Anbindung einer relationalen Datenbank an JAVA

Helge Janicke, Niels-Peter de Witt, Karsten Wolke

2. Mai 2001

Inhaltsverzeichnis

1 Vorwort

Im Rahmen der Veranstaltungen Software-Technik und Datenbanken wurde uns die Aufgabe gestellt, Möglichkeiten zur Anbindung von relationalen Datenbanken an Java-Applikationen zu erarbeiten. Der Schwerpunkt der Aufgabe liegt jedoch in der Weitervermittlung des von uns angeeigneten Wissens an die anderen Projektgruppen, welche die Konzepte und entstandene Klassen für ihre Programme nutzen sollen. Um eine möglichst vollständige Einführung in das Thema zu geben, haben wir uns entschlossen die wichtigsten Punkte in einer zweistündigen Veranstaltung vorzutragen. Natürlich kann in dieser Zeit nicht das komplette Wissen, das zur Realisierung einer Datenbankanbindung gehört, vermittelt werden, so daß zusätzlich noch Schulungen und eine Online-Dokumentation erstellt werden.
Der hier vorliegende Artikel gibt die Inhalte wieder, die während des Vortrags behandelt wurden. Wir hoffen, daß es eine Hilfe darstellt, sich zügig in das Thema einzuarbeiten.
Viel Spaß beim Lesen

Helge, Niels und Karsten

Online Dokumentation finden Sie unter www.furchur.de
Zu erreichen sind wir bei Fragen und Anregungen unter der Addresse:
DatabaseSupport.web.de

2 Allgemeines zum Thema

Der folgenden Teil soll als Überblick dienen. Es werden die verwendeten Begriffe erläutert und eine Übersicht über die Entstehung und Geschichte der einzelnen Konzepte der Datenbanken und deren Anbindungsmöglichkeiten gegeben.

2.1 ODBC

ODBC ist die Abkürzung für **O**pen **D**ata**b**ase **C**onnectivity. Microsoft erkannte bereits Anfang der 90er Jahre, daß zwar viele Programmierer Windows als Plattform akzeptieren, dennoch aber ihre Daten lieber auf einem Großrechner (UNIX oder WINDOWS) in einer bestimmten Datenbank halten. Damit auch Endanwenderwerkzeuge, in denen der Datenbankzugriff eine wichtige Rolle spielt, wie z.B das Office Paket, nicht für jeden Datenbanktyp eigene Befehle anbieten müssen, hat Microsoft, in Zusammenarbeit mit anderen Datenbankherstellern, Anfang der 90er Jahre ODBC ins Leben gerufen. ODBC basiert auf einem „Baukastenprinzip“, bei dem vereinfacht formuliert der untere Teil des Baukastens „weiß“, wie eine bestimmte Datenbank angesprochen wird. Der obere Teil des Baukastens dagegen weiß nichts über die Datenbank, sondern ist lediglich in der Lage, ein Kommando (über eine Mittelschicht) an den unteren Teil zur Bearbeitung weiterzureichen. Programmierer sprechen über die ODBC-API-Funktionen nur den oberen Teil an und sind in der Lage, verschiedene Datenbanken ansprechen zu können, ohne Änderungen am Programm vornehmen zu müssen. Voraussetzung ist, daß für die Datenbank ein ODBC-Treiber existiert. ODBC ist in erster Linie für relationale Datenbanken konzipiert und an SQL als Abfragesprache gebunden. OLE DB (OLE steht für Object Linking and Embedding (*Objekte verknüpfen und einbetten*) und DB für Database) besitzt als „Nachfolger“von ODBC diese Einschränkung nicht.

Interessant ist das Prinzip, wie mit ODBC eine Datenbank ausgewählt wird. Anstelle eines konkreten Datenbanknamens mit genauer Pfadangabe, wird bei Zugriff lediglich ein sogenannter Data Source Name (DSN) übergeben. Was muß man sich unter einem DSN vorstellen? Ein DSN ist nichts anderes als eine Zusammenfassung aller Angaben, die für den Zugriff auf eine Datenbank benötigt werden. Dazu gehören der Name oder Verzeichnispfad der Datenbank, sowie das für den Zugriff benötigte Kennwort. Diese Angaben werden unter einem Namen zusammengefaßt - dem DSN. Ein DSN wird über den ODBC-Manager (in der Windows-Systemsteuerung) angelegt. Dieser speichert die angegebenen Informationen in der Registry (früher unter Windows 3.1 noch in der ODBC.INI-Datei). Ein DSN muß mindestens den Namen der Datenbank enthalten.

2.2 JDBC

JDBC ist die Abkürzung für Java Database Connectivity und bezeichnet einen Satz von Klassen und Methoden, um von Java aus relationale Datenbanksysteme zu nutzen. Das JDBC Projekt wurde 1996 gestartet und die Spezifikation im Juni 1996 festgelegt. Die Klassen sind ab dem JDK 1.1 im Core-Paket integriert. Mit der JDBC-API und den JDBC-Treibern wird eine wirksame Abstraktion von Datenbanken erreicht, so daß durch eine einheitliche Programmierschnittstelle die Funktionen unterschiedlicher Datenbanken genutzt werden können.

Das Erlernen verschiedenener Zugriffsmethoden für unterschiedliche Datenbanken entfällt. Wie eine spezielle Datenbank wirklich aussieht, wird durch die Abstraktion verheimlicht. Jede Datenbank besitzt ihr eigenes Protokoll (eventuell auch Netzwerkprotokoll), allerdings ist deren Implementierung nur dem Datenbanktreiber bekannt. Das Modell von JDBC setzt auf dem X/OPEN SQL-Call-Level-Interface (CLI) auf und bietet somit die gleiche Schnittstelle wie ODBC. Dem Programmierer stellt JDBC Funktionen zur Verfügung, um Verbindungen zu Datenbanken aufzubauen, Datensätze zu lesen oder neue Datensätze zu erstellen. Zusätzlich können Tabellen aktualisiert und Prozeduren serverseitig ausgeführt werden. Im folgenden werden die für einen Zugriff auf eine relationale Datenbank mit JDBC nötigen Schritte genannt:

- Installieren der JDBC-Datenbank-Treiber.
- Eine Verbindung zur Datenbank über den entsprechenden JDBC-Treiber für das verwendete DBMS aufbauen.
- Eine SQL-Anweisung erzeugen.
- Ausführen der SQL-Anweisung.
- Das Ergebnis der Anweisung holen.
- Schließen der Datenbankverbindung.

Von einem Java-Programm aus kann mit dem jetzigen Modell (JDBC) auf relationale Datenbanken zugegriffen werden. Für die stark im Kommenden objektorientierten Datenbanken existiert noch kein fertiges Konzept. Dies liegt unter anderem daran, daß zur Zeit die meisten Datenbanken noch relational orientiert sind und Standards fehlen. Mit JDBC-2 Treibern ist jedoch ein Übergang zum SQL3-Standard geschaffen, in dem objektorientierte Konzepte eine größere Rolle spielen.

2.3 Die Architektur von JDBC

JDBC ist keine eigene Datenbank, sondern eine Schnittstelle zwischen einer SQL-Datenbank und der Applikation, die sie benutzen will. Bezüglich der Architektur der zugehörigen Verbindungs-, Anweisungs- und Ergebnisklassen unterscheidet man vier Typen von JDBC-Treibern:

2.3.1 Treibertypen

- Steht bereits ein ODBC-Treiber zur Verfügung, so kann er, mit Hilfe der im Lieferumfang enthaltenen JDBC-ODBC-Bridge, in Java-Programmen verwendet werden. Diese Konstruktion bezeichnet man als **Typ-1-Treiber**. Mit seiner Hilfe können alle Datenquellen, für die ein ODBC-Treiber existiert, in Java-Programmen genutzt werden. Der JDBC-Treiber ist damit nichts anderes, als eine Fassade, die eine dahinter liegende ODBC-Datenquelle verwendet.
- Zu vielen Datenbanken gibt es neben ODBC-Treibern auch spezielle Treiber des jeweiligen Datenbankherstellers. Setzt ein JDBC-Treiber auf einem solchen proprietären Treiber auf, bezeichnet man ihn als **Typ-2-Treiber**.

- Wenn ein JDBC-Treiber komplett in Java geschrieben und auf dem Client keine spezielle Installation erforderlich ist, der Treiber zur Kommunikation mit einer Datenbank aber auf eine funktionierende Middleware (Database Access Server) angewiesen ist, über den die echten Treiber der Datenbank angesprochen werden, handelt es sich um einen **Typ-3-Treiber**.
- Falls ein JDBC-Treiber komplett in Java geschrieben ist und die JDBC-Calls direkt in das erforderliche Protokoll der jeweiligen Datenbank umsetzt, handelt es sich um einen **Typ-4-Treiber**.

Während die Typ-1- und Typ-2-Treiber lokal installierte und konfigurierte Software erfordern (die jeweiligen ODBC- bzw. herstellerspezifischen Treiber), ist dies bei Typ-3- und Typ-4-Treibern normalerweise nicht der Fall. Hier können die zur Anbindung an die Datenbank erforderlichen Klassendateien zusammen mit der Applikation oder dem Applet aus dem Netz geladen und ggfs. automatisch aktualisiert werden. Nach der Veröffentlichung von JDBC gab es zunächst gar keine Typ-3- oder Typ-4-Treiber. Mittlerweile haben sich aber alle namhaften Datenbankhersteller zu Java bekannt und stellen auch Typ-3- oder Typ-4-Treiber zur Verfügung. Daneben gibt es eine ganze Reihe von Fremdherstellern, die JDBC-Treiber zur Anbindung bekannter Datenbanksysteme zur Verfügung stellen.

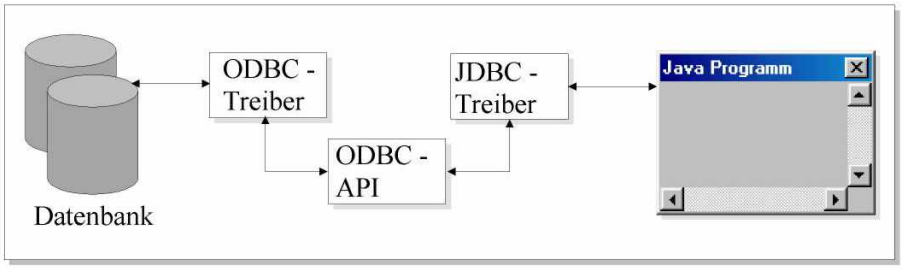
SUN stellt eine Suchmaschine bereit, auf der Sie wahrscheinlich auch einen Treiber für die von Ihnen verwendete Datenbank finden.
Die Adresse ist: http://industry.java.sun.com/products/jdbc/drivers

2.3.2 Standards

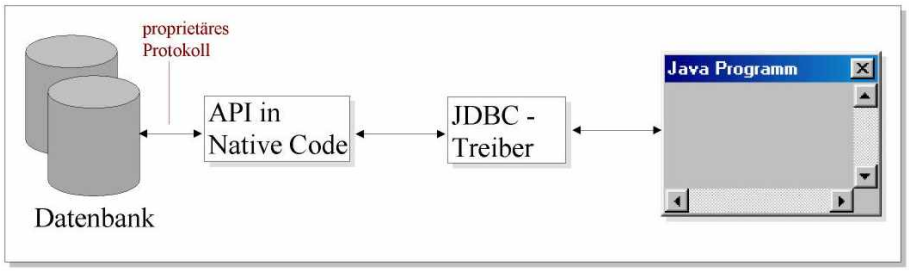
JDBC übernimmt die Aufgabe eines „Transportprotokolls“zwischen Datenbank und Anwendung und definiert damit zunächst noch nicht, welche SQL-Kommandos übertragen werden dürfen und welche nicht. Tatsächlich verwendet heute jede relationale Datenbank ihren eigenen SQL-Dialekt. Eine Portierung auf eine andere Datenbank ist meist aufwendig. Um einen minimalen Anspruch an Standardisierung zu gewährleisten, fordert SUN von den JDBC-Treiberherstellern, mindestens den SQL-2 Entry-Level-Standard von 1992 zu erfüllen. Mit Hilfe einer von SUN erhältlichen Testsuite können die Hersteller ihre JDBC-Treiber auf Konformität testen. Da praktisch alle großen Datenbanken in ihrer Funktionalität weit über besagten Standard hinausgehen, ist bei Verwendung solcher Features möglicherweise mit erheblichem Portierungsaufwand zu rechnen.

2.4 SQL

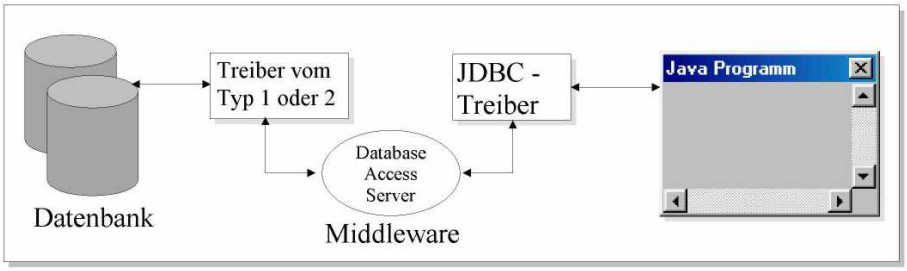
SQL (Structured Query Language - strukturierte Abfragesprache) ist eine relationale Datenbanksprache. Die Sprache umfaßt unter anderem Anweisungen, um Daten eingeben, ändern, löschen und schützen zu können. Zu SQL als Datenbanksprache sind einige Anmerkungen zu machen. Da SQL eine relationale Datenbanksprache ist, wird sie unter die nicht-prozedurorientierten Datenbanksprachen eingeordnet. Das heißt der Benutzer muß „nur“angeben, mit welchen Daten er arbeiten möchte, nicht, wie der Zugriff auf die Daten im Einzelnen erfolgen soll. Bei Sprachen wie COBOL, PASCAL, BASIC oder FORTRAN handelt es sich um typische prozedurorientierte Sprachen.



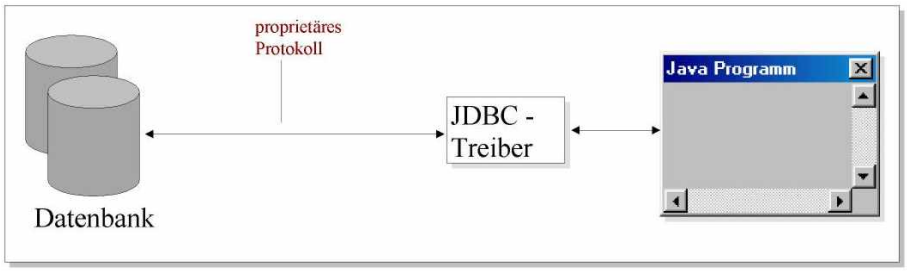
Der JDBC Treiber vom Typ 1 definiert die JDBC - Bridge.



Der JDBC-Treiber vom Typ 2 verwendet nativen Code.



Der JDB-Treiber vom Typ 3 sind rein in Java implementiert.



Der JDBC-Treiber vom Typ 4 ist plattformunabhängig.

Abbildung 1: Darstellung der verschiedenen Treibertypen.

SQL kann auf zwei Weisen verwendet werden. Zum einen interpretierend: Eine SQL-Anweisung wird auf dem Terminal oder Mikrocomputer eingegeben und direkt weiterverarbeitet. Das Ergebnis erscheint direkt auf dem Bildschirm; zum anderen als sogenannte embedded SQL. Hierbei sind die SQL-Anweisungen in ein Programm eingebettet, daß in einer anderen Sprache geschrieben ist. Die Ergebnisse der SQL-Anweisungen sind in diesem Fall nicht direkt für den Benutzer sichtbar, sondern werden durch das Programm, das sie „umhüllt", bearbeitet. Zahlreiche Hersteller von Datenbank-Management-Systemen haben SQL bereits als Datenbanksprache implementiert. SQL ist nicht der Name für ein bestimmtes Produkt eines bestimmten Herstellers.

2.4.1 Die Enstehungsgeschichte von SQL

Die Geschichte von SQL steht in engem Zusammenhang mit der Entwicklungsgeschichte eines IBM-Projekts namens System R. Im Verlauf dieses Projekts wurde ein experimentelles relationales Datenbank-Management-System entwickelt, daß den gleichen Namen trug wie das Projekt selbst: System R. Dieses System wurde im IBM-Forschungslaboratorium in San Jose, Kalifornien konstruiert. Das Projekt sollte beweisen, daß die benutzerfreundlichen Vorteile des relationalen Modells in ein System integriert werden konnten, welches allen Anforderungen eines modernen Datenbank-Management-Systems entsprach. Als Datenbanksprache für System R wurde eine Sprache mit dem Namen Sequel (**S**tructured **E**nglish **Que**ry **L**anguage) ausgewählt. Die erste Veröffentlichung über diese Sprache wurde von R.F. Boyce und D.D. Chamberlin geschrieben. Der Name wurde später aufgrund von Warenzeichengesetzen in SQL geändert (Der Name SQL wird jedoch häufig noch „*siekwel*"ausgesprochen.) Das Projekt System R wurde in drei Phasen ausgeführt. In der ersten Phase, **Phase Null** (von 1974 bis 1975), wurde SQL nur teilweise integriert. Das *join* (um Daten aus verschiedenen Tabellen miteinander zu kombinieren) wurde noch nicht integriert und das System lediglich als Einzelplatzversion konzipiert. Diese Phase diente lediglich der Überprüfung, wie weit ein solches System überhaupt integrierbar ist. Die Phase Null wurde erfolgreich abgeschlossen. Der für Phase Null geschriebene Programmcode wurde 1976 mit Beginn der **Phase Eins** beiseite gelegt und ein völlig neuer, das gesamte System umfassender Code entwickelt. Dies bedeutet unter anderem, daß die Mehrplatzfähigkeit und das *join* aufgenommen wurden. Die Entwicklung von Phase Eins fand in den Jahren 1976 und 1977 statt. In **Phase Zwei** wurde System R ausgewertet, indem es von IBM auf verschiedenen unternehmenseigenen Anlagen und bei einer Reihe von wichtigen Kunden installiert wurde. Die Auswertungsphase fand in den Jahren 1978 und 1979 statt. Im Jahre 1979 wurde das Projekt System R abgeschlossen.

Auf der Grundlage der in diesen Jahren gewonnenen Kenntnisse und der Technologie, die in den drei Phasen entwickelt wurde, konnte SQL/DS konstruiert werden. SQL/DS war das erste relationale Datenbank-Management--System von IBM, das auf den Markt kam. 1981 folgte die SQL/DS-Version für das Steuersystem DOS/VSE und 1983 die VM/CMS-Version. Im gleichen Jahr wurde auch DB2 angekündigt. Bei DB2 handelt, es sich um das System für die Steuersysteme MVS/370 und MVS/XA. Zur Entwicklung von System R hat IBM zahlreiche Veröffentlichungen herausgegeben, und zwar zu einer Zeit, in der auf Konferenzen und in Seminaren relationale Datenbank-Management-Systeme ausgiebig diskutiert wurden. So war es nicht weiter verwunderlich, daß auch an-

dere Hersteller damit begannen relationale Systeme aufzubauen. Etliche dieser Hersteller, zum Beispiel ORACLE, implementierten SQL als Datenbanksprache. In den letzten Jahren sind zahlreiche SQL Implementierungen hinzugekommen. Auch bereits vorhandene Datenbank-Management-Systeme sind später um SQL erweitert worden.

2.4.2 Spezifikationen

Eng verbunden mit der Sprache SQL ist das **A**merican **N**ational **S**tandards **I**nstitute (ANSI), das mehrere international anerkannte Standards zu SQL erarbeitete und auch heute noch erarbeitet (ANSI 1986, 1989 und 1992). Dieser internationale Standard definiert die Datenstrukturen und grundlegenden Operationen auf SQL-Daten. Er beschreibt die funktionellen Fähigkeiten für das Erzeugen, den Zugriff, die Verwaltung, die Steuerung und den Schutz von SQL-Daten. Im Laufe der Zeit hat sich die Sprache SQL von einer reinen Abfragesprache zu einer Sprache entwickelt, mit der sich auch Datenbanken erstellen und die Sicherheit des Datenbanksystems verwalten lassen.

2.4.3 SQL2 oder SQL-92

Der Standard SQL2 oder SQL-92 wurde im Jahre 1992 verabschiedet. SQL-92 definiert drei mögliche Ebenen des Sprachumfangs:

Eingangsstufe (Entry Level) Enthält die gesamte Funktionalität von SQL1 einschließlich der referentiellen Integrität und der Unterstützung für Modulsprache und eingebettete SQL-Schnittstellen für sieben verschiedene Programmiersprachen.

Zwischenstufe (Intermediate Level) Zusätzliche Funktionalität für Schemaänderungen, dynamisches SQL, Isolationsebenen für die Transaktionsverarbeitung, kaskadierte referentielle Löschaktionen, Zeilen- und Tabellenausdrücke, Union Joins, Schnittmengen von Tabellen (Intersection), Differenzoperationen, Domänen, CASE-Ausdrücke, Typumwandlungen, mehrere Zeichensätze, Datums-/Zeit-Typen und Zeichenstrings variabler Länge.

Stufe des vollständigen Standards (Full Level) Diese Stufe schließt aufgeschobene Bedingungsprüfung, Namensbedingungen, selbstreferenzierende Aktualisierungen und Löschung, kaskadierte referentielle Aktualisierungsaktionen, Unterabfragen in Testklauseln, bildlauffähige Cursor, Zeichenübersetzungen, Bitstring-Datentypen, temporäre Tabellen und einfache Annahmen (Assertions) ein.

2.4.4 SQL3

Diese Spezifikation ist gerade in Arbeit und soll folgende Merkmale aufweisen:

- Unterstützung aktiver »Regeln«, sogenannte Trigger
- Unterstützung abstrakter Datentypen
- Unterstützung von mehreren Nullzuständen

- Rekursive UNION-Operation für Abfrageausdrücke
- Unterstützung für Aufzählungs- und boolsche Datentypen
- Unterstützung für SENSITIVE Cursor (d.h. Cursor, die Änderungen an den zugrundeliegenden Daten widerspiegeln)

2.4.5 Grundzüge von SQL

Die Anweisungen der Sprache SQL lassen sich den allgemeinen Kategorien Datendefinitionssprache (DDL), Datenmanipulationssprache (DML) und Datensteuerungssprache (DCL) zuordnen.

Datendefinitionssprache
Die Datendefinitionssprache kommt vornehmlich für administrative Zwecke zum Einsatz. Typische Vertreter der DDL sind die SET-Anweisungen zum Einstellen von Optionen und die `CREATE`-Anweisungen zum Erstellen von Datenbankobjekten.

Datenmanipulationssprache
Die Datenmanipulationssprache (DML - Data Manipulation Language) ist eine Teilmenge der Datenbanksprache, mit der sich Daten abrufen und bearbeiten lassen. Mit den Anweisungen der Datenmanipulationssprache werden Daten in Objekten ausgewählt, aktualisiert und gelöscht oder in die Objekte eingefügt. Dabei handelt es sich um die Objekte, die mit Hilfe der DDL definiert wurden. Beispiele für die DML sind `SELECT`, `INSERT`, `UPDATE`.

Datensteuerungssprache
Mit den Anweisungen der Datensteuerungssprache (DCL - Data Control Language) steuert man die Berechtigungen für Datenbankobjekte. Typische Anweisungen sind `GRANT` und `REVOKE`.

2.5 Allgemeines über Datenbanken

2.5.1 Von Dateien zu Datenbanken

Schreibt man ein Programm, dann sind die im Programm verwendeten und erzeugten Daten nur solange vorhanden, wie das Programm läuft. Mit dem Ende des Programms gehen die Daten „verloren“, sofern man vorher nicht besondere Vorkehrungen getroffen hat. Diese Vorkehrungen bestehen darin, daß man vor Programmende die Daten, die man bei einem späteren Programmlauf wieder benötigt, in Dateien speichert. Deren Inhalt ist dauerhaft (persistent), d.h. er steht solange zur Verfügung, bis er explizit gelöscht oder überschrieben wird. Dateien befinden sich auf externen Speichermedien. Die Inhalte von Dateien können nicht direkt von Programmen bearbeitet werden. Daher müssen die Inhalte von Programmvariablen in bzw. von Dateien explizit kopiert werden. Charakteristisch für diese Art der persistenten Datenhaltung ist außerdem, daß der Aufbau der abgespeicherten Daten und die damit verbundene Semantik im entsprechenden Programm definiert sind. Die Unternehmenspraxis führte zu weitergehenden Anforderungen:

- Es zeigte sich, daß verschiedene Programme oft die gleichen Daten benötigen. Daher ist es nicht sinnvoll, daß jedes Programm gleiche Daten für sich

verwaltet. Dies führt zu Doppelerfassungen von Daten und redundanter Speicherung. Nach längerer Zeit erhält man inkonsistente Datenbestände. Ziel ist es daher, die Daten eines Unternehmens oder eines größeren Anwendungsbereichs integriert zu verwalten.

- Die integrierte Verwaltung macht es sinnvoll, daß nicht jedes Programm für sich die Datenelemente und Strukturen festlegt, sondern daß die Datenbeschreibungen von den Programmen unabhängig gespeichert werden.

2.5.2 Datenbanksysteme

Ein **D**aten**b**anks**y**stem (DBS) sorgt für die:

- dauerhafte (persistente),
- zuverlässige und
- unabhängige Verwaltung sowie die
- komfortable,
- flexible und
- geschützte Verwendung
- großer,
- integrierter und
- mehrfachbenutzbarer Datenbanken

Ein Datenbanksystem besteht aus einer oder mehreren **D**aten**b**anken (DB), einem **D**ata **D**ictionary (DD) und einem **D**aten**b**ank-**M**anagement-**S**ystem (DBMS)

$$\text{DBS} = \text{DBMS} + \text{DD} + \text{nDB} \ (\text{n} \geq 1)$$

Eine Datenbank enthält die Gesamtheit der Daten eines Anwendungsbereichs. Im Data Dictionary wird das Datenbankschema gespeichert, das den Aufbau der Daten der Datenbanken beschreibt. Das Datenbank-Management-System verwaltet und kontrolliert zentral unter Berücksichtigung des Datenbankschemas im DD, die in der Datenbank oder den Datenbanken abgelegten Datenbestände.

Zuverlässige Verwaltung bedeutet, daß das Datenbank-Management-System über Mechanismen verfügt, um die Konsistenz, die Integrität und die Unversehrtheit der Daten (kein Verlust und keine Verfälschung aufgrund technischer Fehler) sicherzustellen. Im Fehlerfall muß ein Wiederanlauf (recovery) des Datenbanksystems möglich sein.

Unabhängige Verwaltung bedeutet, daß die Programme, die ein Datenbanksystem benutzen, und das Datenbanksystem selbst weitgehend unabhängig voneinander sind. Strukturelle und andere interne Änderungen, die für den jeweils anderen nicht von Bedeutung sind, wirken sich nicht aus (Datenunabhängigkeit). Das bedeutet, daß die Daten in den Datenbanksystemen einheitlich beschrieben werden können, unabhängig von den jeweiligen Anwendungsprogrammen. Die Beschreibung erfolgt im Datenbankschema.

Komfortable Verwendung bedeutet, daß über eine höhere, abstrakte Schnittstelle mit der Datenbank kommuniziert wird. Man muß sich nicht um die Details der Speicherung einzelner Datenelemente kümmern.

Flexible Verwendung bedeutet, daß man mit Hilfe spezieller Anfragesprachen ad hoc, d.h. ohne eigentliche Programmierung, auf die Daten zugreifen kann.

Geschützte Verwendung bedeutet, daß Daten vor unberechtigtem Zugriff geschützt werden können (Datenschutz).

Große Datenbank bedeutet, daß sie nicht vollständig im Arbeitsspeicher gehalten werden kann.

Integrierte Datenbank bedeutet, daß alle Daten redundanzarm gespeichert werden, selbst wenn sie von verschiedenen Anwendungen stammen bzw. für verschiedene Anwendungen verwendet werden. Das hat zur Folge, daß nicht jedes Anwendungsprogramm alle Daten benötigt, sondern nur bestimmte Ausschnitte. Es muß daher möglich sein, Sichten (**views**) auf Teile der Datenbank zu definieren, die für ein Anwendungsprogramm oder einen Benutzer relevant sind. In der Regel liegt daher eine zweistufige Arbeitsweise mit Elementen einer Datenbank vor: Im ersten Schritt werden die benötigten Daten ausgewählt (Aufgabe des DBMS) und im zweiten Schritt erfolgt die eigentliche Bearbeitung (Aufgabe des Anwendungsprogramms). Ein DBMS benötigt daher leistungsfähige Auswahlmechanismen (in Form von Anfragesprachen), standardisierte Suchverfahren und effiziente Speicherungsstrategien für große Datenmengen.

Mehrfachbenutzbare Datenbank bedeutet, daß die Daten „gemeinsam“von mehreren Programmen, u. U. sogar gleichzeitig, verwendet werden können. Der parallele Zugriff mehrerer Programme oder Benutzer auf denselben Datenbestand muß daher koordiniert werden.

Damit ein Datenbanksystem die beschriebenen Leistungen erbringen kann und damit die Anwendungsprogramme nicht allein für die Semantik der Daten zuständig sind, müssen die Inhalte einer Datenbank durch Angaben über ihre Bedeutung beschrieben werden. Im sogenannten Datenmodell wird festgelegt,

- durch welche Eigenschaften Datenelemente charakterisiert werden können,
- wie die Struktur der Datenelemente aussehen kann,
- welche Konsistenzbedingungen einzuhalten sind und
- welche Operationen zum Speichern, Auffinden, Ändern und Löschen von Datenelementen erlaubt sind.

Syntax und Semantik eines Datenmodells werden durch eine Definitionssprache (DL) und eine Manipulationssprache (ML) festgelegt. Das (Datenbank-) Schema beschreibt eine konkrete Datenbank, d.h. das Datenmodell wird auf einen speziellen Einsatzfall angewandt. Das heute in der Praxis am meisten verwendete Datenmodell für Datenbanksysteme ist das relationale. Das hierarchische Datenmodell und das Netzwerkmodell sind heute technisch überholt und werden daher nicht weiter betrachtet. Zunehmende Bedeutung erhält das objektorientierte Modell.

2.6 Relationale Datenbanken

Der Begriff der relationalen Datenbanken wurde 1970 von E.F. Codd eingeführt. In dem Artikel *A Relational Model of Data for Large Shared Data Banks* wurde die theoretische Grundlage für relationale Datenbanken festgelegt: Das sogenannte relationale Datenmodell. Im Unterschied zu anderen Datenbanksystemen (netzwerkartigen bzw. hierarchischen Systemen), basiert das relationale Modell völlig auf den mathematischen Grundlagen der relationalen Algebra. Das Grundelement einer relationalen Datenbank ist die Tabelle. Aus Benutzersicht besteht jede relationale Datenbank nur aus Tabellen. Eine Tabelle setzt sich aus Reihen und Spalten zusammen, d.h. Sie beinhaltet keine, eine oder mehrere Reihen und eine oder mehrere Spalten. Das Objekt, das genau zu einer Reihe und einer Spalte gehört heißt Datenwert oder Datum. Liegt ein ER-Model (Entity Relationship Modell) vor, dann wird jede Entitätsmenge auf eine Tabelle gleichen Namens abgebildet. Jede Spalte einer Tabelle repräsentiert ein Attribut der Entitätsmenge. Jedes Attribut besitzt einen Attributtyp. In jeder Zeile kann eine Entität der Entitätsmenge, auch Tupel genannt, gespeichert werden. Die Definition der Datenstrukturen duch Tabellen bezeichnet man als logisches Schema. Die formale Definition des logischen Schemas geschieht durch die Datendefinitionssprache (data definition language, DDL), die das DBMS zur Verfügung stellt. Bei der DDL handelt es sich in der Regel um Sprachen der 4. Generation. Als Standard hat sich SQL etabliert.

Nach Abarbeitung der CREATE-Befehle durch das DBMS ist die Datenbank eingerichtet, aber noch leer, d. h. ohne Daten. Eine eingerichtete Datenbank kann von Benutzern und Anwendungsprogrammen verändert werden (Eintragen, Ändern, Löschen von Daten). Außerdem können Anfragen (queries) an die Datenbank gestellt werden. Für diese Aufgaben stellt das DBMS eine Datenmanipulationssprache (data manipulation language, DML) zur Verfügung. Die DML kann als eigenständige Dialog-Sprache implementiert sein, so daß ein Benutzer direkt im Dialog mit der Datenbank arbeiten kann. Die DML kann aber auch zum Schreiben von Anwendungsprogrammen verwendet werden. Zu beachten ist jedoch, daß eine DML in der Regel über keine Kontrollstrukturen und Prozedurkonzepte verfügt. Daher ist die Erstellung umfangreicher Programme problematisch. Als Alternative kann eine DML aber auch in eine klassische Programmiersprache eingebettet sein, d.h. die Programmiersprache ist um DML-Kommandos erweitert, die von einem Precompiler in ausführbare Zugriffsmodule übersetzt werden. Als DML wird heute in der Regel ebenfalls die Sprache SQL verwendet. Einige relationale Datenbanken sind:

- MS Access
- Oracle
- MySQL
- DB2
- Dbase
- Foxpro
- Informix

- Ingres
- MS SQL Server
- Sybase

2.7 Objektorientierte Datenbanken

2.7.1 Grundsätze

Die Grundsätze, die dem objektorientierten Datenmodell zugrunde liegen, sind einfach. Man kann die reale Welt als eine Vielfalt von miteinander in Beziehung stehenden Objekte auffassen und diese Objekte auf unterschiedlichen Abstraktionsebenen detailliert untersuchen. Betrachtet man beispielsweise einen Baum, dann beschäftigt man sich nicht mit seinem komplizierten Aufbau aus Blättern, Zweigen, Stamm und Wurzeln, sondern man sieht ein unteilbares Objekt mit bestimmten Eigenschaften. Manchmal werden die Blätter des Baumes detailliert betrachtet, aber auch hier gilt das Blatt als ein Objekt mit Eigenschaften wie Farbe, Struktur und Form. Man kann bis auf die Molekularstruktur eines Blattes absteigen, aber auch dann kann ein Molekül als ein Objekt mit wohldefinierten Eigenschaften und Verhalten angesehen werden.

Ein Anwendungsgebiet in Objekte und Beziehungen zu zerlegen, ist eine gängige Technik in der Systemanalyse; das erweiterte Entity-Relationship-Modell ist einer von vielen solcher deduktiven Ansätze. Die traditionellen Datenbankmodelle, das relationale Modell eingeschlossen, tendieren jedoch dazu, die Objekte in künstliche Strukturen zu zerlegen. Objektorientierte Datenbanken andererseits operieren auf der Basis einer anwendungsorientierten Darstellung von der Analyse- bis in die Realisierungsphase. Dieser Ansatz erlaubt es den Anwendern, die Datenbank als eine Sammlung von komplexen, untereinander in Beziehung stehenden Objekten zu sehen, die auf der für ihre Anwendung erforderlichen Detaillierungsstufe betrachtet werden können. Somit unterstützt das objektorientierte Modell eine natürlichere Darstellung der realen Anwendungswelt.

In einer objektorientierten Datenbank ist jedes Objekt ein Exemplar einer Klasse. Die Objekte, die zu einer Klasse gehören, werden alle durch die Klassendefinition beschrieben. Anstelle der Beschreibung individueller Objekte, konzentriert sich damit der objektorientierte Ansatz auf die Muster zulässiger Zustände und Verhaltensmuster, die einer ganzen Klasse von Objekten gemeinsam sind, z.B. der Klasse von Personen, Autos, Büchern, Flugzeugen etc. Der Zustand eines Objektes wird durch Eigenschaften oder Attribute implementiert. Im Gegensatz zu relationalen Datenbanken sind diese Eigenschaften nicht auf atomare Datentypen eingeschränkt, sondern können ihrerseits komplexe Objekte sein. So kann beispielsweise ein Baumobjekt die Eigenschaft „Blatttyp“haben, die, wie oben beschrieben, selbst ein komplexes Objekt ist und eine entsprechende Klassendefinition hat. Das Verhalten eines Objektes wird durch eine Menge von Prozeduren (Methoden) implementiert, die zusammen mit den Eigenschaften verbunden sind. Diese Klassenstruktur, die sowohl die Eigenschaften als auch das Verhalten umfaßt, ist die natürliche Abstraktionseinheit in objektorientierten Systemen. Eine beliebige Zahl von Exemplaren einer Klasse existieren nebeneinander; sie haben unterschiedliche Identität, stimmen aber in der Struktur

und im operationalen Verhaltensmuster überein. Diese Objekte nennt man Objekte derselben Klasse. Bei der Definition einer Klasse ist es wichtig, entsprechend den Grundsätzen der schrittweisen Verfeinerung nicht von vorneherein eine bestimmte Datenrepräsentation festzulegen. Es sollte eine umfassende, genaue und eindeutige Beschreibung der Klasse vorliegen, ohne beispielsweise die Details ihrer darunterliegenden Darstellung ins Kalkül gezogen zu haben. Die Auswahl der sichtbaren Details wird sowohl durch die beabsichtigte Anwendung der Klasse als auch durch deren Benutzer bestimmt. Das vordergründige Ziel ist es, den Anwendern Systemdetails, die für die Anwendung irrelevant sind, zu ersparen, damit sie sich auf wichtige Details konzentrieren können. Objektorientierte Systementwicklung ist in erster Linie eine Datenabstraktionstechnik

2.7.2 Anforderungen an objektorientierter Datenbanksysteme

Objektorientierte Datenbanksysteme, abgekürzt OODBS oder auch ODBS für Objektdatenbanksysteme, sind im Laufe der achtziger Jahre in Forschungslabors entwickelt worden und seit wenigen Jahren kommerziell verfügbar. Eine große Streitfrage betrifft die Definition eines objektorientierten Datenbanksystems und vor allem die Überlegungen, welche Eigenschaften zwingend von einem solchen System unterstützt werden müssen.

Prinzipien der Objektorientierung

- Komplexe Objekte
- Objektidentität
- Datenkapselung
- Typen und Klassen
- Vererbung
- Polymorphismus
- Vollständigkeit
- Erweiterbarkeit

Datenbankgrundsätze

- Dauerhaftigkeit
- Große Datenbestände
- Mehrbenutzerbetrieb
- Rekonstruierbarkeit
- Ad-hoc-Abfragemöglichkeiten

Zu diesen Anforderungen, die für jedes ernstzunehmende Datenbank-Management-System gelten, kommen weitere hinzu, die sich aus dem objektorientierten Umfeld ergeben. So muß eine ODB die:

- Objektidentität verwalten,

- Objekte beliebiger Komplexität speichern,
- Klassen der Objekte und deren Vererbungshierarchie kennen,
- Methoden der Klassen kennen und Polymorphismus unterstützen,
- eine passende Abfragesprache bieten,
- verlustfreies Lesen und Schreiben der Objekte zumindest einer Programmiersprache ermöglichen.

Um Mißverständnisse zu vermeiden, einige Erläuterungen zu den Anforderungen:

Komplexität Ein Objekt kann, bedingt durch seine Klassendefinition, weitere sogenannte Komponentenobjekte und Objektreferenzen beinhalten. Ein Komponentenobjekt ist ein abhängiges Objekt, da es nicht ohne das es beinhaltende Objekt existieren kann.

Objektidentität ist die Vorbedingung, um komplexe Objekte inhaltlich korrekt in der Datenbank abbilden zu können. Sie erlaubt es, inhaltlich gleiche Objekte zu unterscheiden. Objektdatenbanken können daher die wichtige Frage beantworten, ob zwei Referenzen auf zwei gleiche Objekte oder auf ein und dasselbe Objekt verweisen.

Objektdatenbanken bieten in der Regel auch Objekte ohne Identität. Objekte atomarer Klassen besitzen normalerweise keine Identität.

Das Schema einer Objektdatenbank speichert das persistente Objektmodell. Es beinhaltet die Vererbungshierarchie der Klassen, den Aufbau der einzelnen Klassen mit Hilfe der atomaren Klassen und Komponentenklassen sowie die Methoden der Klassen.

Ein Schema ist kein statisches Gebilde. Eine Objektdatenbank muß die Fähigkeit besitzen, das Schema einer bestehenden Datenbank ändern zu können. Hierbei müssen eventuell die Objekte der betroffenen Klassen konvertiert werden. Diese Konversion heißt auch Schema-Evolution. Die Schema-Evolution einer Objektdatenbank ist eine sehr komplexe Prozedur. Jede kommerzielle Datenbank ist in der Lage, das Hinzufügen eines neuen Komponentenobjektes automatisch durchzuführen. Einige bieten eine Programmierschnittstelle, die das eventuelle komplexe Initialisieren der neuen Komponenten ermöglicht.

Die höhere Komplexität der gespeicherten Daten verlangt nach einer Abfragesprache, die dieser gewachsen ist. Eine Abfrage sollte eine Objektmenge und nicht nur Attributwerte liefern können. Die Abfrage muß - falls das Resultat eine Objektmenge ist - die Objektidentität wahren, das heißt die zutreffenden Objekte und keine Kopien liefern. SQL kann die obigen Forderungen nicht erfüllen. Die meisten Objektdatenbanken verfügen daher über eigene proprietäre Abfragesprachen. Diese erfüllen zwar die erwähnten Kriterien, werden aber wegen ihrer Datenbankabhängigkeit von vielen Entwicklern als notwendiges Übel angesehen. Einige Objektdatenbanken unterstützen auch standardnahe Zugriffe via SQL und/oder ODBC. Sie interpretieren hierzu Objektmengen als Tabellen, Objekte als Sätze und Attribute als Spalten einer relationalen

Tabelle. Diese SQL-Implementierungen sind sehr nützlich, da sie dem Meer relationaler Hilfsprogramme, etwa Reportgeneratoren, einen Zugang zu den Objektdaten ermöglichen. Die Object Database Management Group (ODMG) definierte 1993 eine SQL-ähnliche Abfragesprache: die Object Query Language (OQL), die inzwischen von mehreren Objektdatenbanken zumindest teilweise integriert worden ist. OQL besitzt im Gegensatz zu SQL aber keine Einfüge- und Änderungsoperationen (wie z.B. `insert` in SQL). Um solche Operationen auszuführen, müssen Anwendungsprogramme geschrieben werden. OQL besitzt die grundsätzliche `SELECT-FROM-WHERE` Struktur von SQL-Anfragen, ist jedoch nicht mit SQL kompatibel.

2.8 ODMG

ODMG ist eine Vereinigung der Hersteller von Objektdatenbanken. Das Ziel der Organisation ist es, gemeinsame und für die Mitglieder verbindliche Standards für Objektdatenbanken festzulegen. Stimmberechtigte Mitglieder sind:

- GemStone Systems
- IBEX Computing
- O2 Technology
- Object Design
- Objectivity
- POET Software
- UniSQL
- Versant Object Technology

ODMG hat in den letzen Jahren einige Spezifikation veröffentlicht. Sie beinhalten in der Version 1.2 von 1993:

OQL Object Query Language, eine Abfragesprache für Objektdatenbanken, die eine Obermenge von ANSI-SQL 2.0 ist.

DML Data Manipulation Language. DML gibt die ODMG-konforme Sprachschnittstelle für C++ und Smalltalk vor.

ODL Die Object Definition Language standardisiert die Beschreibung des Datenbankschemas.

In der Version 2.0 vom März 1997:

Vorgabe einer JAVA Schnittstelle

Weitere Informationen und nähere Informationen zu den Spezifikationen finden Sie unter: http://www.odmg.org

Einige objektorientierte Datenbanken sind:

- ObjectStore

- Objectivity
- POET
- Versant
- Jasmine
- ODBA2

2.9 Relationale vs. objektorientierter Datenbanksysteme

Im relationalen und objektorientierten Datenbanken gibt es zum Teil gravierende Unterschiede. Die wichtigsten Unterschiede werden hier in einer Tabelle zusammentragen.

Relationales DBS	Objektorientiertes DBS
Wertidentität (wertbasiert) • Identifikation über Schlüssel • Verweise über Fremdschlüssel und zusätzliche Tabellen	**Objektidentität (OID)** • jedes Objekt hat Identität, unabhängig von seinen Attributwerten • Verweise über OIDs
einfache Objekte • Attributtypen = elementare Typen (integer, smallint, decimal, float, char, varchar) • fest definiert	**komplexe Objekte** • Attributtypen = beliebige Typen • Typkonstrukturen struct, Set, Bag, List, Array • benutzerdefinierbar
sichtbare Attribute • strikte Trennung zwischen Datenstrukturen (Schemata) und Anwendungsoperationen • wenige generische Operationen	**bedingt gekapselte Attribute** • Zugriff nur über bereitgestellte Operationen • keine Kapselung für lesende Zugriffe

Fortsetzung...	
Relationales DBS	Objektorientiertes DBS
keine Anbindung an Programmiersprachen • eigenes Typsystem • proprietäre, erweiterte deklarative Sprache einsetzen oder • deklarative Sprache in prozedurale Sprache einbetten • „impedance mismatch“zwischen den Sprachen	**enge Anbindung an Programmiersprachen** • Typsystem Programmiersprache Datenbank integriert • DB-Schema kann in Programmiersprache beschrieben werden • Spracherweiterung um OML (object manipulation language) • Anbindung an OQL (object query language)
Serverorientiert • Zentralisierung von Datenbanken auf Servern	**Clientorientiert** • Verteilung von Objekten auf vernetzten auf Clients
Datenspeicherung • Attributwerte werden gespeichert	**Objektspeicherung** • Attributwerte & Operationen werden gespeichert - heute in der Regel nur Attribute
persistent • In Programmen deklarierte Daten sind transient. • DDL-Teil von SQL	**persistent & transient** • Objekte können persistent oder transient sein
Schema-Definitionssprache (DL)	**Schema-Definitionssprache (DL)** • ODL oder PL-ODL
Manipulationssprache (ML) • DML-Teil von SQL • eingebettetes SQL	**Manipulationssprache (ML)** • OQL • OML in C++, Java bzw. Smalltalk
Eingabe von Daten • mit insert	**Eingabe von Daten** • nur über Anwendungsprogramm, nicht mit OQL
Semantik der Anwendung • verschwindet (nur Tabellen und Fremdschlüssel)	**Semantik der Anwendung** • bleibt erhalten (durch explizite Definition von Relationen)

2.9.1 Fazit

Momentan entspricht das Image objektorientierter Datenbanken etwa dem der relationalen, als diese noch neu waren. Auch wenn Kritiker die grundsätzliche Architektur der Tabellenführer für gut befanden, bemängelte man ihre schwache Leistungsfähigkeit im Vergleich zu den bestehenden Systemen und sagte ihnen keine große Zukunft voraus. Auch wenn Objektdatenbanken bei weitem noch nicht so ausgereift sind, wie ihre relationalen Vorläufer und immer noch Standards fehlen - die Objektverwalter werden immer populärer, nicht zuletzt, weil sich komplexe Objekte darin einfach sinnvoller verwalten lassen. Wie heißt es so schön: *Ein Objekt in einer relationalen Datenbank zu speichern ist, als ob man einen Wagen vollständig auseinandermontiert, um ihn zu parken.*

2.10 Die Zukunft der Datenbanken: OLAP(?)

In den achtziger und neunziger Jahren hat es wichtige Fortschritte im Bereich der Datenbanktechnologien gegeben (Stichwort: »Verteilte und objektorientierte Datenbanken« oder der 1992 verabschiedete ANSI 92-SQL-Standard). Dennoch hat keine dieser Entwicklungen den Datenbankbereich so grundlegend prägen können, wie es mit dem relationalen Modell und SQL der Fall war. Das könnte sich allerdings in den nächsten Jahren (endlich) wieder ändern. Unter den Stichworten Knowledge Management, Data Warehousing und vor allem On-Line Analytical Processing, kurz OLAP, werden neue Verfahren beschrieben, nach denen verknüpfte Tabellen multidimensional in Form von (unsichtbaren, also nur logisch existierenden) Würfeln dargestellt werden. Auf diese Weise lassen sich die Daten unter Gesichtspunkten auswerten, die mit dem relationalen Modell alleine nicht oder nur aufwendig möglich wären. Anwendungen für OLAP gibt es mehr, als Sie wahrscheinlich zunächst vermuten würden. Allerdings lohnt sich der Aufwand nur dann, wenn große Datenmengen im Spiel sind, aus denen in kürzester Zeit neue Erkenntnisse gewonnen werden sollen. Ein Beispiel von vielen sind jene Einzelhändler, bei denen ein großer Teil der Kundschaft bereits mit der EC-Karte bezahlt und so (oft ohne es zu wissen) mit jeder Transaktion neben dem Verkaufsdatum auch Anschriftendaten hinterläßt. Zusammen mit mikrodemoskopischen Daten (etwa dem Kaufkraftquotienten für eine bestimmte Region) sitzt der Einzelhändler auf einer wahren »Daten-Schatztruhe«, die er nur noch mit dem passenden Schlüssel öffnen muß:

- Wieviel Prozent der über die letzte Zeitungsbeilage angesprochenen Kunden waren innerhalb der darauffolgenden Tage in meinem Laden und haben für mehr als 100 DM etwas gekauft?
- Wie anziehend ist ein Sonderangebot auf eine bestimmte Region?
- Wo werden die meisten Shorts im Winter verkauft?

Fragen über Fragen, die sich mit einfachem SQL nicht oder nur mit großem Aufwand beantworten lassen.

3 Beispiel-Datenbank

In diesem Kapitel wird erläutert wie eine Datenbank in die ODBC Schnittstelle eingebunden wird und wie auf diese Schnittstelle auch über ein Netzwerk zugegriffen werden kann. Zunächst wird eine Datenbank modelliert und generiert. Das Konzept der hier verwendete Datenbank stammt aus der Veranstaltung Datenbanken und wird hier nicht näher erläutert. Da diese Datenbank zur Erklärung dient und im weiteren immer wieder darauf zugegriffen wird, sollten Sie wissen wie diese aufgebaut ist.

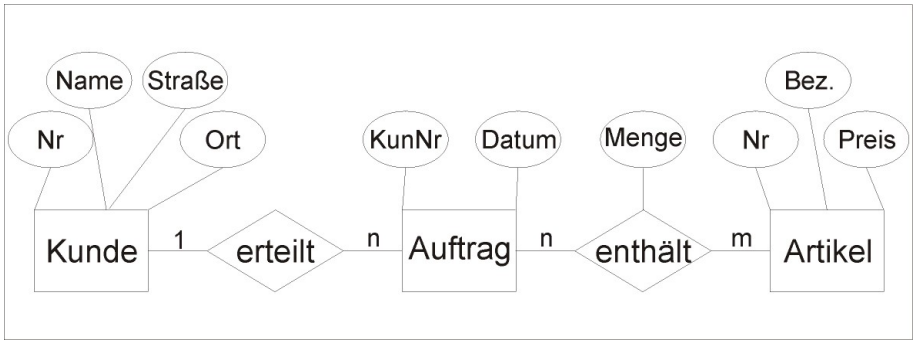


Abbildung 2: Das ER-Modell der Beispiel-Datenbank

Dieses Modell wird jetzt in eine Datenbank übertragen. Dazu durchläuft es den Vorgang der Normalisierung, auf den hier nicht näher eingegangen wird. Das Ergebnis der Normalisierung ist das Relationenschema, welches einfach in eine Datenbank übertragen werden kann. Da später über die ODBC Schnittstelle

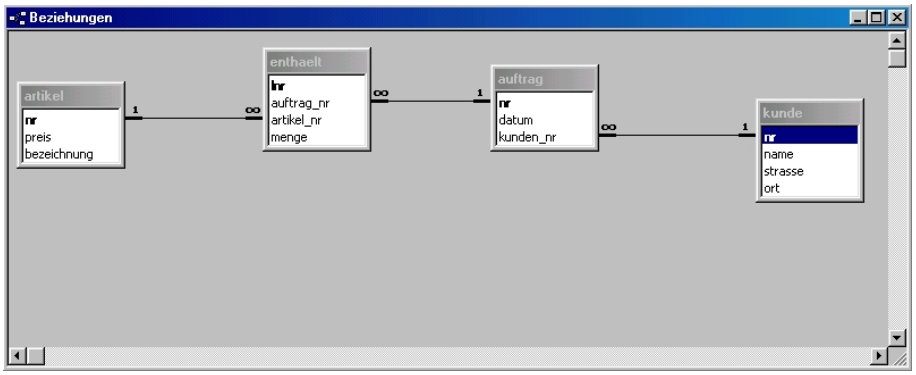


Abbildung 3: Darstellung der Beziehungen

auf die Datenbank zugegriffen werden soll, steht es uns frei ein Datenbank-Management-System zu wählen. Wir wählten Microsoft Access, da die Installation der Treiber für dieses System sehr einfach ist. Prinzipiell ist es egal für welches System Sie sich entscheiden, wichtig ist nur, daß es einen ODBC Treiber für dieses System gibt. Bei den von Microsoft vertriebenen Systemen ist

dies natürlich der Fall. Nach der Generierung der Datenbank mit Microsoft Access 2000 erhalten wir folgendes Beziehungsschema das dem ER-Modell sehr ähnlich sieht. Da später Daten aus dieser Datenbank gelesen verändert bzw. hinzugefügt werden, wird das Data-Dictionary mit den Typen, die wir für die Attribute deklariert haben, notiert. Dies vereinfacht den Entwurf der Prozeduren (Methoden) für den Programmierer wesentlich.

Entity-Typ	`kunde`	*zur Speicherung von Kundendaten*
Attribute	Bedeutung	Datentyp
`nr`	Nr für den Kunden	`Long Integer` *Primärschlüssel*
`name`	enthält den Namen des Kunden	`String 50`
`strasse`	enthält die Straße des Kunden	`String 30`
`ort`	enthält den Ort des Kunden	`String 30`
Entity-Typ	`artikel`	*zur Speicherung von Artikeldaten*
Attribute	Bedeutung	Datentyp
`nr`	Nr für des Artikel	`Long Integer` *Primärschlüssel*
`preis`	Preis des Artikels	`Single`
`bezeichnung`	enthält eine nährere Beschreibung des Artikels	`String 70`
Beziehungstyp	`enthält`	*realisiert die Beziehung zwischen Auftrag und Artikel*
Attribute	Bedeutung	Datentyp
`lnr`	laufende nr	`Long Integer` *Primärschlüssel*
`auftrag_nr`	enhält die nr des Auftrags	`Long Integer`
`artikel_nr`	enhält die nr des Artikels	`Long Integer`
`menge`	enthält die Anzahl des bestellten Artikels	`integer`
Entity-Typ	`auftrag`	*zur Speicherung aller Aufträge*
Attribute	Bedeutung	Datentyp
`nr`	Nr des Auftrags	`Long Integer` *Primärschlüssel*
`datum`	Auftragsdatum	`Datum`
`kunden_nr`	von welchem kunde der Auftrag aufgegeben wurde	`Long Integer`

Jetzt kann die Datenbank, die im weiteren als Auftagsverwaltung bezeichnet wird, in die ODBC-Schnittstelle eingebunden werden. Die Erklärungen und Screenshots der Fenster beziehen sich hier auf ein Windows 9x System. Bei Windows NT Versionen müssen Sie damit rechnen das noch weitere Optionen (wie z.B Rechteverteilung) in den Fenstern zur Verfügung stehen. Die eigentliche ODBC Schnittstelle ist bereits in Windows integriert. Sie befindet sich meist in der Systemsteuerung unter derm Begriff *ODBC ...* . Unter Windows 95 Systemen steht meist nur *ODBC*. Unter Windows 98 der Punkt *ODBC Datenquelllen (32 Bit)* usw. . Wenn Sie das jeweilige Programm starten sehen Sie ein Fenster, welches ungefähr so aussieht.

Um die Datenbank in der ODBC-Schnittstelle einzurichten wählen Sie in diesem Fenster die Option *Hinzufügen*.

Daraufhin erhalten Sie folgendes Fenster:

In diesem Fenster sehen Sie alle ODBC Treiber die auf Ihrem System eingerichtet sind. Standardmäßig sind die meisten von Microsoft entwickelten Treiber schon installiert. Wenn Sie eine Datenbank verwenden für die hier kein Treiber

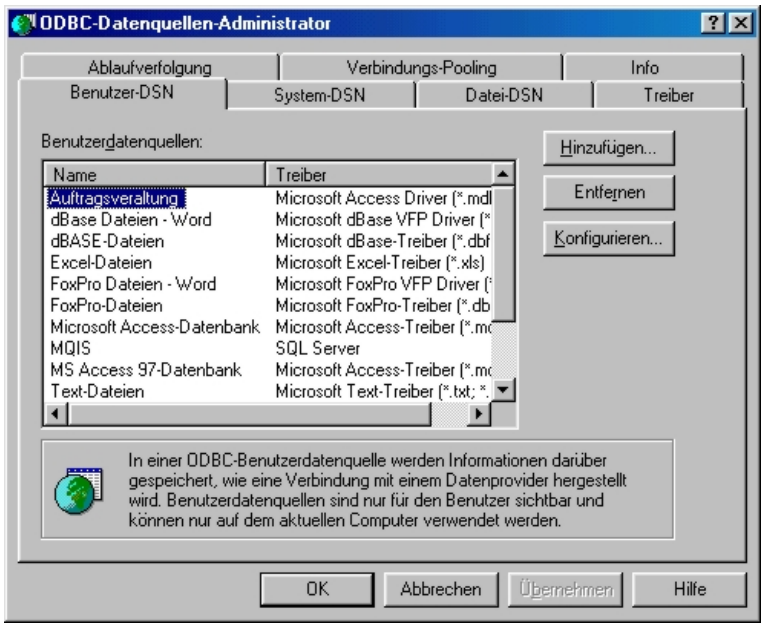


Abbildung 4: Step 1 *ODBC Datenquellen Administrator*

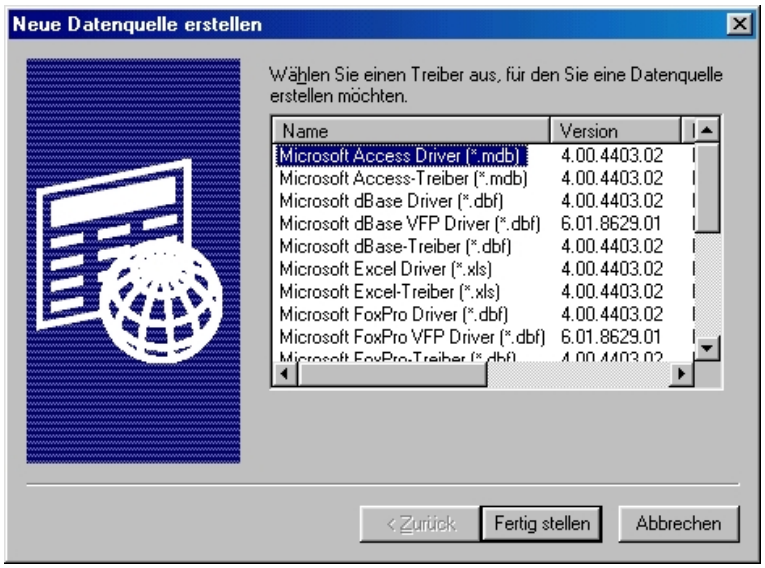


Abbildung 5: Step 2 *Neue Datenquellen erstellen*

aufgelistet ist, wenden Sie sich bitte an den Datenbank-Hersteller und erkundigen Sie sich dort ob es einen ODBC Treiber gibt, und wie Sie diesen installieren müssen. Wählen Sie nun den Treiber für Ihre Datenbank und bestätigen diesen in dem Sie die Option **Fertig stellen** wählen. In der hier vorgestellten

Auftragsverwaltung also den **Microsoft Access Treiber (*.mdb)**. Im darauffolgen Fenster werden nun genaue Angaben zur Datenbank gefordert. Der Aufbau dieses Fensters hängt vom jeweiligen Treiber ab!
Bei Access sieht dies folgendermaßen aus:
Sie müssen unbedingt einen Datenquellennamen (DSN) angeben. Über die-

Abbildung 6: Step 3 *ODBC Microsoft Access Setup*

sen Namen wird später auf die Datenbank zugegriffen. Genauere Angaben zum DSN finden Sie im Allgemeinen Teil unter ODBC. Die optionale Beschreibung dient dazu nähere Angaben zur Datenbank zu machen. Um festzulegen welche Datenbank unter dem DSN Namen angesprochen werden soll, wählen Sie bitte die Option **Auswählen** und wählen dort ihre Datenbank aus. Diese Angaben reichen nun bereits aus um über die ODBC Schnittstelle auf die Datenbank zuzugreifen. Um die Verknüpfung zu übernehmen, wählen Sie nun die Option **OK**. Nun erscheint das Anfangsfenster wieder. In der hier aufgeführen Liste wird nun auch der von Ihnen gewählte DSN Name dargestellt.

3.1 Über ein Netzwerk auf die ODBC Schnittstelle zugreifen

In diesem Kapitel werden Sie lernen wie Sie über ein Netzwerk auf eine Datenbank zugreifen können. Um dies zu erreichen muß auf dem System, auf dem sich die Datenbank befindet, ein Server installiert werden. Die Firma NetDirect stellt einen freien Server names *JdataConnect* zur Verfügung. Sie können diesen von der Homepage von NetDirect downloaden. (http://www.j-netdirect.com) Der hier vorgestellte Treiber bezieht sich auf die Version 2.0. Er greift auf die ODBC Schnittstelle zu und kann somit alle Datenbanken unterstützen die einen Treiber für ODBC bieten. Die Installation ist selbsterklärend. Nach der Installation befindet sich eine Verknüpfung im Start-Menü ihres Windows Systems. Um den Server zu starten wählen Sie bitte die Verknüpfung *Run JData Server*. Dieser öffnet nun eine DOS-Konsole und listet dort Informationen über Zugriff, etc.

Abbildung 7: Step 4 *ODBC Microsoft Access Setup - Auftragsverwaltung*

auf. Sie können nun bereits über ein Netzwerk auf die in der ODBC Schnittstelle installierten Datenbanken zugreifen. Um den Server näher zu konfigurieren, müssen Sie das Programm *JdataAdmin.exe* starten. Bei der Installation wird keine Verknüpfung dieses Tools im Startmenü erstellt! Sie müssen also "zu Fuß" dieses Programm starten. Sie finden es im Installationspfad (Standardmäßig:
`Program Files`
`NetDirect`
`JDataConnect`). Nähere Angaben zu diesem Server entnehmen Sie bitte den Hilfeseiten.

4 Überblick

4.1 Anforderungen an die Schnittstelle zur Datenbank

Die Anforderungen, die wir an die Schnittstelle gestellt haben, lassen sich in einige knappe Worte fassen.

- einfach anzuwenden
- einfach zu erlernen
- so allgemein wie möglich

Wir haben uns alle Mühe gegeben, diese Punkte soweit es uns möglich war zu erfüllen. Ob uns das tatsächlich gelungen ist wird sich in den folgenden Kapiteln zeigen.

4.2 Klassendiagramm

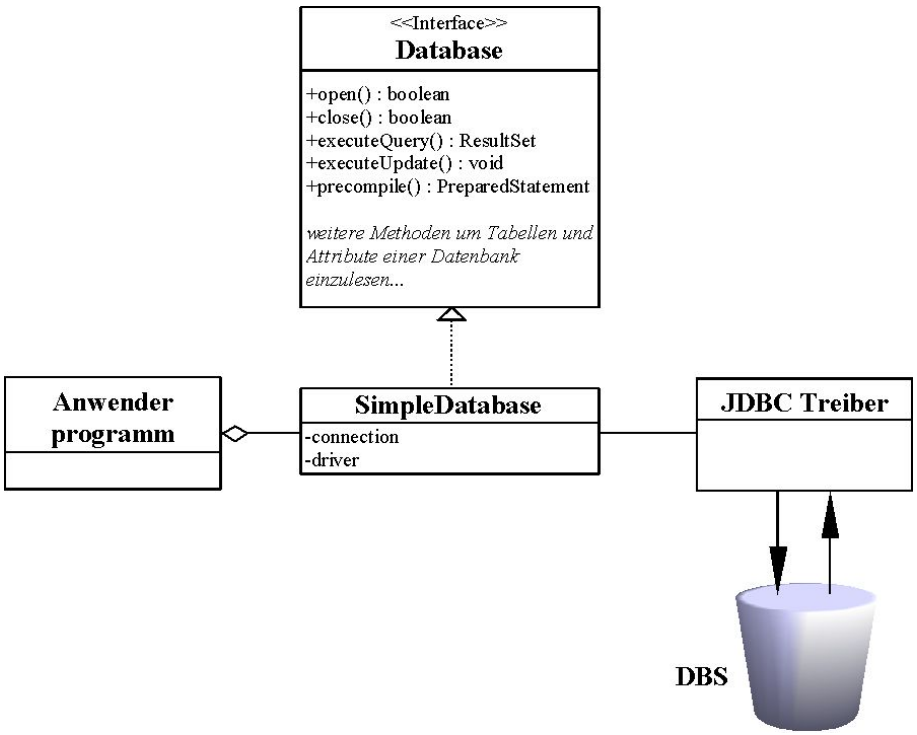


Abbildung 8: Das Klassendiagramm der Schnittstellenanbindung.

4.3 Die Datenbank-Schnittstelle

Um die Datenbank-Schnittstelle so allgemein wie möglich zu halten, haben wir uns entschieden, ein Interface zu definieren. In diesem Interface sind Methoden deklariert, die notwendig sind um eine Datenbank ansprechen zu können. Wie

diese Methoden implementiert sein sollen wird hier nicht näher spezifiziert. Dies ist Aufgabe einer Klasse, die dieses Interface einbindet. Eine solche Klasse wird in den folgenden Kapiteln näher beschrieben. Bei der Betrachtung der Schnittstelle kommt es nur auf eine generelle Funktionalität an, die vorhanden sein muß, um eine Datenbank ansprechen zu können.
Die Arbeitsweise der Schnittstelle läßt sich in drei grundlegende Bereiche unterteilen:

- Aufbau einer Verbindung
- Kommunikation
- Abbau der Verbindung

Desweiteren gibt es noch Möglichkeiten der Performancesteigerung, die am Ende dieses Kapitels noch näher erläutert werden.

4.3.1 Aufbau einer Verbindung

Bevor eine Kommunikation mit der Datenbank stattfinden kann, muß eine Verbindung zu einer bestehenden Datenbank aufgebaut werden. Dabei muß die zu verwendende **Datenbank** im System bekannt sein. In unserem Fall geschieht das über das Einbinden der Datenbank in die ODBC-Schnittstelle. Um von Java aus auf die Datenbank zugreifen zu können, benötigt man einen **Treiber**, der die Kommunikation steuert. Hier stehen mehrere Treiber von verschiedensten Firmen zur Verfügung. Desweiteren besteht bei fast allen Datenbank Systemen die Möglichkeit **Benutzer** mit bestimmten Zugriffsrechten einzurichten. Meist muß sich ein Benutzer mit einem **Passwort** authorisieren.
Diese Angaben können allerdings von einem Datenbank System zum anderen verschieden sein. Aus diesem Grund ist die Spezifikation nicht Bestandteil der Schnittstelle, sondern bleibt den implementierenden Klassen überlassen. In der Schnittstelle wird definiert, daß es eine Methode zum Öffnen der Datenbank geben muß. Diese Methode heißt:

```
public void open()
```

Beim Aufruf dieser Methode ist es möglich, das Fehler (Exceptions) auftreten. Diese müssen vom Anwenderprogramm abgefangen werden. Diese Ausnahmezustände sind:

SQLException ein initialisierender SQL Aufruf ist gescheitert.

ClassNotFoundException Der Treiber konnte nicht geladen werden.

InstantiationException Es konnte keine neue Instanz des Treibers geladen werden.

IllegalAccessException Die Treiberklasse konnte nicht initialisiert werden.

Nachdem die Datenbankverbindung hergestellt wurde, kann der aktuelle Status der Verbindung mit der Methode

```
public boolean isClosed()
```

abgefragt werden. Ein Rückgabewert von `true` bedeutet, daß die Verbindung nicht mehr existiert.

4.3.2 Kommunikation

Die Kommunikation mit der Datenbank findet über SQL statt. Es gibt zwei Varianten eine SQL-Anweisung an die Datenbank zu schicken:

- `public ResultSet executeQuery(String statement)`
  Die Methode `executeQuery` wird bei Anfragen an die Datenbank benutzt, die ein Ergebnis zurückliefern können. Im allgemeinen ist dies der SQL-Befehl `SELECT`. Das Ergebnis wird in Form eines ResultSet zurückgeliefert. Wie ein ResultSet ausgewertet werden kann wird in einem der folgenden Kapitel näher erläutert.

- `public void executeUpdate(String statement)`
  Diese Methode wird benutzt, wenn der SQL-Befehl keine Rückgabewerte hat. Zum Beispiel der Befehl `CREATE`.

Die Daten werden beim Aufruf dieser Methoden nicht sofort übertragen - sondern zwischengespeichert. Eine Übertragung an die Datenbank findet erst statt, wenn explizit die Methode

```
public void commit()
```

aufgerufen wird. Es gibt zwar die Möglichkeit die Übertragung zu automatisieren, allerdings ist dies aus Performancegründen nicht sinnvoll. Daher wird empfohlen, das implementierende Klassen diese Funktionalität abstellen und es dem Anwenderprogramm überlassen die Daten explizit zu übertragen.

Zusätzlich zu den Methoden, mit denen SQL-Anweisungen an die Datenbank gesendet werden können, werden noch weitere Methoden definiert, die es gestatten die Inhalte der Datenbank auszulesen. Unter Inhalten verstehen wir hier Tabellen und deren Attribute. Hier eine Liste der Methoden die für diese Funktionalität vorgesehen sind:

- `public String[] getTablenames()`
- `public ResultSet getTableDescription(String tableName)`
- `public ResultSet getTableDescriptions()`
- `public String[] getColumnNames(String tableName)`
- `public String[] getColumnNames(ResultSet result)`
- `public ResultSet getColumnDescription(String tableName, String columnName)`
- `public ResultSet getColumnDescription(String tableName, int columnIndex)`
- `public ResultSet getColumnDescriptions(String tableName)`
- `public boolean tableExists(String tableName)`
- `public boolean columnExists(String tableName, String columnName)`
- `public boolean columnExists(String tableName, int columnIndex)`
- `public int getColumnCount(String tableName)`
- `public int getColumnCount(ResultSet result)`
- `public int getTableCount()`
- `public java.util.Vector getColumnNamesVector(ResultSet result)`
- `public java.util.Vector getColumnNamesVector(String tableName)`
- `public java.util.Vector getTablenamesVector()`

Die Funktion und Handhabung der Methoden werden in einem der folgenden Kapiteln näher erläutert. Ansonsten ist auch eine vollständige Dokumentation der Schnittstelle als JavaDoc Dokument erhältlich. Bei allen Methoden, die der Kommunikation dienen, können Fehlerzustände eintreten. Zwei Fehlerzustände, die von fast allen Methoden dieses Bereichs ausgeworfen werden können, sind die `NotConnectedException`, wenn keine Verbindung zur Datenbank besteht, sowie eine `SQLException`, wenn der Zugriff auf die Datenbank fehlschlug. Weitere Fehlerzustände finden Sie in der JavaDoc Dokumentation beschrieben.

4.3.3 Abbau der Verbindung

Wenn das Anwenderprogramm keine Datenbankzugriffe mehr benötigt, sollte die Verbindung zur Datenbank abgebaut werden. Spätestens jedoch wenn das Anwenderprogramm beendet wird. Um eine Datenbankverbindung zu schließen ist die Methode

```
public void close()
```

definiert. Wird die Verbindung korrekt abgebaut, so werden alle noch nicht gesendeten Daten vor dem Verbindungsabbau übertragen. Auch beim Abbau einer Verbindung können Fehlerzustände (Exceptions) eintreten. Eine `SQLException` wird ausgeworfen, wenn der Zugriff auf die Datenbank fehlschlug.
Eine `NotConnectedException` wird generiert, wenn keine Verbindung zur Datenbank existiert.

4.3.4 Performancesteigerung

Um häufig verwendete SQL-Anweisungen schneller verarbeiten zu können, gibt es die Möglichkeit SQL-Anweisungen im Vorfeld zu generieren und später nur noch variable Parameter zu ändern. Dies ist gerade bei größeren SQL-Befehlen sinnvoll. Solche vorbereiteten SQL-Anweisungen nennt man PreparedStatements. Sie können sobald eine Datenbankverbindung besteht mit dem Befehl

```
public PreparedStatement precompile(String statement)
```

erstellt werden.

5 Prototyp DBTest

Der Prototyp **DBTest.jar** ist eine Beispielanwendung um auf Datenbanken zuzugreifen.
Sie wurde mit JBuilder4 entwickelt und benutzt unser eigenes Package **de.web.databasesupport.tools.database** und den Treiber **JData2.0.**
Als Funktionsumfang besitzt er zwei verschiedene Datenbanktreiber, zum einen den JDBC:ODBC-Treiber von SUN, zum anderen den JData2.0-Treiber von NetDirect (www.j-netdirect.com).
Der Treiber von SUN ermöglicht es auf Datenbanken, die lokal in der ODBC-Schnittstelle eingebunden sind, zuzugreifen.
Der JData2.0-Treiber ermöglicht es eine Datenbank auch über eine Netzwerkverbindung anzusprechen. Hierfür ist ein Server erforderlich. Dieser ist frei erhältlich bei der Firma NetDirect. Dieser Server läuft nur auf Windows-Rechnern und kann diverse Datenbanken, die in der ODBC-Schnittstelle des Rechners eingebunden sind, verwalten. Folgende Datenbanken können verwaltet werden:

- MS Access
- Oracle
- MySQL
- DB2
- dBase
- FoxPro
- Informix
- Ingres
- MS SQL Server
- ODBC
- Sybase

Beim Öffnen einer Datenbank, werden alle verfügbaren Tabellen dargestellt. Beim Anklicken einer Tabelle werden alle Informationen über die Tabelle im Fenster Results angezeigt. Desweiteren werden alle Attribute der Tabelle aufgelistet. Über diese kann man Informationen erhalten, indem man sie im Fenster Columnnames anklickt.
Es besteht auch die Möglichkeit SQL-Befehle einzugeben, die über den Knopf <Execute SQL-Cmd> an die Datenbank geschickt werden. Lieferte dieser SQL-Befehl ein Ergebnis (ResultSet) zurück, so wird dies im Fenster Results angezeigt.
Treten Fehler auf, z.B. ein ungültiger SQL-Befehl, Datenbank geschlossen, ... so bekommt man eine Beschreibung des Fehler im unteren Fenster eingeblendet.

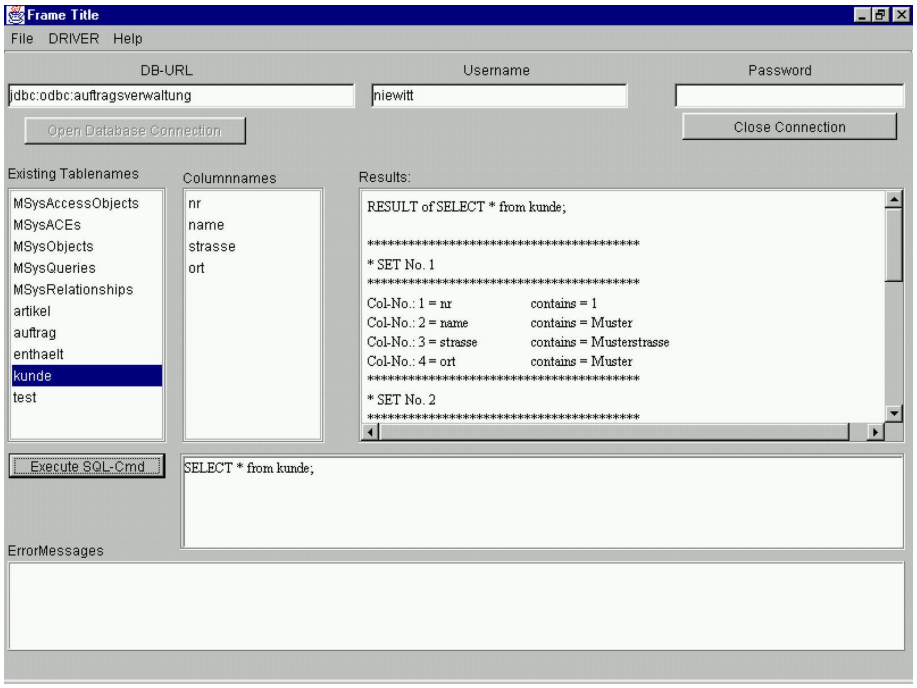


Abbildung 9: Screenshot des Prototypen DBTest

6 Die wichtigsten Interfaces aus dem Paket: java.sql

- Driver
- Connection
- ResultSet
- ResultSetMetaData
- DatabaseMetaData
- Statement
- PreparedStatement

6.1 Was ist das Interface Driver?

Ein Driver ist ein Interface, das jeder Treiber implementieren muß.
Ein Treiber kann wie folgt für eine eigene Database-Klasse eingebunden werden:

```
Driver drv = (Driver) Class.forName(jdbcDriver).newInstance();
```

Um für Ihre Datenbank den passenden Treiber zu finden, bietet SUN eine Suchmaschiene an.
Diese ist unter:

http://industry.java.sun.com/products/jdbc/drivers

zu finden.

(Detailierte Informationen finden Sie in der API unter: `java.sql.Driver`)

6.2 Was ist eine Connection?

Eine Connection ist eine Verbindung zu einer Datenbank.
Über sie werden Befehle wie das Öffnen, Schließen, Ausführen von Anfragen, das Vorkompilieren von SQL-Befehlen an die Datenbank weitergeleitet.
Eine Connection kann wie folgt aufgebaut werden:

```
Connection con = java.sql.DriverManager.getConnection(dbURL, Properties);
```

(Detailierte Informationen finden Sie in der API unter: `java.sql.Connection`)

6.3 Was ist ein ResultSet?

Ein ResultSet ist eine Tabelle mit einer gewissen Anzahl von Zeilen (row) und Spalten (column). Es kann z.B. durch den Aufruf `executeQuery(sql-statement)` mit Daten gefüllt, wobei die Zeilen immer einen zusammenhängenden Datensatz repräsentieren. Die Spalten sind die einzelnen Informationen in diesem Datensatz und haben Namen und möglicherweise Inhalte.
Die Auswertung wird im weiterem Verlauf noch betrachtet.

(Detailierte Informationen finden Sie in der API unter: `java.sql.ResultSet`)

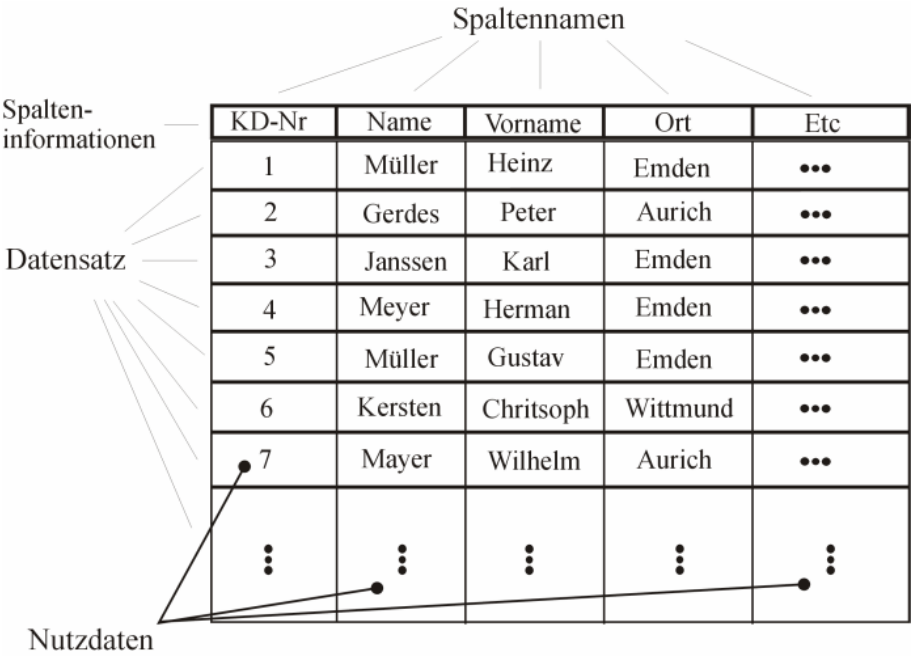


Abbildung 10: Beispielhafter Aufbau eines ResultSets

6.4 Was ist das Interface ResultSetMetaData?

Dieses Interface ermöglicht es, Informationen aus einem ResultSet zu erhalten, z.B. der Spaltenname, wieviele Spalten überhaupt im ResultSet vorhanden sind, sowie weitere Funktionalitäten, auf die hier nicht näher eingegangen wird..
Ein ResultSetMetaData-Objekt wird duch die Methode: `ResultSet.getMetaData()` erzeugt.

(Detailierte Informationen finden Sie in der API unter: `java.sql.ResultSetMetaData; java.sql.ResultSet`)

6.5 Was ist das Interface DatabaseMetaData?

Dieses Interface erlaubt es dem Benutzer, unter Verwendung der Connection, Informationen über die Datenbank zu bekommen. Dies sind z.B. Informationen über die einzelnen Tabellen, verwendete Treiber, etc. .
Ein DatabaseMetaData-Objekt wird durch die Methode: `Connection.getMetaData()` erzeugt.

(Detailierte Informationen finden Sie in der API unter: `java.sql.DatabaseMetaData; java.sql.Connection`)

6.6 Was ist ein Statement?

Ein Statement wird verwendet um SQL-Befehle auszuführen. Dieses Statement wird über `Statement.execute()`, `.executeQuery()` oder `.executeUpdate()` ausgeführt und liefert je nach verwendeter Methode ein Ergebnis zurück, z.B. `executeQuery()` liefert ein ResultSet zurück.

(Detailierte Informationen finden Sie in der API unter: `java.sql.Statement`)

6.7 Was ist ein PreparedStatement?

Um ein PreparedStatement zu erhalten übergibt man einen String, der den SQL-Befehl enthält, an die Methode `Connection.prepareStatement()`. Diese liefert ein Objekt des Typs `PreparedStatement` zurück. Ein PreparedStatement ist direkt mit der jeweiligen `Connection` verbunden. Das heißt nach dem zusammenbrechen der Verbindung muß das PreparedStatement neu erzeugt werden. Es besteht die Möglichkeit Platzhalter (**"?"**) in den SQL-Befehl einzubinden. Diese müssen dann vor dem Ausführen des Befehls mit den Methoden `setString()`, `setInt()`, etc. auf konkrete Werte gesetzt werden. Ausgeführt wird ein PreparedStatement mit der Methode `PreparedStatement.executeQuery()`, `PreparedStatement.execute()` oder `PreparedStatement.executeUpdate()`.

(Detailierte Informationen finden Sie in der API unter: `java.sql.PreparedStatement`)

7 Implementation Schnittstelle

In diesem Kapitel wird erklärt, wie die einzelnen Methoden der Klasse `SimpleDatabase` zu verwenden sind und welche Eigenheiten und interne Methodenaufrufe die einzelne Methode hat.

7.1 Methoden zur Datenbankanbindung

`void open()`

Diese Methode öffnet eine Verbindung(Connection) zu einer Datenbank, wenn der Treiber, die Datenbank-URL sowie Benutzer und Passwort gültig sind. Nach unseren Erfahrungen scheint es MS-Access gleichgültig zu sein, was man als Benutzername und Passwort angibt.

`void close()`

Diese Methode schließt die Datenbankverbindung.

`boolean isClosed()`

Der boolsche Wert dieser Methode zeigt an, ob die Datenbankverbindung offen (`false`) oder geschlossen (`true`) ist. Alle Methoden in SimpleDatabase, die einen Zugriff auf die Datenbankverbindung benötigen, haben als Eingangsabfrage ob die Datenbank geöffnet ist. Liefert das `isClosed()` dort ein `false`, so wird eine `NotConnectedException` geworfen.

7.2 Methoden zur Datenbankverwaltung(SQL)

`void commit()`

Diese Methode sorgt dafür, daß die Daten auch in die Datenbank übernommen werden. Sie ist notwendig, da beim Initialisieren der Verbindung `autoCommit()` auf `false` gesetzt wird. AutoCommit bedeutet, daß die Datenbank in gewissen Abständen die Daten in die Tabellen übernimmt. Da es aber von Datenbank zu Datenbank unterschiedlich ist, wann dies geschieht, halten wir es für sinnvoll, daß diese Übernahme der Daten vom Programmierer selbst vorgenommen wird. Ein nicht sachgemäßes Schließen der Datenbankverbindung (z.B. Abbruch des Programms,...), ohne ein `commit()`, kann sonst zu Datenverlust führen.

`ResultSet executeQuery(String sqlstatement)`

Führt ein SQL-Befehl aus und liefert das Ergebnis in einem ResultSet zurück. Liefert der SQL-Befehl kein ResultSet zurück (z.B. `UPDATE, CREATE, ADD, ...`) so wird eine SQLException geworfen. Es können auch leere ResultSets entstehen.

`void executeUpdate(String sqlstatement)`

Führt ein SQL-Befehl aus. Um diese Änderung in die Datenbank zu übernehmen, muß noch die Methode `commit()`ausgeführt werden. Liefert der SQL-Befehl ein ResultSet zurück (z.B. `SELECT`) so wird eine `SQLException` geworfen.

`PreparedStatement precompile(String sqlstatement)`

Erzeugt ein PreparedStatement aus dem SQL-Befehl mit Platzhaltern. Hier wird nicht kontrolliert ob der SQL-Befehl gültig ist oder formale Mängel aufweist. Dies geschieht erst beim Ausführen eines PreparedStatement.

7.3 Methoden über den Datenbankaufbau

`ResultSet getTableDescriptions()`

Liefert ein ResultSet zurück, in dem alle Tabellen(auch Systeminterne), die in der Datenbank vorkommen, mit detailierten Informationen zu den einzelnen Tabellen.
Die einzelnen Spalten bedeuten folgendes:

1. `TABLE_CAT` String → der Katalog (Pfad zur Datenbank auf dem Server)
2. `TABLE_SCHEM` String → ein Tabellen-Schema
3. `TABLE_NAME` String → der Tabellenname
4. `TABLE_TYPE` String → der Typ der Tabelle. (TABLE, VIEW,...)
   (Detailierte Informationen finden Sie in der API unter: `java.sql.DatabaseMetaData`)
5. `REMARKS` String → Bemerkungen zur Tabelle

`ResultSet getTableDescription(String tablename)`

Liefert ein ResultSet zurück, in dem Informationen zu der gegeben Tabelle stehen. Was die einzelnen Spalten bedeuten ist unter `getTableDescriptions()` nachzulesen. Ist der `tablename = null`, so liefert diese Methode Informationen zu allen Tabellen zurück.
Verwendet intern die Methode `getTableDescriptions()`.

`String[] getTableNames()`

Liefert alle Tabellennamen als String-Array zurück.
Verwendet intern die Methode `getTableDescriptions()`.

`java.util.Vector getTableNamesVector()`

Liefert alle Tabellennamen als Vector mit String-Objekten zurück.
Verwendet intern die Methode `getTableDescriptions()`.

`int getTableCount()`

Liefert die Anzahl der Tabellen in einer Datenbank zurück.

`ResultSet getColumnDescriptions(String tablename)`

Liefert Informationen über alle Spalten einer Tabelle zurück. Die einzelnen Spalten in dem ResultSet bedeuten folgendes:

1. `TABLE_CAT` String → der Katalog (Pfad zur Datenbank auf dem Server)
2. `TABLE_SCHEM` String → ein Tabellen-Schema
3. `TABLE_NAME` String → der Tabellenname

4. `COLUMN_NAME` String → der Spaltenname
5. `DATA_TYPE` short → den DatenTyp aus java.sql.Types
6. `TYPE_NAME` String → DatenTyp Namen
7. `COLUMN_SIZE` int → Bei Chars oder Text ist das die Anzahl der Zeichen, bei Zahlen die Genauigkeit
8. `BUFFER_LENGTH` wird nicht verwendet
9. `DECIMAL_DIGITS` int → Die Anzahl der Nachkommastellen
10. `NUM_PREC_RADIX` int → Die Basis der Zahl (normalerweise 10 oder 2)
11. `NULLABLE` int → ist null als Inhalt erlaubt ? (s. `DatabaseMetaData.getColums()`)
12. `REMARKS` String → Beschreibung der Spalte
13. `COLUMN_DEF` String → Default-Wert
14. `SQL_DATA_TYPE` int → wird nicht verwendet
15. `SQL_DATETIME_SUB` int → wird nicht verwendet
16. `CHAR_OCTET_LENGTH` int → Maximale Anzahl von Bytes bei Char-Typen
17. `ORDINAL_POSITION` int → Position der Spalte in der Tabelle
18. `IS_NULLABLE` String → "YES" = kann `null` sein - "NO" darf nicht `null` sein
19. `ORDINAL` int → Position der Spalte in der Tabelle

(Detailierte Informationen finden Sie in der API unter: `java.sql.DatabaseMetaData.getColumns()`)

`ResultSet getColumnDescription(String tablename, String columnname)`

Liefert Informationen über eine Spalte in einer Tabelle zurück. Ist `tablename = null`, so werden alle Spalten in der Datenbank gesucht, die den Namen `columnname` haben.
Ist `columnname = null`, so werden alle Spalten einer Tabelle zurückgeliefert.
Sind beide `null`, so werden alle Spalten aus allen Tabellen zurückgeliefert. Die Beschreibung der einzelnen Spalten im ResultSet kann aus `getColumnDescriptions(...)` entnommen werden.

`ResultSet getColumnDescription(String tablename, String columnindex)`

Liefert eine Informationen über eine Spalte in einer Tabelle zurück. Ist `tablename = null`, so werden alle Spalten in der Datenbank gesucht, die den Index `columnindex` haben.
Ist `columnindex` kleiner als 1 oder größer als die die Anzahl der Spalten in der Tabelle, so wird eine `ArrayIndexOutOfBoundsException` geworfen.
Die Beschreibung der einzelnen Spalten im ResultSet kann aus `getColumnDescriptions(...)` entnommen werden.

`String[] getColumnames(String tablename)`

Liefert alle Spaltennamen einer Tabelle als String-Array zurück. Ist `tablename = null`, so werden alle Spaltennamen aller Tabellen zurückgeliefert.
Verwendet intern die Methode `getColumnDescriptions(...)`.

`java.util.Vector getColumnNamesVector(String tablename)`

Liefert alle Spaltennamen einer Tabelle als Vector mit String-Objekten zurück. Ist `tablename = null`, so werden alle Spaltennamen aller Tabellen zurückgeliefert.
Verwendet intern die Methode `getColumnDescriptions(...)`.

`int getColumnCount(String tablename)`

Liefert die Anzahl der verfügbaren Spalten einer Tabelle zurück. Ist `tablename = null`, so liefert sie die Anzahl aller verfügbaren Spalten in der Datenbank zurück.

`boolean tableExists(String tablename)`

Liefert nur dann `true` zurück, wenn es eine Tabelle mit dem Namen `tablename` gibt, ansonsten `false`.

`boolean columnExists(String tablename, String columnname)`

Liefert nur dann `true` zurück, wenn es in einer Tabelle den Spaltennamen `columnname` gibt. Ist `tablename = null`, so werden alle Tabellen auf den Spaltennamen `columnname` durchsucht.

`boolean columnExists(String tablename, int columnindex)`

Liefert nur dann `true` zurück, wenn es in einer Tabelle den Spaltenindex `columnindex` gibt. Ist `tablename = null`, so werden alle Tabellen auf den Spaltenindex `columnindex` durchsucht. Existiert eine Spalte, die diesen Spaltenindex besitzt, so wird auch `true` zurückgeliefert.

7.4 Methoden zur ResultSetAnalyse

`String[] getColumnames(ResultSet result)`

Liefert alle Spaltennamen aus einem ResultSet als String-Array zurück. Ist das ResultSet leer, oder `null`, so werden Exceptions geworfen.

`java.util.Vector getColumnNamesVector(ResultSet result)`

Liefert alle Spaltennamen einer Tabelle als Vector mit String-Objekten zurück. Ist das ResultSet leer, oder `null`, so werden Exceptions geworfen.

`int getColumnCount(ResultSet result)`

Liefert die Anzahl der verfügbaren Spalten eines ResultSets zurück. Auch ein leeres ResultSet kann Spalten enthalten. Ist das `ResultSet = null`, so wird eine `NullPointerException` geworfen.

8 Verwendung der Klasse SimpleDatabase

Aufbau einer Verbindung zu einer Beispiel-Datenbank "`auftragsverwaltung`"[1]:

Um eine Verbindung zu einer Datenbank aufzubauen, wird eine Treiberadresse, die Datenbank-URL, sowie optional (je nach Treiber) Benutzername und Passwort benötigt. Eine Datenbank-URL besteht aus folgenden drei Bestandteilen:

1. Datenbankprotokoll, bei JDBC ist dies immer jdbc
2. Unterprotokoll der Treiber, der eine oder mehrere Datenbanken ansprechen kann
3. Name, der die anzusprechende Datenbank identifiziert.

Getrennt werden die drei Bestandteile durch ein **":"**.
Mit diesen Angaben wird eine neues Exemplar der Klasse `SimpleDatabase` erzeugt. Der Treibername muß als String übergeben werden. Die Namen der bisher vorhandenen zwei Treiber (von SUN und NetDirect) sind in `SimpleDatabase` als Konstanten definiert:

```
public final static String JDBC_ODBC_DRIVER = "sun.jdbc.odbc.JdbcOdbcDriver";
public final static String JDATA2_0_DRIVER = "Jdata2_0.sql.$Driver";
```

Beispiel:

```
1.  String username = "admin";
2.  String password = "test";
3.  String dbURL = "jdbc:JDataConnect://127.0.0.1/auftragsverwaltung";
4.  String driver = SimpleDatabase.JDATA2_0_DRIVER;
5.  Database db = new SimpleDatabase(driver, dbURL, username, password);
```

Um die Datenbank ansprechen zu können, muß noch eine Verbindung aufgebaut werden.
Hierzu wird die Methode `SimpleDatabase.open()` verwendet.
Bei dieser Methode müssen verschiedene Exceptions abgefangen werden, da Fehler beim Laden des Treibers auftreten können, als auch eine fehlerhafte Datenbank-URL angegeben werden konnte.

Beispiel:

```
1.  try {
2.    db.open();
3.  } catch (ClassNotFoundException exc) {
4.    // Kann den Treiber nicht laden. Überprüfen des Classpath
5.  } catch (InstantiationException exc) {
6.    // Kann keine neue Instanz des Treibers anlegen
7.  } catch (IllegalAccessException exc) {
```

[1] Nähere Informationen zur Beispieldatenbank "auftragsverwaltung" finden Sie im Kapitel 3 auf Seite 23

```
8.    // Wenn die Treiberklasse nicht erreichbar ist
9.  } catch (SQLException exc) {
10.    // Wenn das setzen von AutoCommit auf false nicht geklappt hat
11.  }
```

Nachdem die Datenbank-Verbindung aufgebaut ist, können verschiedene Aktionen ausgeführt werden:

- Das Ausführen von SQL-Anweisungen
- Das Vorkompilieren von PreparedStatements
- Das Ausführen von PreparedStatements
- Informationen über die Datenbank holen
- Das Schließen der Datenbankverbindung

Hier je ein Beispiel zu SQL-Statements und PreparedStatements:

Beispiel SQL-Statement:

```
1.  String sqlStatement = "SELECT * FROM kunde WHERE nr < 100 ;";
2.  ResultSet result = null;
3.  try {
4.    result = db.executeQuery(sqlStatement);
5.  } catch (SQLException exc) {
6.    System.err.println("Error executing SQL-Statement. "+
        exc.getMessage());
7.  } catch (NotConnectedException exc) {
8.    System.err.println(
        "Could not execute Statement. Database is closed.");
9.  }
```

Beispiel PreparedStatement:

```
1.  // Vorbereitung des PreparedStatements
2.  String sqlStatement = "SELECT ? FROM kunde WHERE nr < ? ;";
3.  ResultSet result = null;
4.  PreparedStatement preSQL = null;
5.  // vorkompilieren des SQL-Statements und Erhalt eines PreparedStatement
6.  try {
7.    preSQL = db.precompile(sqlStatement);
8.  } catch (SQLException exc) {
9.    System.err.println("Error precompiling SQL-Statement. "+
        exc.getMessage());
10. } catch (NotConnectedException exc) {
11.   System.err.println("Could not execute Statement. Database is closed.");
12. }
13. // Setzen der Platzhalter und ausführen des PreparedStatement
14. try {
15.   preSQL.setString(1, "name");
```

```
16.   preSQL.setInt(2, 100);
17.   result = preSQL.executeQuery();
18. } catch (SQLException exc) {
19.   System.err.println("Error executing SQL-Statement. "+
        exc.getMessage());
20. } catch (NotConnectedException exc) {
21.   System.err.println("Could not execute Statement. Database is closed.");
22. }
```

Bei beiden Beispielen erhält man ein ResultSet. Wie man dieses auswertet wird in einem späteren Kapitel erläutert.
Wenn vom Anwenderprogramm nicht mehr auf die Datenbank zugegriffen wird, muß die Datenbank wieder geschlossen werden. Dies geschieht mit der Methode: `SimpleDatabase.close()`.

Beispiel zum schließen einer Datenbank:

```
1.  try {
2.     db.close();
3.  } catch (SQLException exc) {
4.    System.err.println("Error closing the Database.");
5.    exc.printStackTrace();
6.  } catch (NotConnectedException exc) {
7.    System.err.println("Database already closed.");
8.  }
```

Dieser Codeausschnitt schließt die Datenbankverbindung. Falls ein Fehler auftritt, wird je nach Fehlerfall eine Meldung ausgegeben.

Ein kleines Beispielprogramm zum Aufbau einer Verbindung, Ausführen einer SQL-Anweisung, Auswerten eines ResultSets und Schließen der Verbindung, ist im Anhang im Kapitel 2 auf Seite 52 zu finden.

9 ResultSet: Der Umgang und die Auswertung

Ein von der Klasse `SimpleDatabase` erzeugtes ResultSet kann nur von vorne nach hinten durchlaufen werden. Es empfiehlt sich auch die einzelnen Spalten der Reihe nach auszuwerten.

Ein Beispiel:

```
1.  public void showResult(ResultSet result) {
2.    try {
3.      while (result.next()) {
4.        String[] names = db.getColumnNames(result);
5.        int i;
6.        for (i=1; i<=db.getColumnCount(result); i++) {
8.          // result.getObject(i) holt das Object aus der i-ten Column,
9.          // bei festen Typen kann auch getString()
10.         // oder getInt() ... verwendet werden
11.         System.out.print("Col-No.: "+ i +" = "+ names[i-1] +" = ");
12.         System.out.println(result.getObject(i));
13.       }
14.     }
15.   result.close()
16.   } catch (SQLException exc) {
17.     // ResultSet könnte keine Zeile enthalten
18.     System.out.println("Error examining ResultSet\n"+
          exc.getMessage());
19.   }
20. }
```

Bei einem neu generierten ResultSet steht der Zeiger für den aktuellen Datensatz vor dem ersten möglichen Datensatz. Um an den ersten Datensatz zu gelangen muß man den Zeiger um einen Datensatz vor bewegen. Dazu wird die Methode `ResultSet.next()` aufgerufen.
Die Methode `ResultSet.next()` gibt einen boolschen Wert zurück, der besagt, ob es noch einen nächsten Datensatz gibt.
Ist das ResultSet leer, also kein einziger Datensatz vorhanden, so wirft die Methode `ResultSet.next()` eine `SQLException` aus.
Damit ist es möglich, sich durch ein ResultSet von vorne nach hinten zu bewegen.
Im nächsten Schritt werden nun die Daten der einzelnen Datensätze extrahiert. Dafür besitzt ein ResultSet die Methoden:

- `String getString(int spaltenindex)`
- `int getInt(int spaltenindex)`
- `boolean getBoolean(int spaltenindex)`
- `Object getObject(int spaltenindex)`
- `...`

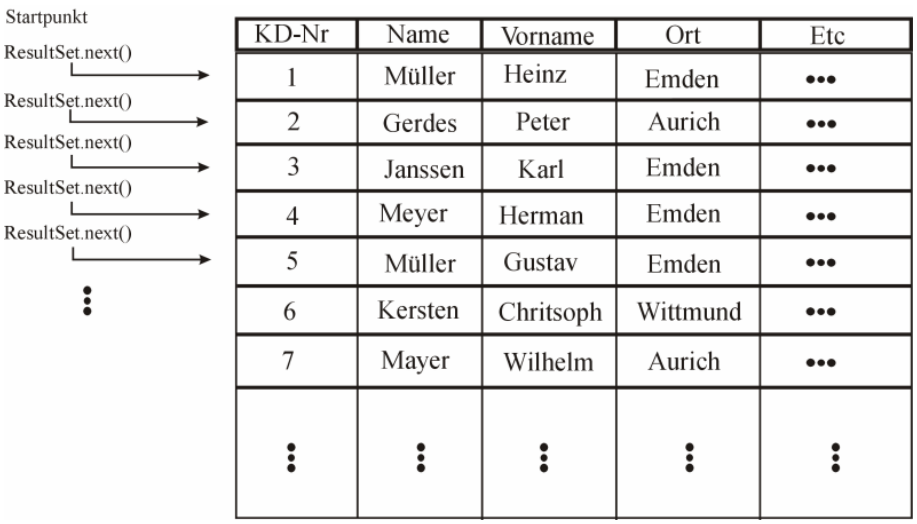


Abbildung 11: Ablauf der Datensatzauswahl

(Detailierte Informationen finden Sie in der API unter: `java.sql.ResultSet`)

Die einzelnen Attribute des ResultSets werden bei **1** beginnend durchnummeriert. Werden unterschiedliche Datentypen erwartet, so gibt es die Möglichkeit die Methode `ResultSet.getObject(index)` zu verwenden.

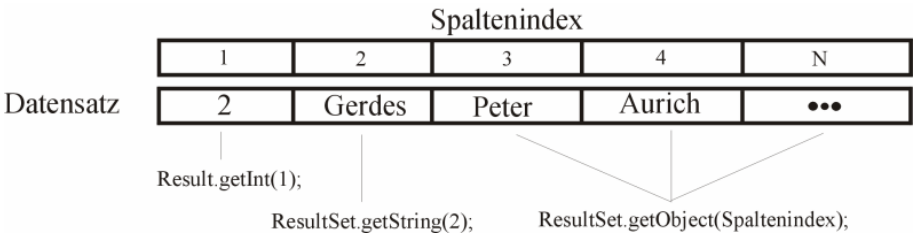


Abbildung 12: Auszug aus einem Datensatz

Handelt es sich aber um bekannte Datentypen, so empfiehlt es sich, die entsprechenden Methoden für diesen Datentyp zu verwenden, da diese wesentlich effizienter arbeiten. z.B. für den Datentyp VARCHAR (String) die Methode `ResultSet.getString(index)`, für den Datentyp boolean die Methode `ResultSet.getBoolean(index)`, ...
Ein aus dem ResultSet gelesenes Object kann auch null sein. Ein ResultSet muß nach Gebrauch geschlossen werden. Hierzu dient die Methode `ResultSet.close()`.

Ein kleines Beispielprogramm zum Aufbau einer Verbindung, Ausführen einer SQL-Anweisung, Auswerten eines ResultSets und Schließen der Verbindung, ist im Anhang im Kapitel 2 auf Seite 52 zu finden.

10 Spezielle Anbindung an die Beispiel Datenbank

10.1 Konzeptionelle Überlegung

Grundlegend für die Erstellung einer Datenbank und der Planung einer Objektorientierten Anwendung ist das Fachkonzept. In einem Fachkonzept wird nur der Zusammenhang zwischen Objekten der realen Welt, die modelliert werden soll, betrachtet. Durch das Erstellen eines ER-Modells wird die reale Datenwelt auf eine Datenbank abgebildet. Nach der Normalisierung (die Umsetzung auf ein relationales Schema) werden die Objekte der realen Welt oft auf mehrere Tabellen verteilt. In einer Java-Klasse lassen sich jedoch ohne weiteres Wiederholungsgruppen darstellen, so daß die Umsetzung auf ein Klassenkonzept doch leicht von den Entities des ER-Modells abweichen kann.
Ziel einer Datenbankanbindung ist es, Objekte einer Klasse in der Datenbank ablegen zu können und bei Bedarf wiederherzustellen. Die Wiederherstellung der Objekte kann anhand von bestimmten Kriterien erfolgen und sich dabei alle Möglichkeiten der Sprache SQL zunutze machen. Eine einfache Lösung ist es, für jede Klasse des Klassenkonzepts, eine zusätzliche Klasse zu schreiben, die Methoden zum Lesen und Schreiben von Exemplaren dieser Klasse enthält. Eine solche Klasse bezeichnen wir im Folgenden als Broker-Klasse.

10.2 Klassendiagramm

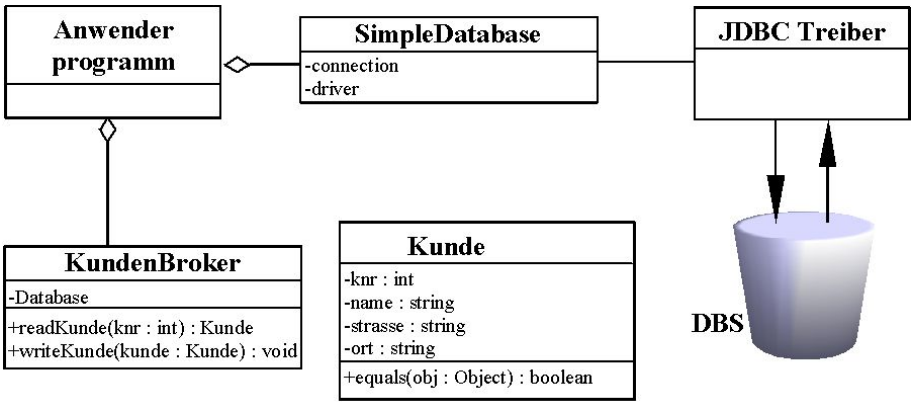


Abbildung 13: Das Klassendiagramm des Brokers.

10.3 Beispiel einer Broker-Klasse

Um das Beispiel so klein (und verständlich) wie möglich zu halten, wird in den weiteren Kapiteln eine Broker-Klasse für einen Kunden (Fachkonzept) erstellt. Diese Klasse nennen wir KundenBroker.
Ein Broker muß auf jeden Fall über eine Verbindung zur Datenbank verfügen. Dazu verwenden wir in dem Beispiel die Klasse SimpleDatabase, die in den vorherigen Kapiteln vorgestellt wurde. Wir verzichten darauf, daß der Broker

die Verbindung selbst aufbaut und übergeben eine bereits etablierte Verbindung beim Konstruktoraufruf. Die Kommunikation mit der Datenbank geschieht über SQL. Da nur einige spezielle SQL-Befehle verwendet werden, benutzen wir PreparedStatements. Ein PreparedStatement repräsentiert einen SQL-Ausdruck für den noch variable Parameter gesetzt werden können. Solche PreparedStatements müssen jedoch über die Datenbankverbindung vorkompiliert werden. Dies geschieht im Konstruktor des Brokers.

Hier die im Beispiel verwendeten SQL-Befehle:

1. `String strKundeByKnr = ''SELECT kunde.nr, kunde.name, kunde.strasse, kunde.ort FROM kunde WHERE kunde.nr = ? ;'';`

   Diese Anweisung wird in dem PreparedStatement preKundeByKnr vorkompiliert.

2. `String strUpdateKundeByKnr = ''UPDATE kunde SET kunde.name = ? , kunde.strasse = ? , kunde.ort = ? WHERE knr = ? ;'';`

   Diese Anweisung wird in dem PreparedStatement preUpdateKundeByKnr vorkompiliert.

Die Fragezeichen in dem SQL-Ausdruck wirken ungewohnt. Sie dienen als Platzhalter für die Parameter, die während des laufenden Programms geändert werden können. Es empfiehlt sich, die Werte direkt vor dem Ausführen der SQL-Anweisung zu setzen, da beim Ausführen der Anweisung sichergestellt sein muß, daß alle Parameter gesetzt wurden. Wie das geschieht wird bei der Beschreibung der Methoden erklärt. Zwei einfache Methoden zeigen die Funktionsweise der PreparedStatements.

- `void writeKunde(Kunde kunde)`
- `Kunde readKunde(int knr)`

10.3.1 Der Konstruktor

Der Konstruktor sieht dann wie folgend aus:

```
1  public KundenBroker(Database db) throws SQLException,
   NotConnectedException {
2    // Referenz auf die Datenbank merken
3    this.db = db;
4
5    // Die SQL-Anweisungen vorkompilieren
6    this.preKundeByKnr = db.precompile(strKundeByKnr);
7    this.preUpdateKundeByKnr = db.precompile(strUpdateKundeByKnr);
8  }
```

Die angegebenen Fehlerzustände (Exceptions) können beim vorkompilieren auftreten und werden an die Methode, die den Broker erzeugt weitergereicht.

10.3.2 Die Methode `writeKunde`

Diese Methode schreibt ein Exemplar der Klasse Kunde in die Datenbank. Ein Kunde wird durch seine Kunden Nummer (Knr) eindeutig identifiziert. In diesem Beispiel ist es nur möglich einen bereits bestehenden Kunden Datensatz wieder zurück in die Datenbank zu schreiben.

```
1  public void writeKunde(Kunde kunde) throws SQLException,
   NotConnectedException {
2    preUpdateKundeByKnr.clearParameters();
3    preUpdateKundeByKnr.setString(1, kunde.getName());
4    preUpdateKundeByKnr.setString(2, kunde.getStrasse());
5    preUpdateKundeByKnr.setString(3, kunde.getOrt());
6    preUpdateKundeByKnr.setInt(4, kunde.getKnr());
8    preUpdateKundeByKnr.executeUpdate();
9    db.commit();
10 }
```

Abbildung 14: Quelltext der Methode `writeKunde(Kunde kunde)`

10.3.3 Die Methode `readKunde`

Diese Methode erzeugt ein Objekt vom Typ Kunde indem es die Kundendaten (Attribute) aus der Datenbank einliest. Auch hier wird der Kunde eindeutig über das Attribut Kunden Nummer (knr) identifiziert.

```
1  public Kunde readKunde(int knr) throws SQLException {
2    Kunde returnKunde = null;
3    preKundeByKnr.clearParameters();
4    preKundeByKnr.setInt(1, knr);
5    ResultSet result = preKundeByKnr.executeQuery();
6    if (result.next()) {
7      returnKunde = new Kunde();
8      returnKunde.setKnr(result.getInt(1));
9      returnKunde.setName(result.getString(2));
10     returnKunde.setStrasse(result.getString(3));
11     returnKunde.setOrt(result.getString(4));
12   }
13   result.close();
14   return returnKunde;
15 }
```

Abbildung 15: Quelltext der Methode `readKunde(int knr)`

11 Abbildung von Objekten in relationalen Datenbanken

In diesem Abschnitt werden Konzepte zur allgemeinen Umsetzung von Java-Objekten auf relationale Datenbanken, sowie Probleme die bei der Umsetzung auftreten, erläutert. Unter der allgemeinen Umsetzung verstehen wir das Speichern und Restaurieren von konkreten Java-Objekten.
Die wichtigste Überlegung ist, welche Informationen über ein Objekt überhaupt gespeichert werden müssen, damit es sich wiederherstellen läßt. Zu diesen Informationen gehören mit Sicherheit die Attribute der Klasse, sowie der Klassenname. Betrachten wir zunächst die Umsetzung einer sehr einfachen Klasse.

11.1 Ein einfaches Beispiel

Um den prinzipiellen Ablauf darzustellen beschränken wir uns bei dem Beispiel auf eine Klasse mit einem primitiven Datentyp als einziges Attribut.

```
1  public class Entity {
2    public int i = 5;
3  }
```

Soll ein Exemplar der Klasse Entity in einer relationalen Datenbank gespeichert werden, so wird in der Datenbank eine Tabelle mit dem Namen der Klasse (Entity) erzeugt und in dieser Tabelle ein Attribut (i=5), vom Typ Integer, eingetragen. Beim Wiederherstellen eines speziellen Objektes muß eine neue Instanz der Klasse Entity angelegt werden und deren Attribut i auf den Wert 5 gesetzt werden. Da das Attribut public ist, stellt dies kein weiteres Problem dar. Auf das Problem der Zugriffsrestriktion werden wir später noch eingehen.
Bis jetzt ist es nur möglich ein Objekt anhand seines Attributes i zu identifizieren. Dies ließe zwar zu Objekte auf Gleichheit zu prüfen, nicht aber auf die Identität. Desweiteren ist in der Tabelle Entity noch kein Primärschlüssel definiert, der einen Datensatz eindeutig identifiziert. Wir können also noch kein konkretes Objekt wiederherstellen.

11.2 Die Identität von Objekten

In Java wird die Identität von Objekten über deren Referenzen geprüft. Verweisen zwei Referenzen (x und y) auf das selbe Objekt (x == y) so sind die Objekte identisch. Dies ist auch die Funktionalität der Methode `boolean Object.equals(Object o)`. In einer relationalen Datenbank sind hingegen zwei Datensätze identisch, wenn sie den gleichen Primärschlüssel besitzen.
Die Schwierigkeit besteht nun darin die Referenzen auf den Primärschlüssel abzubilden. Dies ist nicht unmittelbar möglich, da Referenzen nur zur Laufzeit des Java-Programmes existieren und somit nicht persistent sind. Das heißt: Wenn man die Referenz eines Objektes in einer Datenbank speichert, so besteht keine Möglichkeit nach einem Neustart des Java-Programmes diese Referenz wiederherzustellen.
Um trotzdem die Möglichkeit einer Identitätsprüfung zu haben, erweitern wir unsere Java-Klasse Entity um ein Attribut. Dieses Attribut dient dazu ein Objekt der Klasse eindeutig zu identifizieren. Wir nennen das Attribut OID (Object Identification Number).

```
1  public class Entity {
2    private byte[] OID = theFactory.getUniqueOID();
3    public int i = 5;
4  }
```

Die OID wird in der Datenbank als Primärschlüssel für jede Tabelle verwendet. Zwar würde es prinzipiell reichen zu garantieren, daß die OID innerhalb einer Klasse (Tabelle) eindeutig ist, es empfiehlt sich jedoch sie über die ganze Datenbank hinweg eindeutig zu halten. Das bedeutet: Jedes Objekt wird durch einen Datensatz in der Datenbank repräsentiert, der über eine datenbankweit eindeutige OID identifiziert wird.
Beim Erzeugen neuer Objekte ist nun dafür zu sorgen, daß dem Objekt eine neue, noch nicht vorhandene, OID zugeordnet wird. Dies läßt sich mit einer Objekt-Fabrik realisieren. Ein Fabrik-Muster arbeitet wie folgend: Es wird sichergestellt, daß von der Klasse (in unserem Beispiel Entity) keine Exemplare direkt erstellt werden können (Realisierung z.B über eine innere Klasse). Exemplare der Klasse Entity lassen sich nur von einer Fabrik-Klasse (factory) über einen Methodenaufruf erzeugen. Die für die Erzeugung zuständige Klasse stellt

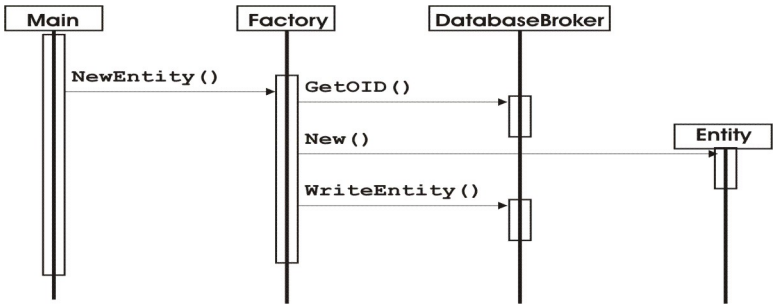


Abbildung 16: Ablaufdiagramm für Anwendungsfall Objekt neu erzeugen

sicher, daß keine OIDs vergeben werden, die in der Datenbank schon verwendet sind.
Damit innerhalb des Java-Programms keine Redundanzen, z.B durch mehrmaliges Wiederherstellen eines Objektes, entstehen. Ist es notwendig, das im Java-Programm immer nur ein Exemplar mit der gleichen OID existiert. Dies läßt sich über ein abgewandeltes Singleton-Muster verwirklichen. Unter einem Singleton-Muster wird normalerweise verstanden, daß sichergestellt wird, daß nur ein Exemplar der Klasse existieren kann. In unserem Fall müßten wir die Einschränkung ausweiten, so daß nur Exemplare mit verschiedenen OIDs existieren können.
Sinnvoll ist es mit Sicherheit dieses mit einem Proxymuster zu verbinden, damit bei Anfragen nach Objekten, die bereits im Java-Programm wiederhergestellt wurden, nur die Referenz auf das bereits wiederhergestellte Objekt zurückgeliefert wird. Es ist dafür zu sorgen, daß die Klasse, die das Proxymuster verwirklicht, darauf achtet die, vom Java-Programm nicht mehr referenzierten Objekte, selbst auch zu dereferenzieren, damit der Mechanismus der Garbage-Collection anlaufen kann.

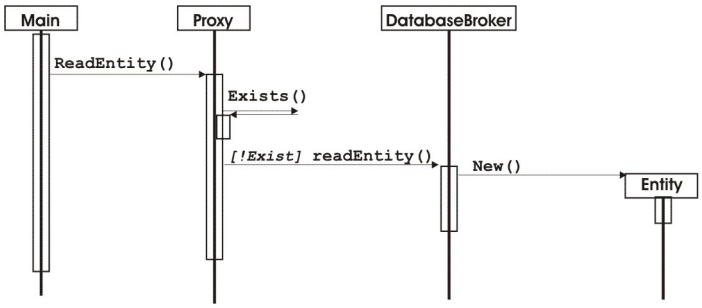


Abbildung 17: Ablaufdiagramm für Anwendungsfall Objekt lesen

11.3 Ein Objekt als Attribut

In dem einfachen Beispiel besteht die Klasse Entity ausschließlich aus Attributen primitiven Typs. Wie wird ein Attribut in der Datenbank abgebildet, das selbst wieder ein Objekt ist?

```
1  public class Entity {
2    public byte[] OID = theFactory.getUniqueOID();
3    public int i = 5;
4    public Object obj = new AnyClass();
5  }
```

Dafür wird für die Klasse des Objekts eine neue Tabelle angelegt, falls diese nicht schon existiert. Über die entsprechende Fabrikmethode wird ein neuer Datensatz mit einer neuen OID angelegt. In der aggregierenden Klasse wird, für das Attribut, die OID des Attribut-Objekts eingetragen. Auf diese Weise können Objekte auch von mehreren anderen Objekten aggregiert werden, ohne das redundante Informationen gespeichert werden müssen.

11.4 Abbildung der Kapselung

Ein weiteres Problem der Wiederherstellung ergibt sich durch das Prinzip der Kapselung. Dieses besagt, daß es Attribute und Methoden gibt, die nicht öffentlich zugänglich sind. Die Kapselung wird unter Java durch die Schlüsselworte `private` (nur innerhalb der Klasse) und `protected` (innerhalb des Paketes oder in Subklassen) angegeben, die Standardbesetzung ist das Methoden und Attribute nur innerhalb des selben Paketes sichtbar sind.

```
1  public class Entity {
2    private byte[] OID = theFactory.getUniqueOID();
3    protected int i = 5;
4    Object obj = new AnyClass();
5  }
```

Für die Klasse, welche die Wiederherstellung der Objekte übernimmt, bedeutet dies, daß sie nur die Attribute wiederherstellen kann, die explizit als public (öffentlich - für alle zugreifbar) deklariert sind. Da ein Objekt ohne gekapselte

Attribute nicht brauchbar ist, muß es für die Rekonstruktion eines Objektes eine Möglichkeit geben die Kapselung zu umgehen.
Wahrscheinlich bietet sich über den Mechanismus der Reflexion diese Möglichkeit. Die Serialisation von Objekten sowie bestimmte Debugging-Möglichkeiten stoßen auf ähnliche Anforderungen.

11.5 Einschätzung der Möglichkeit eine allgemeine Anbindung zu implementieren

Die oben ausgeführten Ansätze sind nur Konzepte. Die Umsetzung dieser Konzepte wird auf eine ganze Reihe von Problemen stoßen. Als Beispiele seien hier nur die Zählung der Referenzen für das Singleton-Proxy-Muster und die Schwierigkeit die Kapselung zu umgehen genannt. Prinzipiell scheint es allerdings Lösungen für diese Probleme zu geben, so daß es möglich sein müsste, ein Packet zu entwickeln, welches Objekte in relationalen Datenbanken abbilden kann. Allerdings ist dafür einiges an Forschungsarbeit zu leisten und der Aufwand für ein solches Projekt nur schwer einzuschätzen.

A Codebeispiel für den Umgang mit SimpleDatabase

Ein kleines Codebeispiel unter Verwendung des Package `de.web.databasesupport.tools.database` und der Datenbank "`auftragsverwaltung`"[2]:

```
1.  import de.web.databasesupport.tools.database.*;
2.  import java.sql.*;
3.
4.  class ExampleDatabaseUsage   {
5.
6.    ResultSet result=null;
7.    Database db;
8.    String sqlStatement = "SELECT * FROM kunde WHERE nr < 100 ;";
9.    String username = "admin";
10.   String password = "test";
11.   String dbURL =
        "jdbc:JDataConnect://127.0.0.1/auftragsverwaltung";
12.   String driver = SimpleDatabase.JDATA2_0_DRIVER;
13.
14.   public ExampleDatabaseUsage() {
15.     openDatabase();
16.     result = executeSQL();
17.     showResult(result);
18.     closeDatabase();
19.   }
20.
21.   public static void main(String[] args) {
22.     ExampleDatabaseUsage exDBUse = new ExampleDatabaseUsage();
23.   }
24.
25.   public void openDatabase() {
26.     // Anlegen einer neuen Datenbank Connection
27.     // benötigt wird hierfür der Treiber, die URL der Datenbank,
28.     // sowie Username und Password
29.     db = new SimpleDatabase(driver, dbURL, username, password);
30.     try {
31.       // versuchen die Datenbank-Connection zu öffnen
32.       db.open();
33.     } catch (Exception exc) {
34.       // Falls Datenbank nicht geöffnet werden konnte
35.       // (z.Bsp. falscher Treiber, falsche URL, etc.)
36.       System.err.println(
            "Error openening the Database. Programm terminated.");
37.       exc.printStackTrace();
38.       System.exit(0);
```

[2]Nähere Informationen zur Beispieldatenbank "auftragsverwaltung" finden Sie im Kapitel 3 auf Seite 23

```
39.     }
40.   }
41.
42.   public ResultSet executeSQL() {
43.     ResultSet rv = null;
44.     try {
45.       // SQL-Abfrage ausführen. Liefert ein ResultSet zurück.
46.       rv = db.executeQuery(sqlStatement);
47.     } catch (SQLException exc) {
48.       System.err.println("Error executing SQL-Statement. " +
            exc.getMessage());
49.     } catch (NotConnectedException exc) {
50.       System.err.println(
            "Could not execute Statement. Database is closed.");
51.     }
52.     return rv;
53.   }
54.
55.   public void showResult(ResultSet result) {
56.     int counter=0;
57.     System.out.println("ResultSet = ");
58.     try {
59.       // Solange es eine neue Zeile im ResultSet gibt
60.       while (result.next()) {
61.         counter++;
62.         System.out.println("****************************************");
63.         System.out.println("* SET No. "+counter);
64.         System.out.println("****************************************");
65.         // Anzahl der Columns aus dem ResultSet herausziehen
66.         int i = 1;
67.         // Namen der Columns aus dem ResultSet herausziehen
68.         String[] names = db.getColumnNames(result);
69.         for (i=1; i<=db.getColumnCount(result); i++)   {
71.           // result.getObject(i) holt das Object
72.           // aus der i-ten Column (allgemeines Object)
73.           // bei festen Typen kann auch getString()... verwendet werden
74.           System.out.print("Col-No.: "+ i +" = "+ names[i-1]);
75.           System.out.println(" = "+ result.getObject(i));
80.         }
81.       }
82.     } catch (SQLException exc) {
83.       // ResultSet könnte keine Zeile enthalten
84.       System.out.println("Error examining ResultSet\n"+ exc.getMessage());
85.     }
86.   }
87.
88.   public void closeDatabase() {
89.     try {
90.       // versuchen die Datenbank-Connection zu schließen
91.       db.close();
```

```
92.     } catch (SQLException exc) {
93.       System.err.println("Error closing the Database.");
94.       exc.printStackTrace();
95      } catch (NotConnectedException exc) {
96        System.err.println("Database already closed.");
97      }
98    }
99.}
```

B Codebeispiel für eine Broker-Klasse

B.1 Quelltext `Kunde.java`

```
/** This class represents a Kunde. */
public class Kunde {

  /** The knr (primary key - see concept) of the Kunde. */
  private int knr = 0;

  /** The name property of the Kunde */
  private String name = null;

  /** The strasse property of the Kunde */
  private String strasse = null;

  /** The ort property f the Kunde */
  private String ort = null;

  /** Constructor */
  public Kunde() {
  }

  /** Constructor.
    * @param knr The knr property of the Kunde.
    * @param name The name property of the Kunde.
    * @param strasse The strasse property of the Kunde.
    * @param ort The ort property of the Kunde.
    */
  public Kunde(int knr, String name, String strasse, String ort) {
    setKnr(knr);
    setName(name);
    setStrasse(strasse);
    setOrt(ort);
  }

  /** Sets the knr property.
    * @param knr The new knr.
    */
  public void setKnr(int knr) {
    this.knr = knr;
  }

  /** Sets the name property.
    * @param name The new name.
    */
  public void setName(String name) {
    this.name = name;
  }
```

```
/** Sets the strasse property.
  * @param strasse The new strasse.
  */
public void setStrasse(String strasse) {
  this.strasse = strasse;
}

/** Sets the ort property.
  * @param ort The new ort.
  */
public void setOrt(String ort) {
  this.ort = ort;
}

/** Returns the knr property.
  * @return The knr.
  */
public int getKnr() {
  return this.knr;
}

/** Returns the name property.
  * @return The name.
  */
public String getName() {
  return this.name;
}

/** Returns the strasse property.
  * @return The strasse.
  */
public String getStrasse() {
  return this.strasse;
}

/** Returns the ort property.
  * @return The ort.
  */
public String getOrt() {
  return this.ort;
}

/** Overwrites equals in class Object.
  * A Kunde is compared by its' knr property.
  * kunde1.equals(kunde2) is equivalent to kunde1.getKnr() == kunde2.getKnr()
  * @param obj The Kunde to compare with.
  * @returns True if both Kunde objects have the same knr.
  */
public boolean equals(Object obj) {
  if ((obj instanceof Kunde) && (this.knr == ((Kunde) obj).knr)) {
```

```
      return true;
    }
    return false;
  }
}
```

B.2 Quelltext KundenBroker.java

```
import de.web.databasesupport.tools.database.*;
import java.sql.*;

/** The KundenBroker class is responsible for reading and
    writing Kunde objects
  * from and to the database. This class holds a reference
    to a Database object
  * which represents the database connection. The connection
    have to be established
  * when instanciating this class, because it uses PreparedStatements.
  * A NotConnectedException will be thrown if the connection
    is not established.
  */
public class KundenBroker {

  /** String containing the SQL statement used to
      read a Kunde object. */
  private final static String strKundeByKnr =
  "SELECT kunde.nr, kunde.name, kunde.strasse," +
  " kunde.ort FROM kunde WHERE kunde.nr = ? ; ";

  /** String containing the SQL statement used to
      write a Kunde object. */
    private final static String strUpdateKundeByKnr =
    "UPDATE kunde SET kunde.name = ? , kunde.strasse = ? ," +
    " kunde.ort = ? WHERE knr = ? ; ";

  /** The Database object representing the connection. */
  private Database db = null;

  /** The PreparedStatement used to read a Kunde object. */
  private PreparedStatement preKundeByKnr = null;

  /** The PreparedStatement used to write a Kunde object. */
  private PreparedStatement preUpdateKundeByKnr = null;

  /** Constructor.
    * Ensure that the Database connection is established when
    * instanciating this class.
    * @param db The Database object representing the connection.
    */
  public KundenBroker(Database db) throws SQLException,
  NotConnectedException {
    this.db = db;
    this.preKundeByKnr = db.precompile(strKundeByKnr);
    this.preUpdateKundeByKnr = db.precompile(strUpdateKundeByKnr);
  }
```

```
  /** Writes a Kunde object to the database.
    * A Kunde object is identified by the Kunde.knr property.
      The name, strasse, ort
    * property is updated.
    * @param kunde The Kunde object to write to the database.
    * @throw SQLException If there is no Kunde with the specified knr
    *   or the used SQLStatement (strUpdateKundeByKnr) was wrong.
    * @throw NotConnectedException If the Database connection
      is not established.
    */
  public void writeKunde(Kunde kunde) throws SQLException,
  NotConnectedException {
    preUpdateKundeByKnr.clearParameters();
    preUpdateKundeByKnr.setString(1, kunde.getName());
    preUpdateKundeByKnr.setString(2, kunde.getStrasse());
    preUpdateKundeByKnr.setString(3, kunde.getOrt());
    preUpdateKundeByKnr.setInt(4, kunde.getKnr());
    preUpdateKundeByKnr.executeUpdate();
    db.commit();
  }

  /** Reads a Kunde object from the database.
    * @param knr The Knr property of the Kunde to retrieve.
    * @return The Kunde. null if no Kunde was found.
    * @throw SQLException if the used SQLStatement (strKundeByKnr)
      was wrong.
    */
  public Kunde readKunde(int knr) throws SQLException {
    Kunde returnKunde = null;
    preKundeByKnr.clearParameters();
    preKundeByKnr.setInt(1, knr);
    ResultSet result = preKundeByKnr.executeQuery();
    if (result.next()) {
      returnKunde = new Kunde();
      returnKunde.setKnr(result.getInt(1));
      returnKunde.setName(result.getString(2));
      returnKunde.setStrasse(result.getString(3));
      returnKunde.setOrt(result.getString(4));
    }
    result.close();
    return returnKunde;
  }

}
```

B.3 Quellcode KundeTest.java

```
import de.web.databasesupport.tools.database.*;
import java.sql.*;

/** Simple Tester class for Kunde and KundeBroker. */
public class KundeTester {
  Database db = null;
  KundenBroker broker = null;

  public KundeTester() throws Exception {
    db = new SimpleDatabase(SimpleDatabase.JDBC_ODBC_DRIVER,
      "jdbc:odbc:auftragsverwaltung");
    db.open();
    broker = new KundenBroker(db);
  }

  public static void main(String[] args) {
    try {
      KundeTester test = new KundeTester();
      Kunde k = test.broker.readKunde(2);
      System.out.println("knr = " + k.getKnr() +
        "\nname = " + k.getName());
      test.db.close();
    } catch (Exception exc) {
      exc.printStackTrace();
    }
  }
}
```

C Verwendete Warenzeichen und Disclaimer

Die Informationen in diesem Artikel wurden von uns sorgfältig aufgearbeitet. Dennoch können Fehler nicht vollständig ausgeschlossen werden. Die Autoren übernehmen keine juristische Verantwortung oder irgendeine Haftung für eventuell verbliebene Fehler und deren Folgen.
Alle Warennamen werden ohne Gewährleistung der freien Verwendbarkeit benutzt und sind möglicherweise eingetragene Warenzeichen.

Abbildungsverzeichnis