces sind innerhalb der Middleware implementiert. Das Datasource-Interface liegt in der DatasourceImpl-Klasse und die DatasourceFactory in der DatasourceFactoryImpl-Klasse vor. Wie die Namen schon andeuten, lassen sich die Instanzen der Datasources nur über die DatasourceFactory erzeugen. Dies ist der Tatsache geschuldet, daß ein DatasourceImpl-Objekt einen Proxy ([GOF01, Seite 254]) auf eine IDatasource-Instanz des zuständigen Treibers darstellt. Neben dem Weiterleiten der Aufrufe in die entsprechende Treiberklasse hat die DatasourceImpl-Klasse aber noch drei weitere wichtige Funktionen:

1. Prüfen der Berechtigungen und Anpassung der Aufrufe: Nachdem ein Client einen Aufruf in die Datasource, also eine Instanz der DatasourceImpl-Klasse, abgesetzt hat, wird er dort auf die Zulässigkeit abhängig vom zugeordneten UserContext überprüft. Jeder Datasource können positive und negative Berechtigungen zugewiesen werden. Positive Berechtigungen enthalten eine beliebige Kombination der Erlaubnisse zum selektieren, ändern, löschen und erzeugen von Datensätzen. Solche positiven Berechtigungen sind immer auf eine gesamte Datenquelle bezogen. Die negativen Berechtigungen können attributweise in einer Datasource gesetzt werden und verhindern dann den entsprechenden Funktionszugriff nur für das Attribut, für welches sie gesetzt wurden. Beispielsweise hat eine Rolle generell das positive Recht zum SELECT auf eine Datasource, welche die Mitarbeiterinformationen enthält, aber das negative Recht auf das Attribut, unter dem das Gehalt gespeichert wird. Bei einem Zugriff auf die Datasource durch einen Benutzer, der Mitglied dieser Rolle ist, wird nun transparent das Gehalt nicht mit zurückgeliefert. Diese Überprüfung und Anpassung eines Funktionsaufrufs findet in der DatasourceImpl-Klasse statt. Dazu wird zunächst eine Liste der für den jeweiligen Funktionsblock zulässigen Attribute erzeugt, die dann um die Attribute reduziert wird, die sich nicht in der Originalanfrage befanden. Diese automatische Änderung der Anfrage mag Verwirrung hervorrufen, weil üblicherweise eine solche (fehlerhafte) Anfrage mit einer Exception abgebrochen würde. Da die Middleware aber als 2-tier Software entwickelt wurde, ist es eher wünschenswert, den Clients eine flexible Schnittstelle anzubieten. Alternativ würde nämlich jeder Client selbst diese aufwendige Anpassung vornehmen müssen.

2. Aliasing/Unaliasing: Wenn ein Client eine Anfrage an eine DatasourceImpl-Instanz stellt, werden die Aliasnamen der Attribute genutzt. Dies geschieht um eine Abstraktion zu den physikalischen Namen zu etablieren. Diese Aliasnamen sind im Backend natürlich nicht bekannt und müssen daher entsprechend durch die DatasourceImpl-Instanz ersetzt werden, bevor der Aufruf in die IDatasource weitergeleitet wird. Wenn der Aufruf dann wieder zurückkehrt, gilt das umgekehrte Problem auch für die Ergebnis-Objekte. Sie enthalten die physikalischen Namen

und müssen daher vor der Rückgabe an den Client mit den Aliasnamen versehen werden.

3. Locks: Die Middleware bietet die Möglichkeit, bestimmte Datensätze für einen manipulativen Zugriff zu sperren. Diese Sperre wird durch die DatasourceImpl-Klasse realisiert, indem sie eine Liste der gesperrten Datentupel hält, die in der Datasource liegen, auf die die Instanz verweist, und bei jedem UPDATE oder DELETE die Selektionseinschränkung um die Tupel aus der Liste erweitert.

Neben der Erzeugung der DatasourceImpl-Instanzen ist die DatasourceFactoryImpl-Klasse für alle Strukturmanipulationen zuständig. Es lassen sich alle virtuellen Strukturen der Meta-Repository (Backends, Datasources, Attribute) anlegen, manipulieren und löschen. Desweiteren erzeugt und cachet die DatasourceFactoryImpl-Klasse die Backends (siehe 4.3.1) und hält die Meta-Informationen über die virtuellen Strukturen in gecacheter Form vor, da ständig auf diese zugegriffen werden muss und der sonst jeweils nötige Backendzugriff die Performance negativ beeinflussen würde. Als wichtigste Funktion implementiert die DatasourceFactory auch die konkreten Backendzugriffe, mit deren Hilfe die virtuellen Strukturen gespeichert werden. Dazu wird am Start der DatasourceFactory eine Backend-Instanz erzeugt, die nicht innerhalb einer kapselnden DatasourceImpl-Instanz läuft, sondern direkt genutzt wird. So verfügt die Middleware selbst über die Möglichkeit, die Meta-Repository mittels einem der verfügbaren Backend-Treiber zu verwalten.

4.3.1. Treiber

Die Middleware hat für den konkreten physikalischen Backendzugriff zwei abstrahierende Interfaces spezifiziert, die ein Treiber implementieren muß, will er innerhalb der Middleware funktionieren. Die Struktur eines klassischen Treibers zeigt die Abbildung 4.3 auf Seite 37. Die Trennung der Interfaces hat einen wesentlichen Grund in der Strukturierung der Treiber. Die Backend-Implementierung wird immer als Singleton in der Middleware verwaltet, während die Zahl der IDatasources unbegrenzt ist. Das Backend ist als Singleton vorgesehen, um dem Treiber bestimmte Funktionen in einer garantierten Umgebung zu ermöglichen. In der Backend-Implementierung des generischen SQL-Treibers ist das beispielsweise der ConnectionPool, mit dem die physikalischen Verbindungen in eine Datenbank gecachet werden. Funktional ist die Backend-Implementierung für die folgende Punkte zuständig:

1. Das Erzeugen der IDatasource-Instanzen geschieht innerhalb der Backend-Instanz in der getDatasource() Methode. Verlangt ein Client den Zugriff auf eine Datasource, ruft er an der DatasourceFactory die Funktion getDatasource() mit dem Namen oder der ID der Datasource und dem eigenen Benutzerkontext auf. Die DatasourceFactory prüft zunächst, ob der Benutzer überhaupt auf Datasources zugreifen darf und sollte dies der Fall sein, ob das entsprechende Backend existiert. Sollte bisher keine Instanz der Backend-Implementierung des zugeordneten Treibers existieren, wird zuallererst diese Treiberkomponente erzeugt. Nun kann

an dieser Backend-Instanz die Methode getDatasource() aufgerufen werden, wobei der physikalische Name der gewünschten Datasource übergeben wird, damit dann eine IDatasource-Instanz erzeugt werden kann, die in die DatasourceFactory zurückgeliefert wird. Die Backend-Instanz hat dabei durchaus die Möglichkeit, abhängig von der gewünschten Datasource unterschiedliche Implementierungen des IDatasource-Interface zu verwenden. Allerdings ist die Wiederverwendung einer bereits einmal erzeugten IDatasource-Instanz verboten, sie muß immer neu instantiiert werden. Zurück in der DatasourceFactory wird die gerade frisch erzeugte IDatasource-Instanz in einer DatasourceImpl-Instanz gekapselt, wobei zusätzlich die virtuellen Strukturinformationen der Datasource und der bei dem Aufruf mitgelieferte Benutzerkontext mit übergeben werden. Dies hat zur Konsequenz, daß die Datasource nun jeden Aufruf ihrerseits mit diesem Nutzerkontext autorisiert.

2. Das Finden von Datasources im Backend muß vom Treiber bereitgestellt werden, um das Einbinden von bereits vorhandenen Strukturen als Datasources zu erleichtern. Dazu kann ein Client einen einschränkenden String mit den Wildcards * und ? nutzen oder sich alle Datasources liefern lassen, die physikalisch im Backend existieren. Bevor ein Client allerdings diese Liste bekommt, wird sie um die Datasources bereinigt, die bereits in der Middleware bekannt sind.

3. Die Beschreibung einer Datasource ist der dritte Funktionsblock. Dabei wird dem Backend der physikalische Name mitgeteilt und eine Middleware-kompatible Strukturbeschreibung mit Hilfe der Meta-Objekte (siehe 4.5 auf Seite 44) zurückerwartet. Diese Funktion macht das Browsen von vorhandenen Datasources erst wirklich sinnvoll, weil so ein Client sich erst die Datasource sucht, die in der Middleware bekannt gemacht werden soll, und dann anhand des Namens sofort der Aufbau der Datasource ermittelt werden kann. Es müssen lediglich noch die Aliasnamen der einzelnen Attribute entsprechend den Anforderungen des Clients gesetzt werden.

Das IDatasource-Interface ist so gehalten, daß es den Zugriff auf exakt eine physikalische Ressource gestattet. An Funktionen hat eine Implementierung die folgenden Punkte immer zu unterstützen:

1. Die Instanz muß den physikalischen Namen der zugeordneten Datasource nennen können.

2. Jede Implementierung muß den eigenen Funktionsumfang darstellen können. Dazu ist eine Methode getCapabilities() im Interface definiert, die in der Implementierung einen Long Wert zurückliefern soll, in den bitweise die unterstützten Methoden eincodiert sind. Diese Bits sind im IDatasource-Interface bereits spezifiziert und können dann mittels der Bitoperation OR miteinander verknüpft werden.

3. Für jedes Attribut muß ein Treiber ermitteln können, ob es sich um ein binäres Attribut handelt. Solche Attribute werden von der Verwendung in Selection-Chains (siehe 4.3.3 auf Seite 39) ausgeschlossen.

4. Existenzprüfung ist eine weitere notwendige Funktion. Sie muß ermitteln können, ob die physikalische Struktur, wie sie innerhalb der Middleware eingetragen ist, überhaupt im Backend existiert.

5. Äußerst wichtig ist desweiteren eine Funktion, die die Liste der Schlüsselattribute bereitstellt.

Die nun folgenden Funktionen sind innerhalb des IDatasource-Interfaces zwar spezifiziert, müssen aber nicht implementiert werden. Die Methoden werfen in diesem Fall einfach nur eine NotSupportedException.

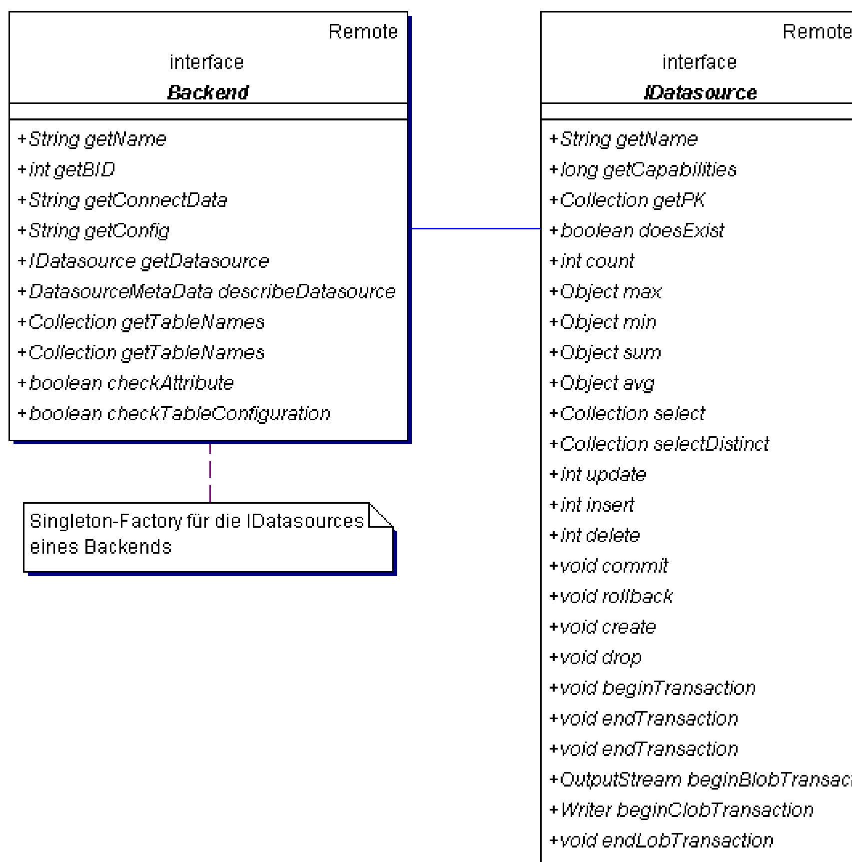

Abbildung 4.3.: Aufbau eines Middleware-Treibers

1. Eine erste Gruppe umfasst alle Methoden, die normalerweise innerhalb einer SELECT-Anweisung aufgerufen werden und dort mathematische Funktionen auf selektierten Ergebnistupeln ausführen. Im Interface handelt es sich dabei um avg(), count(), min(), max() und sum(). Wirklich Sinn machen diese Aufrufe natürlich nur auf zahlenbasierten Attributen.

2. Die wichtigste Gruppe umfasst die Gruppe der Selektionen. Es handelt sich dabei um zwei Methoden, zum einen das select(), zum anderen das selectDistinct(). Es werden jeweils die Liste mit den gewünschten Attributen, die Selection-Chain und die Liste mit den ORDER BY-Attributen übergeben.

3. Wichtig ist ebenso die insert() Funktion, die als einzigen Parameter einen Record übergeben bekommt, der in der Datasource gespeichert werden soll.

4. Zur Gruppe der manipulativen Methoden gehören zwei Funktionen: die update() Methode bekommt eine Selection-Chain und einen Record übergeben und ersetzt für alle passenden Datensätze alle Attributinhalte durch die Inhalte im Record-Objekt. Die delete() Methode löscht alle Datensätze, die der übergebenen Selection-Chain entsprechen.

5. Etwas globalere Methoden sind die parameterlosen create() und drop() Methoden, die die gesamte Datasource physikalisch löschen beziehungsweise erzeugen.

6. Zur Transaktionsunterstützung existieren vier Methoden: beginTransaction() und endTransaction() beginnen und beenden eine Transaktion, wobei mit letzterem auch ein commit() oder ein rollback(), also die Rücknahme der ausgeführten Operationen, durchgeführt werden kann. Befindet sich die IDatasource bereits in einer Transaktion, können auch alle bereits erfolgten Manipulationen mittels commit() finalisiert oder mittels rollback() zurückgenommen werden. Diese Aufrufe sind final und nicht mehr korrigierbar; allerdings befindet sich die Datasource danach immer noch in einer Transaktion.

7. In der letzten Funktionsgruppe finden sich die Methoden für die LobTransaktionen. Zum einen zwei Varianten (für BLOBs und CLOBs) zum Beginnen einer solchen Transaktion und eine Methode zum Beenden der Transaktion (mit commit() oder rollback()). Da die Datasource sich in der Zeit einer LobTransaktion in einer Transaktion befindet, kann auch immer nur eine LobTransaktion pro Datasource parallel durchgeführt werden.

4.3.2. Datentupel-Kapselung

Die Daten welche aus einer Datasource gelesen wurden oder in eine solche geschrieben werden sollen, müssen in einer strukturierten, aber Backend-unabhängigen Variante vorliegen. Zu diesem Zweck wurden zwei Klassen eingeführt, der Record und der CRecord, der die erste Klasse um speziellere Funktionen erweitert. Abbildung 4.4 auf der nächsten Seite zeigt diese Struktur deutlich. Während die erste Klasse lediglich Methoden für einen einfacheren Zugriff auf die enthaltenen Elemente bereithält, ist die CRecord-Klasse wesentlich umfangreicher. Ein CRecord wird immer dann innerhalb der Treiberschicht erzeugt, wenn alle Attribute abgefragt werden. So wird sichergestellt, daß alle Schlüsselattribute enthalten sind. Ein CRecord kann sich dementsprechend auch selbst wieder im Backend auf den aktuellen Stand bringen, eine LobTransaktion auf ein enthaltenes Attribut beginnen, sowie den Datensatz innerhalb der Middleware locken

(siehe dazu Seite 33 und Seite 26). Dies geschieht durch einen Link innerhalb des CRecords auf die DatasourceFactory sowie die Mitgabe des Nutzerkontext bei der Erzeugung des CRecords.

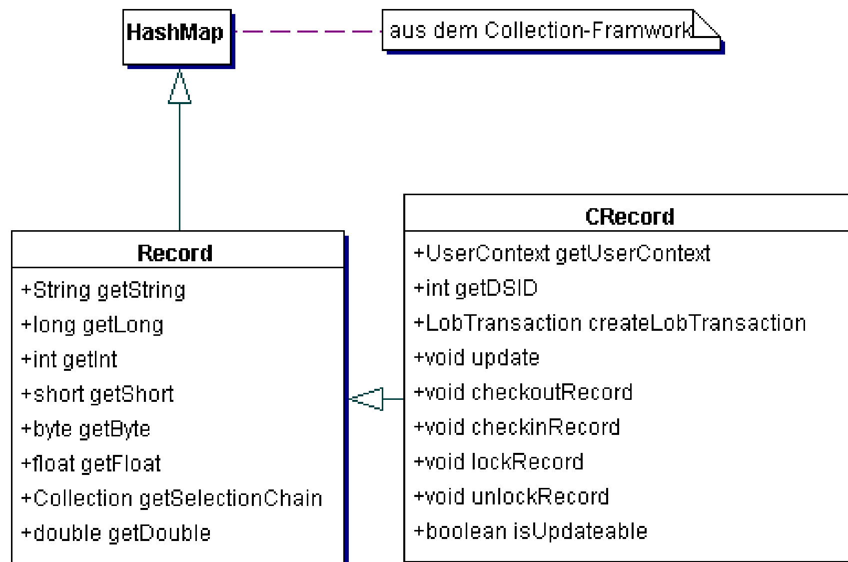

Abbildung 4.4.: Klassendiagramm der Klassen, die Datentupel enthalten

4.3.3. Selection-Chains

Neben der Abstraktion der Daten, die aus einer Datasource stammen, müssen auch die Bedingungen, die zur Einschränkung von Selektionen genutzt werden, abstrahiert von einem Backend zur Verfügung stehen. Dieses Problem wurde mit Hilfe eines Objektcontainers namens Selection gelöst. Ein solcher Container enthält einen Verknüpfungsoperator (AND, OR, AND NOT und OR Not), einen Attributbezeichner, sowie einen Wert, der für den gewünschten Zielwert steht. Wenn Anfragen verschachtelt werden sollen, kann ein Selection-Objekt auch eine Liste von anderen Selection-Objekten aufnehmen.

Diese Kette von Selection-Objekten wird Selection-Chain genannt. Auf Treiberebene besteht bei fast jeder Methode die Möglichkeit, eine Selection-Chain zu übergeben. Bevor die vom Client definierte Selection-Chain allerdings im Treiber ankommt, muß sie natürlich aliased werden. Es werden also die Aliasnamen, die ein Client für die Attributbezeichner benutzt hat, durch die physikalischen Namen, wie sie im Backend vorkommen, ersetzt. Im Treiber muß die Selection-Chain nun in ein kompatibles Format umgewandelt werden; in den entwickelten Datenbanktreibern wird beispielsweise die Selection-Chain in einen SQL-WHERE-String umgewandelt.

4.4. Interner Aufbau der Meta-Repository

Die virtuellen Strukturen, die den Client-Anwendungen zur Verfügung gestellt werden, stellen eine einheitliche Sicht auf unterschiedliche Attributmengen dar. Diese Sichten werden innerhalb der Middleware genutzt, um die divergierenden Formate und Systeme der Backends für Clients zu vereinheitlichen. Die geschieht mittels Datasources, die strukturiert mit den Daten gefüllt werden. Im folgenden soll dieser interne Aufbau der Meta-Repository dargestellt werden. Dazu wird jede Datasource mit ihren Strukturen erst beschrieben und dann die inhaltliche Verknüpfung mit anderen Datasources näher erläutert. Ein Treiber, der als internes Backend diese Strukturen verwalten kann, muß die folgenden Funktionen unterstützen: count(), max(), select(), selectDistinct(), update(), insert(), create(), drop(), sowie alle Transaktionsfunktionen.

Einen Überblick über die Meta-Repository mit ihren internen Referenzen zeigt Abbildung 4.5.

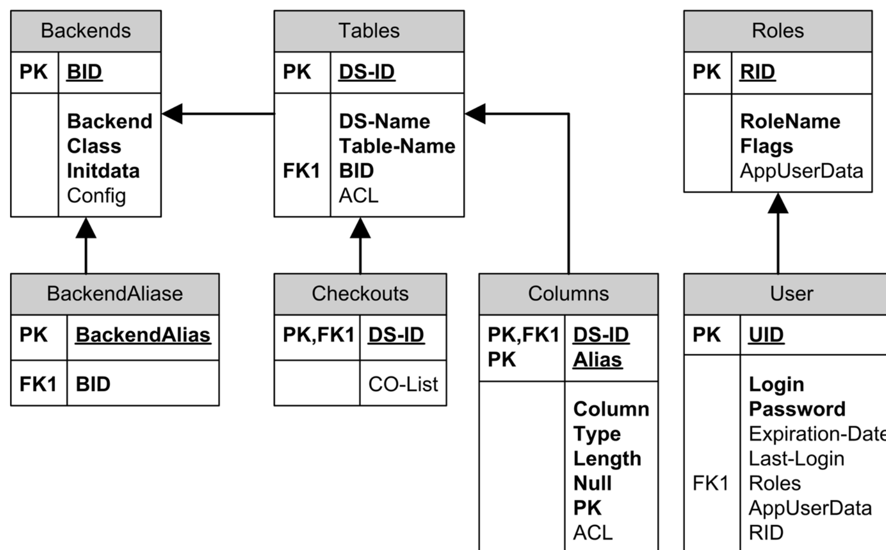

Abbildung 4.5.: Interne Verknüpfungen der Meta-Repository

4.4.1. Tables

Die Datasource Tables beschreibt die grundsätzlichen Daten einer Datasource. Ruft ein Client eine Funktion auf, die sich auf eine Datasource bezieht, wird anhand der Datasource Tables ermittelt, welche IDs die zuständigen Verbundobjekte, wie etwa das Backend haben.

Die Backend-ID bezieht sich auf die Datasource Backends.

Name	Datentyp	Null	PK	Erläuterungen
DS-ID	Integer	o	x	eindeutige ID der Datasource
DS-Name	Varchar (64)	o	o	eindeutiger Name der Datasource
Table-Name	Varchar (64)	o	o	physikalischer Name der Struktur
Backend-ID	Integer	o	o	ID des zugehörigen Backends
ACL	Java-Object	x	o	Objekt für die positive ACL Liste

Tabelle 4.1.: Struktur der internen Datasource Tables

4.4.2. Columns

Die Datasource COLUMNS beschreibt die Attribute einer Datasource. Dabei entspricht ein Datentupel jeweils einem Attribut. Mit der ID der gesuchten Datasource lassen sich also alle Attribute finden.

Die Datasource-ID (DSID) bezieht sich auf das entsprechende Attribut in der Datasource TABLES.

Name	Datentyp	Null	PK	Erläuterungen
DSID	Integer	o	x	eindeutige ID der Datasource
Alias	Varchar (64)	o	x	Alias-Bezeichner des Attributs
Column	Varchar (64)	o	o	physikalischer Name des Attributs
Type	Small-Int	o	o	ID des Datentyps
Length	Integer	o	o	Parameter zum Datentyp
Null	Bit	o	o	Nullwerte erlaubt
PK	Bit	o	o	Teil der Schlüsselmenge
ACL	Java-Object	x	o	Objekt für die negative ACL-Liste

Tabelle 4.2.: Struktur der internen Datasource Columns

4.4.3. Checkouts

Die Datasource CHECKOUTS speichert alle gesicherten Datensätze, die über einen Neustart der Middleware hinaus gesichert werden sollen. Dies geschieht mittels serialisierter Objekte, die jeweils alle Checkouts, die eine Datasource betreffen, speichern. Beim Start der Middleware wird nach der Initialisierung diese Datasource ausgelesen und die Daten werden in die im Speicher befindlichen Checkout-Listen wieder eingesetzt. Nähere Erläuterungen finden sich unter 4.7 auf Seite 48.

Die Datasource-ID (DSID) bezieht sich auf das entsprechende Attribut in der Datasource TABLES.

Name	Datentyp	Null	PK	Erläuterungen
DSID	Integer	o	x	eindeutige ID der Datasource
CO-List	Java-Object	x	o	serialisierte Liste der Checkouts

Tabelle 4.3.: Struktur der internen Datasource Checkouts

4.4.4. Backends

Diese Datasource speichert die Daten, die ein Backend, entsprechend der Feststellung auf Seite 24, benötigt. Insbesondere wenn ein Treiber initialisiert wird, also das Backend-Singleton erzeugt wird, wird die Datasource Backends ausgelesen. Anschließende Aufrufe werden direkt in die dann im Speicher befindliche Instanz gestellt, anstatt einen aufwendigen Datasourcezugriff durchzuführen. Prinzipiell wird in der Middleware zum Erzeugen des Backend der Inhalt des Feldes Class ausgelesen und es wird versucht eine Instanz der Klasse zu erzeugen, wobei die Initialisierungsdaten übergeben werden. Der implementierte generische Treiber setzt beispielsweise voraus, das der Initialisierungsstring in einer Variante der Form {<JDBC-Treiber-Klasse> <JDBC-URL> <Username> <Password>} spezifiziert wird.

Name	Datentyp	Null	PK	Erläuterungen
BID	Integer	o	x	eindeutige ID des Backend
Backend	Varchar (64)	o	o	eindeutiger Name des Backend
Class	Varchar (255)	o	o	Klassenname des Backend-Singleton
Initdata	Varchar (255)	o	o	erster Initialisierungs-String
Configdata	Varchar (255)	x	o	zweiter Initialisierungs-String

Tabelle 4.4.: Struktur der internen Datasource Backends

4.4.5. Backend-Aliase

Jedem Backend können Aliasbezeichner zugewiesen werden. Dies ist eine weitere Abstraktionsebene für Clients, deren gewünschter Backendbezeichner dem Backend zugeteilt wird, dessen Inhalte dem Client bekannt gemacht werden sollen. Dabei ist wichtig, daß der originale Backendbezeichner, wie er auch in der Datasource Backends steht, in dieser Datasource ebenfalls auftaucht, um doppelte Bezeichner zu vermeiden.

Die Backend-ID (Bid) bezieht sich auf das entsprechende Attribut in der Datasource Backends.

4.4.6. User

Die Datasource User speichert alle Informationen, die einen Benutzer betreffen. Neben den Hauptdaten zählen dazu die Liste der Rollen, deren Rechte kumulativ verknüpft

Name	Datentyp	Null	PK	Erläuterungen
BID	Integer	o	o	eindeutige ID des Backend
Backend-Alias	Varchar (255)	o	x	eindeutiger Bezeichner des Backend

Tabelle 4.5.: Struktur der internen Datasource Backend-Aliase

werden, und das AppUserData-Objekt. Erläuterungen zu diesem Objekt finden sich bei 3.3.5 auf Seite 27.

Das Attribut ROLES bezieht sich auf die Role-ID der Datasource ROLES.

Name	Datentyp	Null	PK	Erläuterungen
UID	Integer	o	x	eindeutige ID des Benutzers
Login	Varchar (32)	o	o	Loginname
Password	Varchar (32)	o	o	Passwort
Expiration-Date	Timestamp	x	o	Zeitpunkt, zu dem der Account abläuft
Last-Login	Timestamp	x	o	letzter erfolgreicher Login
Roles	Varchar (255)	x	o	Liste der Rollen des Nutzers
AppUserData	Java-Object	x	o	Client-Daten für den Nutzer

Tabelle 4.6.: Struktur der internen Datasource User

4.4.7. Roles

Die Datasource ROLES enthält die Informationen der Rollen im System der Middleware. Dazu zählen neben der ID, den Rechte-Bits und dem Namen auch wieder AppUserData-Objekt, das einem Client die Möglichkeit verschafft, an einer Rolle weitere Informationen zu speichern. Ausführliche Erklärungen zu diesem Objekt und der dahinter stehenden Motivation finden sich bei 3.3.5 auf Seite 27.

Name	Datentyp	Null	PK	Erläuterungen
RID	Integer	o	x	eindeutige ID der Rolle
Role-Name	Varchar (64)	o	o	Name der Rolle
Flags	Big-Int	o	o	Rechtebits
AppUserData	Java-Object	x	o	Client-Daten für die Rolle

Tabelle 4.7.: Struktur der internen Datasource Roles

4.5. Meta-Informationen

Die Meta-Repository, in der die virtuellen Strukturen der Middleware gehalten werden, ist durch Objekte verfügbar. Dies ist notwendig, um die Informationen strukturiert für Clients bereitstellen zu können. Dabei werden die folgenden Strukturen unterschieden:

- Die Informationen, die ein Backend betreffen (entsprechend der Zuordnung unter 3.2.1 auf Seite 24), liegen in der BackendMetaData-Klasse. Hier werden der Name (und die Aliase) des Backends und die Informationen, die für die korrekte Konfiguration des Treibers vonnöten sind, bereitgestellt:

  Program 4.1: BackendMetaData.java

  ```
  1 public String getBackendName ()
  2 public String getClassName ()
  3 public String getConnectData ()
  4 public String getConfigData ()
  5 public int getBID ()
  6 public Collection getAliases ()
  7 public void setAliases (Collection aliases )
  ```

- Alle Informationen, die einem Attribut zugeordnet werden (entsprechend der Zuordnung unter 3.2.2 auf Seite 24), sind in der AttributeMetaData-Klasse gespeichert:

  Program 4.2: AttributeMetaData.java

  ```
  1 public AccessControlList getACL ()
  2 public String getAlias ()
  3 public int getLength ()
  4 public String getName ()
  5 public boolean getNullable ()
  6 public int getType ()
  7 public boolean getPK ()
  ```

- Auch die Informationen der Meta-Repository über eine Datasource (entsprechend der Zuordnung unter 3.2.3 auf Seite 25) können über ein Objekt abgefragt werden, entsprechend heißt die Klasse DatasourceMetaData:

  Program 4.3: DatasourceMetaData.java

  ```
  1 public int getDSID ()
  2 public String getDSName ()
  3 public String getTableName ()
  4 public int getBID ()
  5 public AccessControlList getACL ()
  6 public Collection getAttributes ()
  7 public void setAttributes (Collection attributes)
  8 public Collection getInternalPK ()
  9 public Collection getPK ()
  ```

Wichtig ist dabei zu beachten, daß auch wenn einige der vorgestellten Klassen Setter-Methoden besitzen, diese nur objektlokal arbeiten und nicht die virtuellen Strukturen in der Meta-Repository ändern. Um die Strukturen zu ändern oder neue zu erstellen, müssen die Methoden der DatasourceFactory genutzt werden.

4.6. Provider

Entsprechend der Spezifikation in 4.3.1 auf Seite 35 wurden mehrere Treiber erstellt. Als einfache Lösung ein Treiber für das OpenSource-DBMS Mysql und als komplexe Lösung, die auch Transaktionen unterstützt, ein Oracle-Treiber. Um der Doppelarbeit zu entgehen, war es sinnvoll, einen generischen JDBC-basierten Datenbanktreiber zu schreiben, dessen Methoden abhängig von den spezifischen Implementierungen nur noch überschrieben werden müssen. Der generische Treiber hat einen Umfang von über 2500 Zeilen, der Mysql-Treiber benötigte dann mit seinen Anpassungen nur noch 1100 Zeilen und der Oracle Treiber nur noch 1200 Zeilen.

4.6.1. Generischer SQL-Treiber

Der generische SQL-Treiber wurde als Basis für die Datenbanken entwickelt, die per JDBC ansprechbar sind, um die bei jeder Datenbank auftretenden Problemstellungen nicht immer wieder neu lösen zu müssen. Dazu wurden die potentiell möglichen SQL-Kommandos so partitioniert, daß jeder darauf aufbauende Treiber nur noch die entsprechenden Methoden mit den differierenden SQL-Bestandteilen zu überschreiben braucht. Als Beispiel mag nachfolgend die Struktur der IDatasource-Implementierung für das select() diese Partitionierung zeigen:

Program 4.4: SQLDatasource Implementierung

```
1  public class SQLDatasourceImpl implements IDatasource {
2      public Collection select  ( ... )
3      protected StringBuffer createSelectPrefix  ( ... )
4      protected StringBuffer createSelectDistinctPrefix  ( ... )
5      protected StringBuffer createSelectColumnDivider ( ... )
6      protected StringBuffer createSelectOrderDivider  ( ... )
7      protected StringBuffer createSelectColumnList  ( ... )
8      protected StringBuffer createSelectFrom  ( ... )
9      protected StringBuffer createSelectWhere  ( ... )
10     protected StringBuffer createSelectOrder  ( ... )
11     protected StringBuffer createSelectOrderList  ( ... )
12     protected StringBuffer createSelectPostSQL ( ... )
13     protected StringBuffer createFunctionSelectPostSQL  ( ... )
14 }
```

In der Aufrufsequenz (Abbildung 4.6 auf der nächsten Seite als UML-Sequenzdiagramms) werden die Verschachtelungen noch deutlicher.

Außerdem wurde ein generischer ConnectionPool entwickelt, der jedes Java-Objekt als Connection verwalten kann. Die spezifischen Implementierungen, die den ConnectionPool nutzen wollen, müssen nur noch drei Funktionen überschreiben. Das sind die Methoden zur Erzeugung, Validierung und Löschung einer Connection.

Desweiteren müssen in jedem Treiber die Datentypkonversionen backendspezifisch realisiert werden. Dabei handelt es sich um die Konvertierungen, die auf die Werte angewendet werden, die in der Datenbank gespeichert werden sollen oder von selbiger als Rückgabewerte geliefert wurden. Ein Beispiel: eine Datasource hat ein Attribut vom Datentyp BIT, also einem boolschen Wert. Da die Datenbank Mysql aber keinen entsprechenden Typen kennt, wird stattdessen der Datentyp TINYINT als zu nutzender

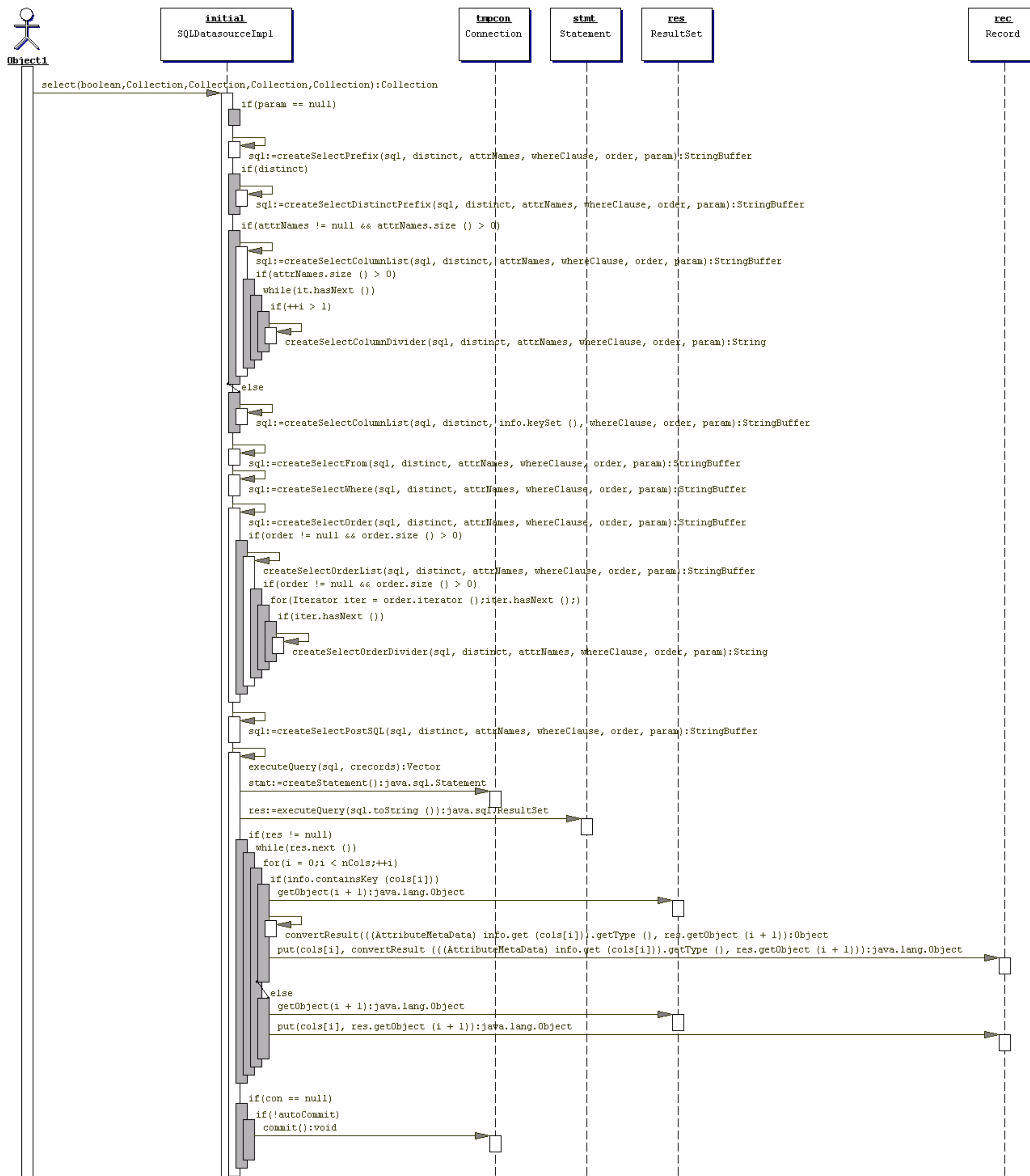

Abbildung 4.6.: Aufrufpartitionierung des generischen SQL-Treibers. Deutlich sind die internen Queraufrufe zu erkennen, die den eigentlichen SQL String erzeugen und daher bei abgeleiteten Klassen lediglich partiell überschrieben werden müssen.

Typ festgelegt. Wenn nun ein Boolean geschrieben werden soll, wandelt der Mysql-Treiber den Boolean in einen Zahlwert um, wobei nur die Werte 0 (für falsch) und 1 (für wahr) auftreten können. Wird später aus dieser Datasource wieder gelesen, weiß der Treiber aufgrund der Meta-Informationen, daß der Client einen Boolean erwartet und konvertiert daher den Zahlwert entsprechend wieder zurück.

Eine Besonderheit besteht in der Implementierung der Transaktionen. Da die Datasource nicht für die gesamte Lebenszeit eine physikalische Connection in das Backend halten soll, wird nach dem Bilden des zu verwendenden SQL-Strings eigentlich immer dynamisch eine JDBC-Connection aus dem ConnectionPool geholt, die dann zum Absenden eines Statements und dem Auslesen des ResultSets genutzt werden kann. Eine Ausnahme bildet allerdings die Transaktion, die in JDBC immer auf derselben Connection erfolgen muß. Daher wird in der beginTransaction() Methode eine Connection geholt und in einer lokalen Referenz gebunden. Anschließend wird diese Connection in den Transaktionsmodus versetzt. Wird nun ein Funktionsaufruf ausgeführt, prüft die Implementierung zuallererst, ob lokal eine Connection vorliegt, was bedeutet, daß sich die IDatasource in einer Transaktion befindet, und benutzt bei positivem Bescheid diese dann auch, oder holt sich sonst eine neue Connection aus dem ConnectionPool. Ein abschließendes endTransaction() liefert die gebundene Connection wieder in den ConnectionPool zurück und die lokale Referenz wird auf null zurückgesetzt.

Eine Funktion die nicht implementiert werden konnte, sind LobTransaktionen, da es für diesen Typ keinen generischen Lösungsansatz gibt.

4.6.2. Mysql-Treiber

Der Mysql-Treiber basiert auf dem generischen Treiber, der um die Spezialitäten von Mysql erweitert wurden. Dazu zählen die Mysql-spezifischen Datentypkonvertierungen, die Lieferung der in einer Mysql-Datenbank vorhandenen Tabellen und die Prüfung, ob eine Tabelle existiert. Eine Besonderheit gab es bei Implementierung des LOB-Protokolls, da Mysql derzeit keine Streams für LOBs unterstützt und daher auf Objekte, die komplett im Speicher gehalten werden, zurückgegriffen werden musste. Dazu wird im Treiber ein Byte-Array erstellt, das per Stream einem Client zur Verfügung gestellt wird. Ist der Schreibvorgang abgeschlossen, wird innerhalb des Treibers das im Speicher liegende Objekt mittels eines normalen UPDATE finalisiert. Während einer Lob-Transaktion befindet sich die eigentliche Datasource nicht in einer Transaktion.

Eine Transaktionsunterstützung konnte nicht implementiert werden, da die entsprechenden Funktionen in Mysql nicht bereitstehen.

4.6.3. Oracle-Treiber

Ebenso wie der Mysql-Treiber, basiert auch der Oracle-Treiber auf der generischen SQL-Lösung. Bei den Änderungen handelt es sich hauptsächlich um die immer notwendige Anpassung der Datentypkonvertierungen, die Änderung des ConnectionPools bezüglich der bei den Oracle JDBC-Treibern geänderten Connectionvalidierung, sowie die Implementierung des LOB-Protokolls, das bei Oracle als echte Streamlösung möglich ist.

Die Lob-Unterstützung basiert auf einer Transaktion, für die das Tupel, das das zu füllende Blobfeld enthält, mit einem Oracle spezifischen SELECT FOR UPDATE gesperrt wird. Das daraufhin gelieferte Blob- beziehungsweise Clob-Objekt wird mittels Adapterklasse ([GOF01, Seite 171]) zum Client getunnelt. Nach dem Abschluss der Schreiboperation wird die Transaktion beendet. Die Datasource-Klasse ist während einer Lob-Operation in einer Transaktion.

4.7. Sicherung von Datensätzen

Die Sicherung erfolgt über eine Liste von Objekten des Typs LockItem. Eine Instanz von LockItem kapselt genau einen zu sperrenden Datensatz, indem eine Selection-Chain für diesen Tupel gespeichert wird. Für jede Datasource existiert eine Liste, die die Locks innerhalb dieser Datasource speichert. Bei einem UPDATE- oder DELETE-Aufruf wird die durch den Client übergebene Selection-Chain um die in der für diese Datasource vorhandenen Liste gespeicherten LockItems erweitert. Dies geschieht dann entsprechend in der negierten Form. Ein Beispiel: es wurde ein Datensatz mit den Schlüsselwerten $id = 10$ && $foo = abc$ gesichert. Nun kommt eine UPDATE-Anweisung mit der Selection-Chain WHERE $id > 5$. Diese würde dann erweitert zur Selection-Chain WHERE $id > 5$ AND NOT ($id = 10$ AND $foo = abc$).

4.8. Verteilung

Bei der Arbeit mit RMI sind einige große Unterschiede in der Art der Verwendung von entfernten und lokalen Objekten festzuhalten. Es ist auf den ersten Blick nicht feststellbar, ob ein Objekt lokal in der Virtual Machine oder als entferntes Objekt in einer anderen Virtual Machine vorliegt. Dieses ist aber entscheidend für die Verarbeitung von Methodenaufrufen. Parameter, die an lokale Objekte übergeben werden, werden per *call-by-reference* übermittelt. Dies ist bei Serverobjekten anders, hier werden Parameter immer als *call-by-value* übergeben, das bedeutet, sie werden in ihren Werten kopiert. Dazu werden die Parameter serialisiert, also in einen über einen Socket übermittelbaren Bytestrom umgewandelt. Die Gegenseite erstellt dann aus dem Bytestrom wiederum neue Objekte, die Kopien der ursprünglichen Parameter darstellen. Neben den physikalisch gebundenen Objekten musste abgewogen werden, inwieweit die häufige Nutzung als Parameter für nachfolgende Methodenaufrufe ein Objekt als Serverobjekt qualifiziert. Zu diesen Fällen gehören beispielsweise alle Objekte der Benutzerverwaltung, da diese zur Identifizierung immer gebraucht werden. Ein weiterer Punkt ist der Speicherverbrauch. Werden *call-by-value* Aufrufe sehr parallelisiert, kann es vorkommen, daß inhaltlich identische Objekte mehrfach den Speicher der Middleware belegen. Durch den Einsatz eines einzelnen Objektes, dessen Referenz den verschiedenen Clients zur Verfügung gestellt wird, kann dieses Problem umgangen werden. Zu den wichtigen Objekten, die als Server-Objekte immer in der virtuellen Maschine (VM) der Middleware laufen zählen die folgenden:

1. DatasourceFactory: die zentrale Klasse der ganzen Middleware ist natürlich ein Server-Objekt und ein Singleton, weil die Klasse alle Zugriffe auf die virtuellen Strukturen der Meta-Repository durchführt und außerdem als Factory alle Datasources und Backends bereitstellt.

2. Backends: Alle Implementierungen des Backend-Interfaces laufen als Singleton (ausführlich Begründung siehe 4.3.1 auf Seite 35) und als Server-Objekte in derselben virtuellen Maschine wie die DatasourceFactory. Hauptsächlich ist das aufgrund der gewünschten Verwaltung der physikalischen Ressourcen durch die Backend-Implementierung der Fall. Zusätzlich muß der Zugriff auf ein Backend aus der Middleware heraus möglichst effizient erfolgen können.

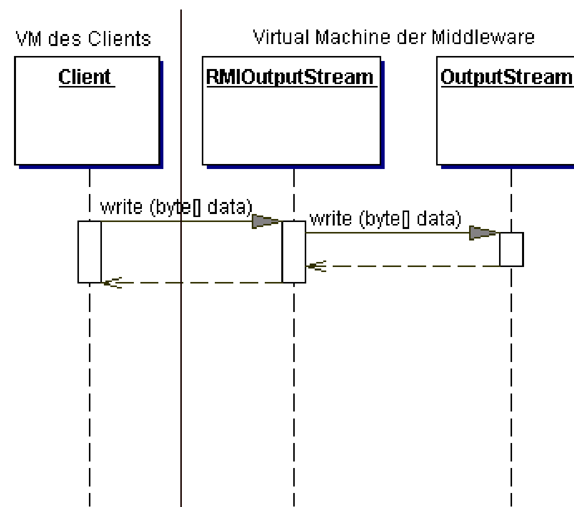

Abbildung 4.7.: Wrapperfunktion der Hilfsklassen für Streams

3. IDatasources: Die Implementierungen des IDatasource-Interface sind dementsprechend auch Server-Objekte, weil sie (zumindestens temporär) physikalische Verbindungen, etwa auch in das lokale Dateisystem, beinhalten können.

4. DatasourceImpl: Die Implementierung des Datasource-Interface als DatasourceImpl, das grundsätzlich als Proxy ([GOF01, Seite 254]) auf eine IDatasource-Instanz verstanden werden kann, ist aus Performancegründen ein Server-Objekt. Da viele Aufrufe innerhalb der Klasse in die Datasource beziehungsweise in Treiber-Klassen gehen, sollten diese Aufrufe keinen unnötigen RMI-Ballast erzeugen. Dies wird insbesondere daran deutlich, daß ein DatasourceImpl-Aufruf mehrere Aufrufe aus der DatasourceImpl-Instanz in den Treiber beziehungsweise in die DatasourceFactory nach sich zieht.

5. LobTransaction: Die Implementierung der LobTransaction ist ebenso wie die vorangegangene Klasse aus Perfomancegründen ein Server-Objekt. In der Klasse erfolgen mehrere Aufrufe sowohl in eine Datasource, als auch in die DatasourceFactory.

6. Benutzersystem: Die Factory, die die Benutzerinformationen verwaltet ist auch ein Server-Objekt. Zum einen, weil sie natürlich auch direkt die Meta-Repository bearbeitet, zum anderen, weil sie so oft funktional benötigt wird, daß Performanceüberlegungen eine Rolle gespielt haben. Neben der UserFactory sind auch die Benutzer- und Rollenimplementierungen als Server-Objekte ausgelegt. Zusätzlich laufen sie auch als Singleton. Letzteres geschieht, um Manipulationen an den wichtigen Klassen zu verhindern und um die in der Regel häufig parallelen Informationen nicht mehrfach verwalten zu müssen. Desweiteren ist die Erzeugung eines Benutzerkontextes auch keine unkomplizierte Angelegenheit, weil alle Rollen, in denen der Nutzer Mitglied ist, ebenso mit gelesen werden müssen, um die Rechte und Nutzerobjekte in ihren Informationen zu addieren.

7. Hilfsklassen: Zusätzlich gibt es einige Hilfsklassen, die als Server-Objekte entwickelt wurden. Es handelt sich dabei hauptsächlich um Adapter ([GOF01, Seite 171]), die Streams, die physikalisch gebunden sind, im Sinne von RMI verteilungsfähig zu machen. Sie haben in aller Regel die Methoden, die auch die Streams besitzen und eine Stream-Instanz als interne Referenz, auf die dann alle Anfragen einfach umgelenkt werden. Abbildung 4.7 auf der vorherigen Seite soll das verdeutlichen.

5. Client-Anwendungen

Um die entwickelte Middleware auf Stabilität und Praktikabilität testen zu können, wurden mehrere Clients entwickelt, die im folgenden ausführlich vorgestellt werden.

5.1. Administrationsclient

Die Verwaltung der Middleware geschieht mittels eines graphischen Clients, der auf Basis

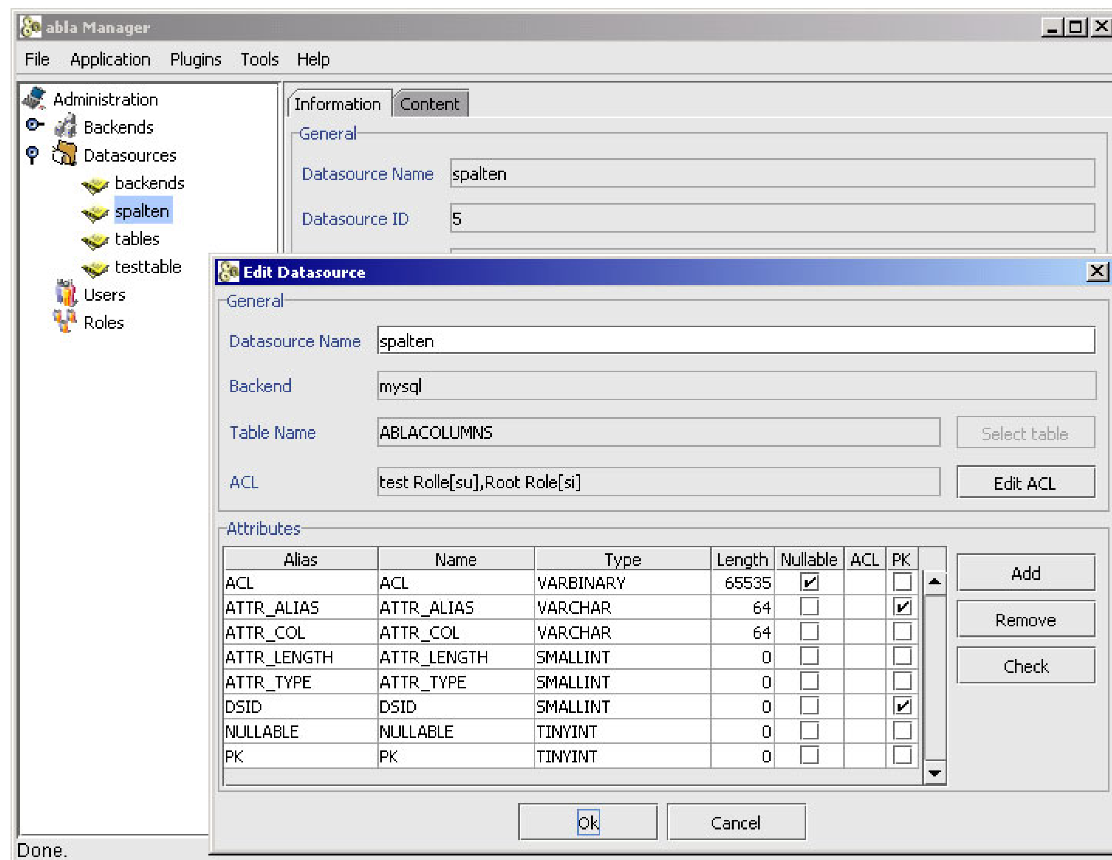

Abbildung 5.1.: Screenshot des Administrationsclients mit geöffnetem Dialog für eine Datasource mit den sichtbaren Attributen und den definierten ACL-Rechten

der Bibliothek Swing erstellt wurde. Der Client stellt dabei alle Funktionen bereit, die zur Einstellung und Konfiguration der Middleware benötigt werden. Dabei wurde darauf geachtet, daß der Administrationsclient keine speziellen Funktionen nutzt, sondern wie ein ganz normaler Client auf die Middleware zugreift. So ist auch ein nachträglicher Austausch gegen eine alternative Software möglich.

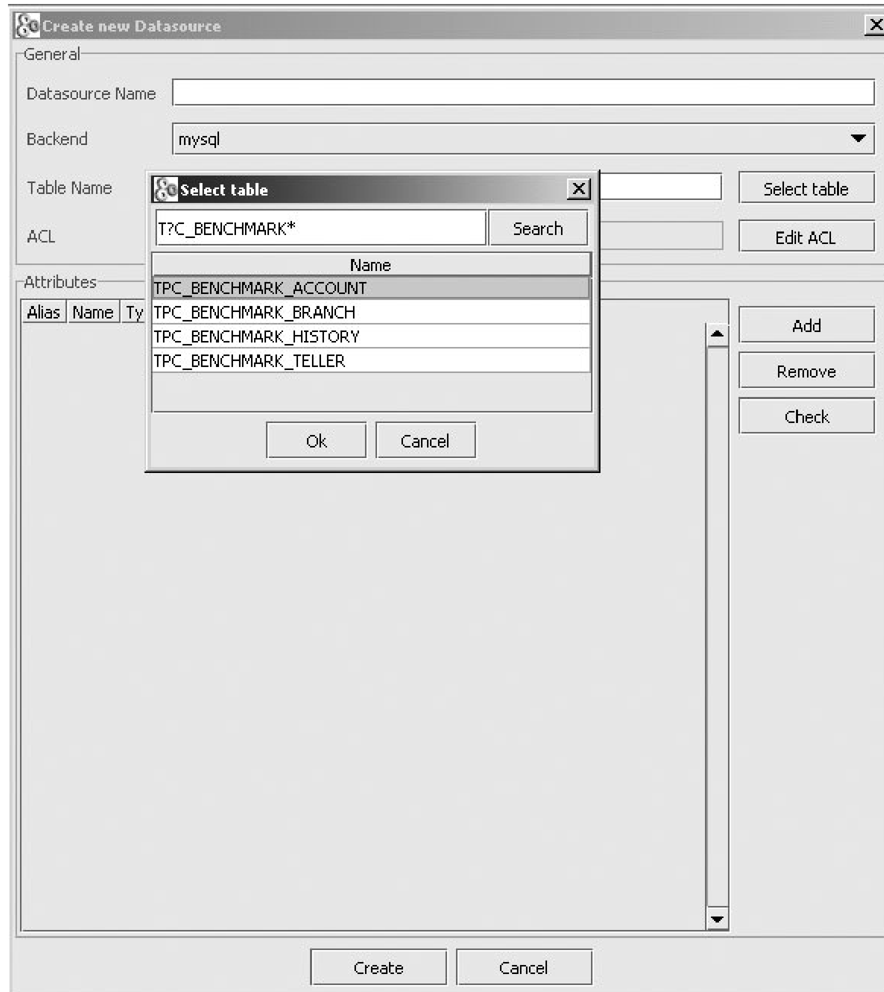

Abbildung 5.2.: Suchdialog für bereits in einem Backend existierende Datasources, dessen angezeigte Ergebnisse unter Verwendung der Wildcards gebildet wurden

Der Administrationsclient bietet die Verwaltung aller Rechtebits, der Rollen und deren Zuordnung zu den entsprechenden Benutzerkonten. Desweiteren können die Rollen in die AccessControlLists (ACL) der Datasources eingetragen werden, wodurch die Positiv- und die Negativrechte realisiert werden. Natürlich können auch Backends und Datasources über den Client in der Middleware bekannt gemacht werden, wobei die Eintragung von Datasources auch über eine Browsing-Funktion erfolgen kann. Dazu wählt

der Benutzer das Backend, in dem sich die einzutragende Datenquelle befindet, aus und kann dann in dem sich öffnenden Fenster die Liste der anzuzeigenden Datasources durch die Verwendung der üblichen Suchwildcards (* für $0-n$ viele beliebige Zeichen und ? für 1 beliebiges Zeichen) einschränken (ein Beispiel für eine solche Suche findet sich in Abbildung 5.2 auf der vorherigen Seite) und abschließend, durch die Auswahl einer Datenquelle, diese innerhalb der Middleware eintragen. Die Analyse der Struktur der Datenquelle geschieht in diesem Verlauf dann automatisch und muß gegebenenfalls nur noch durch eine Änderung der Aliasattributbezeichner ergänzt werden.

Neben diesen Basisfunktionen wurde der Administrationsclient um eine dynamische Plugin-Funktion ergänzt. Dies resultiert aus der genutzten Verteilungsplattform, durch die der Client beliebig viele Middleware-Server verwalten kann. In einigen Fällen kann es aber wünschenswert sein, nachträglich den Funktionsumfang des Administrationsclient zu erweitern. Um dann nicht jeden installierten Client mit dieser Funktion manuell ausstatten zu müssen, wurde eine Möglichkeit geschaffen, Plugins innerhalb der Middleware abzulegen, um sie dann dynamisch den sich anmeldenden Administrationsclients zur Verfügung zu stellen. Der unter Punkt 5.2 vorgestellte Backend-Tester ist als ein solches Plugin realisiert.

5.2. Backendtester

Während der Entwicklung der Treiber traten immer die identischen Probleme auf. Vor allem das Erstellen der Datentypmappings (siehe dazu die Diskussion unter 4.3 auf Seite 33) stellte einen aufwendigen Teil der Arbeit dar. Um anschließend die Treiber auch aus einer Praxissituation heraus testen zu können, wurde ein Testprogramm entwickelt, das ein Backend auf die Konformität bezüglich der Datentypspezifikation überprüft. Dazu werden die Grenzen der Datentypen in eine Tabelle geschrieben und anschließend wieder gelesen um sie zu validieren. Zum Testen des Datentyps BIT beispielsweise wird sowohl der Wert TRUE als auch der Wert FALSE in die Datenquelle geschrieben und anschließend wieder gelesen, wobei ein Test auf Korrektheit durchgeführt wird. Identisch wird mit den numerischen und den zeichenbasierten Datentypen verfahren. Zum Test der Lob-Fähigkeiten werden eine eingelesene Bild- und eine Textdatei in die Datasource geschrieben und dann beim Lesen byteweise mit den extern weiterhin vorliegenden Dateien verglichen.

Die zu diesem Zweck benötigten Datasources werden für jeden zu testenden Datentyp neu angelegt, um die bei manchen Datentypen benötigten wandelbaren Parameter einbinden zu können. Wenn ein Attribut beispielsweise als nicht mit Nullwerten füllbar gekennzeichnet wurde, muß beim Schreiben eines solchen Werts eine Exception geworfen werden.

5.3. Beispielhafter Kommandozeilenclient

Da das ganze Middlewaresystem in seiner konkreten Handhabung aus der obigen Beschreibung schwer zu verstehen ist, folgt hier ein kleines Client-Programm, dessen einzige

Funktion es ist, den Inhalt einer Datasource auszugeben. Dazu werden die wichtigsten Schritte, die bei der Kommunikation mit der Middleware notwendig sind, erklärt. Das Beispielprogramm ist allerdings in seinem Aufbau sehr reduziert, da auftretende Exceptions nicht abgefangen werden, und es vorausgesetzt wird, das als Parameter der Name der auszulesenden Datasource, der Loginname und das Passwort übergeben werden. Das Programm soll lediglich das grundsätzliche Verständnis für die Client-Entwicklung auf Basis der Middleware erleichtern.

Program 5.1: Test-Programm

```
1  import org.abla.server.datasource.Datasource;
2  import org.abla.server.datasource.DatasourceFactory;
3  import org.abla.server.usermgmt.AccessException;
4  import org.abla.server.usermgmt.UserContext;
5  import org.abla.server.usermgmt.UserFactory;
6  import org.abla.server.usermgmt.UserSystemException;
7  public class CmdTest
8  {
9      public static void main (String[] args)
10     {
11         DatasourceFactory dsfac = null;
12         UserFactory usfac = null;
13         UserContext uscon = null;
14         String loginName = args[1];
15         String password = args[2];
16         int rmi_port = 1099;
17         String host = "localhost";
18         String serverapp = "DatasourceFactory";
19         try
20         {
```

Zunächst wird die DatasourceFactory per Naming-Service lokalisiert:

```
21             String target = "//" + host + ":" + rmi_port + "/" + serverapp;
22             dsfac = (DatasourceFactory) java.rmi.Naming.lookup (target);
23         }
24         catch (Exception e)
25         {
26             System.out.println ("Could not connect to Server.\nServer not running or wrong IP−
                   Adress.\n" + e);
27         }
28         try
29         {
```

Nach der Lokalisierung der DatasourceFactory, wird die UserFactory, über die der Login erfolgt, als Objekt verfügbar gemacht.

```
30             usfac = dsfac.getUserFactory ();
31         }
32         catch (java.rmi.RemoteException err)
33         {
34         }
35         try
36         {
```

Nun der eigentlich Loginvorgang, bei dem man das UserContext-Objekt erhält.

```
37             uscon = usfac.login (loginName, password);
```

```
38              if (uscon == null)
39              {
40                  throw new AccessException ("Login failed.");
41              }
42          }
43          catch (Exception e)
44          {
45          }
46          try
47          {
```

Für den Zugriff auf die Datasource benötigt man natürlich noch selbige. Daher folgt hier der entsprechende Lookup.

```
48              Datasource ds = dsfac.getDatasource (uscon, args[0]);
```

Beispielhaft wird ein SELECT * auf der Datasource ausgeführt, wobei das * durch die leere Liste dargestellt wird.

```
49              java.util.Collection coll = ds.select (new java.util.LinkedList ());
50              int i = 0;
```

Per Iterator kann die Collection nun durchlaufen werden, um die Record-Objekte anzuzeigen.

```
51              for (java.util.Iterator it = coll.iterator (); it.hasNext ();)
52              {
53                  System.out.println ((++i) + "..." + it.next ());
54              }
55          }
56          catch (Exception e)
57          {
58              System.out.println ("err\n" + e);
59          }
60      }
61  }
```

6. Performance-Untersuchungen

6.1. Einleitung

Eine Einordnung der Middleware in vorhandene Lösungen erfordert die Durchführung eines Belastungs- und Funktionstests. Der Funktionstest erfolgt durch den Konfigurationsclient, mit dem alle Middleware-Funktionen verfügbar sind. Für einen Belastungstest muß ein entsprechender Benchmark gewählt werden. Im Bereich der Datenbanken hat sich dabei das Transaction Processing Performance Council, kurz TPC, hervorgetan. Diese Organisation besteht aus verschiedensten Firmen, die mit ihrem Engagement für vergleichbare Performance-Messungen sorgen wollen. Die TPC selbst beschreibt sich als "non-profit corporation founded to define transaction processing and database benchmarks and to disseminate objective, verifiable TPC performance data to the industry." [TPC01]. Zu diesem Zweck werden Benchmarks für verschiedene Anwendungsszenarien definiert, die dann von den jeweiligen Kandidaten implementiert werden. Die Durchführung eines solchen Benchmarks soll neben dem Vergleich der Performance der Middleware mit einer äquivalenten direkten Implementierung auch die Stabilität der entwickelten Lösung testen.

6.2. Auswahl des Benchmarks

Bei der Konzeption der Middleware lag das Hauptaugenmerk auf der Erstellung einer Software, die den Anschluß an möglichst viele Systeme ermöglicht. Durch diese Designentscheidung wurde der Funktionsumfang im Vergleich etwa zu einer Datenbank eingeschränkt. Es lassen sich Datensätze lesen, manipulieren und wieder schreiben. Auf die Möglichkeit, Datensätze miteinander zu verknüpfen, wie man es von Joins in Datenbanken kennt, wurde bewußt verzichtet, weil diese Funktion den Code erheblich aufgebläht, und vor allem bei der Entwicklung eines neuen Treibers einen erheblichen Mehraufwand nach sich gezogen hätte. Aus diesem Grund wurden Joins aus der Middleware herausgezogen und liegen als externe Funktionen vor. Damit ist es möglich, Joins durchzuführen, ohne die einfache Treiberstruktur aufgeben zu müssen. Zu den implementierten Joins gehören die Funktionen leftOuterJoin(), naturalJoin() und naturalJoinFirstOccurenceOnly(), denen jeweils zwei Collections mit Ergebnissen und ein Verknüpfungsoperator übergeben werden. Die entsprechend verknüpften Records werden dann als neue Collection zurückgeliefert.

Eine Software, die als Middleware nicht direkt eine Schnittstelle zu einem Benutzer darstellt, sondern als Grundlage für andere Programme dient, muß sinnvollerweise auf

den reinen Datendurchsatz getestet werden. Zusätzliche Funktionen, wie etwa die Aktualisierung einer Client-Anwendung, würde zur qualitativen Beurteilung der entwickelten Middleware nichts beitragen.

Ein Benchmark der diese Vorbedingungen erfüllt, ist der TPC-B Benchmark, der nur die atomaren Funktionen benötigt, die auch durch die Middleware bereitgestellt werden.

6.3. TPC-B Benchmark

Der TPC-B besteht aus einer Sequenz von SELECT, INSERT und UPDATE-Aufrufen, die auf einem genau definierten Datenbestand ausgeführt werden. Die zu diesem Zweck definierte Datenstruktur soll eine Bank darstellen. Dabei kann diese Bank beliebig viele Filialen, branches genannt, haben. Jede Filiale hat mehrere Kassierer, teller genannt. Zusätzlich besitzt die Bank viele Kunden, wobei jeder genau ein Konto (account) besitzt. Die Datenbasis repräsentiert die monetären Positionen aller Entitäten (branch, teller, account) und die Geschichte der durchgeführten Transaktionen. Diese Transaktion stellt alle Aktionen dar, die vollzogen werden, wenn ein Kunde auf sein Konto etwas einzahlt oder sich auszahlen lässt. Dabei wird eine solche Transaktion immer von einem Kassierer in einer Filiale durchgeführt. Diese TPC-B Transaktion wird beliebig oft ausgeführt, woraus sich aus der Verrechnung mit der abgelaufenen Zeit eine Kennzahl der Transaktionen pro Sekunde ergibt. Die Kennzahl bezieht sich aber nicht auf die atomaren Aktionen, sondern der Begriff der Transaktion bezieht sich auf eine komplette TPC-B Befehlssequenz, wie oben beschrieben. Die Standard-Transaktion hat nach der Spezifikation die folgende Form:

Program 6.1: TPC-B Transaktion

```
1  BEGIN TRANSACTION
2          Update Account where Account_ID = Aid:
3                  Read Account_Balance from Account
4                  Set Account_Balance = Account_Balance + Delta
5                  Write Account_Balance to Account
6          Write to History:
7                  Aid, Tid, Bid, Delta, Time_stamp
8          Update Teller where Teller_ID = Tid:
9                  Read Teller_Balance from Teller
10                 Set Teller_Balance = Teller_Balance + Delta
11                 Write Teller_Balance to Teller
12         Update Branch where Branch_ID = Bid:
13                 Read Branch_Balance from Branch
14                 Set Branch_Balance = Branch_Balance + Delta
15                 Write Branch_Balance to Branch
16 COMMIT TRANSACTION
17 Return Account_Balance to driver
```

Die Variablen Aid, Bid, Tid und Delta werden vor dem Beginn jeder Transaktion nach folgenden Regeln Regeln bestimmt:

1. Teller: es wird zufällig aus dem gesamten Spektrum der verfügbaren Kassierer eine ID ausgewählt, die dann die Tid (für Teller-ID) bildet.

2. Branch: da jeder Teller genau einer Branch zugeordnet ist, ergibt sich aus der Tid automatisch auch die Branch-ID (Bid).
3. Account: die Generierung der Account-ID geschieht mittels der folgenden Regeln
   - Ziehung einer Zufallszahl X aus dem Bereich $[0, 1]$
   - wenn X<0.85 oder nur eine Branch existiert, wird die Account-ID zufällig aus der Gesamtmenge aller Accounts der Branch gewählt
   - wenn X>=0.85 und mehrere Branches existieren, wird eine zufällige Account-ID aus der Menge der Accounts, die nicht innerhalb der vorher ermittelten Branch liegen, ausgewählt
4. Delta: der zu bewegende Betrag wird zufällig aus dem Bereich $[-999999, +999999]$ ausgewählt

Entsprechend diesen Vorgaben wurde eine Implementierung für die Middleware durchgeführt. Um die alternative Implementierung auf einer äquivalenten Stufe bereitzustellen, fiel die Wahl auf eine direkte Datenbankanbindung über die in 2.2.1 auf Seite 18 vorgestellte Schnittstelle JDBC unter der Programmiersprache Java. Die Sourcecodes der entscheidenden Methoden, die jeweils die TPC-B-Transaktion durchführen, ist im Anhang B auf Seite 71 beigelegt.

6.4. Durchführung

Als Testsysteme waren zwei Rechner zum Einsatz. Auf einem Rechner mit einem Prozessor vom Typ Celeron-450MHz und 392MB Arbeitsspeicher lief die Oracle-Datenbank, die die benötigten Tabellen bereitstellte. Die Middleware war auf einem PentiumIII-600MHz mit 192MB Arbeitsspeicher installiert. Es kam auf allen beteiligten Systemen Windows2000 als Betriebssystem zum Einsatz. Beide Rechner waren über ein 10Base2 Netzwerk verbunden. Als Java-Umgebung wurde eine Standard-Installation der Java2 Standard Edition Version 1.3 eingesetzt.

Die Messreihen wurden dreimal mit zurückgesetzter Datenbank durchgeführt und gingen dann als Mittelwert in die Auswertung ein, um eventuelle Fehler und Abweichungen auszugleichen.

6.5. Ergebnisse

Im folgenden finden sich die Ergebnisse der Benchmark-Läufe in graphischer Form. Eine Diskussion und Auswertung der Ergebnisse findet sich im Anschluss an die Performance-Grafiken auf Seite 63.

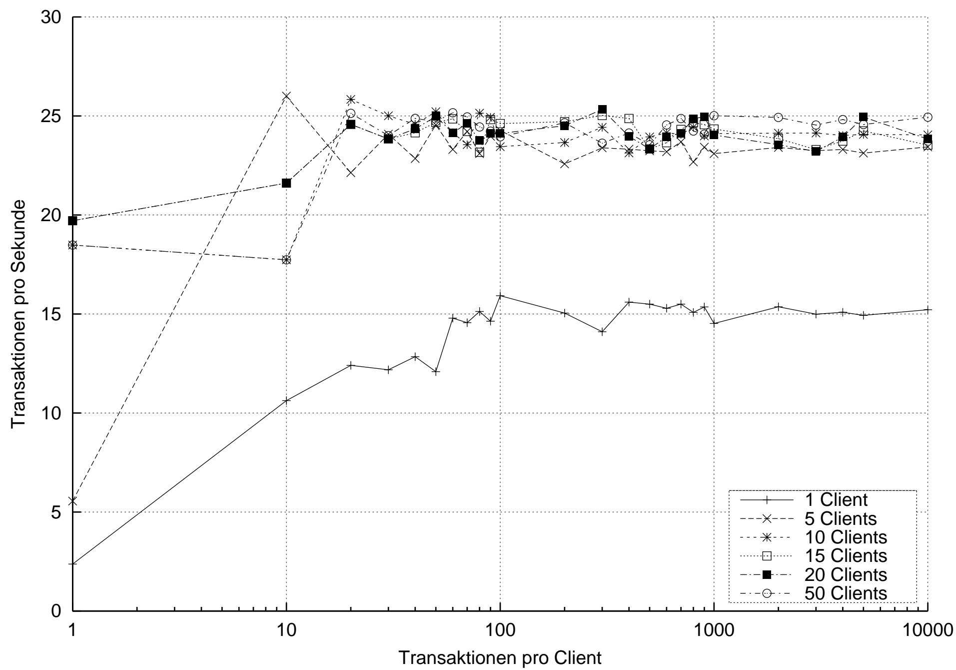

Abbildung 6.1.: Performance-Übersicht der Middleware mit der Darstellung aller Benchmarkläufe die über die Middleware durchgeführt wurden

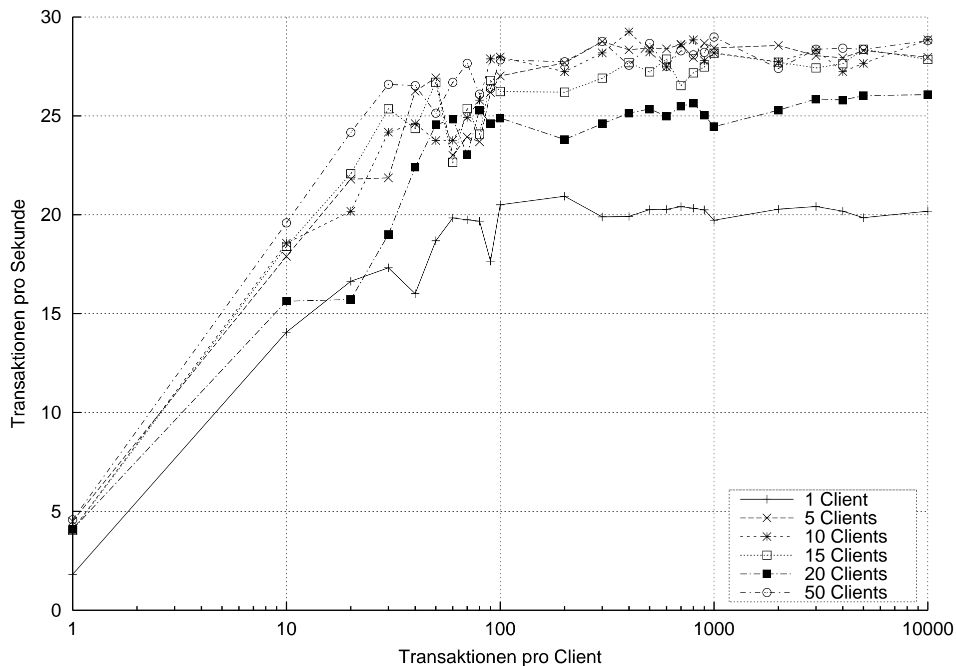

Abbildung 6.2.: Performance-Übersicht der direkten JDBC-Implementierung mit der Darstellung aller Benchmarkläufe

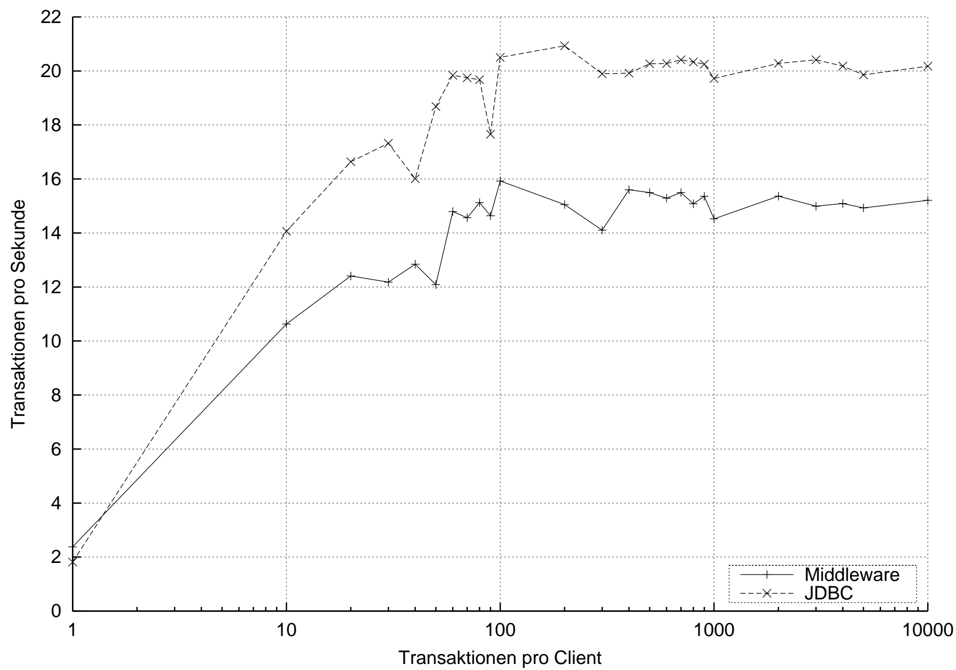

Abbildung 6.3.: Vergleich der Performance zwischen der Middleware und der direkten JDBC-Verbindung bei einem Client

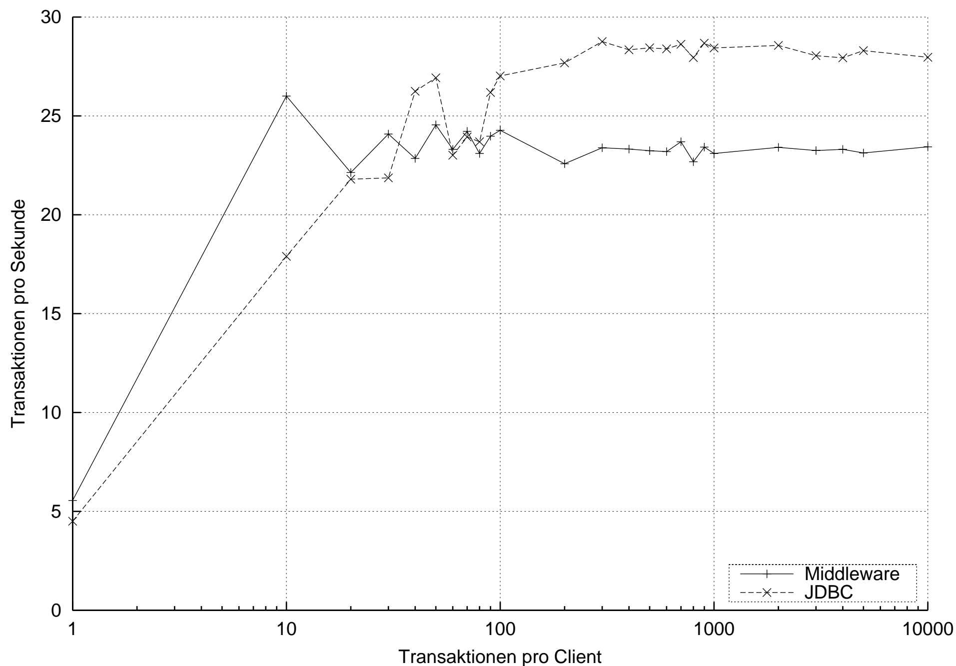

Abbildung 6.4.: Vergleich der Performance zwischen der Middleware und der direkten JDBC-Verbindung bei fünf parallelen Clients

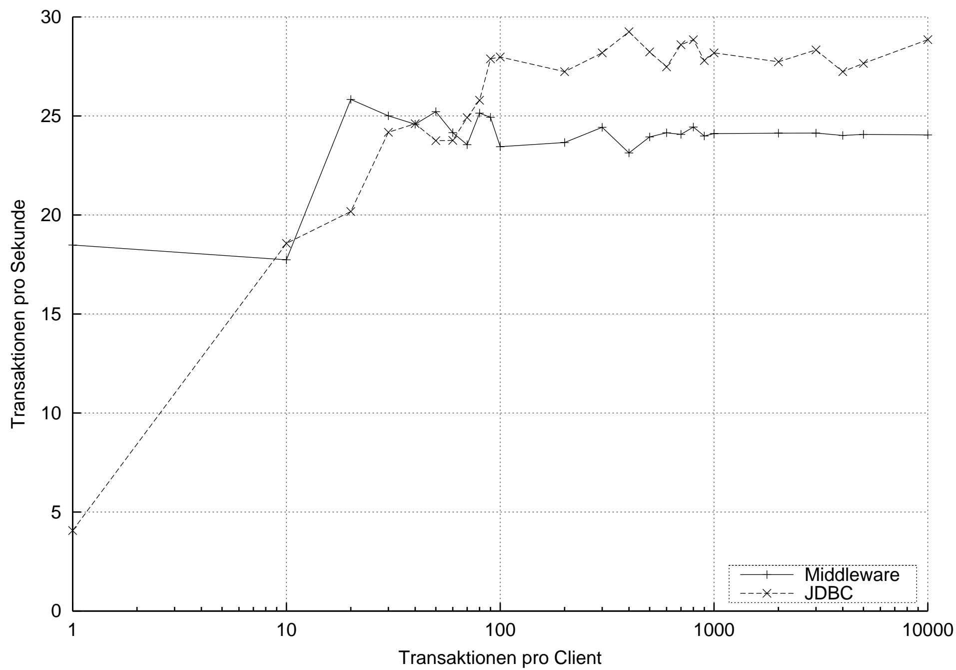

Abbildung 6.5.: Performance bei zehn parallelen Clients mit dem sichtbaren Einfluss des ConnectionPools der Middleware bei wenigen Transaktionen

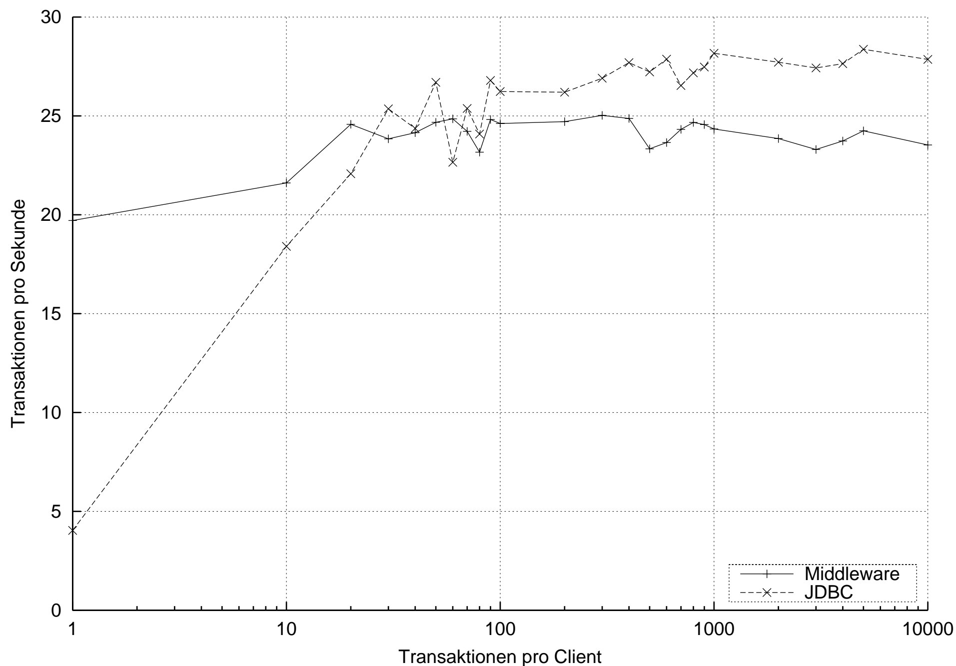

Abbildung 6.6.: Performance bei fünfzehn parallelen Clients mit dem erkennbaren Einfluss des ConnectionPools der Middleware bei wenigen Transaktionen

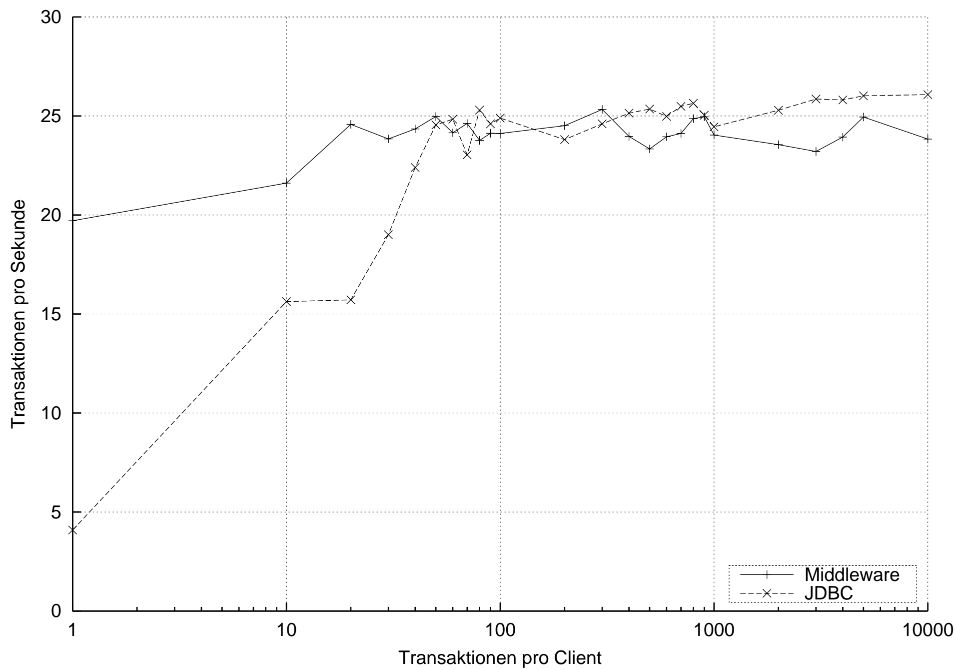

Abbildung 6.7.: Performance bei zwanzig parallelen Clients mit dem erkennbaren Einfluss des ConnectionPools der Middleware bei wenigen Transaktionen

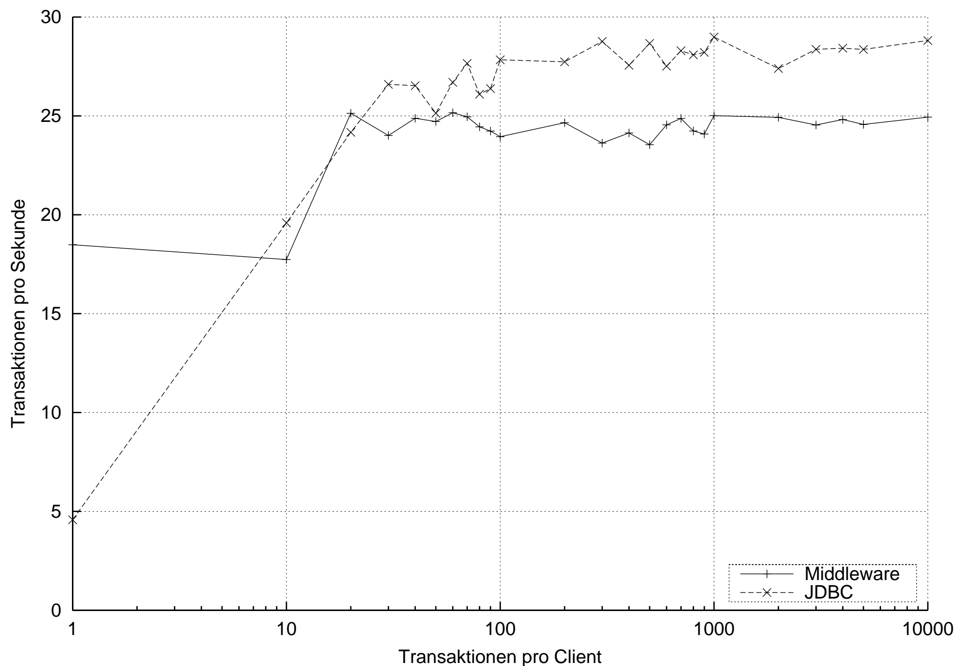

Abbildung 6.8.: Performance bei fünfzig parallelen Clients mit dem erkennbaren Einfluss des ConnectionPools der Middleware bei wenigen Transaktionen

6.6. Auswertung

Der Performance-Unterschied zwischen den beiden Test-Implementierungen lässt sich auf mehrere architektonische Unterschiede zurückführen. Der JDBCBenchmark allokiert für jeden Client eine eigene Datenbankverbindung, die dann für jede TPC-B-Transaktion genau ein Statement-Objekt erzeugt, über welches alle Aktionen ablaufen. Zusätzlich werden lediglich acht Strings gebildet, die die SQL-Kommandos enthalten, die in die Datenbank gesendet werden. Schließlich werden die Ergebnisse der Datenbankzugriffe als ResultSet-Objekte zurückgeliefert. Das ergibt pro TPC-B-Transaktion innerhalb eines Clients genau dreizehn Objekte, die erzeugt werden müssen. Die Middleware dagegen ist wesentlich komplexer. Jeder Zugriff auf die Middleware muß mittels der spezifizierten Objekte, etwa den Selection-Chains und den Record-Objekten, erfolgen. Desweiteren hält die Middleware intern einen ConnectionPool, der die Datenbankverbindungen verwaltet. Bei jeder Aktion innerhalb des Middleware-Benchmarks muß erst eine Verbindung aus dem ConnectionPool geholt werden und erst dann kann ein Statement-Objekt erzeugt werden, mit dem dann die singuläre Aktion durchgeführt werden kann. Insgesamt werden beim Benchmark in der Middleware-Implementierung 21 Objekte pro TPC-B-Transaktion erzeugt, wobei die Objekte, die dann innerhalb der Middleware infolge der aufgerufenen Methoden und in Abhängigkeit vom benutzten Treibers kreiert werden, nicht mitgezählt werden. Damit lassen sich die folgenden (groben) Abschätzungen errechnen:

$$\begin{aligned} \mathrm{JDBC-Objekte} &= \mathrm{nClients} * \mathrm{nTransaktionen} * 13 + 1 \\ \mathrm{Middleware-Objekte} &= \mathrm{nClients} * \mathrm{nTransaktionen} * 21 \end{aligned}$$

In einem praktischen Beispiel von 20 Clients mit je 1000 Transaktionen wird der Unterschied deutlich:

$$\begin{aligned} \mathrm{JDBC-Objekte} &= 20 * 1000 * 13 + 1 = 260001 \\ \mathrm{Middleware-Objekte} &= 20 * 1000 * 21 = 420000 \end{aligned}$$

Das zeigt, daß bereits auf der Implementierungsebene des Benchmarks die Middlewarelösung komplexere und damit zeitaufwendige Strukturen erfordert. Zusätzlich muß innerhalb der Middleware noch funktional die Anpassung der eingesendeten Strukturen an das verwandte Backend sowie das Ressourcenmanagement erfolgen. Diese Komplexitätsunterschiede erklären die gefundenen Geschwindigkeitsunterschiede.

Das bei niedrigen Transaktionszahlen auftretende Problem der geringeren Geschwindigkeit ist auf den Aufwand beim Start zurückzuführen. Jeder Client muß zunächst eine Datenbankverbindung bzw. die Verbindung zur Middleware aufbauen, und die entsprechenden Selektionsbedingungen erzeugen. Bei geringen Transaktionszahlen sind diese Aufrufe zu großen Teilen für die Gesamtlaufzeit des Benchmarks verantwortlich. Bei steigenden Client- oder Transaktionszahlen und damit steigender Gesamtdauer des Benchmarklaufs fallen diese Faktoren nicht mehr ins Gewicht. Wenn die absolute Laufzeit des Benchmarks soweit angestiegen ist, daß der mit dem Start verbundene Aufwand keine Rolle mehr spielt, laufen die Benchmarks unabhängig von den benutzten Nutzerzahlen beziehungsweise Transaktionen pro Benutzer mit vergleichbarer Geschwindigkeit.

Das bedeutet, sowohl die Parallelisierung als auch die Steigerung der Dauer haben wenig Einfluss auf den gemessenen Durchsatz.

Beim Betrachten der Ergebnisse fällt bei den Messungen mit hohen Client-Zahlen (siehe Grafik 6.5 auf Seite 61, 6.6 auf Seite 61, 6.7 auf Seite 62 und 6.8 auf Seite 62) ein Phänomen auf: Die Middleware-Implementierung startet bei wenigen Transaktionen pro Client mit wesentlich höheren Transaktionsraten als die JDBC-basierte Lösung. Die Ursache hierbei ist in den parallelen Datenbankverbindungen zu sehen, die vom JDBC-Client zunächst erzeugt werden müssen, während die Middleware einen bereits vorinitialisierten ConnectionPool bereitstellt. Da das Erzeugen einer Verbindung in die Datenbank wesentlich zeitaufwendiger ist als die übliche Objekterzeugung, fällt sie bei hohen Nutzerzahlen dementsprechend mehr ins Gewicht, als bei den Läufen mit wenigen Clients.

Ein weiteres interessantes Ergebnis des Benchmarks ist die obere Schranke, gegen die beide Varianten laufen. Die JDBC-basierte Implementierung überschreitet nicht die 30 Transaktionen pro Sekunde und die Middleware-Lösung nur selten die 25 Transaktionen pro Sekunde. Der hier auftretende limitierende Faktor ist eindeutig die Oracle-Datenbank, die den Rechner, auf dem die Datenbank installiert war, auslastete. Der Client-Rechner war nie am Limit, sondern verfügte stets über Reserven.

Eine noch ergänzend durchgeführte Test-Messung auf einem besser ausgestattetem Datenbanksystem mit Dual-PentiumII-800MHz unter Linux zeigte eine leicht verbesserte Skalierbarkeit der JDBC-Implementierung, die auf eine schlechtere Java-Implementierung unter Linux zurückzuführen ist, bei der zum einen die Erzeugung neuer Objekte mehr ins Gewicht fällt und zum anderen die Auflösung der Locks zwischen den Threads nicht so feinkörnig ist, wie beim vergleichbaren Java Development Kit für die Windows-Plattform.

6.7. Fazit

Die durchgeführten Messungen zeigen, daß die Middleware im Vergleich mit einer direkten Backendanbindung rund 20% an reiner Durchsatzleistung verliert. Dieser Verlust ist aber mit den innerhalb der Software vorgenommenen Abstraktionen und dem damit verbundenen Aufwand durchaus im erwarteten Rahmen. Die Software ist bei geringen Transaktionszahlen und vielen Clients sogar schneller als eine direkte Anbindung, da hier ein Client durch das automatische Ressourcenpooling profitieren kann.

Die Middleware empfiehlt sich also insbesondere für stark parallelisierte Anwendungen, die mit vielen Clients und wenigen Transaktionen arbeiten. Außerdem sind Anwendungen, die mit vielen Backendsystemen zurecht kommen müssen, ein empfehlenswertes Einsatzszenario.

7. Abschluss und Ausblick

7.1. Abschluss

Die in dieser Arbeit vorgelegten Konzepte haben gezeigt, daß es möglich ist, durch die Einführung neuer Abstraktionsebenen, differierende komplexe Systeme einheitlich anbindbar zu machen. Die dadurch verfügbaren Techniken lösen viele Probleme die beim alltäglichen Zugriff auf attributbasierte Datenspeicher auftreten, von Syntaxunterschieden bis zu komplett differierenden Anschlußvarianten. Die in der Arbeit vorgestellten Lösungen wurden funktional implementiert und demonstrieren damit die Korrektheit und Praxistauglichkeit der Verfahren. Durch die Nutzung der Middleware mittels der verschiedenen Client-Anwendungen, von der Verwaltungsanwendung bis zum Benchmark, wurde auch die Stabilität und Variabilität für verschiedene Einsatzzwecke bewiesen. Insbesondere die mehrtägig laufenden Benchmarks haben die Stabilität auch unter hoher Last eindrucksvoll gezeigt. Der Vergleich zur direkten Datenbankanbindung zeigte zwar die erwarteten Performanceeinbußen, die sich aber innerhalb eines akzeptablen Rahmens befanden. Insbesondere für hoch parallelisierte Anwendungen bietet sich die Middleware an, da die entsprechend benötigten Cachingroutinen transparent für den Client durch die Middleware bereitgestellt und verwaltet werden.

Die unter Punkt 1.2 auf Seite 14 spezifizierten Aufgabenstellungen werden nun auf ihre Erfüllung überprüft:

1. Es ist gelungen sowohl für den Client, als auch für die Treiber, klar definierte Schnittstellen zu entwerfen, die den Clients die benötigte Flexibilität bieten, aber die Komplexität der Treiber nicht unnötig steigern.

2. Die Middleware ist durch den Einsatz der Programmiersprache Java plattformunabhängig.

3. Das Lesen und Schreiben von Daten, sowie die Benutzung von Selection-Chains geschieht abstrahiert und unabhängig vom letztendlich verwandten Backend. Ebenso sind die Meta-Daten, die die Backend- und Datenquellenstrukturen beschreiben, durch abstrahierte Objekte verfügbar.

4. Der Zugriff auf eine Datenquelle geschieht auf eine einheitliche Art und Weise, völlig unabhängig vom Backend, in der die Datenquelle liegt, die angesprochen wird.

5. Die Treiberarchitektur wurde erfolgreich entsprechend der Anforderung implementiert, daß zur Laufzeit und ohne Funktionseinschränkungen weitere Systeme durch die Einfügung neuer Treiber im System bekannt gemacht werden können.

6. Bei der Verteilungsplattform fiel die Wahl auf die Remote Method Invocation (RMI), die später auch durch den Einsatz von RMI-IIOP Corba-kompatibel gemacht werden kann.

7. Die Treiberarchitektur wurde so gestaltet, daß beispielsweise auch Treiber für nur les- aber nicht schreibbare Datenquellen entwickelt werden können.

8. Wenn ein Backend Transaktionen unterstützt, können diese durch den Treiber bereitgestellt werden.

9. Das Backend-Singleton ermöglicht es einem Treiber, die für die eigene Funktion notwendigen Ressourcen durch den Einsatz eines Pools möglichst effizient zu verwalten.

10. Die rollenbasierte Benutzerverwaltung ist nicht nur für die innerhalb der Middleware benutzten Rechte vorbereitet, sondern bietet auch den Anwendungen, die auf die Middleware aufbauen, eine Erweiterung um zusätzlich benötigte Rechte.

11. Die Sperrung von Datensätzen wurde erfolgreich implementiert und steht in einer zweistufigen Variante zur Verfügung. So können Datensätze bis zu einem Neustart der Middleware oder auf Dauer, also auch über einen solchen Neustart hinaus, vor manipulierenden Zugriffen, die über die Middleware erfolgen, geschützt werden. Dieser Schutz ist dabei treiberunabhängig und bedeutet keinen Mehraufwand bei der Implementierung eines Treibers.

12. Durch den konsequenten Einsatz von Aliasnamen für alle Strukturen die innerhalb der Middleware bereitstehen, ist es möglich, eine Client-Anwendung ohne genaue Kenntnisse des Zielsystems zu entwickeln, weil die Anpassung erst bei der Installation zu erfolgen braucht.

13. Die Middleware wurde funktional erfolgreich mit einer komplexen Konfigurationsanwendung getestet, die die komplette Verwaltung der Middleware ermöglicht.

14. Die Stabilität wurde erfolgreich bei den Benchmarkläufen gezeigt, die zum Teil mehrere Tage durchliefen, ohne daß die Middleware einen Fehler produzierte.

Es ist gelungen alle Aufgabenstellungen und Ziele erfolgreich zu lösen und zu implementieren. Dabei mussten keine Einschränkungen vorgenommen werden, sondern die nun vorliegende Software ist funktional vollständig und praxistauglich.

Bei der Arbeit wurden einige Aspekte nicht implementiert, weil sie keine neuen Erkenntnisse gebracht hätten, wie die Einbindung von dokumentenbasierten Systemen, oder zu zeitaufwendig, wie ein Treiber für XML-Dateien, gewesen wären. Trotzdem werden nachfolgend Lösungskonzepte auch für diese Probleme vorgestellt.

7.1.1. Lösunganstz für dokumentbasierte Systeme

Die Beschränkung dieser Arbeit auf attributbasierte System geschah nicht aus einer

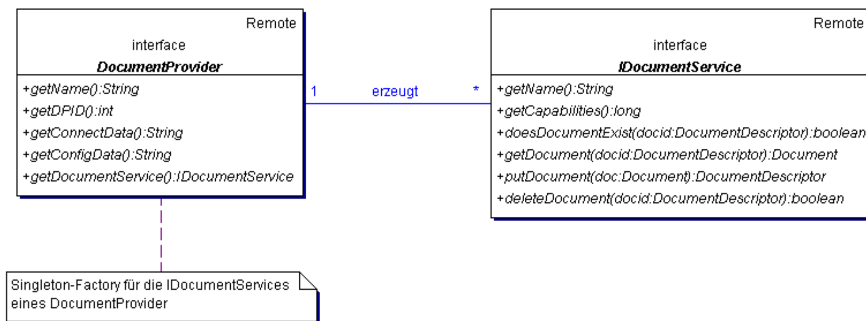

Abbildung 7.1.: Vorschlag für den Aufbau von dokumentenbasierten Treibern

Überlegung der Nichtübertragbarkeit der Lösungen auf dokumentenbasierten Systeme heraus, sondern lediglich zur Eingrenzung der erheblichen Funktionsmenge. Die Konzepte lassen sich auch problemlos auf dokumentenbasierte Systeme erweitern. Das unter 4.3.1 auf Seite 35 vorgestellte Treiberkonzept läßt sich fast komplett übernehmen. Das Backend-Interface würde als abgeändertes DocumentProvider-Interface wieder als Singleton zur Verwaltung der parallelen IDocumentServices dienen, die dann jeweils den Zugriff auf ein konkretes Dokument-Backend ermöglichen. Als neue Abstraktionsebene sind die Dokumentattribute einzuführen, mit Hilfe derer ein abgespeichertes Dokument wiedergefunden werden kann. Diese Abstraktion würde mittels eines DocumentDescriptor-Objekts realisiert, dass vom DocumentService mit den benötigten Daten zum Wiederfinden des Dokumentes gefüllt wird. Den konkreten Zugriffsmethoden getDocument() und deleteDocument() würde dann dieser DocumentDescriptor übergeben. Das eigentliche Dokument muß auch abstrahiert werden, da, wie schon in 3.3.7 auf Seite 29 gezeigt wurde, große Dokumente am besten mittels Streams übertragen werden. Diese Lese-/Schreibstreams würden dann innerhalb des Document-Objekts gekapselt und der Client-Anwendung zur Verfügung gestellt werden. Die letzte noch im IDocumentService-Interface (siehe Abbildung 7.1) genannte Methode getCapabilities() wird erneut benötigt, um den Umfang der Implementierungen verschiedener Treiber zur Laufzeit überprüfen zu können. Beispielsweise könnte ein Treiber der auf ReadOnly-Medien arbeitet, die Methode deleteDocument() nicht implementieren und würde dies dann über getCapabilities() bekanntgeben.

7.1.2. Lösungsansatz für XML-Formate

Ebenso interessant wäre die Implementierung eines Treibers für das Format XML, der dann auf entsprechenden Dateien arbeiten könnte. Auf diese Implementierung wurde in dieser Arbeit aufgrund der erkennbaren Komplexität verzichtet. XML ist eine Textauszeichnungssprache für Dokumente, die strukturierte Informationen enthalten. Die

XML-Spezifikation definiert einen einheitlichen Weg, um weitere Auszeichnungselemente zu Dokumenten hinzuzufügen.

Ein potentieller Lösungsansatz würde auf lokalen XML-Dateien arbeiten, die als Datasources ins System eingebunden werden. Das Backend-Singleton würde eine konfigurierbare Anzahl von JDOM XML-Bäumen lokal verwalten, die dann bei einer Anfrage an eine Datasource-Instanz weitergegeben werden, wo sie dann entsprechend bearbeitet werden. JDOM ermöglicht die einfache Darstellung eines XML-Dokuments durch Java-Strukturen. Dabei bietet JDOM alle Funktionen, um ein Dokument auf simple und effiziente Weise zu lesen, zu manipulieren und zu schreiben. Um auch Suchanfragen realisieren zu können, könnten die verwendeten Selection-Chains in eine XPath-Struktur umgewandelt werden, um die Sucheinschränkungen in optimierter Form auch direkt innerhalb den XML-Strukturen durchführen zu können. Die Sprache XPath dient zur Adressierung von Teilen eines XML-Dokuments. Um das parallele Verarbeiten von XML-Dateien zu ermöglichen, könnte ein abgewandelter ConnectionPool die JDOM-Bäume halten und diese bei INSERT und UPDATE-Funktionsaufrufen für einen weiteren Zugriff sperren. So wären Leseoperationen weiterhin effizient und die Transaktionssicherheit bei manipulierenden Operationen wäre trotzdem gewährleistet.

7.2. Ausblick

Dem praktischen Einsatz der Middleware in professionellen Produkten steht nichts im Weg, da die Strukturen mit Blick auf eine klare Definition und verständliche Kapselung entwickelt wurden. Natürlich ist die Middleware nicht für alle Anwendungsfälle geeignet, da beispielsweise Software, die komplexe Anfragen durchführen muß oder bei der es auf einen möglichst hohen Datendurchsatz ankommt, mit einer direkten Verbindung in eine Datenbank, mit Hilfe derer auch alle Funktionen genutzt werden können, besser gedient ist. Ein sinnvolles Anwendungsszenario ist die Verknüpfung von Daten aus mehreren Datenquellen. Ein Unternehmen etwa, welches in einem CRM-System die Kundendaten führt und in einer externen Datenbank die dazugehörigen Informationen für die Produkte, die die Kunden erworben haben, könnte mit Hilfe der Middleware solche, bisher disjunkten, Informationsblöcke in einer Informationsstruktur zusammenführen.

Die Middleware bietet weiterhin viele Vorteile bei der Entwicklung von großen Softwareprojekten, die auf umfangreiche, bereits existierende Datenbankbestände zugreifen und dabei viele verschiedene Zielplattformen unterstützen müssen, weil die Bandbreite der zu unterstützen Systeme nicht zur Entwicklungszeit festgelegt werden kann. Insbesondere aufgrund der in die Middleware ausgelagerten backendspezifischen Komplexität bringt die Softwareentwicklung auf Basis der Middleware einen enormen Zeitvorteil, was unter dem zunehmenden Wunsch, die "Time-To-Market", also die Dauer von der Planung bis zur Produktreife, zu minimieren einen entscheidenden Vorteil bedeuten kann.

Ein weiterer Vorteil ist die Variabilität hinsichtlich des Formats der Client-Anwendung, da durch die Nutzung der Verteilungsplattform RMI beliebig viele Anwendungen auf eine entsprechend konfigurierte Middleware-Instanz zugreifen können. Das entwickelte Benutzersystem ermöglicht es desweiteren, komplexe Benutzerstrukturen für jede Client-

Software zu realisieren, da es sich um ein dynamisch erweiterbares System handelt. So ist es beispielsweise möglich, mit Hilfe der Middleware ein Benutzersystem bei einem Kunden zu definieren, das Rechte, die während der Entwicklung noch gar nicht bekannt waren, dynamisch zuweist und verwaltet.

Letztendlich sind mit der entwickelten Software viele komplexe Anwendungsszenarien denkbar, deren Realisierung auch Aufgabe kommender Diplomarbeiten sein kann.

A. Datentypen

Die in der Middleware eingeführte Datentyp-Abstraktion macht eine Spezifikation der von Clients verwendbaren Datentypen notwendig. Die nachfolgende Liste zeigt die Datentypen, die verfügbar sind, mit ihren Minimal- und Maximalwerten.

Bezeichner	Minimum	Maximum
Bit	false	true
Tinyint	-127	128
Smallint	-32768	32767
Integer	-2147483648	2147483647
Bigint	0x8000000000000000L	0x7fffffffffffffffL
Real	1.40129846432481707e-45f	3.40282346638528860e+38f
Float, Double	4.94065645841246544e-324	1.79769313486231570e+308
Numeric, Decimal		
Char	1 Zeichen	255 Zeichen
Varchar	1 Zeichen	4000 Zeichen
Longvarchar		2 Gigabyte
Date		
Time		
Timestamp		
Binary	1 Byte	255 Bytes
Varbinary	1 Byte	255 Bytes
Longvarbinary		2 Gigabyte
Java Object		2 Gigabyte
Blob		2 Gigabyte
Clob		2 Gigabyte
Sequence Int	0	2147483647

Tabelle A.1.: Datentypspezifikation

B. TPC-B Benchmark

Der für die Performanceuntersuchungen notwendige Benchmark wurde in zwei verschiedenen Varianten implementiert. Zum einen in einer Lösung, die auf der Middleware basiert und eine Variante, die direkt auf die Datenbank zugreift. Aufgrund der Unterschiede der Implementierungen sind beide Versionen mit der entscheidenden Methode, die die TPC-B Transaktion durchführt, angefügt.

B.1. JDBC-Benchmark

Program B.1: JDBC-Implementierung

```
1  public class JDBCBench
2  {
3
4            [...]
5
6          int doOne (int bid, int tid , int aid , int delta)
7          {
8              try
9              {
10                 Statement Stmt = con.createStatement ();
11                 String Query = "SELECT Abalance FROM accounts WHERE Aid = " + aid;
12                 ResultSet RS = Stmt.executeQuery (Query);
13                 Stmt.clearWarnings ();
14                 int oldAccountBalance = 0;
15                 if (RS.next ())
16                 {
17                     oldAccountBalance = RS.getInt (1);
18                 }
19                 RS.close();
20                 Query = "UPDATE accounts SET Abalance = " + oldAccountBalance + " + "
                        + delta + " ";
21                 Query += "WHERE Aid = " + aid;
22                 Stmt.executeUpdate (Query);
23                 Stmt.clearWarnings ();
24                 Query = "SELECT Abalance FROM accounts WHERE Aid = " + aid;
25                 RS = Stmt.executeQuery (Query);
26                 Stmt.clearWarnings ();
27                 int aBalance = 0;
28                 while (RS.next ())
29                 {
30                     aBalance = RS.getInt (1);
31                 }
32                 RS.close();
33                 Query = "INSERT INTO history(Tid, Bid, Aid, delta) ";
```

```
34                      Query += "VALUES (";
35                      Query += tid + ",";
36                      Query += bid + ",";
37                      Query += aid + ",";
38                      Query += delta + ")";
39                      Stmt.executeUpdate (Query);
40                      Stmt.clearWarnings ();
41                      Query = "SELECT Tbalance FROM tellers WHERE Tid = " + tid;
42                      RS = Stmt.executeQuery (Query);
43                      Stmt.clearWarnings ();
44                      int oldTBalance = 0;
45                      if (RS.next ())
46                      {
47                          oldTBalance = RS.getInt (1);
48                      }
49                      RS.close();
50                      Query = "UPDATE tellers SET Tbalance = " + oldTBalance + " + " + delta
                            + " ";
51                      Query += "WHERE Tid = " + tid;
52                      Stmt.executeUpdate (Query);
53                      Stmt.clearWarnings ();
54                      Query = "SELECT Bbalance FROM branches WHERE Bid = " + bid;
55                      RS = Stmt.executeQuery (Query);
56                      Stmt.clearWarnings ();
57                      int oldBBalance = 0;
58                      if (RS.next ())
59                      {
60                          oldBBalance = RS.getInt (1);
61                      }
62                      RS.close();
63                      Query = "UPDATE branches SET Bbalance = " + oldBBalance + " + " + delta
                            + " ";
64                      Query += "WHERE Bid = " + bid;
65                      Stmt.executeUpdate (Query);
66                      Stmt.clearWarnings ();
67                      Stmt.close();
68                      return aBalance;
69                  }
70                  catch (SQLException E)
71                  {
72                      incrementFailedTransactionCount ();
73                  }
74                  return 0;
75              }
76          }
77  }
```

B.2. Middleware-Benchmark

Program B.2: Middleware-Implementierung

```
1   package org.abla.client.cmd;
2
3   public class TPCBenchmark
4   {
5
```

```
6         [...]
7 
8       int doOne (int bid, int tid , int aid , int delta)
9       {
10          try
11          {
12              Collection what = new LinkedList ();
13              what.add(TPCBenchmark.abalance);
14              Selection sel = new Selection (TPCBenchmark.aid, new Integer (aid));
15              Collection colwhere = DatasourceUtils.createCollection(sel);
16              Collection olddata = account.select(what, colwhere);
17              rec.put(abalance, new Integer (((Record)olddata.iterator().next()).getInt(
                    TPCBenchmark.abalance)+delta));
18              account.update(rec, colwhere);
19              rec.clear();
20              int newabalance = 0;
21              olddata = account.select(what, colwhere);
22              for (Iterator it = olddata.iterator (); it.hasNext ();)
23              {
24                  Record record = (Record) it.next ();
25                  newabalance = record.getInt(TPCBenchmark.abalance);
26              }
27              rec.put(TPCBenchmark.aid,new Integer (aid));
28              rec.put(TPCBenchmark.bid,new Integer (bid));
29              rec.put(TPCBenchmark.tid,new Integer (tid));
30              rec.put(TPCBenchmark.delta,new Integer (delta));
31              rec.put(TPCBenchmark.time,new Date());
32              history.insert(rec);
33              rec.clear();
34              sel = new Selection (TPCBenchmark.tid, new Integer (tid));
35              what.clear();
36              what.add(TPCBenchmark.tbalance);
37              colwhere = DatasourceUtils.createCollection(sel);
38              olddata = teller.select(what, colwhere);
39              rec.put(tbalance, new Integer (((Record)olddata.iterator().next()).getInt(
                    TPCBenchmark.tbalance)+delta));
40              teller.update (rec, colwhere);
41              rec.clear();
42              sel = new Selection (TPCBenchmark.bid, new Integer (bid));
43              what.clear();
44              what.add(TPCBenchmark.bbalance);
45              colwhere = DatasourceUtils.createCollection(sel);
46              olddata = branch.select(what, colwhere);
47              rec.put(bbalance, new Integer (((Record)olddata.iterator().next()).getInt(
                    TPCBenchmark.bbalance)+delta));
48              branch.update (rec, colwhere);
49              return newabalance;
50          }
51          catch (Exception e)
52          {
53              incrementFailedTransactionCount ();
54          }
55          return 0;
56      }
57    }
58  }
```

C. Glossar

ACL: Access Control List sind Listen, die der Verwaltung von Rechten dienen

Adapter: Muster, um die Schnittstelle einer Klasse an eine von einem Client erwartete Schnittstelle anzupassen

Aliasing: Ersetzung der Aliasbezeichner eines Attributs durch den physikalischen Bezeichner

API: Application Programming Interface, dokumentierte Schnittstelle, um aus einem Programm heraus auf ein anderes Programm oder eine Programmbibliothek zugreifen zu können

Backend: in dieser Arbeit unter zwei Aspekten verwendet; als Backend allgemein für ein System, das strukturierte Daten bereitstellt, und als Backend als Teil eines Middleware-Treibers

Checkouts: Sperre von Datensätzen innehalb der Middleware zur Laufzeit und über einen Neustart hinaus

Corba: Common Object Request Broker Architecture ist ein Hardware- und Softwareunabhängiger Middleware-Standard zur Kommunikation von Objekten in einem Netzwerk

CRM: Customer Relationship Management, Software zur Adressierung von Kunden mit den richtigen Produkten und Diensten, durch die Zusammenführung aller dementsprechend wichtigen Informationen

Datasource: in dieser Arbeit unter zwei Aspekten verwendet; als Datasource allgemein für eine Datenquelle, aus der sich strukturierte Daten auslesen lassen, und als Datasource als Teil eines Middleware-Treibers

DBMS: Datenbankmanagementsystem, das als standardisierte Software die Definition, Verwaltung, Verarbeitung und Auswertung der in den Datenbanken enthaltenen Daten sicherstellt

EJB: Enterprise JavaBeans, Komponenten, die in einem standardisierten Container als Server-Anwendung laufen, und Geschäftslogik implementieren; der Container stellt Transaktionen, Ressourcen-Management, Security etc. bereit

ERP: Integriertes System zur effizienten Geschäftsplanung unter Einbindung möglichst vieler Geschäftsinformationen und -abläufe

IDL: Interface Definition Language, spezifiziert programmiersprachenunabhängig eine Objektschnittstelle

IIOP: Internet Inter-ORB Protocol, Transport-Protokoll, das Teil CORBAs für verteilte Systeme ist

ISO: International Standards Organization, nicht-staatliche Organisation (gegründet 1947) mit Mitgliedern aus über 140 Ländern, die weltweit die Standardisierung koordinieren soll

J2SE: Java2 Standard Edition

JDBC: die Java Database Connectivity ist eine einheitliche Schnittstelle, um in einem Java-Programm auf Datenbanken zugreifen zu können, Architektur siehe Abbildung 2.1 auf Seite 17

JDOM: ermöglicht einfache Darstellung eines XML-Dokuments durch Java-Strukturen und durch die Bereitstellung von Funktionen, um ein Dokument auf simple und effiziente Weise zu lesen, zu manipulieren oder zu schreiben

LOB: Large Object, Begriff aus dem Datenbankumfeld, der für einen binären (BLOB) oder zeichenbasierten (CLOB) Datentyp steht und zur Speicherung großer Inhalte gedacht ist

Locks: Sperre von Datensätzen innerhalb der Middleware zur Laufzeit

Middleware: Softwarekomponente für den Datenaustausch zwischen getrennten Softwaremoduln

ODBC: Open Database Connectivity, durch Microsoft eingeführte Schnittstelle für den Datenbankzugriff

OMG: Object Management Group, gegründet 1989, versucht Industriestandards im Bereich der verteilten Systeme und der Softwareentwicklung zu setzen

ORB: Object Request Broker, Basiskomponente für die Kommunikation in verteilten Anwendungen; dient dazu, Client/Server-Beziehungen zwischen Objekten aufzubauen

Proxy: Muster, um den Zugriff auf ein Objekt mit Hilfe eines vorgelagerten Stellvertreterobjekts zu kontrollieren

RMI: Remote Method Invocation ist eine Möglichkeit für Software-Entwickler, bei der Nutzung der Programmiersprache Java, objektorientierte Systeme zu entwerfen, bei denen Objekte auf unterschiedlichen Rechnern in einem Netzwerk kommunizieren

Rolle: mit einem Namen versehene Sammlung von Rechten, die dann den Mitgliedern der Rolle zugeteilt werden

Singleton: Muster, das sicherstellt, daß von einer Klasse nur eine Instanz existiert

SQL: Structured Query Language ist eine strukturierte und durch die ISO standardisierte Sprache, mit deren Hilfe Inhalte in Datenbanken abgefragt und manipuliert werden können

Swing: Bibliothek für die Programmiersprache Java, mit derer Hilfe betriebssystemunabhängige graphische Programmoberflächen erstellt werden können

TCO: Total Cost of Ownership steht für die Gesamtkosten, die ein Rechnersystem verursacht, neben den reinen Anschaffungskosten etwa auch die Schulungskosten, die Wartung und der Support uvm.

Time-To-Customer: Dauer, um ein Produkt soweit zu entwickeln, um es bei einem Kunden einsetzen zu können

Time-To-Market: Dauer, um ein Produkt soweit zu entwickeln, um es auf dem Markt verkaufen zu können

TPC: Transaction Processing Council ist eine Vereinigung von Firmen zur Schaffung von Benchmarks, die Produkte innerhalb gewisser Szenarien vergleichbar machen sollen

Unaliasing: Ersetzung der physikalischen Bezeichner eines Attributs durch den Aliasnamen

WORM: Write Once Read Many, häufig bei optischen Medien, die nur einmal beschrieben werden können und hauptsächlich zur Datensicherung eingesetzt werden, verwandter Begriff

XML: Textauszeichnungssprache für Dokumente, die strukturierte Informationen enthalten, wobei die XML-Spezifikation einen einheitlichen Weg, um weitere Auszeichnungselemente zu Dokumenten hinzuzufügen, spezifiziert

XPath: dient zur Adressierung von Teilen eines XML-Dokuments

Literaturverzeichnis

[BL01] Bloch, J. (2001). *Effective Java: Programming Language Guide*. Addison Wesley Longman, Inc

[DECZ01] Deitsch, A. & Czarnecki, D. (2001). *Java Internationalization*. Sebastopol: O'Reilly & Associates, Inc

[GOF01] Gamma, E. & Helm, R. & Johnson, R. & Vlissides, J. (2001). *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software* (5. Auflage). Bonn: Addison Wesley

[GE99] Geary, D. (1999). *Graphic Java 2: Mastering the JFC* (3. Auflage). Upper Saddle River, NJ: Prentice Hall PTR

[GUVE00] Gumbel, M. (2000). *Java Standard Libraries.* Bonn: Addison Wesley

[LEA00] Lea, D. (2000). *Concurrent Programming in Java - Second Edition: Design Principles and Patterns* (2. Auflage). Addison Wesley Longman, Inc

[LA00] McLaughlin, B. (2000). *Java and XML*. Sebastopol: O'Reilly & Associates, Inc

[OE01] Oestereich, B. (2001). *Die UML-Kurzreferenz für die Praxis*. München: Oldenbourg Wissenschaftsverlag

[RA99] Prof. Dr. Rahm, E. (1998/99). *Datenbanksysteme I - Vorlesungsscript zum Wintersemester 1998/99*. Leipzig: Institut für Informatik

[RO99] Roman, E. (1999). *Mastering Enterprise JavaBeans and the Java 2 Platform, Enterprise Edition*. New York: John Wiley & Sons, Inc

[SCH01] Schmidt, M. & Demmig, T. (2001). *SQL Ge-Packt*. Bonn: mitp

[TPC01] Transaction Processing Performance Council (2001). *About the TPC* (WWW). Available: WWW: http://www.tpc.org/information/about/abouttpc.asp

[VE99] Venners, B (1999). *Inside the Java 2 Virtual Machine* (2. Auflage). Blacklick, OH: McGraw-Hill

[VO99] Vossen, G. (1999). *Datenbankmodelle, Datenbanksprachen und Datenbankmanagement-Systeme* (3. Auflage). München: Oldenbourg Wissenschaftsverlag

[WHFI99] White, S. & Fisher, M. & Cattell, R. & Hamilton, G. & Hapner, M. (1999). *JDBC API Tutorial and Reference, Second Edition* (3. Auflage). Addison Wesley Longman, Inc

Erklärung

Ich versichere, daß ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Leipzig, der 19. November 2001