1 Zugriffsschnittstellen

Dieses Kapitel führt in drei Zugriffsschnitstellen auf Datenbanken und persistente Objektspeicher ein. Die Darstellung skizziert daher zunächst die auf das relationale Speicherungs- und Zugriffsparadigma ausgelegte JDBC-Schnittstelle.
Davon ausgehend wird die Kapselung von JDBC-basierter Persistenzlogik durch Enterprise Java Beans entwickelt. Hierbei steht die Ausprägungsform der bean managed persistence im Vordergrund, da sie die weitestgehenden Eingriffsmöglichkeiten für den Programmierer bietet. Abschließend wird mit den *Java Data Objects* ein jüngerer Ansatz zur Realisierung transparenter Speicherung eingeführt, der gleichzeitig verschiedenste Persistenzdienstleiter unterstützt.

1.1 Java Database Connectivity

Motivation

Häufig besteht der Wunsch oder die Notwendigkeit auf bereits vorliegende Datenbestände, die durch ein Datenbankmanagementsystem (DBMS) verwaltet werden, in einer Applikationsprogrammiersprache zuzugreifen. Dabei soll die Anbindung der benötigten Datenquelle nicht problemspezifisch wieder und wieder neu entwickelt werden, sondern sollte sich auf ähnliche Datenanbindungsprobleme übertragen lassen.
Vor diesem Hintergrund liegt es nahe sich an den Typen der verfügbaren und kommerziell bedeutsamen DBMS zu orientieren und herstellerspezifische Entwicklungen außer Acht zu lassen. Gleichzeitig offenbaren sich hierbei Standardisierungsbemühungen wie die Sprache *SQL* zum Zugriff auf relationale DBMS als lohnenswerter Ansatz der Etablierung einer generischen und übertragbaren Schnittstelle.

Die Idee zur Schaffung einer solchen generischen Schnittstelle für den Zugriff auf relationale DBMS geht zurück auf eine Initiative der *SQL Access Group*, welche später in der Vereinigung mit der *X/Open* Group aufging, die zwischenzeitlich in *Open Group* umbenannt wurde. Das dort konzipierte programmiersprachenunabhängige *SQL Call Level Interface* (SQL/CLI) konnte sich dank der Umsetzung unter dem Namen Open Database Connectivity (ODBC) durch die Firma Microsoft und die parallel erfolgte internationale Normierung unter dem Titel *SQL/CLI* breit am Markt etablieren.

Die für die Programmiersprache Java adaptierte Variante des Zugriffs auf relationale DBMS wird durch SUN Microsystems unter dem Namen *Java Database Connectivity* (JDBC) propagiert und

stellt eine auf ODBC konzeptionell aufbauende und auf die spezifischen Bedürfnisse dieser Applikationsprogrammiersprache optimierte Untermenge des SQL/CLI-Standards dar.

Konzept und Grundidee

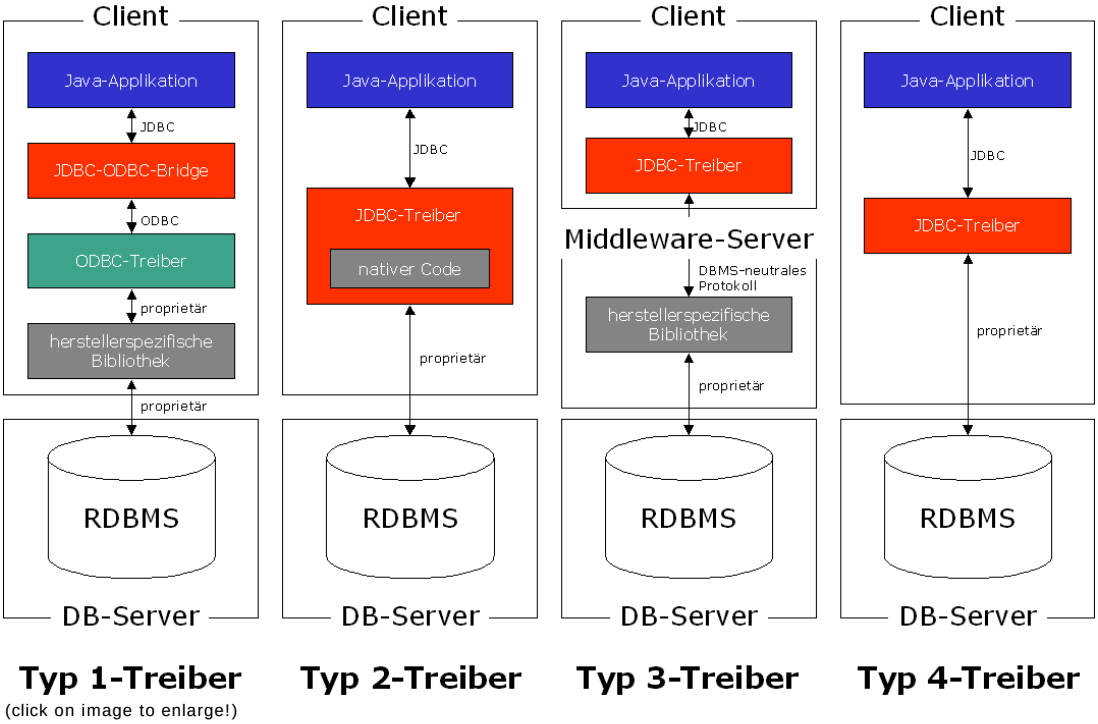
Von den Vorgängeransätzen übernommene Grundidee der Schnittstelle ist es den physischen Zugriff auf das Datenbankmanagementsystem durch eine von der Applikation spearierte wiederverwendbare Softwarekomponente, den sog. *JDBC-Treiber*, abzuwickeln.

Dieser Treiber vermittelt zwischen der Javaapplikation und dem verwendeten DBMS. Hierbei muß für jedes DBMS ein auf es abgestimmter JDBC-Treiber verwendet werden, da lediglich die Schnittstelle zur Applikation, nicht jedoch die zum DBMS, standardisiert ist.

Diesem Treiber obliegt die Abwicklung der gesamten Kommunikationsvorgänge mit dem DBMS. Er setzt jedoch selbst keine datenbankspezifischen Funktionalitäten, wie Syntax- oder Plausibilitätsprüfungen der übermittelten Kommandos um. Etwaige Fehlerprüfungen können, ebenso wie Anfrageoptimierungen, daher erst seitens des DBMS vorgenommen werden.
Der Vorteil dieses Vorgehens liegt in der Generizität des JDBC-Treibers. Er kann ohne aufwendige Logikanteile als reine uninterpretierende Vermittlungsschicht zwischen Applikation und DBMS umgesetzt werden, wodurch schlanke Implementierungen ermöglicht werden.

Die JDBC-Spezifikation detailliert den Treiberbegriff zusätzlich hinsichtlich der gewählten technischen Umsetzung aus. So werden die vier in Abbildung 1 dargestellten Treibertypen gemäß ihrer Charakteristika beschrieben und unterschieden.

Abbildung 1: *JDBC-Treibertypen*



(click on image to enlarge!)

Die historisch älteste Variante bildet der **Typ 1 Treiber**. Strenggenommen verkörpert er selbst keinen Datenbanktreiber, sondern lediglich eine Umsetzungsschicht die einem existierenden ODBC-Treiber vorgeschaltet wird.
Die Abbildung belegt diesen Treibertyp daher mit dem Begriff *JDBC-ODBC-Bridge*, da er lediglich den Brückenschlag zwischen den beiden Standards vornimmt und sich in der konkreten Anwendung auf die Umsetzung der zwischen den beiden Protokollen beschränkt, ohne realen Zugriff auf die Datenbank zu erhalten.
Dieser ist dem ODBC-Treiber vorbehalten, der im allgemeinen Falle mit einer weiteren Umsetzungsstufe kommuniziert, welche die generischen ODBC-Aufrufe in konkrete DBMS-spezifische wandelt.
Während sowohl der JDBC-ODBC-Brückentreiber als auch der ODBC-Treiber selbst für verschiedene DBMS verwendet werden können, muß für jedes konkrete DBMS eine herstellerspezifische, d.h. an das verwendete DBMS angepaßte, Bibliothek vorliegen.

Für den Fall eines **Typ 2 Treiber**s entfällt diese durch ODBC geschaffene zusätzliche Indirektionsstufe zugunsten der Adaption der Konversionskomponente, welcher die Wandlung der Aufrufe in das DBMS-native Protokoll obliegt, an das JDBC-Protokoll und ihrer Integration in den JDBC-Treiber selbst.

Die Natur der Kommunikation des Java-Anteils des Treibers mit den Nativen ist im Rahmen der durch die JDBC-Spezifikation gegebenen Definition nicht festgelegt.
Durch die integration der DBMS-nativen Treiberanteile in den JDBC-Treiber muß dieser für jedes anzusprechende DBMS neu erstellt werden. Eine Wiederverwendung der JDBC-spezifischen Anteile die für die Clientkommunikation eingesetzt werden kann hierbei nicht erfolgen.

Der Fall der (partiellen) Konkretisierung dieser Kommunikationsbeziehung zu einem beliebigen *DBMS-neutralen Protokoll* wird durch einen **Typ 3 Treiber** aufgegriffen.
Hier wird die DBMS-spezifische Komponente (in der Abbildung grau dargestellt) als vom JDBC-Treiber separiertes Modul aufgefaßt, daß mit diesem mittels eines festgelegten neutralen Protokolls kommuniziert.
Durch diese Separierung, die auch durch Installation auf physisch getrennten Maschinen --- der DBMS-spezifische Anteil könnte beispielsweise auf einem Middleware-Server untergebracht werden --- fundiert werden kann, gelingt die Wiederverwendung des JDBC-Treiberanteils, der mit verschiedenen DBMS-spezifischen Bibliotheken über das gewählte Protokoll kommunizieren kann.

Der **Typ 4 Treiber** stellt die letzte durch die JDBC-Spezifikation vorgesehene Ausprägung dar. Er konzipiert eine vollständig in Java implementierte Zugriffsschicht, die in sich geschlossen ist. Sie besitzt daher lediglich die notwendige JDBC-Schnittstelle zur Kommunikation mit der Java-Applikation und eine DBMS-Spezifische zum Zugriff auf die Datenquelle.
Die Vorteile dieser Architekturvariante liegen in ihrer Portabilität und den geringen Installations und Wartungsaufwänden, die aus der Reduktion der Kommunikationsbeziehungen resultieren. So kann ein solcher Treiber durch einfache Integration in die Java-Applikation verwendet werden und bedarf keiner Installationen oder Modifikationen an der verwendeten Ausführungsumgebung.
Gleichzeitig offenbart sich diese Lösung jedoch als technisch aufwendig in der Umsetzung, sobald DBMS verschiedener Hersteller angesprochen werden sollen, da die JDBC-Anteile des Treibers nicht separat wiederverwendet werden können.

Hinsichtlich des Laufzeitverhaltens zeigt sich deutlich die Schwäche der Typ 1 Treiber, welche in der inhärent notwendigen Doppelkonversion (JDBC zu ODBC und ODBC zu nativem Aufruf) begründet liegt. Daher sind Treiber dieses Typs als Übergangserscheinung hin zu „echten“ JDBC-Treibern, d.h. Treibern der restlichen Typen, anzusehen und sollten in Produktivumgebungen nicht eingesetzt werden.
Die Vorteile der Typ 2 und 3 Treiber seitens der Ausführungsgeschwindigkeit liegen in den nativen Codeanteilenbegründet, welche für das jeweilige verwendete DBMS optimiert werden können.
Zwar spricht der leichte Installations- und Adminstrationsaufwand eindeutig für Typ 4 Treiber, jedoch fallen diese in ihrer Leistungsfähigkeit durch die ausschließliche Verwendung der Programmiersprache Java teilweise deutlich hinter Treiber des Typs 2 und 3, mit unter sogar hinter solche des Typs 1, zurück. Sie verkörpern jedoch den aus konzeptioneller Sicht zu bevorzugenden Ansatz hinsichtlich Portabilität und Vergleichbarkeit der erzielten quantitativen Ergebnisse.
Typischerweise kommen im produktiven Einsatz jedoch Treiber der Typen 2 und 4 zum Einsatz, die entweder durch den Hersteller des DBMS mitgeliefert werden (Typ 2) oder auf der Basis publizierter Schnittstellen plattformunabhängig für genau ein spezifisches DBMS entwickelt wurden (Typ 4).

Generell formuliert das JDBC-Konzept auf dieser Ebene noch keine Einschränkung hinsichtlich der unterstützten DBMS-Typen und ist generell auf verschiedenste Datenquellen anwendbar. Durch die Struktur des API und die verfügbaren Treiber kristallisieren sich jedoch relationale DBMS als Hauptanwendungsgebiet dieser Zugriffsschnittstelle heraus.

Im folgenden wird die Verwendung des Typ 4 Treibers Connector/J im Zusammenspiel mit dem RDBMS *MySQL* betrachtet.

Die Beispiele basieren auf einer Demodatenbank, deren Struktur und Inhalte nachfolgend angegeben sind.

Die Tabelle EMPL0YEE

```
+----------+-------+---------+-----------+------------+--------------------------
+------+----------+-----------+------+
| FNAME    | MINIT | LNAME   | SSN       | BDATE      | ADDRESS                  |
SEX  | SALARY   | SUPERSSN  | DNO  |
+----------+-------+---------+-----------+------------+--------------------------
+------+----------+-----------+------+
| John     | B     | Smith   | 123456789 | 1965-01-09 | 731 Fondren, Houston, TX |
M    | 30000.00 | 333445555 |    5 |
| Franklin | T     | Wong    | 333445555 | 1955-12-08 | 638 Voss, Houston, TX    |
M    | 40000.00 | 888665555 |    5 |
| Joyce    | A     | English | 453453453 | 1972-07-31 | 5631 Rice, Houston, TX   |
F    | 25000.00 | 333445555 |    5 |
```

```
| Ramesh   | K     | Narayan | 666884444 | 1962-09-15 | 975 Fire Oak, Humble, TX |
M    | 38000.00 | 333445555 |    5 |
| James    | E     | Borg    | 888665555 | 1937-11-10 | 450 Stone, Houston, TX   |
M    | 55000.00 |      NULL |    1 |
| Jennifer | S     | Wallace | 987654321 | 1941-06-20 | 291 Berry, Bellaire, TX  |
F    | 43000.00 | 888665555 |    4 |
| Ahmad    | V     | Jabbar  | 987987987 | 1969-03-29 | 980 Dallas, Houston, TX  |
M    | 25000.00 | 987654321 |    4 |
| Alicia   | J     | Zelaya  | 999887777 | 1968-07-19 | 3321 Castle, Spring, TX  |
F    | 25000.00 | 987654321 |    4 |
+----------+-------+---------+-----------+------------+--------------------------
+------+----------+-----------+------+
```

Umsetzung in der Java-API

Das Klassendiagramm der Abbildung 2 zeigt die zentralen Klassen des Paketes `java.sql`. Auffallend ist, daß alle Elemente des dargestellten Pakets -- abgesehen von den definierten Exceptionklassen -- als Schnittstellen ausgelegt sind. Durch diese Mimik wird die Organisation der JDBC-Schnittstelle deutlich. Die API legt lediglich das Verhalten hinsichtlich seiner Semantik und die Einzeloperationen durch Definition ihrer Parameter fest, die konkrete DBMS-spezifische Implementierung dieser Operationen wird durch den JDBC-Treiber bereitgestellt.

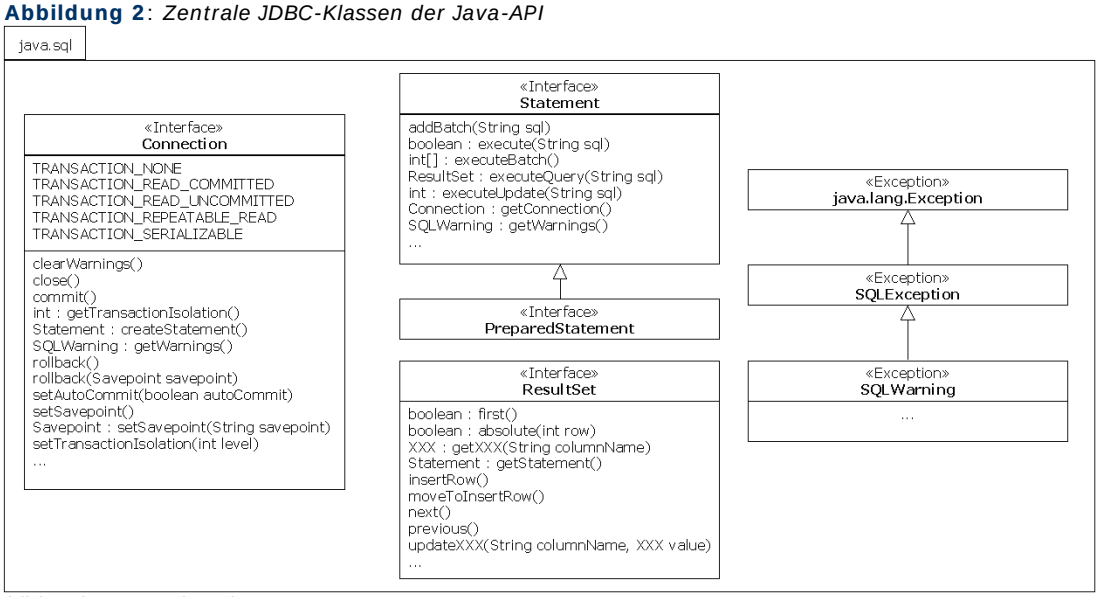
Zentrale Klasse der JDBC-API ist die Schnittstelle `Connection`. Sie bildet die Kommunikationsverbindungen zum DBMS ab und bietet notwendige Verwaltungsoperationen. Hierunter fallen insbesondere auch die Aufrufe zur Transaktionssteuerung.

Die Schnittstelle `Statement` realisiert genau eine aus Javasicht atomare Datenbankaktion. Diese muß hierbei aus minimal einem Aufruf an das DBMS bestehen, kann aber eine Reihe separater Aufrufe zu einem *Batch* bündeln.
Als Sonderform sieht die API die Spezialisierung `PreparedStatement` vor, die es gestattet parametrisierte Anfragen zwischenzuspeichern, die nach Belegung der Parameterfelder an das DBMS übergeben werden. Hierdurch wird ein einfacher Mechanismus zur Wiederverwendung von DBMS-Aufrufen etabliert.

Liefert eine DBMS-Anfrage Ergebnistupel, wo werden diese konform zur Schnittstelle `ResultSet` verwaltet. Diese Schnittstelle erlaubt die lesende Traversierung der vom DBMS gelieferten Tupel ebenso wie ihre Aktualisierung im Hauptspeicher und das anschließende Zurückschreiben in die Datenbank.
Die in der Abbildung nur durch *getXXX* und *updateXXX* angedeuteten Operationen existieren in Ausprägungen für alle unterstützten Datentypen, wobei *XXX* den Namen des Typs bezeichnet.

Ferner definiert die API eine mit `SQLWarning` eine Ausnahme zur Behandlung auftretender Fehlersituationen sowie eine Reihe weiterer, in der Abbildung 2 nicht dargestellter Klassen wie beispielsweise verschiedene Datentypen.

Abbildung 2: *Zentrale JDBC-Klassen der Java-API*



(click on image to enlarge!)

Zugriff auf die Datenbank

Beispiel 1 zeigt den Ablauf zur Aufnahme einer Verbindung mit der Datenbank `jdbctest` auf dem lokalen Rechner (`localhost`).

Zunächst muß die Klasse des gewählten JDBC-Treibers (im Beispiel `com.mysql.jdbc.Driver` vor ihrer Verwendung geladen werden. Dies geschieht durch den Aufruf der statischen Methode `forName` auf der Klasse `Class`.
Der zu ladende Treiber muß hierbei die JDBC-Schnittstellenklasse `Driver` implementieren um später durch die JDBC-API verwendet werden zu können.
Gleichzeitig mit dem dynamischen Ladevorgang erfolgt die Registrierung des Treibers beim JDBC-`DriverManager`, der die Verwaltung der geladenen DB-Treiber übernimmt.

Nach dem erfolgreichen Laden des Treibers wird durch den Aufruf von `getConnection` (Zeile 16) die Verbindung zur Datenbank hergestellt. Die anzusprechende Datenbank wird hierbei durch eine URI der Form `jdbc:mysql://`*`DB-Server/DB-Name`* repräsentiert (Zeile 17). Zusätzlich können ein zur Anmeldung am DB-System benötiger Benutzer (Zeile 18) und sein Paßwort (Zeile 19) übergeben werden.

Beispiel 1: Aufbau einer Datenbankverbindung

```
(1)import java.sql.DriverManager;
(2)import java.sql.SQLException;
(3)import com.mysql.jdbc.Connection;
(4)
(5)public class JDBCConnect {
(6)     public static void main(String[] args) {
(7)             try {
(8)                     Class.forName("com.mysql.jdbc.Driver");
(9)             } catch (ClassNotFoundException e) {
(10)                    System.err.println("Driver class not found");
(11)                    e.printStackTrace();
(12)            }
(13)            Connection con = null;
(14)            try {
(15)                    con =
(16)                            (Connection) DriverManager.getConnection(
(17)                                    "jdbc:mysql://localhost/jdbctest/",
(18)                                    "mario",
(19)                                    "thePassword");
(20)            } catch (SQLException e1) {
(21)                    System.err.println("Error establishing database connection");
(22)                    e1.printStackTrace();
(23)            }
(24)
(25)    }
(26)}
```

Download des Beispiels

Zusätzlich stellen die Klassen `Driver` und `DriverManager` die Möglichkeit der Abfrage von verbindungsunabhängigen Verwaltungsinformationen zur Verfügung.

Beispiel 2: Ermittlung von Informationen über Treiber und Treibermanager

```
(1)import java.sql.Driver;
(2)import java.sql.DriverManager;
(3)import java.util.Enumeration;
(4)
(5)public class JDBCDriver {
(6)
(7)     public static void main(String[] args) {
(8)             try {
(9)                     Class.forName("com.mysql.jdbc.Driver");
(10)            } catch (ClassNotFoundException e) {
(11)                    System.err.println("Driver class not found");
(12)                    e.printStackTrace();
(13)            }
(14)
(15)            System.out.println(
(16)                    "DriverManager:\nlogin timeout=" + DriverManager.
getLoginTimeout());
(17)
(18)            Enumeration e = DriverManager.getDrivers();
```

```
(19)            while (e.hasMoreElements()) {
(20)                    Driver drv = (Driver) e.nextElement();
(21)
(22)                    System.out.println(
(23)                            "Driver="
(24)                                    + drv.getClass().getName()
(25)                                    + "\nmajor version="
(26)                                    + drv.getMajorVersion()
(27)                                    + "\nminor version="
(28)                                    + drv.getMinorVersion()
(29)                                    + "\nJDBC compliant="
(30)                                    + drv.jdbcCompliant());
(31)            }
(32)
(33)    }
(34)}
```

Download des Beispiels
Download der Ergebnisdatei

Beispiel 2 zeigt die Ermittlung des durch den `DriverManager` für alle durch ihn verwalteten Treiber global definierten Login Timouts, der angibt wie lange beim Anmeldevorgang an der Datenbank auf eine Rückmeldung gewartet wird.
Zusätzlich werden für alle verwalteten Treiber der Klassenname sowie Daten zur Version und zum Stand der JDBC-Unterstützung ermittelt und ausgegeben.
Der JDBC-Unterstützungsstand gibt an, ob ein gegebener Treiber die Konformitätstests der Firma SUN bestanden hat. Voraussetzung hierfür ist u.a. die vollständige Unterstützung des SQL 92-Standards (entry level).
Diese Interpreatation von Spezifikationskonformität verwundert etwas, da alle JDBC-Treiber mit Ausnahme der inhärent DB-neutralen Typ 1 Treiber DBMS-spezifisch realisiert sind. Aus diesem Grunde bewertet der Konformitätstest vielmehr den Umsetzungsgrad des SQL-Standards in dem via JDBC genutzten DBMS als die Güte des JDBC-Treibers selbst.

Seit der JDBC-Schnittstellenversion 2 ist neben der „klassischen“ Zugriffsvariante auch eine auf dem Java Naming and Directory Interface (JNDI) basierende Zugriffsmethodik definiert, deren Verwendung --- abgesehen von der geänderten Mimik im Aufbau der DB-Verbindung --- identisch gestaltet ist.

Jedoch ist, wie in JNDI üblich, vor dem Zugriff ein benanntes Objekt beim JNDI-Dienst zu registrieren.
Im Falle von JDBC ist dies ein Objekt welches die Schnittstelle `DataSource` implementiert.

Der Code des Beispiels 3 zeigt die notwendigen Schritte zur Registrierung eines `MysqlDataSource`-Objekts, der durch den MySQL-JDBC-Treiber gelieferten Implementierung der Schnittstelle `DataSource`.

Beispiel 3: Ablage von Verbindungsinformation in einem JNDI-Verzeichnis

```
(1)import java.util.Hashtable;
(2)import javax.naming.Context;
(3)import javax.naming.InitialContext;
(4)import javax.naming.NamingException;
(5)import com.mysql.jdbc.jdbc2.optional.MysqlDataSource;
(6)
(7)public class JDBCConnect2Server {
(8)
(9)     public static void main(String[] args) {
(10)            Hashtable env = new Hashtable();
(11)            env.put(
(12)                    Context.INITIAL_CONTEXT_FACTORY,
(13)                    "com.sun.jndi.fscontext.RefFSContextFactory");
(14)            env.put(Context.PROVIDER_URL, "file:/tmp/registry");
(15)
(16)            MysqlDataSource ds = new MysqlDataSource();
(17)            ds.setDatabaseName("jdbctest");
(18)            Context ctx = null;
(19)            try {
(20)                    ctx = new InitialContext(env);
(21)            } catch (NamingException ne) {
(22)                    ne.printStackTrace();
```

```
(23)            }
(24)
(25)            try {
(26)                    ctx.rebind("jdbc/mySrc", ds);
(27)            } catch (NamingException ne) {
(28)                    ne.printStackTrace();
(29)            }
(30)    }
(31)}
```

Download des Beispiels

Entsprechend der modifizierten Ablage der Verwaltungsinformation ändert sich die Erzeugung der Datenbankverbindung beim Zugriff. Hier wird nun zunächst über einen Zugriff auf den JNDI-Verzeichnisdienst das benannte `DataSource`-Objekt (es trägt den Namen `jdbc/mySrc` ermittelt. Anschließend wird durch das dem Verzeichnisdienst entnommene `DataSource`-Objekt die Datenbankverbindung (d.h. das `Connection`-Objekt) erzeugt.
Alle weiteren Schritte zur Interaktion mit der Datenbank verlaufen dann identisch zur im Beispiel 1 gezeigten Verbindungsaufnahme.
Der Code des Beispiels 4 zeigt die notwendigen Schritte zur Ermittlung der Referenz auf das Objekt des Typs `DataSource` aus dem JNDI-Verzeichnis, sowie die Erzeugung des `Connection`-Objekts.

Beispiel 4: Verbindungsaufbau unter Nutzung von JNDI

```
(1)import java.sql.Connection;
(2)import java.sql.SQLException;
(3)import java.util.Hashtable;
(4)import javax.naming.Context;
(5)import javax.naming.InitialContext;
(6)import javax.naming.NamingException;
(7)import javax.sql.DataSource;
(8)
(9)public class JDBCConnect2 {
(10)    public static void main(String[] args) {
(11)            Hashtable env = new Hashtable();
(12)            env.put(
(13)                    Context.INITIAL_CONTEXT_FACTORY,
(14)                    "com.sun.jndi.fscontext.RefFSContextFactory");
(15)            env.put(Context.PROVIDER_URL, "file:/tmp/registry");
(16)            Context ctx = null;
(17)            try {
(18)                    ctx = new InitialContext(env);
(19)            } catch (NamingException ne) {
(20)                    ne.printStackTrace();
(21)            }
(22)            DataSource ds = null;
(23)            try {
(24)                    ds = (DataSource) ctx.lookup("jdbc/mySrc");
(25)            } catch (NamingException ne) {
(26)                    ne.printStackTrace();
(27)            }
(28)            Connection con = null;
(29)            try {
(30)                    con = ds.getConnection("mario", "thePassword");
(31)            } catch (SQLException sqle) {
(32)                    sqle.printStackTrace();
(33)            }
(34)    }
(35)}
```

Download des Beispiels

Auffallend ist die Ablage des Datenbanknamens im Verzeichnisdienst mittels des Methodenaufrufs `setDatabaseName`. Diese Verschiebung der Information wird durch die geänderte Mimik der Erzeugung des `Connection`-Objekts impliziert. So sieht die Implementierung dieser Methode für die Klasse `DataSource` keine Möglichkeit zur gleichzeitigen Übergabe von Anmeldenamen, Paßwort und Datenbank vor.
Vielmehrnoch ist es sogar möglich diese Daten allesamt innerhalb des JNDI-Verzeichnisdienstes abzulegen. (Für diesen Zweck stehen die Methoden `setUser` bzw. `setPassword` zur Verfügung.) Als Konsequenz hiervon kann der Verbinungswunsch durch Aufruf der Methode `getConnection` ohne

weitere Parameter erfüllt werden.
Diese Umsetzungsweise ist vor ihrer Realisierung hinsichtlich des damit eintretenden Verlustes an Sicherheit zu prüfen, da in ihrer Folge eine Datenbankverbindung allein durch Kenntnis des JNDI-residenten Namens des `DataSource`-Objektes erfolgen kann.

Generell wählen JDBC-Umsetzungen den Weg jede Ausprägung eines `Connection`-Objekts in eine physische Datenbankverbindung abzubilden. Dieses, durchaus der intuitiven Semantik der `Connection`-Klasse entsprechende Vorgehen kann jedoch in realen Applikationen, begründet in der Vielzahl der durch das DBMS zu verwaltenden Verbindungen, zu Zugriffsengpässen führen.
Aus diesem Grunde definiert die JDBC-Schnittstelle Operationen zur Zusammenfassung „gleichartiger" Zugriffe. Hierzu zählen Zugriffe die unter derselben Nutzerkennung auf dieselbe Datenbank abgewickelt werden. Diese Zugriffsform tritt insbesondere bei Anwendungen auf, die über nur einen in der Datenbank eingetragenen Anwender verfügen und die gesamte Nutzerverwaltung datenbanktransparent applikationsseitig abwickeln.
Zur Optimierung von Zugriffen dieser Natur sieht die JDBC-Schnittstelle das sog. *Connection Pooling* vor, welches gleichartige Zugriffe bündelt.
Das Beispiel 5 zeigt eine Umsetzung:

Beispiel 5: Verbindungsaufbau unter Nutzung von Connection Pooling

```
(1)import java.sql.DriverManager;
(2)import java.sql.SQLException;
(3)import javax.sql.PooledConnection;
(4)import com.mysql.jdbc.Connection;
(5)import com.mysql.jdbc.jdbc2.optional.MysqlPooledConnection;
(6)
(7)public class JDBCConnection3 {
(8)     public static void main(String[] args) {
(9)             try {
(10)                    Class.forName("com.mysql.jdbc.Driver");
(11)            } catch (ClassNotFoundException cnfe) {
(12)                    System.err.println("Driver class not found");
(13)                    cnfe.printStackTrace();
(14)            }
(15)            Connection con = null;
(16)            try {
(17)                    con =
(18)                            (Connection) DriverManager.getConnection(
(19)                                    "jdbc:mysql://localhost/jdbctest/",
(20)                                    "mario",
(21)                                    "thePassword");
(22)            } catch (SQLException e1) {
(23)                    System.err.println("Error establishing database connection");
(24)                    e1.printStackTrace();
(25)            }
(26)
(27)            PooledConnection pc = new MysqlPooledConnection(con);
(28)
(29)            java.sql.Connection con1 = null;
(30)            try {
(31)                    con1 = pc.getConnection();
(32)            } catch (SQLException sqle) {
(33)                    sqle.printStackTrace();
(34)            }
(35)    }
(36)}
```

Download des Beispiels

Statt für jede gewünschte Datenbankverbindung ein zusätzliches Objekt des Type `Connection` zu erzeugen wird die erzeugte Verbindung zur Konstruktion eines Objektes, welches Konform zur Schnittstelle `PooledConnection` definiert ist, verwendet. Dieses verwaltet sort für die Verwaltung der DB-Verbindung und stellt dieselbe physische Verbindung verschiedenen Anfragern zur Verfügung.
Konsequenterweise wird daher eine neue Verbindung nicht mehr vom `DriverManager` angefordert, sondern durch die Methode `getConnection` der aus der Verwaltungsstruktur entnommenen `PooledConnection` beantragt.

Aufgrund der Unterstützung des SQL-Sprachumfanges, durch unveränderte textuelle Propagation an das DBMS sind durch JDBC im Allgemeinen alle Facetten der Datenbanksprache nutzbar, sofern

sie durch das verwendete DBMS Unterstützung finden. Hierunter fallen:

- Data Definition Language.
  Zur Erzeugung eines Datenmodells.
- Data Manipulation Language.
  Zur Modifikation der verwalteten Daten.
- Data Retrieval Language.
  Zur Anfrage der in einer Datenbank gespeicherten Daten.
- Data Control Language.
  Zur Festlegung und Kontrolle von Zugriffsberechtigungen.

JDBC reflektiert jedoch nicht diese Sprach(-sub-)klassen selbst in der API, sondern sieht vielmehr ausschließlich zwei Formen des Zugriffs vor. Solche die tabellenwerte Resultate liefern und solche, deren Ausführung lediglich primitivwertige Rückgabewerte liefert.

Primitivwertige Zugriffe

Primitivwertige Datenbankzugriffe liefern, abgesehen von Fehler- oder Warnmeldungen, lediglich die Anzahl der geänderten Tupel, falls zutreffend, oder 0 zurück.
Aus dieser Festlegung lassen sich diejenigen SQL-Anweisungstypen ableiten, welche als primitivwertiger Zugriff realisiert sind. Hierunter fallen alle Operationen der Datendefinition wie `CREATE` oder `ALTER TABLE` sowie alle Einfüge- (`INSERT`) Änderungs- (`UPDATE`) und Löschvorgänge (`DELETE`). Darüberhinaus alle Operationen zur Administration der Datenbank durch Rechtevergabe (`GRANT`, `REVOKE`).

Zugriffe dieser Art werden generell durch die Methode `executeUpdate`, oder einer Abart davon, realisiert.

Beispiel 6: Erstellung einer neuen Tabelle

```
(1)import java.sql.DriverManager;
(2)import java.sql.SQLException;
(3)import com.mysql.jdbc.Connection;
(4)import com.mysql.jdbc.Statement;
(5)
(6)public class JDBCCreateTable {
(7)     public static void main(String[] args) {
(8)             try {
(9)                     Class.forName("com.mysql.jdbc.Driver");
(10)            } catch (ClassNotFoundException e) {
(11)                    System.err.println("Driver class not found");
(12)                    e.printStackTrace();
(13)            }
(14)            Connection con = null;
(15)
(16)            try {
(17)                    con =
(18)                            (Connection) DriverManager.getConnection(
(19)                                    "jdbc:mysql://localhost/jdbctest/",
(20)                                    "mario",
(21)                                    "thePassword");
(22)            } catch (SQLException e1) {
(23)                    System.err.println("Error establishing database connection");
(24)                    e1.printStackTrace();
(25)            }
(26)
(27)            Statement  stmt = null;
(28)            try {
(29)                    stmt = (Statement) con.createStatement();
(30)            } catch (SQLException e2) {
(31)                    System.err.println("Error creating SQL-Statement");
(32)                    e2.printStackTrace();
(33)            }
(34)            String createTab = new String("CREATE TABLE EMPLOYEE(" +
(35)                    "FNAME VARCHAR(10) NOT NULL," +
(36)                    "MINIT VARCHAR(1)," +
(37)                    "LNAME VARCHAR(10) NOT NULL," +
(38)                    "SSN INTEGER(9) NOT NULL," +
(39)                    "BDATE DATE," +
(40)                    "ADDRESS VARCHAR(30)," +
(41)                    "SEX ENUM('M','F')," +
(42)                    "SALARY REAL(7,2) UNSIGNED," +
```

```
(43)                    "SUPERSSN INTEGER(9)," +
(44)                    "DNO INTEGER(1));");
(45)            try {
(46)                    System.out.println("result="+stmt.executeUpdate(createTab));
(47)            } catch (SQLException e3) {
(48)                    System.err.println("Error creating table EMPLOYEE");
(49)                    e3.printStackTrace();
(50)            }
(51)    }
(52)}
```

Download des Beispiels
Download der Ergebnisdatei

Beispiel 6 zeigt die notwendigen Schritte zur Erstellung der Tabelle `EMPLOYEE` in der Datenbank.

Nach dem (üblichen) Verbindungsaufbau (Zeile 8-24) wird in Zeile 26 eine Variable des Typs `Statement` deklariert. Auch bei `Statement` handelt es sich um eine durch die JDBC-API vordefinierte Schnittstelle die als Bestandteil des JDBC-Treibers durch von einer Klasse implementiert wird.
Ausgehend von der etablierten Datenbankverbindung wird durch Aufruf der Methode `createStatement` eine konkrete Ausprägung konform zur `Statement`-Schnittstelle erzeugt (Zeile 28).

Der Aufruf von `executeUpdate` übergibt das als Zeichenkette abgelegte SQL-Kommando an die Datenbank zur Ausführung.
Da durch `CREATE TABLE` keine Tupeländerungen vorgenommen werden ist das Resultat des Aufrufs der Rückgabewert `0`.

Beispiel 7 zeigt mit dem `ALTER TABLE`-Kommando eine weitere Anwendung der `executeUpdate`-Methode.
Auch in diesem Falle wird als Resultat `0` geliefert, da die Definition des Primärschlüssels keine Änderungen an den verwalteten Datensätzen vornimmt.

Beispiel 7: Modifikation der Tabellendefinition

```
(1)import java.sql.DriverManager;
(2)import java.sql.SQLException;
(3)import com.mysql.jdbc.Connection;
(4)import com.mysql.jdbc.Statement;
(5)
(6)public class JDBCAlterTable {
(7)     public static void main(String[] args) {
(8)             try {
(9)                     Class.forName("com.mysql.jdbc.Driver");
(10)            } catch (ClassNotFoundException e) {
(11)                    System.err.println("Driver class not found");
(12)                    e.printStackTrace();
(13)            }
(14)            Connection con = null;
(15)
(16)            try {
(17)                    con =
(18)                            (Connection) DriverManager.getConnection(
(19)                                    "jdbc:mysql://localhost/jdbctest/",
(20)                                    "mario",
(21)                                    "thePassword");
(22)            } catch (SQLException e1) {
(23)                    System.err.println("Error establishing database connection");
(24)                    e1.printStackTrace();
(25)            }
(26)
(27)            Statement stmt = null;
(28)            try {
(29)                    stmt = (Statement) con.createStatement();
(30)            } catch (SQLException e2) {
(31)                    System.err.println("Error creating SQL-Statement");
(32)                    e2.printStackTrace();
(33)            }
(34)            String createTab =
(35)                    new String("ALTER TABLE EMPLOYEE ADD PRIMARY KEY (SSN);");
```

```
(36)            try {
(37)                    System.out.println("result=" + stmt.executeUpdate
(createTab));
(38)            } catch (SQLException e3) {
(39)                    System.err.println("Error altering table EMPLOYEE");
(40)                    e3.printStackTrace();
(41)            }
(42)    }
(43)}
```

Download des Beispiels
Download der Ergebnisdatei

Beispiel 8: Einfügen von Werten

```
(1)import java.sql.DriverManager;
(2)import java.sql.SQLException;
(3)import com.mysql.jdbc.Connection;
(4)import com.mysql.jdbc.Statement;
(5)
(6)public class JDBCInsert1 {
(7)     public static void main(String[] args) {
(8)             try {
(9)                     Class.forName("com.mysql.jdbc.Driver");
(10)            } catch (ClassNotFoundException e) {
(11)                    System.err.println("Driver class not found");
(12)                    e.printStackTrace();
(13)            }
(14)            Connection con = null;
(15)
(16)            try {
(17)                    con =
(18)                            (Connection) DriverManager.getConnection(
(19)                                    "jdbc:mysql://localhost/jdbctest/",
(20)                                    "mario",
(21)                                    "thePassword");
(22)            } catch (SQLException e1) {
(23)                    System.err.println("Error establishing database connection");
(24)                    e1.printStackTrace();
(25)            }
(26)
(27)            Statement stmt = null;
(28)            try {
(29)                    stmt = (Statement) con.createStatement();
(30)            } catch (SQLException e2) {
(31)                    System.err.println("Error creating SQL-Statement");
(32)                    e2.printStackTrace();
(33)            }
(34)
(35)            try {
(36)                    System.out.println("result=" + stmt.executeUpdate("INSERT
INTO EMPLOYEE VALUES('John', 'B', 'Smith', 123456789, '1965-01-09', '731 Fondren,
Houston, TX', 'M', 30000, 333445555, 5);"));
(37)                    System.out.println("result=" + stmt.executeUpdate("INSERT
INTO EMPLOYEE VALUES('Franklin', 'T', 'Wong', 333445555, '1955-12-08', '638 Voss,
Houston, TX', 'M', 40000, 888665555, 5);"));
(38)                    System.out.println("result=" + stmt.executeUpdate("INSERT
INTO EMPLOYEE VALUES('Alicia', 'J', 'Zelaya', 999887777, '1968-07-19', '3321 Castle,
Spring, TX', 'F', 25000, 987654321, 4);"));
(39)                    System.out.println("result=" + stmt.executeUpdate("INSERT
INTO EMPLOYEE VALUES('Jennifer', 'S', 'Wallace', 987654321, '1941-06-20', '291
Berry, Bellaire, TX', 'F', 43000, 888665555, 4);"));
(40)                    System.out.println("result=" + stmt.executeUpdate("INSERT
INTO EMPLOYEE VALUES('Ramesh', 'K', 'Narayan', 666884444, '1962-09-15', '975 Fire
Oak, Humble, TX', 'M', 38000, 333445555, 5);"));
(41)                    System.out.println("result=" + stmt.executeUpdate("INSERT
INTO EMPLOYEE VALUES('Joyce', 'A', 'English', 453453453, '1972-07-31', '5631 Rice,
Houston, TX', 'F', 25000, 333445555, 5);"));
(42)                    System.out.println("result=" + stmt.executeUpdate("INSERT
INTO EMPLOYEE VALUES('Ahmad', 'V', 'Jabbar', 987987987, '1969-03-29', '980 Dallas,
Houston, TX', 'M', 25000, 987654321, 4);"));
```

```
(43)                    System.out.println("result=" + stmt.executeUpdate("INSERT
INTO EMPLOYEE VALUES('James', 'E', 'Borg', 888665555, '1937-11-10', '450 Stone,
Houston, TX', 'M', 55000, null, 1);"));
(44)            } catch (SQLException e3) {
(45)                    System.err.println("Error inserting values into table
EMPLOYEE");
(46)                    e3.printStackTrace();
(47)            }
(48)    }
(49)}
```

Download des Beispiels
Download der Ergebnisdatei

Beispiel 8 zeigt den Einfügevorgang von acht Werten in die durch die vorangegangenen Beispiele erzeugte Tabelle EMPLOYEE.
Jeder der Einfügevorgänge der Zeilen 36-43 führt im Rahmen einer separaten Datenbankkommunikation sequentiell genau einen Einfügevorgang durch, was durch den Rückgabewert 1 dokumentiert wird.

Zwar ist dieses Verfahren praktikabel und erzielt die angestrebten Resultate, jedoch ist es unter Zeiteffizienzgesichtspunkten inadäquat, da sich Einfüge- und Kommunikationsvorgänge zahlenmäßig entsprechen.

Aus diesem Grunde bietet die Schnittstelle Statement die Möglichkeit zur Bündelung einzelner SQL-Aufrufe in einem sog. *Batch* an.

Beispiel 9 zeigt die entsprechende Umgestaltung des vorangegangenen Beispiels.

Beispiel 9: Einfügen von Werten mittels eines Batches

```
(1)import java.sql.DriverManager;
(2)import java.sql.SQLException;
(3)import com.mysql.jdbc.Connection;
(4)import com.mysql.jdbc.Statement;
(5)
(6)public class JDBCInsert2 {
(7)     public static void main(String[] args) {
(8)             try {
(9)                     Class.forName("com.mysql.jdbc.Driver");
(10)            } catch (ClassNotFoundException e) {
(11)                    System.err.println("Driver class not found");
(12)                    e.printStackTrace();
(13)            }
(14)            Connection con = null;
(15)
(16)            try {
(17)                    con =
(18)                            (Connection) DriverManager.getConnection(
(19)                                    "jdbc:mysql://localhost/jdbctest/",
(20)                                    "mario",
(21)                                    "thePassword");
(22)            } catch (SQLException e1) {
(23)                    System.err.println("Error establishing database connection");
(24)                    e1.printStackTrace();
(25)            }
(26)
(27)            Statement stmt = null;
(28)            try {
(29)                    stmt = (Statement) con.createStatement();
(30)            } catch (SQLException e2) {
(31)                    System.err.println("Error creating SQL-Statement");
(32)                    e2.printStackTrace();
(33)            }
(34)
(35)            try {
(36)                    stmt.addBatch("INSERT INTO EMPLOYEE VALUES('John', 'B',
'Smith', 123456789, '1965-01-09', '731 Fondren, Houston, TX', 'M', 30000, 333445555,
5);");
(37)                    stmt.addBatch("INSERT INTO EMPLOYEE VALUES('Franklin', 'T',
'Wong', 333445555, '1955-12-08', '638 Voss, Houston, TX', 'M', 40000, 888665555,
```

```
5);");
(38)                    stmt.addBatch("INSERT INTO EMPLOYEE VALUES('Alicia', 'J',
'Zelaya', 999887777, '1968-07-19', '3321 Castle, Spring, TX', 'F', 25000, 987654321,
4);");
(39)                    stmt.addBatch("INSERT INTO EMPLOYEE VALUES('Jennifer', 'S',
'Wallace', 987654321, '1941-06-20', '291 Berry, Bellaire, TX', 'F', 43000,
888665555, 4);");
(40)                    stmt.addBatch("INSERT INTO EMPLOYEE VALUES('Ramesh', 'K',
'Narayan', 666884444, '1962-09-15', '975 Fire Oak, Humble, TX', 'M', 38000,
333445555, 5);");
(41)                    stmt.addBatch("INSERT INTO EMPLOYEE VALUES('Joyce', 'A',
'English', 453453453, '1972-07-31', '5631 Rice, Houston, TX', 'F', 25000, 333445555,
5);");
(42)                    stmt.addBatch("INSERT INTO EMPLOYEE VALUES('Ahmad', 'V',
'Jabbar', 987987987, '1969-03-29', '980 Dallas, Houston, TX', 'M', 25000, 987654321,
4);");
(43)                    stmt.addBatch("INSERT INTO EMPLOYEE VALUES('James', 'E',
'Borg', 888665555, '1937-11-10', '450 Stone, Houston, TX', 'M', 55000, null, 1);");
(44)                    int[] insertCounts = stmt.executeBatch();
(45)            } catch (SQLException e3) {
(46)                    System.err.println("Error inserting values into table
EMPLOYEE");
(47)                    e3.printStackTrace();
(48)            }
(49)    }
(50)}
```

Download des Beispiels

Statt der Einzelübergabe der SQL INSERT-Anweisungen werden diese nun (in Zeile 36-43) in in einem Batch gesammelt. Hierzu werden die SQL-Zeichenketten durch den Aufruf addBatch innerhalb des Statement-Objekts abgelegt und durch Aufruf der Methode executeBatch gesammelt an das DBMS übergeben.
Statt der Einzelresultate wird durch diese Aufrufvariante ein Array geliefert, der die Einzelrückgabewerte der als Batch übergebenen Aufrufe versammelt.

Dies verdeutlicht nochmals das nachfolgende Beispiel. In ihm wird zunächst mittels ALTER TABLE eine neue Tabellenspalte zur Aufnahme des Wochentages der Geburt erstellt und anschließend durch SQL UPDATE-Anweisungen die benötigten Daten aus dem vorhandenen Geburtsdatum ermittelt.
Auch dieses Beispiel bedient sich zur Performancebeschleunigung der Möglichkeiten des Batchaufrufes.

Beispiel 10: Aktualisieren von Tabellendefinitionen und Werten

```
(1)import java.sql.DriverManager;
(2)import java.sql.SQLException;
(3)import com.mysql.jdbc.Connection;
(4)import com.mysql.jdbc.Statement;
(5)
(6)public class JDBCUpdate1 {
(7)     public static void main(String[] args) {
(8)             try {
(9)                     Class.forName("com.mysql.jdbc.Driver");
(10)            } catch (ClassNotFoundException e) {
(11)                    System.err.println("Driver class not found");
(12)                    e.printStackTrace();
(13)            }
(14)            Connection con = null;
(15)
(16)            try {
(17)                    con =
(18)                            (Connection) DriverManager.getConnection(
(19)                                    "jdbc:mysql://localhost/jdbctest/",
(20)                                    "mario",
(21)                                    "thePassword");
(22)            } catch (SQLException e1) {
(23)                    System.err.println("Error establishing database connection");
(24)                    e1.printStackTrace();
(25)            }
(26)
```

```
(27)            Statement stmt = null;
(28)            try {
(29)                    stmt = (Statement) con.createStatement();
(30)            } catch (SQLException e2) {
(31)                    System.err.println("Error creating SQL-Statement");
(32)                    e2.printStackTrace();
(33)            }
(34)
(35)            try {
(36)                    stmt.addBatch("ALTER TABLE EMPLOYEE ADD BDAY VARCHAR(10);");
(37)                    stmt.addBatch("UPDATE EMPLOYEE SET BDAY='Sunday' WHERE
DAYOFWEEK(BDATE)=1;");
(38)                    stmt.addBatch("UPDATE EMPLOYEE SET BDAY='Monday' WHERE
DAYOFWEEK(BDATE)=2;");
(39)                    stmt.addBatch("UPDATE EMPLOYEE SET BDAY='Tuesday' WHERE
DAYOFWEEK(BDATE)=3;");
(40)                    stmt.addBatch("UPDATE EMPLOYEE SET BDAY='Wednesday' WHERE
DAYOFWEEK(BDATE)=4;");
(41)                    stmt.addBatch("UPDATE EMPLOYEE SET BDAY='Thursday' WHERE
DAYOFWEEK(BDATE)=5;");
(42)                    stmt.addBatch("UPDATE EMPLOYEE SET BDAY='Friday' WHERE
DAYOFWEEK(BDATE)=6;");
(43)                    stmt.addBatch("UPDATE EMPLOYEE SET BDAY='Saturday' WHERE
DAYOFWEEK(BDATE)=7;");
(44)                    int[] result = stmt.executeBatch();
(45)                    for (int i=0; i<result.length;i++){
(46)                            System.out.println("Statement No "+i+" changed
"+result[i]+" rows");
(47)                    }
(48)            } catch (SQLException e3) {
(49)                    System.err.println("Error inserting values into table
EMPLOYEE");
(50)                    e3.printStackTrace();
(51)            }
(52)    }
(53)}
```

Download des Beispiels
Download der Ergebnisdatei

Die Ausführung liefert als Resultat:

```
Statement No 0 changed 8 rows
Statement No 1 changed 0 rows
Statement No 2 changed 1 rows
Statement No 3 changed 0 rows
Statement No 4 changed 1 rows
Statement No 5 changed 1 rows
Statement No 6 changed 2 rows
Statement No 7 changed 3 rows
```

So werden durch den `ALTER TABLE`-Aufruf (Indexnummer 0) alle acht Tupel der Tabelle modifiziert, während die nachfolgenden Aufrufe nur Teilmengen davon verändern.

Die nähere Betrachtung der Zeilen 37-43 des Quellcodes von Beispiel 10 zeigt sich, daß diese im Kern denselben Vorgang ausführen, nur jeweils mit variierenden Parametern.
Zur Behandlung von Fällen dieser Problemstellung definiert die JDBC-API die Schnittstelle `PreparedStatement` als Spezialisierung von `Statement`.

Diese Schnittstelle gestattet es Anweisungen die später an die Datenbank übermittelt werden sollen mit Platzhaltern zu versehen und diese vor der Übermittlung mit Werten zu befüllen.
Beispiel 11 zeigt die entprechende Modifikation des vorangegangenen Beispiels.

Beispiel 11: Aktualisieren von Tabellendefinitionen und Werten

```
(1)import java.sql.DriverManager;
(2)import java.sql.SQLException;
(3)import com.mysql.jdbc.Connection;
(4)import com.mysql.jdbc.PreparedStatement;
(5)import com.mysql.jdbc.Statement;
(6)
(7)public class JDBCUpdate2 {
(8)     public static void main(String[] args) {
(9)             try {
(10)                    Class.forName("com.mysql.jdbc.Driver");
(11)            } catch (ClassNotFoundException e) {
(12)                    System.err.println("Driver class not found");
(13)                    e.printStackTrace();
(14)            }
(15)            Connection con = null;
(16)
(17)            try {
(18)                    con =
(19)                            (Connection) DriverManager.getConnection(
(20)                                    "jdbc:mysql://localhost/jdbctest/",
(21)                                    "mario",
(22)                                    "thePassword");
(23)            } catch (SQLException e1) {
(24)                    System.err.println("Error establishing database connection");
(25)                    e1.printStackTrace();
(26)            }
(27)
(28)            Statement stmt = null;
(29)            PreparedStatement pstmt = null;
(30)            try {
(31)                    stmt = (Statement) con.createStatement();
(32)                    pstmt = (PreparedStatement) con.prepareStatement("UPDATE
EMPLOYEE SET BDAY=? WHERE DAYOFWEEK(BDATE)=?;");
(33)
(34)            } catch (SQLException e2) {
(35)                    System.err.println("Error creating SQL-Statement");
(36)                    e2.printStackTrace();
(37)            }
(38)
(39)            try {
(40)                    String[] days=
{"Sunday","Monday","Tuesday","Wednesday","Thursday","Friday","Saturday"} ;
(41)                    stmt.addBatch("ALTER TABLE EMPLOYEE ADD BDAY VARCHAR(10);");
(42)                    for (int i=1; i<8;i++){
(43)                            pstmt.setString(1,days[i-1]);
(44)                            pstmt.setInt(2,i);
(45)                            pstmt.addBatch();
(46)                    }
(47)                    int[] result = stmt.executeBatch();
(48)                    for (int i=0; i<result.length;i++){
(49)                            System.out.println("Statement No "+i+" changed
"+result[i]+" rows");
(50)                    }
(51)            } catch (SQLException e3) {
(52)                    System.err.println("Error inserting values into table
EMPLOYEE");
(53)                    e3.printStackTrace();
(54)            }
(55)    }
(56)}
```

Download des Beispiels
Download der Ergebnisdatei

Im Beispiel wird neben dem Objekt des Typs `Statement` zusätzlich eines des Typs `PreparedStatement` erzeugt (Zeile 32).
Die dem Konstruktor übergebene Anweisung enthält als Sonderzeichen zur Markierung der Platzhalter das Fragezeichen (?).
Die Wochentage werde in Zeile 40, des vereinfachten Zugriffs wegen, als Array definiert.
In den Zeilen 42 mit 46 werden die benötigten SQL-`UPDATE`-Anweisungen dynamisch aus durch

Einsetzen der geeigneten Werte in den vorpräparierten Änderungsausruck erzeugt und einem eigenen Batch zugeordnet. Der Einsetzungsvorgang der benötigten Werte geschieht durch die Methoden `setString` für zeichenkettenartige bzw. `setInt` für den ganzzahlige Parameter. Den Methoden wird jeweils die Position des Parameters, gezählt ab 1 sowie die zu wählende Wertbelegung übermittelt.
Zur Ausführung müssen beide Batches getrennt angefordert werden.

Tabellenwertige Zugriffe

Die in der Praxis quantitativ bedeutendste Klasse von Datenbankzugriffen dürfte zweifellos auf die lesende Ermittlung von bestehenden Daten darstellen, kurzum alle Spielarten der SQL SELECT-Anweisung.

Für Anfragen an die Datenbank steht prinzipiell der gesamte durch das DBMS unterstützte SQL-Umfang zur Verfügung.

Anfragen werden im Gegensatz zu den bisher betrachteten lesenden Zugriffen nicht als primivwerte Methoden realisiert, sondern liefern als Resultat immer eine Tabelle zurück.
Diese wird durch den API-Typ `ResultSet` dargestellt.
Zusätzlich werden Anfragen durch die Methode executeQuery ausgeführt.

Das Beispiel 12 zeigt die generische Extraktion von DB-Daten und den Zugriff auf Metadaten.
Die aus der Datenbank gelesenen Ergebnistupel werden im durch `rs` benannten `ResultSet` abgelegt (Zeile 39). Die Resultatmenge wird mithilfe eines *Cursors* (Datensatzzeiger) traversiert. Hierzu wird der initial auf eine Ausgangsstellung vor dem ersten empfangenen Tupel positionierte Cursor durch Aufruf der Methode `next` solange weitergerückt, bis der letzte Datensatz verarbeitet wurde.

Der Aufruf der Methode`getMetaData` liefert deskriptive Metadaten wie Spaltenzahl sowie deren Bezeichner und Typen für die erstellte Resultattupelmenge.
In Zeile 43 werden diese Metadaten verwendet um die Spaltennamen der extrahierten Attribute anzuzeigen.
Zeile 47-52 liest die einzelnen Werte jedes Tupels mittels `getObject` aus und stellt sie am Bildschirm dar.

Beispiel 12: Auslesen von Daten und Metadaten

```
(1)import java.sql.DriverManager;
(2)import java.sql.ResultSet;
(3)import java.sql.ResultSetMetaData;
(4)import java.sql.SQLException;
(5)
(6)import com.mysql.jdbc.Connection;
(7)import com.mysql.jdbc.Statement;
(8)
(9)public class JDBCSelect1 {
(10)    public static void main(String[] args) {
(11)            try {
(12)                    Class.forName("com.mysql.jdbc.Driver");
(13)            } catch (ClassNotFoundException e) {
(14)                    System.err.println("Driver class not found");
(15)                    e.printStackTrace();
(16)            }
(17)            Connection con = null;
(18)
(19)            try {
(20)                    con =
(21)                            (Connection) DriverManager.getConnection(
(22)                                    "jdbc:mysql://localhost/jdbctest/",
(23)                                    "mario",
(24)                                    "thePassword");
(25)            } catch (SQLException e1) {
(26)                    System.err.println("Error establishing database connection");
(27)                    e1.printStackTrace();
(28)            }
(29)
(30)            Statement stmt = null;
(31)            try {
(32)                    stmt = (Statement) con.createStatement();
(33)            } catch (SQLException e2) {
(34)                    System.err.println("Error creating SQL-Statement");
```

```
(35)                    e2.printStackTrace();
(36)            }
(37)
(38)            try {
(39)                    ResultSet rs = stmt.executeQuery("SELECT * FROM EMPLOYEE;");
(40)                    ResultSetMetaData rsmd = rs.getMetaData();
(41)                    int noColumns = rsmd.getColumnCount();
(42)                    for (int i = 1; i < noColumns; i++) {
(43)                            System.out.print(rsmd.getColumnLabel(i) + "\t");
(44)                    }
(45)                    System.out.println();
(46)
(47)                    while (rs.isLast() == false) {
(48)                            rs.next();
(49)                            for (int i = 1; i < noColumns; i++) {
(50)                                    System.out.print( rs.getObject(i)+"\t" );
(51)                            }
(52)                            System.out.println();
(53)                    }
(54)
(55)            } catch (SQLException e3) {
(56)                    System.err.println("Error selecting values from table
EMPLOYEE");
(57)                    e3.printStackTrace();
(58)            }
(59)    }
(60)}
```

Download des Beispiels
Download der Ergebnisdatei

Neben im Beispiel 12 gezeigten Verarbeitung in exakter der Ablagereihenfolge der Datenbank kann auch durch Definition eines Cursors die Traversierung in inverser Ablagerichtung erreicht werden. Das nachfolgende Beispiel das entsprechende Vorgehen durch anfängliche Positionierung des Cursors ans Ende der empfangenen Daten (d.h. nach dem letzten Datensatz) und anschließendes schrittweises Rückpositionieren durch Aufruf der Methode `previous`.

Beispiel 13: Auslesen von Daten in invertierter Reihenfolge

```
(1)import java.sql.DriverManager;
(2)import java.sql.ResultSet;
(3)import java.sql.SQLException;
(4)import com.mysql.jdbc.Connection;
(5)import com.mysql.jdbc.Statement;
(6)
(7)public class JDBCSelect5 {
(8)     public static void main(String[] args) {
(9)             try {
(10)                    Class.forName("com.mysql.jdbc.Driver");
(11)            } catch (ClassNotFoundException e) {
(12)                    System.err.println("Driver class not found");
(13)                    e.printStackTrace();
(14)            }
(15)            Connection con = null;
(16)
(17)            try {
(18)                    con =
(19)                            (Connection) DriverManager.getConnection(
(20)                                    "jdbc:mysql://localhost/jdbctest/",
(21)                                    "mario",
(22)                                    "thePassword");
(23)            } catch (SQLException sqle) {
(24)                    System.err.println("Error establishing database connection");
(25)                    sqle.printStackTrace();
(26)            }
(27)
(28)            Statement stmt = null;
(29)            try {
(30)                    stmt =
(31)                            (Statement) con.createStatement();
(32)            } catch (SQLException e2) {
(33)                    System.err.println("Error creating SQL-Statement");
```

```
(34)                    e2.printStackTrace();
(35)            }
(36)
(37)            try {
(38)                    ResultSet rs = stmt.executeQuery("SELECT * FROM EMPLOYEE;");
(39)                    rs.afterLast();
(40)                    while (rs.previous()){
(41)                            System.out.println(rs.getString("FNAME"));
(42)                    }
(43)            } catch (SQLException sqle) {
(44)                    System.err.println("Error selecting values from table
EMPLOYEE");
(45)                    sqle.printStackTrace();
(46)            }
(47)    }
(48)}
```

Download des Beispiels
Download der Ergebnisdatei

Ferner kann der Cursor wahlfrei auf eine beliebige Position der Ergebnisrelation gesetzt werden. Das nachfolgende Beispiel zeigt dies. Ferner illustriert es das Vorgehen zur Größenermittlung des resultierenden `ResultSets` durch das Aufrufpaar `last` und `getRow`, welches zunächst den Cursor auf den letzten aus der Datenbank extrahierten Datensatz positioniert und anschließend dessen Nummer liefert.

Beispiel 14: Auslesen von Daten in wahlfreier Reihenfolge

```
(1)import java.sql.DriverManager;
(2)import java.sql.ResultSet;
(3)import java.sql.SQLException;
(4)import com.mysql.jdbc.Connection;
(5)import com.mysql.jdbc.Statement;
(6)
(7)public class JDBCSelect6 {
(8)     public static void main(String[] args) {
(9)             try {
(10)                    Class.forName("com.mysql.jdbc.Driver");
(11)            } catch (ClassNotFoundException e) {
(12)                    System.err.println("Driver class not found");
(13)                    e.printStackTrace();
(14)            }
(15)            Connection con = null;
(16)
(17)            try {
(18)                    con =
(19)                            (Connection) DriverManager.getConnection(
(20)                                    "jdbc:mysql://localhost/jdbctest/",
(21)                                    "mario",
(22)                                    "thePassword");
(23)            } catch (SQLException sqle) {
(24)                    System.err.println("Error establishing database connection");
(25)                    sqle.printStackTrace();
(26)            }
(27)
(28)            Statement stmt = null;
(29)            try {
(30)                    stmt = (Statement) con.createStatement();
(31)            } catch (SQLException e2) {
(32)                    System.err.println("Error creating SQL-Statement");
(33)                    e2.printStackTrace();
(34)            }
(35)
(36)            try {
(37)                    int position = 0;
(38)                    ResultSet rs = stmt.executeQuery("SELECT * FROM EMPLOYEE;");
(39)                    rs.last();
(40)                    int size = rs.getRow();
(41)                    for (int i = 0; i < size; i++) {
(42)                            position = (position + 3) % size;
(43)                            rs.absolute(position + 1);
```

```
(44)                            System.out.println(
(45)                                    "position=" + (position + 1) + ": " + rs.
getString("FNAME"));
(46)                    }
(47)            } catch (SQLException sqle) {
(48)                    System.err.println("Error selecting values from table
EMPLOYEE");
(49)                    sqle.printStackTrace();
(50)            }
(51)    }
(52)}
```

Download des Beispiels
Download der Ergebnisdatei

Wird der benötigte `ResultSet` geeignet (d.h. mit den Parameter `CONCUR_UPDATABLE`) (siehe Zeile 49) initialisiert so können Änderungen, die im Hauptspeicher durch die JDBC-API durchgeführt werden in die Datenbank persistiert werden.
Beispiel 15 zeigt dies exemplarisch für den Einfügevorgang eines neuen Tupels.

Die Voraussetzungen für Einfüge- und Aktualisierungsvorgänge entstprechen denen von *updatable views*, d.h. die Daten dürfen nur aus genau einer Tabelle entnommen sein und müssen den Primärschlüssel enthalten.

Beispiel 15: Auslesen und Einfügen von Daten

```
(1)import java.sql.DriverManager;
(2)import java.sql.ResultSet;
(3)import java.sql.ResultSetMetaData;
(4)import java.sql.SQLException;
(5)import java.sql.Statement;
(6)
(7)import com.mysql.jdbc.Connection;
(8)
(9)public class JDBCSelect2 {
(10)    private static void printResultSet(ResultSet rs) throws SQLException {
(11)            ResultSetMetaData rsmd = rs.getMetaData();
(12)            int noColumns = rsmd.getColumnCount();
(13)            for (int i = 1; i < noColumns; i++) {
(14)                    System.out.print(rsmd.getColumnLabel(i) + "\t");
(15)            }
(16)            System.out.println();
(17)
(18)            while (rs.isLast() == false) {
(19)                    rs.next();
(20)                    for (int i = 1; i < noColumns; i++) {
(21)                            System.out.print( rs.getObject(i)+"\t" );
(22)                    }
(23)                    System.out.println();
(24)            }
(25)
(26)    }
(27)    public static void main(String[] args) {
(28)            try {
(29)                    Class.forName("com.mysql.jdbc.Driver");
(30)            } catch (ClassNotFoundException e) {
(31)                    System.err.println("Driver class not found");
(32)                    e.printStackTrace();
(33)            }
(34)            Connection con = null;
(35)
(36)            try {
(37)                    con =
(38)                            (Connection) DriverManager.getConnection(
(39)                                    "jdbc:mysql://localhost/jdbctest/",
(40)                                    "mario",
(41)                                    "thePassword");
(42)            } catch (SQLException e1) {
(43)                    System.err.println("Error establishing database connection");
(44)                    e1.printStackTrace();
(45)            }
```

```
(46)
(47)            Statement stmt = null;
(48)            try {
(49)                    stmt = (Statement) con.createStatement(ResultSet.
TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE);
(50)            } catch (SQLException e2) {
(51)                    System.err.println("Error creating SQL-Statement");
(52)                    e2.printStackTrace();
(53)            }
(54)
(55)            try {
(56)                    ResultSet uprs = (ResultSet) stmt.executeQuery("SELECT *
FROM EMPLOYEE;");
(57)                    printResultSet(uprs);
(58)                    uprs.moveToInsertRow();
(59)                    uprs.updateString("FNAME","Mario");
(60)                    uprs.updateString("LNAME","Jeckle");
(61)                    uprs.updateInt("SSN",111111111);
(62)                    uprs.insertRow();
(63)                    uprs = (ResultSet) stmt.executeQuery("SELECT * FROM
EMPLOYEE;");
(64)                    printResultSet(uprs);
(65)            } catch (SQLException e3) {
(66)                    System.err.println("Error selecting values from table
EMPLOYEE");
(67)                    e3.printStackTrace();
(68)            }
(69)    }
(70)}
```

Download des Beispiels
Download der Ergebnisdatei

Auf dieselbe Weise können auch Tupel einer Relation verändert werden. Hierzu stehen eine Reihe von updateXXX-Methoden zur Verfügung, wobei XXX für den Typ des zu aktualisierenden Attributs steht.
Nach durchgeführter Modifikation der hauptspeicherresidenten Werte werden diese durch updateRow in die Datenbank rückgeschrieben.
Beispiel 16 zeigt dies:

Beispiel 16: Modifizieren von Daten

```
(1)import java.sql.DriverManager;
(2)import java.sql.ResultSet;
(3)import java.sql.SQLException;
(4)import java.sql.Statement;
(5)
(6)import com.mysql.jdbc.Connection;
(7)
(8)public class JDBCSelect3 {
(9)     public static void main(String[] args) {
(10)            try {
(11)                    Class.forName("com.mysql.jdbc.Driver");
(12)            } catch (ClassNotFoundException e) {
(13)                    System.err.println("Driver class not found");
(14)                    e.printStackTrace();
(15)            }
(16)            Connection con = null;
(17)
(18)            try {
(19)                    con =
(20)                            (Connection) DriverManager.getConnection(
(21)                                    "jdbc:mysql://localhost/jdbctest/",
(22)                                    "mario",
(23)                                    "thePassword");
(24)            } catch (SQLException e1) {
(25)                    System.err.println("Error establishing database connection");
(26)                    e1.printStackTrace();
(27)            }
(28)
(29)            Statement stmt = null;
```

```
(30)            try {
(31)                    stmt =
(32)                            (Statement) con.createStatement(
(33)                                    ResultSet.TYPE_SCROLL_SENSITIVE,
(34)                                    ResultSet.CONCUR_UPDATABLE);
(35)            } catch (SQLException e2) {
(36)                    System.err.println("Error creating SQL-Statement");
(37)                    e2.printStackTrace();
(38)            }
(39)
(40)            try {
(41)                    ResultSet uprs =
(42)                            (ResultSet) stmt.executeQuery("SELECT * FROM
EMPLOYEE;");
(43)                    int namePos = uprs.findColumn("LNAME");
(44)
(45)                    while (uprs.isLast() == false) {
(46)                            uprs.next();
(47)                            if (uprs.getString(namePos).compareTo("Wallace") ==
0) {
(48)                                    uprs.updateString(namePos, "Doe");
(49)                                    uprs.updateRow();
(50)                            }
(51)                    }
(52)
(53)            } catch (SQLException e3) {
(54)                    System.err.println("Error selecting values from table
EMPLOYEE");
(55)                    e3.printStackTrace();
(56)            }
(57)    }
(58)}
```

Download des Beispiels

Analog vollzieht sich der Löschvorgang mittels `deleteRow`:

Beispiel 17: Löschen von Daten

```
(1)import java.sql.DriverManager;
(2)import java.sql.ResultSet;
(3)import java.sql.SQLException;
(4)import java.sql.Statement;
(5)
(6)import com.mysql.jdbc.Connection;
(7)
(8)public class JDBCSelect4 {
(9)     public static void main(String[] args) {
(10)            try {
(11)                    Class.forName("com.mysql.jdbc.Driver");
(12)            } catch (ClassNotFoundException e) {
(13)                    System.err.println("Driver class not found");
(14)                    e.printStackTrace();
(15)            }
(16)            Connection con = null;
(17)
(18)            try {
(19)                    con =
(20)                            (Connection) DriverManager.getConnection(
(21)                                    "jdbc:mysql://localhost/jdbctest/",
(22)                                    "mario",
(23)                                    "thePassword");
(24)            } catch (SQLException e1) {
(25)                    System.err.println("Error establishing database connection");
(26)                    e1.printStackTrace();
(27)            }
(28)
(29)            Statement stmt = null;
```

```
(30)            try {
(31)                    stmt =
(32)                            (Statement) con.createStatement(
(33)                                    ResultSet.TYPE_SCROLL_SENSITIVE,
```

```
(34)                                    ResultSet.CONCUR_UPDATABLE);
(35)            } catch (SQLException e2) {
(36)                    System.err.println("Error creating SQL-Statement");
(37)                    e2.printStackTrace();
(38)            }
(39)
(40)            try {
(41)                    ResultSet uprs =
(42)                            (ResultSet) stmt.executeQuery("SELECT * FROM
EMPLOYEE;");
(43)                    int namePos = uprs.findColumn("LNAME");
(44)
(45)                    while (uprs.isLast() == false) {
(46)                            uprs.next();
(47)                            if (uprs.getString(namePos).compareTo("Smith") == 0)
{
(48)                                    uprs.deleteRow();
(49)                            }
(50)                    }
(51)
(52)            } catch (SQLException e3) {
(53)                    System.err.println("Error selecting values from table
EMPLOYEE");
(54)                    e3.printStackTrace();
(55)            }
(56)    }
(57)}
```

Download des Beispiels

Die bisher betrachteten Varianten extrahieren Daten aus der Datenbank im Stile einer Momentaufnahme (*snapshot*) zum Zeitpunkt der Anfrage. Die einmal angefragten Inhalte können sich jedoch noch zur Laufzeit der zugreifenden JDBC-Applikation datenbankseitig ändern, wenn sie durch eine andere Applikation neu geschrieben werden. Zur Gewährleistung der Konsistenz des extrahierten Snapshots mit den tatsächlichen Datenbankinhalten steht die Operation `rowUpdated` zur Verfügung. Sie ermittelt ob der im Hauptspeicher befindliche Wert mit dem aktuellen Datenbankinhalt übereinstimmt, d.h. ob der DB-Inhalt aktualisiert wurde.
Beispiel 18 zeigt ein Umsetzungsbeispiel.

Beispiel 18: Test auf geänderte Daten

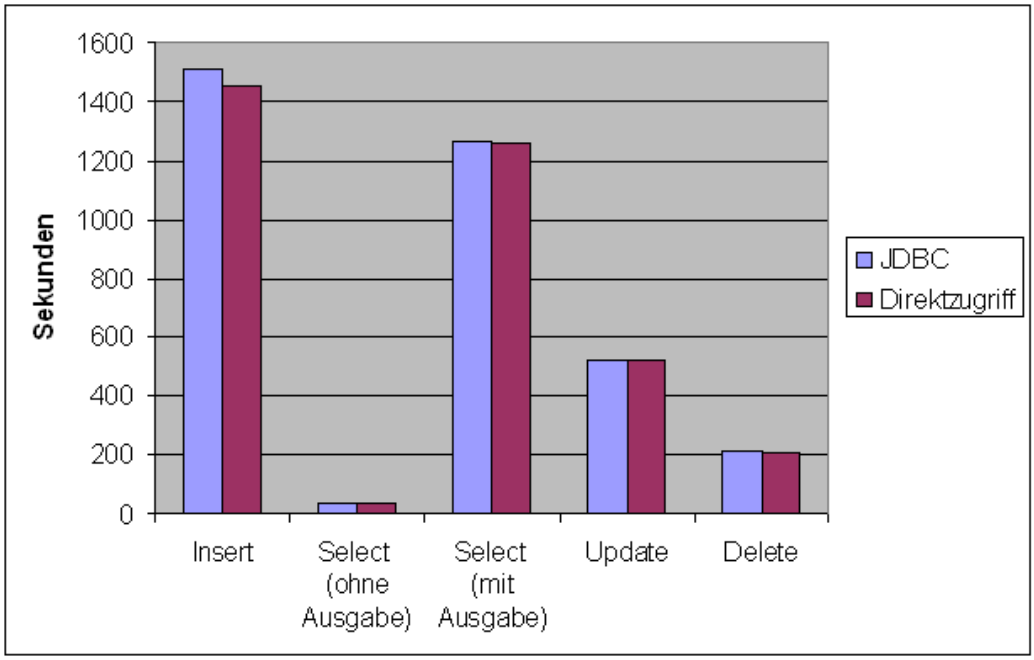
```
(1)import java.sql.DriverManager;
(2)import java.sql.ResultSet;
(3)import java.sql.SQLException;
(4)import com.mysql.jdbc.Connection;
(5)import com.mysql.jdbc.Statement;
(6)
(7)public class JDBCSelect7 {
(8)     public static void main(String[] args) {
(9)             try {
(10)                    Class.forName("com.mysql.jdbc.Driver");
(11)            } catch (ClassNotFoundException cnfe) {
(12)                    System.err.println("Driver class not found");
(13)                    cnfe.printStackTrace();
(14)            }
(15)            Connection con = null;
(16)
(17)            try {
(18)                    con =
(19)                            (Connection) DriverManager.getConnection(
(20)                                    "jdbc:mysql://localhost/jdbctest/",
(21)                                    "mario",
(22)                                    "thePassword");
(23)            } catch (SQLException sqle) {
(24)                    System.err.println("Error establishing database connection");
(25)                    sqle.printStackTrace();
(26)            }
(27)
(28)            Statement stmt = null;
(29)            try {
(30)                    stmt =
```

```
(31)                            (Statement) con.createStatement(
(32)                                    ResultSet.TYPE_SCROLL_SENSITIVE,
(33)                                    ResultSet.CONCUR_UPDATABLE);
(34)            } catch (SQLException sqle) {
(35)                    System.err.println("Error creating SQL-Statement");
(36)                    sqle.printStackTrace();
(37)            }
(38)
(39)            try {
(40)                    ResultSet rs = stmt.executeQuery("SELECT * FROM EMPLOYEE;");
(41)                    rs.absolute(5);
(42)                    System.out.println(rs.getString("FNAME"));
(43)
(44)                    System.out.println("sleeping ...");
(45)                    Thread.sleep(6000);
(46)                    System.out.println("awake ...");
(47)
(48)                    if (rs.rowUpdated() == true) {
(49)                            rs.refreshRow();
(50)                            System.out.println(rs.getString("FNAME"));
(51)                    }
(52)
(53)            } catch (SQLException sqle) {
(54)                    System.err.println("Error selecting values from table
EMPLOYEE");
(55)                    sqle.printStackTrace();
(56)            } catch (InterruptedException ie) {
(57)                    ie.printStackTrace();
(58)            }
(59)    }
(60)}
```

Download des Beispiels

Performancebetrachtungen

Abbildung 3: *JDBC-Geschwindigkeitsvergleich*



(click on image to enlarge!)

Die Abbildung zeigt die Ergebnisse einiger Geschwindigkeitsmessungen als Vergleich zwischen dem Zugriff auf eine MySQL-Datenbank unter Nutzung der Textschnittstelle und der Abwicklung derselben Zugriffe mittels JDBC.

Zur Messung wurde eine nicht-indexierte Datenbank mit 10^7 Einträgen verwendet die aus der Relation `tab` bestand. Deren Tupel wurden aus Paaren von 36-Byte großen UUIDs gemäß dem Spezifikationsentwurf der IETF gebildet.

Zur Zeitmessung wurden folgende Einzeloperationen betrachtet:

- **Insert**: `INSERT INTO tab VALUES(...)`
  Die Werte wurden unter Deaktivierung des Autocommit sequentiell eingefügt.
- **Select (ohne Ausgabe)**: `SELECT COUNT(*) FROM tab`
  Die ermittelte Gesamtzahl wurde nicht am Bildschirm ausgegeben.
- **Select (mit Ausgabe)**: `SELECT * FROM tab`
  Die Resultate wurden durch den Standarclient bzw. eine selbsterstellte (textmode) Java-Implementierung ausgegeben.
- **Update**: `UPDATE tab SET UUID1="X" WHERE UUID1<>"X"`
  Durch die Initialisierung der Werte mit UUID-Einträgen wird sichergestellt, daß alle Tupel aktualisiert werden, da sie in keinem Fall den Wert `X` enthalten.
- **Delete**: `DELETE FROM tab WHERE UUID2<>"X"`
  Durch die Initialisierung der Werte mit UUID-Einträgen wird sichergestellt, daß alle Tupel aktualisiert werden, da sie in keinem Fall den Wert `X` enthalten.

Insgesamt zeigt sich ein ausgewogenes Bild, in welchem der JDBC-Zugriff lediglich bei datenintensiven Zugriffen (große Mengen schreibender Zugriffe bei `INSERT` bzw. große Mengen lesender Operationen bei `SELECT`) im Bereich von fünf Prozent zurückliegt.

Diese enge Vergleichbarkeit der beiden Zugriffsmodi rührt von den Realisierung des eingesetzten JDBC-Treibers her; insbesondere von der Handhabung der physischen Datenbankverbindung auf Ebene des Netzwerkprotokolls.

SQL3-Datentypen

Die JDBC-API unterstützt mit Zugriffsmethoden auf die Datentypen `BLOB`, `CLOB`, `ARRAY`, `Object` und `Ref` bereits eine Untermenge des SQL:1999-Standards. So können, vorausgesetzt das durch JDBC angesprochene DBMS unterstützt dies große unstrukturierte Binär- oder Textdaten sowie einfache verschachtelte Tabellen, mithin NF^2-Strukturen verwaltet werden.

Beispiel 19 zeigt den Zugriff auf ein als eingebettete Tabelle realisiertes mengenwertiges Attribut. Die Beispieldatenbank wurde hierfür wie folgt modifiziert:

```
alter table EMPLOYEE ADD CAR SET('53M91','521R4', 'LLO415', 'XNU457');
update EMPLOYEE set CAR='XNU457' where SSN=123456789;
update EMPLOYEE set CAR='XNU457,521R4'  where SSN="999887777";
```

Beispiel 19: Zugriff auf ein mengenwertiges Attribut

```
(1)import java.sql.Array;
(2)import java.sql.DriverManager;
(3)import java.sql.ResultSet;
(4)import java.sql.SQLException;
(5)import com.mysql.jdbc.Connection;
(6)import com.mysql.jdbc.Statement;
(7)
(8)public class JDBCSelect8 {
(9)     public static void main(String[] args) {
(10)            try {
(11)                    Class.forName("com.mysql.jdbc.Driver");
(12)            } catch (ClassNotFoundException cnfe) {
(13)                    System.err.println("Driver class not found");
(14)                    cnfe.printStackTrace();
(15)            }
(16)            Connection con = null;
(17)
(18)            try {
(19)                    con =
(20)                            (Connection) DriverManager.getConnection(
(21)                                    "jdbc:mysql://localhost/jdbctest/",
(22)                                    "mario",
(23)                                    "thePassword");
(24)            } catch (SQLException sqle) {
(25)                    System.err.println("Error establishing database connection");
(26)                    sqle.printStackTrace();
(27)            }
(28)
(29)            Statement stmt = null;
(30)            try {
(31)                    stmt = (Statement) con.createStatement();
```

```
(32)            } catch (SQLException sqle) {
(33)                    System.err.println("Error creating SQL-Statement");
(34)                    sqle.printStackTrace();
(35)            }
(36)
(37)            try {
(38)                    ResultSet rs = stmt.executeQuery("SELECT * FROM EMPLOYEE;");
(39)                    while (!rs.isLast()) {
(40)                            rs.first();
(41)                            System.out.print(rs.getString("FNAME") + "\t");
(42)                            Array cars = rs.getArray("CAR");
(43)                            ResultSet carsRS = cars.getResultSet();
(44)                            System.out.print("(");
(45)                            while (!carsRS.isLast()) {
(46)                                    rs.first();
(47)                                    System.out.print(carsRS.getString("CAR"));
(48)                                    carsRS.next();
(49)                            }
(50)                            System.out.println(")");
(51)                            rs.next();
(52)                    }
(53)            } catch (SQLException sqle) {
(54)                    System.err.println("Error selecting values from table
EMPLOYEE");
(55)                    sqle.printStackTrace();
(56)            }
(57)    }
(58)}
```

Download des Beispiels

Das Beispiel unterstreicht die Rolle der mengenwertigen Attribute als eingebettete Tabellen. So erfolgt der Zugriff auf die Einzelwerte des Attributs `CAR` identisch zur Ermittlung der Resultatmenge der SQL-Anfrage mittels `getResultSet`. Auch die Traversierung der einzelnen `CAR`-Elemente erfolgt äquivalent.

Die Aufnahme der *large objects* in ihrer Ausprägungsform als *Character Large Objects* (CLOB) oder *Binary Large Objects* (BLOB) stellen eine der zentralen Erweiterungen des SQL:1999-Standards gegenüber seinen Vorgängern dar.
Zwar ist die Ablage großer unstrukturierter Datenobjekte in relationalen Datenbanken konzeptionell durchaus diskussionswert, jedoch in der Praxis oftmals, trotz der teilweise erheblichen Geschwindigkeitseinbußen im Zugriff (so benötigt die Ausführung der Beispielapplikation mit einem 10^6 Byte großen Datenstrom 1,1 Sekunden, während dieselbe Operation dateisystembasiert in 0,1 Sekunde abläuft), gewünscht.
Beispiel 20 zeigt die notwendigen Schritte zur Ablage und erneuten Auslese eines aus einer Datei gewonnen Binärdatenstroms in der Datenbank.
Die Beispieldatenbank wurde hierfür um ein Attribut zur Aufnahme binärer Daten erweitert:
`ALTER TABLE EMPLOYEE ADD binData blob;`

Beispiel 20: Verarbeitung unstrukturierter Binärdaten

```
(1)import java.io.File;
(2)import java.io.FileInputStream;
(3)import java.io.FileOutputStream;
(4)import java.io.IOException;
(5)import java.sql.DriverManager;
(6)import java.sql.PreparedStatement;
(7)import java.sql.ResultSet;
(8)import java.sql.SQLException;
(9)
(10)import com.mysql.jdbc.Connection;
(11)import com.mysql.jdbc.Statement;
(12)
(13)public class JDBCSelect9 {
(14)    public static void main(String[] args) {
(15)            try {
(16)                    Class.forName("com.mysql.jdbc.Driver");
(17)            } catch (ClassNotFoundException cnfe) {
(18)                    System.err.println("Driver class not found");
(19)                    cnfe.printStackTrace();
(20)            }
```

```
(21)            Connection con = null;
(22)
(23)            try {
(24)                    con =
(25)                            (Connection) DriverManager.getConnection(
(26)                                    "jdbc:mysql://localhost/jdbctest/",
(27)                                    "mario",
(28)                                    "thePassword");
(29)            } catch (SQLException sqle) {
(30)                    System.err.println("Error establishing database connection");
(31)                    sqle.printStackTrace();
(32)            }
(33)
(34)            try {
(35)                    File file = new File(args[0]);
(36)                    FileInputStream fis = new FileInputStream(args[0]);
(37)                    PreparedStatement pstmt =
(38)                            con.prepareStatement(
(39)                                    "UPDATE EMPLOYEE SET binData =? WHERE
SSN=123456789");
(40)                    pstmt.setBinaryStream(1, fis, (int) file.length());
(41)                    pstmt.executeUpdate();
(42)                    fis.close();
(43)
(44)                    //read it back from the database
(45)                    Statement stmt = (Statement) con.createStatement();
(46)                    ResultSet rs =
(47)                            stmt.executeQuery(
(48)                                    "SELECT binData FROM EMPLOYEE WHERE
SSN='123456789';");
(49)
(50)                    FileOutputStream fos = new FileOutputStream(args[1]);
(51)                    if (rs.next())
(52)                            fos.write(rs.getBytes(1));
(53)                    fos.close();
(54)
(55)            } catch (SQLException sqle) {
(56)                    System.err.println("Error selecting values from table
EMPLOYEE");
(57)                    sqle.printStackTrace();
(58)            } catch (IOException ioe) {
(59)                    ioe.printStackTrace();
(60)            }
(61)    }
(62)}
```

Download des Beispiels

Die Binärdaten können naturgemäß nicht direkt in die SQL-UPDATE-Anweisung eingebunden werden, sie werden daher einer mittels prepareStatement vorerzeugten Anweisung durch Aufruf der Methode setBinaryStream übergeben.

Transaktionssteuerung

Zur Steuerung des transaktionalen Verhaltens einer JDBC-Anfrage bietet die Klasse Connection verschiedene Methoden an:

- Abfrage der aktuellen Isolationsstufe: getIsolationLevel.
  Hierbei werden fünf Stufen unterschieden:
  - TRANSACTION_NONE: Keinerlei Transaktionsunterstützung
  - TRANSACTION_READ_UNCOMMITTED: Auch nicht durch commit freigegebene Daten werden gelesen. Es können daher *dirty reads*, Nicht-wiederholbare- und Phantomlesevorgänge auftreten.
  - TRANSACTION_READ_COMMITTED: Nur durch commit freigegebene Daten werden gelesen. nichtwiederholbare- und Phantomlesevorgänge können jedoch auftreten.
  - TRANSACTION_REPEATABLE_READ: Innerhalb einer Transaktion können die verarbeiteten Daten nicht durch eine andere Transaktion verändert werden. Das Auftreten von *dirty reads* und nichtwiederholbaren Lesevorgängen ist daher ausgeschlossen, Phantomlesevorgänge sind jedoch weiterhin möglich.
  - TRANSACTION_SERIALIZABLE: Strikte Isolation aller Transaktionen, auf dieser Stufe sind auch

Phantomlesevorgänge ausgeschlossen.
Jede Stufe geht zwar mit einer gesteigerten Qualität der durch eine Transaktion verarbeiteten Daten einher, jedoch senkt gleichzeitig eine strikterere Isolationsstufe die Anzahl der gleichzeitigen Zugriffe auf die Datenbank und damit die Gesamtsystemperformance.

- Setzen der Isolationsstufe: `setTransactionIsolation`
- (De-)Aktivierung der automatischen Freigabe: `setAutoCommit`. Die Übergabe von `true` bewirkt die Aktivierung des Modus bei dem jede Einzelanweisung sofort persistent übernommen und für andere Transaktionen sichtbar wird.
  Standardmäßig ist diese Option aktiviert. Ihr aktueller Zustand kann per `getAutoCommit` ermittelt werden.
- Freigabe von Änderungen: `commit`.
- Rücknahme von Änderungen: `rollback`.
- Rücknahme von Änderungen bis zu definiertem Sicherungspunkt: `rollback(Savepoint s)`.
- Setzen eines Sicherungspunktes: `setSavepoint`.

Beispiel 1 zeigt die Nutzung des Transaktionskonzepts.
Zunächst wird die aktuelle Isolationsstufe ermittelt und geprüft ob das angesprochene DBMS die höchste durch JDBC vorhergesehene Isolationsstufe unterstützt.
Nach Abschaltung der automatischen Änderungsübernahme (`setAutoCommit(false)`) werden zunächst zwei Tupel in die Tabelle `EMPLOYEE` eingefügt, die jedoch nur innerhalb der laufenden Transaktion sichtbar werden, für alle anderen Transaktionen innerhalb des DBMS bleiben die neuen Werte (zunächst) unsichtbar.
Eine angenommene Fehlersituation führt zum Rücksetzen der Transaktion durch (`rollback`).
Nach Abschluß des Programms wurden zwar die beiden ersten Werte lokal in die Datenbank übernommen, aber noch innerhalb der laufenden Transaktion wieder daraus entfernt, weshalb sie zu keinem Zeitpunkt für andere Datenbankbenutzer sichtbar waren.

Beispiel 21: Transaktionsverarbeitung

```
(1)import java.sql.DriverManager;
(2)import java.sql.ResultSet;
(3)import java.sql.SQLException;
(4)
(5)import com.mysql.jdbc.Connection;
(6)import com.mysql.jdbc.Statement;
(7)
(8)public class JDBCTransact1 {
(9)     private static void printContent(Statement stmt) throws SQLException {
(10)            ResultSet rs =
(11)                    stmt.executeQuery("SELECT FNAME,MINIT,LNAME FROM EMPLOYEE;");
(12)            while (!rs.isLast()) {
(13)                    rs.next();
(14)                    System.out.println(
(15)                            rs.getString("LNAME")
(16)                                    + "\t"
(17)                                    + rs.getString("MINIT")
(18)                                    + "\t"
(19)                                    + rs.getString("LNAME"));
(20)            }
(21)    }
(22)    public static void main(String[] args) {
(23)            try {
(24)                    Class.forName("com.mysql.jdbc.Driver");
(25)            } catch (ClassNotFoundException cnfe) {
(26)                    System.err.println("Driver class not found");
(27)                    cnfe.printStackTrace();
(28)            }
(29)            Connection con = null;
(30)
(31)            try {
(32)                    con =
(33)                            (Connection) DriverManager.getConnection(
(34)                                    "jdbc:mysql://localhost/jdbctest/",
(35)                                    "mario",
(36)                                    "thePassword");
(37)            } catch (SQLException sqle) {
(38)                    System.err.println("Error establishing database connection");
(39)                    sqle.printStackTrace();
(40)            }
(41)
(42)            Statement stmt = null;
(43)            try {
```

```
(44)                    stmt = (Statement) con.createStatement();
(45)            } catch (SQLException sqle) {
(46)                    System.err.println("Error creating SQL-Statement");
(47)                    sqle.printStackTrace();
(48)            }
(49)
(50)            try {
(51)                    int transactionIsolation = con.getTransactionIsolation();
(52)                    switch (transactionIsolation) {
(53)                            case Connection.TRANSACTION_NONE :
(54)                                    System.out.println("Transactions are not
supported");
(55)                                    break;
(56)                            case Connection.TRANSACTION_READ_UNCOMMITTED :
(57)                                    System.out.println(
(58)                                            "Dirty reads, non-repeatable reads
and phantom reads can occur");
(59)                                    break;
(60)                            case Connection.TRANSACTION_READ_COMMITTED :
(61)                                    System.out.println(
(62)                                            "Dirty reads are prevented; non-
repeatable reads and phantom reads can occur");
(63)                                    break;
(64)                            case Connection.TRANSACTION_REPEATABLE_READ :
(65)                                    System.out.println(
(66)                                            "Dirty reads and non-repeatable
reads are prevented; phantom reads can occur");
(67)                                    break;
(68)                            case Connection.TRANSACTION_SERIALIZABLE :
(69)                                    System.out.println(
(70)                                            "Dirty reads, non-repeatable reads
and phantom reads are prevented");
(71)                                    break;
(72)                    }
(73)                    if (transactionIsolation < Connection.
TRANSACTION_SERIALIZABLE) {
(74)                            con.setTransactionIsolation(
(75)                                    Connection.TRANSACTION_SERIALIZABLE);
(76)                            if (con.getTransactionIsolation()
(77)                                    != Connection.TRANSACTION_SERIALIZABLE) {
(78)                                    System.out.println(
(79)                                            "cannot set Connection.
TRANSACTION_SERIALIZABLE");
(80)                            } else {
(81)                                    System.out.println(
(82)                                            "reached highest possible isolation
level");
(83)                            }
(84)                    }
(85)
(86)                    con.setAutoCommit(false);
(87)                    stmt.executeUpdate(
(88)                            "INSERT INTO EMPLOYEE VALUES
('Hans','X','Hinterhuber','111111111',NULL,NULL,NULL,NULL,NULL,NULL);");
(89)                    stmt.executeUpdate(
(90)                            "INSERT INTO EMPLOYEE VALUES
('Franz','X','Obermüller','222222222',NULL,NULL,NULL,NULL,NULL,NULL);");
(91)                    printContent(stmt);
(92)                    //suppose error happens here
(93)                    Thread.sleep(5000);
(94)                    boolean error = true;
(95)                    if (error) {
(96)                            con.rollback();
(97)                    } else {
(98)                            stmt.executeUpdate(
(99)                                    "INSERT INTO EMPLOYEE VALUES
('Fritz','X','Meier','333333333',NULL,NULL,NULL,NULL,NULL,NULL);");
(100)                   }
(101)                   printContent(stmt);
(102)           } catch (SQLException sqle) {
(103)                   sqle.printStackTrace();
(104)           } catch (InterruptedException ie) {
```

```
(105)                   ie.printStackTrace();
(106)           }
(107)   }
(108)}
```

Download des Beispiels
Download der Ergebnisdatei

Neben dem Zurücksetzen einer vollständigen Transaktion bietet die JDBC-API auch die Möglichkeit alle Schritte bis zu einem anwenderdefierten aus der Datenbank zu entfernen.
Beispiel 22 zeigt dies unter Verwendung der Methode `setSavepoint` zur Definition eines Sicherungspunktes und `rollback(sp)` zum Zurücksetzen bis zu diesem Sicherungspunkt.

Beispiel 22: Transaktionsverarbeitung mit Sicherungspunkten

```
(1)import java.sql.DriverManager;
(2)import java.sql.ResultSet;
(3)import java.sql.SQLException;
(4)import java.sql.Savepoint;
(5)
(6)import com.mysql.jdbc.Connection;
(7)import com.mysql.jdbc.Statement;
(8)
(9)public class JDBCTransact2 {
(10)    private static void printContent(Statement stmt) throws SQLException {
(11)            ResultSet rs =
(12)                    stmt.executeQuery("SELECT FNAME,MINIT,LNAME FROM EMPLOYEE;");
(13)            while (!rs.isLast()) {
(14)                    rs.next();
(15)                    System.out.println(
(16)                            rs.getString("LNAME")
(17)                                    + "\t"
(18)                                    + rs.getString("MINIT")
(19)                                    + "\t"
(20)                                    + rs.getString("LNAME"));
(21)            }
(22)    }
(23)    public static void main(String[] args) {
(24)            try {
(25)                    Class.forName("com.mysql.jdbc.Driver");
(26)            } catch (ClassNotFoundException cnfe) {
(27)                    System.err.println("Driver class not found");
(28)                    cnfe.printStackTrace();
(29)            }
(30)            Connection con = null;
(31)
(32)            try {
(33)                    con =
(34)                            (Connection) DriverManager.getConnection(
(35)                                    "jdbc:mysql://localhost/jdbctest/",
(36)                                    "mario",
(37)                                    "thePassword");
(38)            } catch (SQLException sqle) {
(39)                    System.err.println("Error establishing database connection");
(40)                    sqle.printStackTrace();
(41)            }
(42)
(43)            Statement stmt = null;
(44)            try {
(45)                    stmt = (Statement) con.createStatement();
(46)            } catch (SQLException sqle) {
(47)                    System.err.println("Error creating SQL-Statement");
(48)                    sqle.printStackTrace();
(49)            }
(50)
(51)            try {
(52)                    int transactionIsolation = con.getTransactionIsolation();
(53)                    switch (transactionIsolation) {
(54)                            case Connection.TRANSACTION_NONE :
(55)                                    System.out.println("Transactions are not
supported");
(56)                                    break;
```

```
(57)                            case Connection.TRANSACTION_READ_UNCOMMITTED :
(58)                                    System.out.println(
(59)                                            "Dirty reads, non-repeatable reads
and phantom reads can occur");
(60)                                    break;
(61)                            case Connection.TRANSACTION_READ_COMMITTED :
(62)                                    System.out.println(
(63)                                            "Dirty reads are prevented; non-
repeatable reads and phantom reads can occur");
(64)                                    break;
(65)                            case Connection.TRANSACTION_REPEATABLE_READ :
(66)                                    System.out.println(
(67)                                            "Dirty reads and non-repeatable
reads are prevented; phantom reads can occur");
(68)                                    break;
(69)                            case Connection.TRANSACTION_SERIALIZABLE :
(70)                                    System.out.println(
(71)                                            "Dirty reads, non-repeatable reads
and phantom reads are prevented");
(72)                                    break;
(73)                    }
(74)                    if (transactionIsolation < Connection.
TRANSACTION_SERIALIZABLE) {
(75)                            con.setTransactionIsolation(
(76)                                    Connection.TRANSACTION_SERIALIZABLE);
(77)                            if (con.getTransactionIsolation()
(78)                                    != Connection.TRANSACTION_SERIALIZABLE) {
(79)                                    System.out.println(
(80)                                            "cannot set Connection.
TRANSACTION_SERIALIZABLE");
(81)                            } else {
(82)                                    System.out.println(
(83)                                            "reached highest possible isolation
level");
(84)                            }
(85)                    }
(86)
(87)                    con.setAutoCommit(false);
(88)                    stmt.executeUpdate(
(89)                            "INSERT INTO EMPLOYEE VALUES
('Hans','X','Hinterhuber','111111111',NULL,NULL,NULL,NULL,NULL,NULL);");
(90)                    Savepoint sp = con.setSavepoint();
(91)                    stmt.executeUpdate(
(92)                            "INSERT INTO EMPLOYEE VALUES
('Franz','X','Obermüller','222222222',NULL,NULL,NULL,NULL,NULL,NULL);");
(93)                    printContent(stmt);
(94)                    //suppose error happens here
(95)                    Thread.sleep(5000);
(96)                    boolean error = true;
(97)                    if (error) {
(98)                            con.rollback(sp);
(99)                    }
(100)                   stmt.executeUpdate(
(101)                           "INSERT INTO EMPLOYEE VALUES
('Fritz','X','Meier','333333333',NULL,NULL,NULL,NULL,NULL,NULL);");
(102)                   printContent(stmt);
(103)                   con.commit();
(104)           } catch (SQLException sqle) {
(105)                   sqle.printStackTrace();
(106)           } catch (InterruptedException ie) {
(107)                   ie.printStackTrace();
(108)           }
(109)   }
(110)}
```

Download des Beispiels

Netzwerkverkehr

Der Netzwerkmitschnitt zeigt den TCP-Kommunikationsverlauf der SQL-Anfrage `SELECT * FROM`

EMPLOYEE;.
Die vom Anfrager an den MySQL-Server übermittelten Datenanteile sind rot hervorgehoben.
Zusätzlich sind die für eine gleichwertige native Kommunikation anfallenden Daten berücksichtigt.
Ihre Anteile entsprechen den zusätzlich durch Fettdruck hervorgehobenen.

```
0000 4  00 00 00 \n 4  .  0  .  1  2  -  s  t  a  n  d  a  r  d  -  l  o  g  00 '  00
00 00 N  4  V
0020 5  G  D  a  9  00 ,     \b 02 00 00 00 00 00 00 00 00 00 00 00 00 00 14 00 00
01 87 00 ÿ  ÿ
0040 ÿ  m  a  r  i  o  00 N  W  N  J  S  ]  L  K  00 03 00 00 02 00 00 00 \t 00 00 00
02 j  d  b  c
0060 t  e  s  t  03 00 00 01 00 00 00 0f 00 00 00 03 S  H  O  W     V  A  R  I  A  B
L  E  S  01 00
0080 00 01 02 19 00 00 02 00 \r V  a  r  i  a  b  l  e  _  n  a  m  e  03 1e 00 00 01
þ  03 01 00 1f
00a0 11 00 00 03 00 05 V  a  l  u  e  03 00 01 00 01 þ  03 01 00 1f 01 00 00 04 þ  \f
00 00 05 \b b
00c0 a  c  k  _  l  o  g  02 5  0  7  00 00 06 07 b  a  s  e  d  i  r  .  /  o  p
t  /  r  a  i  d
00e0 /  m  y  s  q  l  -  s  t  a  n  d  a  r  d  -  4  .  0  .  1  2  -  p  c  -  l
i  n  u  x  -
0100 i  6  8  6  /  18 00 00 07 11 b  i  n  l  o  g  _  c  a  c  h  e  _  s  i  z  e
05 3  2  7  6
0120 8     00 00 \b 17 b  u  l  k  _  i  n  s  e  r  t  _  b  u  f  f  e  r  _  s  i
z  e  07 8  3
0140 8  8  6  0  8  15 00 00 \t \r c  h  a  r  a  c  t  e  r  _  s  e  t  06 l  a  t
i  n  1  á  00
0160 00 \n 0e c  h  a  r  a  c  t  e  r  _  s  e  t  s  Ñ  l  a  t  i  n  1     b  i
g  5     c  z
0180 e  c  h     e  u  c  _  k  r     g  b  2  3  1  2     g  b  k     l  a  t  i  n
1  _  d  e
01a0 s  j  i  s     t  i  s  6  2  0     u  j  i  s     d  e  c  8     d  o  s     g
e  r  m  a  n
01c0 1     h  p  8     k  o  i  8  _  r  u     l  a  t  i  n  2     s  w  e  7     u
s  a  7     c
01e0 p  1  2  5  1     d  a  n  i  s  h     h  e  b  r  e  w     w  i  n  1  2  5
1     e  s  t  o
0200 n  i  a     h  u  n  g  a  r  i  a  n     k  o  i  8  _  u  k  r     w  i  n  1
2  5  1  u  k
0220 r     g  r  e  e  k     w  i  n  1  2  5  0     c  r  o  a  t     c  p  1  2  5
7     l  a  t
0240 i  n  5  15 00 00 0b 11 c  o  n  c  u  r  r  e  n  t  _  i  n  s  e  r  t  02 O
N  12 00 00 \f
0260 0f c  o  n  n  e  c  t  _  t  i  m  e  o  u  t  01 5  17 00 00 \r 15 c  o  n  v
e  r  t  _  c
0280 h  a  r  a  c  t  e  r  _  s  e  t  00 <  00 00 0e 07 d  a  t  a  d  i  r  3  /
o  p  t  /  r
02a0 a  i  d  /  m  y  s  q  l  -  s  t  a  n  d  a  r  d  -  4  .  0  .  1  2  -  p
c  -  l  i  n
02c0 u  x  -  i  6  8  6  /  d  a  t  a  /  13 00 00 0f 0f d  e  l  a  y  _  k  e  y
_  w  r  i  t
02e0 e  02 O  N  19 00 00 10 14 d  e  l  a  y  e  d  _  i  n  s  e  r  t  _  l  i  m
i  t  03 1  0
0300 0  1b 00 00 11 16 d  e  l  a  y  e  d  _  i  n  s  e  r  t  _  t  i  m  e  o  u
t  03 3  0  0
0320 18 00 00 12 12 d  e  l  a  y  e  d  _  q  u  e  u  e  _  s  i  z  e  04 1  0  0
0  \n 00 00 13
0340 05 f  l  u  s  h  03 O  F  F  \r 00 00 14 \n f  l  u  s  h  _  t  i  m  e  01
0  !  00 00 15 11
0360 f  t  _  b  o  o  l  e  a  n  _  s  y  n  t  a  x  0e +     -  >  <  (  )  ~
*  :  "  "  &  |
0380 12 00 00 16 0f f  t  _  m  i  n  _  w  o  r  d  _  l  e  n  01 4  14 00 00 17 0f
f  t  _  m  a
03a0 x  _  w  o  r  d  _  l  e  n  03 2  5  4  1c 00 00 18 18 f  t  _  m  a  x  _  w
o  r  d  _  l
03c0 e  n  _  f  o  r  _  s  o  r  t  02 2  0  1c 00 00 19 10 f  t  _  s  t  o  p  w
o  r  d  _  f
03e0 i  l  e  \n (  b  u  i  l  t  -  i  n  )  \f 00 00 1a \b h  a  v  e  _  b  d  b
02 N  O  0f 00
0400 00 1b \n h  a  v  e  _  c  r  y  p  t  03 Y  E  S  10 00 00 1c 0b h  a  v  e  _
i  n  n  o  d
```

```
0420 b  03 Y  E  S  0e 00 00 1d \t h  a  v  e  _  i  s  a  m  03 Y  E  S  \r 00 00 1e
\t h  a  v  e
0440 _  r  a  i  d  02 N  O  16 00 00 1f \f h  a  v  e  _  s  y  m  l  i  n  k  \b D
I  S  A  B  L
0460 E  D  10 00 00    \f h  a  v  e  _  o  p  e  n  s  s  l  02 N  O  15 00 00 !  10
h  a  v  e  _
0480 q  u  e  r  y  _  c  a  c  h  e  03 Y  E  S  0b 00 00 "  \t i  n  i  t  _  f  i
l  e  00 (  00
04a0 00 #  1f i  n  n  o  d  b  _  a  d  d  i  t  i  o  n  a  l  _  m  e  m  _  p  o
o  l  _  s  i
04c0 z  e  07 2  0  9  7  1  5  2  !  00 00 $  17 i  n  n  o  d  b  _  b  u  f  f  e
r  _  p  o  o
04e0 l  _  s  i  z  e  \b 1  6  7  7  7  2  1  6  -  00 00 %  15 i  n  n  o  d  b  _
d  a  t  a  _
0500 f  i  l  e  _  p  a  t  h  16 i  b  d  a  t  a  1  :  1  0  M  :  a  u  t  o  e
x  t  e  n  d
0520 16 00 00 &  14 i  n  n  o  d  b  _  d  a  t  a  _  h  o  m  e  _  d  i  r  00 19
00 00 '  16 i
0540 n  n  o  d  b  _  f  i  l  e  _  i  o  _  t  h  r  e  a  d  s  01 4  18 00 00
(  15 i  n  n  o
0560 d  b  _  f  o  r  c  e  _  r  e  c  o  v  e  r  y  01 0  1c 00 00 )  19 i  n  n
o  d  b  _  t
0580 h  r  e  a  d  _  c  o  n  c  u  r  r  e  n  c  y  01 8  !  00 00 *  1e i  n  n
o  d  b  _  f
05a0 l  u  s  h  _  l  o  g  _  a  t  _  t  r  x  _  c  o  m  m  i  t  01 1  18 00 00
+  14 i  n  n
05c0 o  d  b  _  f  a  s  t  _  s  h  u  t  d  o  w  n  02 O  N  15 00 00 ,  13 i  n
n  o  d  b  _
05e0 f  l  u  s  h  _  m  e  t  h  o  d  00 1c 00 00 -  18 i  n  n  o  d  b  _  l  o
c  k  _  w  a
0600 i  t  _  t  i  m  e  o  u  t  02 5  0  17 00 00 .  13 i  n  n  o  d  b  _  l  o
g  _  a  r  c
0620 h  _  d  i  r  02 .  /  17 00 00 /  12 i  n  n  o  d  b  _  l  o  g  _  a  r  c
h  i  v  e  03
0640 O  F  F  1f 00 00 0  16 i  n  n  o  d  b  _  l  o  g  _  b  u  f  f  e  r  _  s
i  z  e  07 8
0660 3  8  8  6  0  8  1d 00 00 1  14 i  n  n  o  d  b  _  l  o  g  _  f  i  l  e  _
s  i  z  e  07
0680 5  2  4  2  8  8  0  1c 00 00 2  19 i  n  n  o  d  b  _  l  o  g  _  f  i  l  e
s  _  i  n  _
06a0 g  r  o  u  p  01 2  1d 00 00 3  19 i  n  n  o  d  b  _  l  o  g  _  g  r  o  u
p  _  h  o  m
06c0 e  _  d  i  r  02 .  /  1d 00 00 4  1a i  n  n  o  d  b  _  m  i  r  r  o  r  e
d  _  l  o  g
06e0 _  g  r  o  u  p  s  01 1  1a 00 00 5  13 i  n  t  e  r  a  c  t  i  v  e  _  t
i  m  e  o  u
0700 t  05 2  8  8  0  0  18 00 00 6  10 j  o  i  n  _  b  u  f  f  e  r  _  s  i  z
e  06 1  3  1
0720 0  7  2  19 00 00 7  0f k  e  y  _  b  u  f  f  e  r  _  s  i  z  e  \b 1  6  7
7  7  2  1  6
0740 L  00 00 8  \b l  a  n  g  u  a  g  e  B  /  o  p  t  /  r  a  i  d  /  m  y  s
q  l  -  s  t
0760 a  n  d  a  r  d  -  4  .  0  .  1  2  -  p  c  -  l  i  n  u  x  -  i  6  8
6  /  s  h  a  r
0780 e  /  m  y  s  q  l  /  e  n  g  l  i  s  h  /  17 00 00 9  13 l  a  r  g  e  _
f  i  l  e  s
07a0 _  s  u  p  p  o  r  t  02 O  N  10 00 00 :  \f l  o  c  a  l  _  i  n  f  i  l
e  02 O  N  15
07c0 00 00 ;  10 l  o  c  k  e  d  _  i  n  _  m  e  m  o  r  y  03 O  F  F  \b 00 00
<  03 l  o  g
07e0 03 O  F  F  0f 00 00 =  \n l  o  g  _  u  p  d  a  t  e  03 O  F  F  0b 00 00 >
07 l  o  g  _
0800 b  i  n  02 O  N  16 00 00 ?  11 l  o  g  _  s  l  a  v  e  _  u  p  d  a  t  e
s  03 O  F  F
0820 15 00 00 @  10 l  o  g  _  s  l  o  w  _  q  u  e  r  i  e  s  03 O  F  F  11 00
00 A  \f l  o
0840 g  _  w  a  r  n  i  n  g  s  03 O  F  F  13 00 00 B  0f l  o  n  g  _  q  u  e
r  y  _  t  i
0860 m  e  02 1  0  19 00 00 C  14 l  o  w  _  p  r  i  o  r  i  t  y  _  u  p  d  a
t  e  s  03 O
0880 F  F  1b 00 00 D  16 l  o  w  e  r  _  c  a  s  e  _  t  a  b  l  e  _  n  a  m
e  s  03 O  F
```

```
08a0 F  1b 00 00 E  12 m  a  x  _  a  l  l  o  w  e  d  _  p  a  c  k  e  t  07 1  0
4  7  5  5  2
08c0 !  00 00 F  15 m  a  x  _  b  i  n  l  o  g  _  c  a  c  h  e  _  s  i  z  e  \n
4  2  9  4  9
08e0 6  7  2  9  5  1b 00 00 G  0f m  a  x  _  b  i  n  l  o  g  _  s  i  z  e  \n 1
0  7  3  7  4
0900 1  8  2  4  14 00 00 H  0f m  a  x  _  c  o  n  n  e  c  t  i  o  n  s  03 1  0
0  16 00 00 I
0920 12 m  a  x  _  c  o  n  n  e  c  t  _  e  r  r  o  r  s  02 1  0  17 00 00 J  13
m  a  x  _  d
0940 e  l  a  y  e  d  _  t  h  r  e  a  d  s  02 2  0  1d 00 00 K  13 m  a  x  _  h
e  a  p  _  t
0960 a  b  l  e  _  s  i  z  e  \b 1  6  7  7  7  2  1  6  19 00 00 L  \r m  a  x  _
j  o  i  n  _
0980 s  i  z  e  \n 4  2  9  4  9  6  7  2  9  5  15 00 00 M  0f m  a  x  _  s  o  r
t  _  l  e  n
09a0 g  t  h  04 1  0  2  4  17 00 00 N  14 m  a  x  _  u  s  e  r  _  c  o  n  n  e
c  t  i  o  n
09c0 s  01 0  12 00 00 O  0e m  a  x  _  t  m  p  _  t  a  b  l  e  s  02 3  2     00
00 P  14 m  a
09e0 x  _  w  r  i  t  e  _  l  o  c  k  _  c  o  u  n  t  \n 4  2  9  4  9  6  7  2
9  5  *  00 00
0a00 Q  1f m  y  i  s  a  m  _  m  a  x  _  e  x  t  r  a  _  s  o  r  t  _  f  i  l
e  _  s  i  z
0a20 e  \t 2  6  8  4  3  5  4  5  6  %  00 00 R  19 m  y  i  s  a  m  _  m  a  x  _
s  o  r  t  _
0a40 f  i  l  e  _  s  i  z  e  \n 2  1  4  7  4  8  3  6  4  7  1b 00 00 S  16 m  y
i  s  a  m  _
0a60 r  e  c  o  v  e  r  _  o  p  t  i  o  n  s  03 O  F  F     00 00 T  17 m  y  i
s  a  m  _  s
0a80 o  r  t  _  b  u  f  f  e  r  _  s  i  z  e  07 8  3  8  8  6  0  8  17 00 00 U
11 n  e  t  _
0aa0 b  u  f  f  e  r  _  l  e  n  g  t  h  04 8  1  9  2  14 00 00 V  10 n  e  t  _
r  e  a  d  _
0ac0 t  i  m  e  o  u  t  02 3  0  13 00 00 W  0f n  e  t  _  r  e  t  r  y  _  c  o
u  n  t  02 1
0ae0 0  15 00 00 X  11 n  e  t  _  w  r  i  t  e  _  t  i  m  e  o  u  t  02 6  0  \b
00 00 Y  03 n
0b00 e  w  03 O  F  F  13 00 00 Z  10 o  p  e  n  _  f  i  l  e  s  _  l  i  m  i  t
01 0  F  00 00
0b20 [  \b p  i  d  _  f  i  l  e  <  /  o  p  t  /  r  a  i  d  /  m  y  s  q  l  -
s  t  a  n  d
0b40 a  r  d  -  4  .  0  .  1  2  -  p  c  -  l  i  n  u  x  -  i  6  8  6  /  d  a
t  a  /  l  i
0b60 n  u  x  .  p  i  d  0b 00 00 \  \t l  o  g  _  e  r  r  o  r  00 \n 00 00 ]  04
p  o  r  t  04
0b80 3  3  0  6  14 00 00 ^  10 p  r  o  t  o  c  o  l  _  v  e  r  s  i  o  n  02 1
0  18 00 00 _
0ba0 10 r  e  a  d  _  b  u  f  f  e  r  _  s  i  z  e  06 1  3  1  0  7  2  1c 00 00
`  14 r  e  a
0bc0 d  _  r  n  d  _  b  u  f  f  e  r  _  s  i  z  e  06 2  6  2  1  4  4  14 00 00
a  11 r  p  l
0be0 _  r  e  c  o  v  e  r  y  _  r  a  n  k  01 0  1a 00 00 b  11 q  u  e  r  y  _
c  a  c  h  e
0c00 _  l  i  m  i  t  07 1  0  4  8  5  7  6  13 00 00 c  10 q  u  e  r  y  _  c  a
c  h  e  _  s
0c20 i  z  e  01 0  14 00 00 d  10 q  u  e  r  y  _  c  a  c  h  e  _  t  y  p  e  02
O  N  \f 00 00
0c40 e  \t s  e  r  v  e  r  _  i  d  01 1  17 00 00 f  11 s  l  a  v  e  _  n  e  t
_  t  i  m  e
0c60 o  u  t  04 3  6  0  0  19 00 00 g  15 s  k  i  p  _  e  x  t  e  r  n  a  l  _
l  o  c  k  i
0c80 n  g  02 O  N  14 00 00 h  0f s  k  i  p  _  n  e  t  w  o  r  k  i  n  g  03 O
F  F  17 00 00
0ca0 i  12 s  k  i  p  _  s  h  o  w  _  d  a  t  a  b  a  s  e  03 O  F  F  13 00 00
j  10 s  l  o
0cc0 w  _  l  a  u  n  c  h  _  t  i  m  e  01 2  17 00 00 k  06 s  o  c  k  e  t
0f /  t  m  p  /
0ce0 m  y  s  q  l  .  s  o  c  k  18 00 00 l  10 s  o  r  t  _  b  u  f  f  e  r  _
s  i  z  e  06
0d00 5  2  4  2  8  0  0b 00 00 m  \b s  q  l  _  m  o  d  e  01 0  0f 00 00 n  0b t
a  b  l  e  _
```

```
0d20 c  a  c  h  e  02 6  4  12 00 00 o  \n t  a  b  l  e  _  t  y  p  e  06 I  N  N
O  D  B  14 00
0d40 00 p  11 t  h  r  e  a  d  _  c  a  c  h  e  _  s  i  z  e  01 0  14 00 00 q  \f
t  h  r  e  a
0d60 d  _  s  t  a  c  k  06 1  9  6  6  0  8  1d 00 00 r  \f t  x  _  i  s  o  l  a
t  i  o  n  0f
0d80 R  E  P  E  A  T  A  B  L  E  -  R  E  A  D  0e 00 00 s  \b t  i  m  e  z  o  n
e  04 C  E  S
0da0 T  18 00 00 t  0e t  m  p  _  t  a  b  l  e  _  s  i  z  e  \b 3  3  5  5  4  4
3  2  \r 00 00
0dc0 u  06 t  m  p  d  i  r  05 /  t  m  p  /  1c 00 00 v  07 v  e  r  s  i  o  n  13
4  .  0  .  1
0de0 2  -  s  t  a  n  d  a  r  d  -  l  o  g  13 00 00 w  \f w  a  i  t  _  t  i  m
e  o  u  t  05
0e00 2  8  8  0  0  01 00 00 x  þ  11 00 00 00 03 S  E  T     a  u  t  o  c  o  m  m
i  t  =  1  03
0e20 00 00 01 00 00 00 18 00 00 00 03 S  E  L  E  C  T     *     F  R  O  M     E  M
P  L  O  Y  E
0e40 E  ;  01 00 00 01 \n 19 00 00 02 \b E  M  P  L  O  Y  E  E  05 F  N  A  M  E  03
\n 00 00 01 ý
0e60 03 01 00 00 19 00 00 03 \b E  M  P  L  O  Y  E  E  05 M  I  N  I  T  03 01 00 00
01 þ  03 00 00
0e80 00 19 00 00 04 \b E  M  P  L  O  Y  E  E  05 L  N  A  M  E  03 \n 00 00 01 ý  03
01 00 00 17 00
0ea0 00 05 \b E  M  P  L  O  Y  E  E  03 S  S  N  03 \t 00 00 01 03 03 01 00 00 19 00
00 06 \b E  M
0ec0 P  L  O  Y  E  E  05 B  D  A  T  E  03 \n 00 00 01 \n 03 00 00 00 1b 00 00 07 \b
E  M  P  L  O
0ee0 Y  E  E  07 A  D  D  R  E  S  S  03 1e 00 00 01 ý  03 00 00 00 17 00 00 \b \b E
M  P  L  O  Y
0f00 E  E  03 S  E  X  03 01 00 00 01 þ  03 00 01 00 1a 00 00 \t \b E  M  P  L  O  Y
E  E  06 S  A
0f20 L  A  R  Y  03 07 00 00 01 05 03     00 02 1c 00 00 \n \b E  M  P  L  O  Y  E  E
\b S  U  P  E
0f40 R  S  S  N  03 \t 00 00 01 03 03 00 00 00 17 00 00 0b \b E  M  P  L  O  Y  E  E
03 D  N  O  03
0f60 01 00 00 01 03 03 00 00 00 01 00 00 \f þ  R  00 00 \r 04 J  o  h  n  01 B  05 S
m  i  t  h  \t
0f80 1  2  3  4  5  6  7  8  9  \n 1  9  6  5  -  0  1  -  0  9  18 7  3  1     F  o
n  d  r  e  n
0fa0 ,     H  o  u  s  t  o  n  ,     T  X  01 M  \b 3  0  0  0  0  .  0  0  \t 3  3
3  4  4  5  5
0fc0 5  5  01 5  R  00 00 0e \b F  r  a  n  k  l  i  n  01 T  04 W  o  n  g  \t 3  3
3  4  4  5  5
0fe0 5  5  \n 1  9  5  5  -  1  2  -  0  8  15 6  3  8     V  o  s  s  ,     H  o  u
s  t  o  n  ,
1000    T  X  01 M  \b 4  0  0  0  0  .  0  0  \t 8  8  8  6  6  5  5  5  5  01 5  T
00 00 0f 06 A
1020 l  i  c  i  a  01 J  06 Z  e  l  a  y  a  \t 9  9  9  8  8  7  7  7  7  \n 1  9
6  8  -  0  7
1040 -  1  9  17 3  3  2  1     C  a  s  t  l  e  ,     S  p  r  i  n  g  ,     T  X
01 F  \b 2  5
1060 0  0  0  .  0  0  \t 9  8  7  6  5  4  3  2  1  01 4  W  00 00 10 \b J  e  n  n
i  f  e  r  01
1080 S  07 W  a  l  l  a  c  e  \t 9  8  7  6  5  4  3  2  1  \n 1  9  4  1  -  0  6
-  2  0  17 2
10a0 9  1     B  e  r  r  y  ,     B  e  l  l  a  i  r  e  ,     T  X  01 F  \b 4  3
0  0  0  .  0
10c0 0  \t 8  8  8  6  6  5  5  5  5  01 4  V  00 00 11 06 R  a  m  e  s  h  01 K  07
N  a  r  a  y
10e0 a  n  \t 6  6  6  8  8  4  4  4  4  \n 1  9  6  2  -  0  9  -  1  5  18 9  7
5     F  i  r  e
1100    O  a  k  ,     H  u  m  b  l  e  ,     T  X  01 M  \b 3  8  0  0  0  .  0  0
\t 3  3  3  4
1120 4  5  5  5  5  01 5  S  00 00 12 05 J  o  y  c  e  01 A  07 E  n  g  l  i  s  h
\t 4  5  3  4
1140 5  3  4  5  3  \n 1  9  7  2  -  0  7  -  3  1  16 5  6  3  1     R  i  c
e  ,     H  o  u  s
1160 t  o  n  ,     T  X  01 F  \b 2  5  0  0  0  .  0  0  \t 3  3  3  4  4  5  5  5
5  01 5  S  00
1180 00 13 05 A  h  m  a  d  01 V  06 J  a  b  b  a  r  \t 9  8  7  9  8  7  9  8  7
\n 1  9  6  9
```

```
11a0 - 0 3 - 2 9 17 9 8 0    D a l l a s ,    H o u s t o
n ,    T X 01 M
11c0 \b 2 5 0 0 0 . 0 0 \t 9 8 7 6 5 4 3 2 1 01 4 G 00 00 14 05 J
a m e s 01
11e0 E 04 B o r g \t 8 8 8 6 6 5 5 5 5 \n 1 9 3 7 - 1 1 - 1 0
16 4 5 0
1200 S t o n e ,    H o u s t o n ,    T X 01 M \b 5 5 0 0 0 .
0 0 û 01 1
1220 01 00 00 15 þ
```

Die Analyse des Datenverkehrs zeigt, daß im Falle der JDBC-basierten Kommunikation ein gegenüber der nativen Schnittstelle um 3529 Byte vergrößertes Datenaufkommen ausgetauscht wird. Diese zusätzliche Datenmenge fällt jedoch nur einmal zum Zeitpunkt des JDBC-Verbindungsaufbaus statisch an. (Vgl. Mitschnitt der mehrfachen Ausführung einer SQL-Anfrage innerhalb einer bestehenden JDBC-Verbindung)
Zusätzlich offenbart Zeile 0x40 des Datenverkehrs die verschlüsselte Übermittlung des Paßwortes des Anwenders *mario*. Allerdings werden die per Anfrage ermittelten Nutzdaten (ab Zeile 0xe40) unverschlüsselt über die Netzwerkschnittstelle übertragen und stellen somit ein potentielles Angriffsziel dar.
Abhilfe hierfür kann die Tunnelung des Datenverkehrs, beispielsweise mittels SSH, durch eine sichere Verbindung bieten.

Web-Referenzen 1: Weiterführende Links

- JDBC @ SUN
- JDBC learning center @ SUN
- JDBC Tutorial
- JDBC FAQ @ JGuru.com
- G. Reese: *Database Programming with JDBC and Java*. O'Reilly, 1997
- Verhältnis von X/Open CLI und ODBC

1.2 Enterprise Java Beans

Neben den bereits aus anderen Veranstaltungen bekannten Servlets und den davon abgeleiteten Java Server Pages bildet die Technik der *Enterprise Java Beans* (EJB) einen weiteren zentralen Baustein der Java 2 Enterprise Plattform. Als serverseitige Komponenten kommt den EJBs heute große Bedeutung in der Realisierung komplexer Anwendungen, insbesondere durch Umsetzung der sog. „Business Logik“, d.h. den nicht-interaktiven fachlichen Anwendungsteilen, zu.

Der Begriff der Enterprise Java Bean stützt sich auf dem historisch älteren der *Java Bean*. Eine solche stellt eine abgeschlossene wiederverwendbare Softwarekomponente dar, die nach ihrer Erstellung über festgelegte Schnittstellen parametrisiert und manipuliert werden kann. Hierzu muß eine Bean eine festgelegte Interaktionsschnittstelle bieten, die durch die Java Bean Spezifikation definiert ist. Es handelt sich dabei um eine Reihe von Konventionen, der eine Bean gehorchen muß, jedoch um keine festgelegte API, die durch eine Komponente zu implementieren ist.
Der Begriff der Enterprise Java Bean greift diese inhaltliche Fundierung auf und präzisiert gleichzeitig die technische Umsetzung. So stellt eine Enterprise Java Bean eine Softwarekomponente dar, die in einer festgelegten Ausführungsumgebung, welche durch die EJB-Spezifikation festgelegte Dienste zur Nutzung durch die Beans anbieten kann. Eine solche Ausführungsumgebung wird als *Container* bezeichnet.
Ziel der Trennung in Komponente und Ausführungsumgebung ist die Zielsetzung die Enterprise Java Bean ausschließlich zur Umsetzung fachlicher Aufgaben heranzuziehen und alle infrastrukturellen Fragestellungen wie Betriebsmittelverwaltung, Persistenz oder Sicherheit durch die Ausführungsumgebung in gleicher Weise für alle Komponenten bereitzustellen.
Ein EJB-Container wird zumeist im Rahmen eines Application-Servers bereitgestellt.
Die gelegentlich anzutreffende Hervorhebung der anfänglich für Java Beans intendierten *visuellen Manipulationsmöglichkeit* trifft für Enterprise Java Beans nicht zu und hat sich für Java Beans auch nur begrenzte Bedeutung erlangt.

Spezifikationsgemäß können EJB-Container folgende Eigenschaften offerieren:

- **Betriebsmittelverwaltung**
  Typischerweise verwaltet ein einziger EJB-Container gleichzeitig eine Reihe verschiedener Enterprise Java Beans. Zur Organisation und Aufrechterhaltung der Ausführbarkeit obliegt dem Container die Zuteilung von Betriebsmitteln wie Hauptspeicher, CPU-Zeit oder Netzwerkressourcen an die verwalteten EJB. Hierunter fällt insbesondere auch die Einlagerung, Instanziierung und Entfernung der EJBs selbst.
- **Zustandsverwaltung**
  In praktischen Anwendungen ist oft die Nutzung zustandsbehafteter Kommunikation, die sich über

verschiedene Einzelinteraktionen erstreckt gewünscht. Die hierfür notwendigen technischen Voraussetzungen (Zustandsspeicherung, Korrelation der Einzelinteraktionen) werden durch den Container bereitgestellt.

- **Transaktionsverwaltung**
  Erweiterung der Zustandsverwaltung. Zur Gewährleistung des benötigten Verhaltens müssen EJBs keine eigenen Implementierungen zur Verfügung stellen, sondern können vorhandene Dienste des Containers nutzen.
- **Sicherheit**
  Die Sicherheit von EJBs kann durch Vergabe von Zugriffsrechten und Rollen auf Containerebene gesteuert werden.
- **Persistenz**
  Der interne Zustand einer verwalteten EJB kann wahlfrei in persistiert und zu einem späteren Zeitpunkt wiederhergestellt werden.
- **Entfernter Zugriff**
  Der Zugriff auf EJBs erfolgt mittels Remote Method Invocation und ist daher Lokationstransparent.

Neben den in der Aufzählung dargestellten Eigenschaften dürfen Container zusätzlich Weitere wahlfrei implementieren.

EJB-Typen

Grundsätzlich lassen sich alle EJBs drei Typen zuordnen: *Session Beans*, *Entity Beans* und *Message Driven Beans*. Während erstere hauptsächlich zur Abbildung von Abläufen eingesetzt werden, dienen Entity Beans der Abwicklung von Zugriffen auf Daten. Eine Sonderstellung nehmen die *Message Driven Beans* ein, die lediglich hinsichtlich ihres Kommunikationsverhaltens festgelegt sind.

Session Beans dienen der Abbildung von Abläufen im Rahmen der Programmierung der sog. Business Logik. Die Lebensdauer (d.h. Zeitspanne zwischen Erzeugung im und Entfernung aus dem Hauptspeicher) ist daher identisch mit der einer durch den Client erfolgenden Anfrage. Jede zu einem Zeitpunkt existierende Session Bean repräsentiert daher eine zugehörige Clientinstanz. Nach ihrer internen Ausgestaltung werden *stateless* und *statefull* Session Beans unterschieden. Während Erstere keinen über einen einzigen Aufruf hinausgehenden Zustand verwalten und daher seiteneffektfrei lediglich auf den durch den Aufruf übermittelten Daten operieren erhält das zustandserhaltende Pendant die Daten eines Aufrufs und kann diese auch in nachfolgenden Aufrufen verarbeiten.

Entity Beans sind programmiersprachliche Stellvertreter datenbankresidenter Objekte. Sie dienen dem erleichterten Zugriff auf persistent vorliegende Datenbestände. Ihre interne Realisierung ist eng mit der Technik relationaler Datenbankmanagement System verbunden. So werden Sie durch einen anwenderdefinierten Primärschlüssel dauerhaft identifiziert.

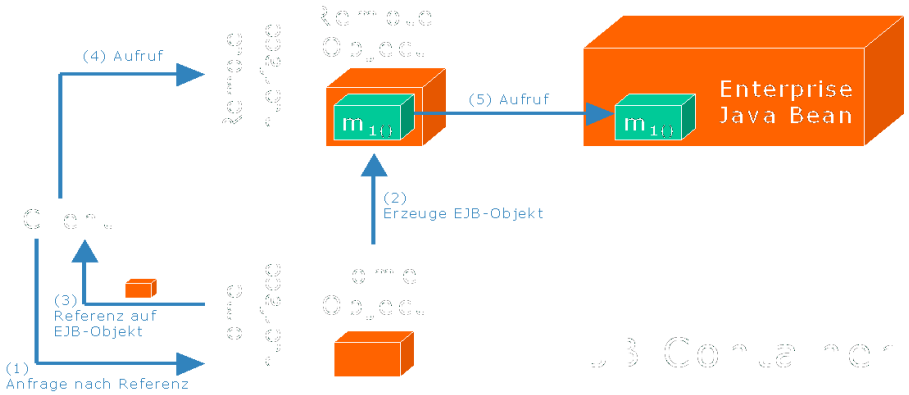
Message Driven Beans sind hinsichtlich ihres Kommunikationsverhaltens auf asynchrone Aufrufe beschränkt. Die Realisierung des eigentlichen Verhaltens wird durch eine Ausprägung eines der anderen Beantypen geboten.

Session Beans

Konzeptionell umfaßt jede EJB-Anwendung, die Session Beans einsetzt, die in Abbildung 4 dargestellten Teile:

- Implementierung der EJB selbst.
- *Remote*-Schnittstelle zum Zugriff auf die durch die Bean publizierten Methoden.
- *Home*-Schnittstelle zur Ermittlung einer Referenz auf das Bean-Objekt.

Abbildung 4: *Aufrufstruktur einer zustandslosen EJB*



(click on image to enlarge!)

Gegenüber der Realisierung als RMI-Anwendung benötigt die Umsetzung als zustandslose Session Bean die Erstellung einer sog. „Home-Schnittstelle“ (*Home Interface*), welche die Operation `create` zur Instanziierung des serverseitigen EJB-Objekts bietet.
Sie ist im Beispiel 23 dargestellt.

Beispiel 23: Home-Schnittstelle einer EJB

```
(1)import java.rmi.RemoteException;
(2)import javax.ejb.CreateException;
(3)import javax.ejb.EJBHome;
(4)
(5)public interface SayHelloHome extends EJBHome {
(6)     public SayHello create() throws RemoteException, CreateException;
(7)}
```

Download des Beispiels

Die anwenderdefinierte Home-Schnittstelle erweitert die durch die Standard-API vorgegebene Schnittstelle `EJBHome`. Diese definiert Operationen zur Entfernung existierender EJB-Objekte aus dem Hauptspeicher (`remove`) sowie zur Ermittlung von Metadaten (`getEJBMetaData`) oder zum Erhalt eines netzwerkunabhängigen Verweises auf das EJB-Objekt (`getHomeHandle`).
Im Einzelnen sind dies die Operationen:

- `EJBMetaData getEJBMetaDate()`
  Liefert ein `EJBMetaData`-Objekt welches einzelne Eigenschaften einer EJB näher beschreibt. Hierzu zählen:
  - Klasse der Home-Schnittstelle
  - Klasse des Primärschlüssels (nur vorhanden sofern es sich um eine Entity Bean handelt)
  - Klasse der Remote-Schnittstelle
  - Boole'scher Wert, der angibt, ob es sich um eine Session Bean handelt
  - Boole'scher Wert, der angibt, ob es sich um eine zustandslose Session Bean handelt
- `HomeHandle getHomeHandle()`
  Liefert ein Objekt des Typs `HomeHandle` zurück, welches eine netzwerkunabhängige Abstraktion des Verweises auf das Home-Objekt realisiert.
- `void remove (Handle h)`
  Entfernt ein durch den Objektverweis (`Handle`) identifiziertes EJB-Objekt aus dem Hauptspeicher.
- `void remove (Object pk)`
  Entfernt ein durch das übergebene Primärschlüsselobjekt identifiziertes EJB-Objekt aus dem Hauptspeicher.

Interessanterweise definiert die Schnittstelle zwar Operationen zur Ermittlung von Daten über bestehende Objekte und zur Entfernung dieser Objekte aus dem Hauptspeicher, nicht jedoch zu ihrer Erzeugung.
Dies liegt in der durch die Programmiersprache Java angestrebten statischen Typsicherheit begründet, die es nicht gestattet Operationen mit variablen Parameterlisten --- wie sie für die zum API-Erstellungszeitpunkt unbekannten spezifischen Initialisierungsparameter aller denkbaren EJBs benötigt würden --- zu versehen.
Aus diesem Grunde definiert die EJB-Spezifikation informell, daß ein diese Schnittstelle erweiternde eigene Home-Schnittstelle zusätzlich die Methode `create`, deren Signatur als Rückgabetyp den Typ der Remote-Schnittstelle vorsehen muß definiert. Zusätzlich enthält diese Operation die zur Initialisierung der Bean benötigten Parameter in ihrer Parameterliste.

Die im Beispiel 24 ist Remote-Schnittstelle dargestellt, deren Ausprägungen von Home-Objekten angesprochenen werden:

Beispiel 24: Remote-Schnittstelle einer EJB

```
(1)import java.rmi.RemoteException;
(2)import javax.ejb.EJBObject;
(3)
(4)public interface SayHello extends EJBObject {
(5)     public String sayHello(String name) throws RemoteException;
(6)}
```

Download des Beispiels

Schnittstellen dieses Typs enthalten ausschließlich die fachlichen Operationen, d.h. die Signaturen der Methoden, die später durch den Client benutzt werden.
Jede Remote-Schnittstelle erweitert zusätzlich die vorgegebene Schnittstelle `EJBObject`, welche,

ähnlich zur Home-Schnittstelle, einige Operationen zur Verwaltung eines EJB-Objektes vorgibt:

- `EJBHome getEJBHome()`
  Liefert die Home-Schnittstelle einer EJB.
- `Handle getHandle()`
  Liefert ein Objekt des Typs `HomeHandle` zurück, welches eine netzwerkunabhängige Abstraktion des Verweises auf das Home-Objekt realisiert.
- `Object getPrimaryKey()`
  Liefert das Primärschlüsselobjekt einer Entity Bean.
- `boolean isIdentical(EJBObject eo)`
  Prüft ob das übergebene EJB-Objekt dasselbe wie das Objekt ist auf dem die Methode ausgeführt wird.
- `void remove()`
  Entfernt das EJB-Objekt aus dem Bean-Container.

Beispiel 25 zeigt die Implementierung der Bean selbst:

Beispiel 25: Realisierung einer Session Bean

```
(1)import java.rmi.RemoteException;
(2)import java.util.Date;
(3)import javax.ejb.EJBException;
(4)import javax.ejb.SessionBean;
(5)import javax.ejb.SessionContext;
(6)
(7)public class HelloWorldBean implements SessionBean {
(8)     public HelloWorldBean() {
(9)     }
(10)
(11)    public String sayHello(String name) {
(12)            return ("Hello " + name + " it is now " + new Date().toString());
(13)    }
(14)
(15)    public void setSessionContext(SessionContext arg0)
(16)            throws EJBException, RemoteException {
(17)    }
(18)    public void ejbRemove() throws EJBException, RemoteException {
(19)    }
(20)
(21)    public void ejbCreate() throws EJBException {
(22)    }
(23)
(24)    public void ejbActivate() throws EJBException, RemoteException {
(25)    }
(26)
(27)    public void ejbPassivate() throws EJBException, RemoteException {
(28)    }
(29)}
```

Download des Beispiels

Die programmiersprachliche Umsetzung der Bean enthält die Methoden der in der Remote-Schnittstelle bekanntgegebenen fachlichen Operationen. Zusätzlich muß ein Konstruktur expliziert werden, dessen Parameterliste mit den für die Operation `create` des Home-Interfaces gegebenen übereinstimmen.
Spezifikationsgemäß muß jede Session Bean die gleichnamige API-Schnittstelle implementieren. Diese definiert einige Operationen zur Behandlung unterschiedlicher Lebenszyklusstadien einer EJB. Hierunter fallen Methoden, die beim Erzeugen (`ejbCreate`), Entfernen (`ejbRemove`), bei der Passivierung (d.h. Auslagerung auf Hintergrundspeicher) (`ejbPassivate`) und dessen Reaktivierung (`ejbActivate`) eines EJB-Objekts durch die Ausführungsumgebung aufgerufen werden.

Bei der zwingend zu implementierenden Schnittstelle `SessionBean` handelt es sich nicht nur um eine Konvention um die Umsetzung der Lebenszyklusschnittstelle sicherzustellen, sondern auch um die Kategorisierung der Bean selbst. So stellt die im Beispiel verwandte Schnittstelle `SessionBean` neben `EntityBean` und `MessageDrivenBean` eine Spezialisierung der (operationslosen) Schnittstelle `EnterpriseBean` dar, deren „Implementierung“ durch eine Klasse lediglich zur Kennzeichnung dieser als EJB herangezogen wird.
Die genannten Spezialisierungen dieser Schnittstelle erfüllen daher sowohl den Zweck der Ausübung des Implementierungszwanges für die in ihnen aufgeführten Operationen als auch den der typisierenden Kennzeichnung.
Darüberhinaus ist `EnterpriseBean` als Spezialisierung der Standard-Schnittstelle `Serializable`

angelegt. In der Konsequenz muß jedes EJB-Objekt durch die Javasprachmechnismen serialisierbar sein. Diese Eigenschaft wird insbesondere für die Passivierung und im Rahmen der Entity Beans genutzt.

Konzeptionell erinnert die Trennung in publizierte fachliche Schnittstelle (Remote-Schnittstelle) und deren technischer Umsetzung durch die EJB an die aus der Betrachtung des Remote Method Invocation Mechanismus bekannte Struktur.
Allerdings weicht die Umsetzung der Bean von der dort anzutreffenden Konvention ab die publizierte Schnittstelle selbst durch die realisierende Klasse zu implementieren. Dies liegt vor allem an der gegenüber RMI veränderten Struktur der publizierten Schnittstelle begründet.
Während RMI für die Schnittstelle die Spezialisierung der operationslosen Standardschnittstelle `Remote` fordert die EJB-Spezifikation die Erweiterung der Schnittstelle `EJBObject`, welche selbst die oben dargestellten Operationen definiert. Aus diesem Grunde würde die Aufnahme der Remote-Schnittstelle, obwohl konzeptionell durchaus zu rechtfertigen, in die Umsetzungsliste der EJB gleichzeitig die Implementierung von zumindest leeren Methodenrümpfen für die in `EJBObject` definierten Operationen notwendig werden lassen.
Abgesehen von dieser Ausnahme rekonstruiert das Verhältnis zwischen EJB und deren Remote-Schnittstelle die aus RMI bekannte Beziehung zwischen Schnittstelle und Umsetzung.

Die Nutzung einer durch eine Java Bean angebotenen Funktionalität erfolgt gemäß dem in Abbildung 4 dargestellten Schema. Ein dies umsetzender Client ist in Beispiel 26 dargestellt.

Beispiel 26: Zugriff auf eine Session Bean

```
(1)import java.rmi.RemoteException;
(2)import javax.ejb.CreateException;
(3)import javax.naming.Context;
(4)import javax.naming.InitialContext;
(5)import javax.naming.NamingException;
(6)import javax.rmi.PortableRemoteObject;
(7)
(8)public class CallHelloWorldBean {
(9)     public static void main(String[] args) {
(10)            try {
(11)                    Context initial = new InitialContext();
(12)                    Object objRef = initial.lookup("helloBean");
(13)
(14)                    SayHelloHome home =
(15)                            (SayHelloHome) PortableRemoteObject.narrow(
(16)                                    objRef,
(17)                                    SayHelloHome.class);
(18)                    SayHello sh = home.create();
(19)
(20)                    System.out.println(sh.sayHello("Mario"));
(21)
(22)            } catch (NamingException ne) {
(23)                    ne.printStackTrace();
(24)            } catch (RemoteException re) {
(25)                    re.printStackTrace();
(26)            } catch (CreateException ce) {
(27)                    ce.printStackTrace();
(28)            }
(29)    }
(30)}
```

Download des Beispiels

Zunächst ermittelt der Client unter Nutzung der JNDI-API eine Referenz auf die EJB. Dies geschieht durch Anfrage (lookup) an den JNDI-Dienst unter Übergabe des bekannten Klarnamens (*helloBean*).
Die erhaltene generische Referenz wird durch Aufruf der statischen Methode `narrow` der Klasse `PortableRemoteObject` typsicher in eine Ausprägung der Home-Schnittstelle konvertiert.
Der Aufruf der in dieser Schnittstelle durch den Anwender definierten `create`-Methode sorgt für die serverseitige Instanziierung der EJB, die als Ausprägung der Remote-Schnittstelle geliefert wird.
Tatsächlich wird nicht das EJB-Objekt selbst durch den Methodenaufruf retourniert, sondern lediglich ein netzwerktransparenter Verweis darauf, der jedoch clientseitig einer lokalen Objektreferenz gleichgestellt verwendet werden kann.
Ferner wird serverseitig zur Kommunikation mit der EJB ein Home-Objekt erzeugt, welchem eine Stellvertreterrolle für den anfragenden Client zukommt.
Der Aufruf der durch die EJB zur Verfügung gestellten Methode erfolgt identisch zu dem einer Lokalen.

Entity Beans

Die zweite zentrale Klasse von Enterprise Java Beans bilden die zur serverseitigen Persistierung von Objekten dienenden *Entity Beans*.
Sie kapseln Datenbankinhalte durch Objekte, die gemäß der EJB-Spezifikation instanziierbar und zugreifbar sind. Die Verwaltung der gekapselten Dateninhalte erfolgt durch ein frei festlegbares Datenbankmanagementsystem, die der Objekte selbst durch den EJB-Container.
Ziel dieser Technik ist es die Komplexität der Persistenzlogik für den Verwender der bereitgestellten Bean vollkommen transparent zu gestalten und serverseitig zu realisieren.
Die Familie der Entity Beans selbst zerfällt in zwei Untertypen, welche sich entlang des Realisierungspunktes der Persistenzlogik separieren: *Bean Managed Persistence*, der Bean obliegt die Umsetzung der Persistenzlogik, und *Container Managed Persistence*, hierbei wird die Persistenzlogik durch den EJB-Container realisiert.

Das nachfolgende Beispiel zeigt die Umsetzung einer Entity Bean mit Bean Managed Persistence. Es kapselt die Verwaltung und den Zugriff auf Objekte, die Personen beschreiben. Jedes `Personen`-Objekt enthält Daten zu Name, Geburtsdatum und Wohnstraße. Der Name dient als eindeutige Identifikation und daher datenbankseitig als Primärschlüssel. Die notwendige Datenbanktabelle wurde erzeugt durch den SQL-Ausdruck:

```
CREATE TABLE PERSON( Name VARCHAR(20) PRIMARY KEY, Birthdate DATE, Street VARCHAR
(30));
```

Wie bereits für die Realisierung von Session Beans eingeführt, werden auch zur Publikation der extern zugänglichen Schnittstellen Home und Remote Interfaces benötigt.

Struktur und Aufbau der Home-Schnittstelle ähnelt konzeptionell der für Session Beans eingeführten. Dieser Schnittstellentyp dient auch für Entity Beans zur Aufnahme der Verwaltungsoperationen zur Erzeugung (`create`) und zur Suche existierender EJBs (`findByPrimaryKey`).
Beispiel 27 zeigt die Home-Schnittstelle des Beispiels.

Beispiel 27: Home-Schnittstelle einer Entity Bean

```
(1)import java.rmi.RemoteException;
(2)import javax.ejb.CreateException;
(3)import javax.ejb.EJBHome;
(4)import javax.ejb.FinderException;
(5)
(6)public interface PersonHome extends EJBHome {
(7)     public Person create(int year, int month, int day, String name, String
street) throws CreateException, RemoteException;
(8)     public Person findByPrimaryKey(String name) throws FinderException,
RemoteException;
(9)}
```

Download des Beispiels

Die Home-Schnittstelle zeigt die `create`-Operation zur Erzeugung einer neuen EJB-Instanz. Ihre Übergabeparameter dienen zur Konstruktion des neuen Objekts und werden durch die Bean-Implementierung interpretiert.
Ferner enthält die Schnittstelle mit `findByPrimaryKey` eine Operation deren Implementierung eine Entity-Bean anhand ihres Primärschlüssels identifiziert und liefert. Aus diesem Grunde erhält die Methode den zu suchenden Wert von Typ des Primärschlüssels übergeben.
Hinsichtlich der verwendeten Typen zeigt sich bereits hier, daß eine Abbildung der durch die Programmiersprache Java bereitgestellten Typen auf die des eingesetzten Persistenzsystems stattfinden muß. In der im Beispiel gewählten Ausführungsform der durch die EJB selbst verwalteten Persistenz muß diese Abbildung manuell durch den Programmierer bereitgestellt werden.

Die Remote-Schnittstelle gibt die für Nutzer der Bean zugänglichen Geschäftsfunktionen wieder. Daher enthält dieser Schnittstellentyp lediglich Operationen zum Zugriff auf die verwalteten Daten, nicht jedoch zur technischen Verwaltung und Interaktion mit dem Bean-Container.
Per Konvention muß diese Schnittstelle als Spezialisierung der Schnittstelle `EJBObject` definiert sein. Diese Standardschnittstelle definiert allgemeine Interaktionsformen, wie Löschen (`remove`), Vergleich (`isIdentical`) und Ermittlung des Primärschlüsselwertes (`getPrimaryKey`) die für alle Entity Bean Objekte gleichermaßen benötigt werden.
`isIdentical` liefert den Vergleich zweier serverseitiger EJB-Objekte und ermittelt so, ob zwei Java-

Objektreferenzen auf dasselbe Datenbankobjekt zugreifen.
Mittels `getPrimaryKey` ermittelt den Wert des Primärschlüssels eines gegebenen EJB-Objekts aus der Datenbank.
Zur Lösung von datenbankresidenten Objekten wird `remove` eingesetzt. Der Aufruf dieser Methode entfernt ausschließlich die durch die EJB repräsentierten Datenbanktupel, die programmiersprachliche Objektrepräsentation bleibt jedoch über die gesamte Laufzeit (sofern nicht durch Gültigkeitsbereiche oder explizite `NULL`-Setzung explizit anders gehandhabt) intakt.

Beispiel 28: Remote-Schnittstelle einer Entity Bean

```
(1)import java.rmi.RemoteException;
(2)import javax.ejb.EJBObject;
(3)
(4)public interface Person extends EJBObject {
(5)     public int getAge() throws RemoteException;
(6)     public String getStreet() throws RemoteException;
(7)     public void setStreet(String street) throws RemoteException;
(8)}
```

Download des Beispiels

Beispiel 29 zeigt den vollständigen Code der Bean. Sie implementiert mit `EntityBean` die Standardschnittstelle alle Entity Beans, welche als Spezialisierung der ausschließlich markierenden Schnittstelle `EnterpriseBean` die notwendigen Basisoperationen zur Abwicklung der persistenten Speicherung.

Im Falle einer *Bean Managed Persistence* enthalten die Methoden der durch die Schnittstelle definierten und der zusätzlich im Rahmen der Spezifikation textuell definierten Operationen die notwendigen Aufrufe zur Ablage eines Objekts in der Datenbank und zu seiner späteren Extraktion daraus.
Im Einzelnen sind dies die Operationen:

Tabelle 1: Persistenzoperationen einer Entity Bean

Operation	Semantik	Zugehörige SQL-Anweisung
`ejbCreate`	Wird nach dem Erzeugen eines Java-Objektes aufgerufen um dieses in der Datenbank abzulegen. Diese Operation ist nicht Bestandteil der Schnittstelle, da ihre Parameter, die den Übergabeparametern des Objektkonstruktors entsprechen, zum Schnittstellenerzeugungszeitpunkt nicht feststehen.	`INSERT`
`ejbFindByPrimaryKey`	Liefert den Wert des Primärschlüssels zurück, sofern ein Datenbankeintrag existiert, der durch diesen Primärschlüssel identifiziert wird. Diese Operation ist nicht Bestandteil der Schnittstelle, da ihre Parameter, die in Typ, Name und Reihenfolge der Zusammensetzung des Primärschlüssels entsprechen, zum Schnittstellenerzeugungszeitpunkt nicht feststehen. Diese Methode wird nicht durch den Anwender direkt aufgerufen, sondern stattdessen auf einer Ausprägung der Home-Schnittstelle das dort zur Verfügung stehende Analogon `findByPrimaryKey`, welches das durch den Primärschlüssel identifizierte EJB-Objekt zurückliefert. Diese Methode greift intern auf ausschließlich den Schlüssel liefernde Methode der Bean zu.	`SELECT`
`ejbRemove`	Entfernt das EJB-Objekt aus der Datenbank.	`DELETE`

ejbStore	Synchronisiert das Java-Objekt mit dem EJB-Objekt und aktualisiert so die Datenbankinhalte. Diese Methode wird nach jedem Zugriff mittels einer in der Remote-Schnittstelle aufgeführten Operation auf das EJB-Objekt ausgeführt.	UPDATE

Beispiel 29: Eine Entity Bean

```
(1)import java.rmi.RemoteException;
(2)import java.sql.Connection;
(3)import java.sql.DriverManager;
(4)import java.sql.ResultSet;
(5)import java.sql.SQLException;
(6)import java.sql.Statement;
(7)import java.util.Calendar;
(8)import java.util.GregorianCalendar;
(9)
(10)import javax.ejb.CreateException;
(11)import javax.ejb.EJBException;
(12)import javax.ejb.EntityBean;
(13)import javax.ejb.EntityContext;
(14)import javax.ejb.FinderException;
(15)import javax.ejb.RemoveException;
(16)
(17)public class PersonBean implements EntityBean {
(18)    //persistent fields
(19)    private String name;
(20)    private GregorianCalendar birthdate;
(21)    private String street;
(22)
(23)    public PersonBean() {
(24)    }
(25)
(26)    //methods which are part of the remote interface
(27)    public int getAge() {
(28)            return (
(29)                    new GregorianCalendar().get(Calendar.YEAR)
(30)                            - birthdate.get(Calendar.YEAR));
(31)    }
(32)    public String getStreet() {
(33)            return street;
(34)    }
(35)    public void setStreet(String street) throws RemoteException {
(36)            if (street.length() <= 30) {
(37)                    this.street = street;
(38)            } else {
(39)                    throw new RemoteException("cannot update street");
(40)            }
(41)    }
(42)    //      ------------------------------------------------------------------
(43)    public String ejbCreate(
(44)            int year,
(45)            int month,
(46)            int day,
(47)            String name,
(48)            String street)
(49)            throws CreateException {
(50)            if (year > 1900
(51)                    && month >= 1
(52)                    && month <= 12
(53)                    && day >= 1
(54)                    && day <= 31
(55)                    && name.length() <= 20
(56)                    && street.length() <= 30) {
(57)
(58)                    this.name = name;
(59)                    this.birthdate = new GregorianCalendar(year, month, day);
(60)                    this.street = street;
(61)                    Statement stmt = getStatement();
```

```
(62)                    String s =
(63)                            new String(
(64)                                    "INSERT INTO PERSON VALUES ('"
(65)                                            + name
(66)                                            + "','"
(67)                                            + birthdate.get(Calendar.YEAR)
(68)                                            + "-"
(69)                                            + birthdate.get(Calendar.MONTH)
(70)                                            + "-"
(71)                                            + birthdate.get(Calendar.DATE)
(72)                                            + "','"
(73)                                            + "'"
(74)                                            + street
(75)                                            + "'"
(76)                                            + ");");
(77)                    try {
(78)                            stmt.executeUpdate(s);
(79)                    } catch (SQLException e) {
(80)                            e.printStackTrace();
(81)                    }
(82)            } else {
(83)                    throw new CreateException("Invalid values supplied");
(84)            }
(85)            return name;
(86)    }
(87)    public void ejbPostCreate(
(88)            int year,
(89)            int month,
(90)            int day,
(91)            String name,
(92)            String street) {
(93)    }
(94)    //      ---------------------------------------------------------------
(95)    public int ejbHomeGetAge() {
(96)            return 0;
(97)    }
(98)
(99)    public String ejbHomeGetStreet() {
(100)           return new String();
(101)   }
(102)
(103)   public void ejbHomeSetStreet(String street) {
(104)   }
(105)
(106)   public Statement getStatement() {
(107)           try {
(108)                   Class.forName("com.mysql.jdbc.Driver");
(109)           } catch (ClassNotFoundException e) {
(110)                   e.printStackTrace();
(111)           }
(112)           Connection con = null;
(113)           try {
(114)                   con =
(115)                           (Connection) DriverManager.getConnection(
(116)                                   "jdbc:mysql://10.0.0.1/Address/",
(117)                                   "mario",
(118)                                   "thePassword");
(119)           } catch (SQLException e1) {
(120)                   e1.printStackTrace();
(121)           }
(122)           try {
(123)                   return ((Statement) con.createStatement());
(124)           } catch (SQLException e2) {
(125)                   e2.printStackTrace();
(126)           }
(127)           return null; //never gets here
(128)   }
(129)
(130)   // ---------------------------------------------------------------
(131)   public void setEntityContext(EntityContext ectx)
(132)           throws EJBException, RemoteException {
(133)   }
```

```
(134)
(135)   public void unsetEntityContext() throws EJBException, RemoteException {
(136)   }
(137)
(138)   public void ejbRemove()
(139)           throws RemoveException, EJBException, RemoteException {
(140)           Statement stmt = getStatement();
(141)           String s = new String("DELETE FROM PERSON WHERE Name='" + name +
"';");
(142)           try {
(143)                   stmt.executeUpdate(s);
(144)           } catch (SQLException e) {
(145)                   e.printStackTrace();
(146)           }
(147)   }
(148)
(149)   public void ejbActivate() throws EJBException, RemoteException {
(150)   }
(151)
(152)   public void ejbPassivate() throws EJBException, RemoteException {
(153)   }
(154)
(155)   public void ejbLoad() throws EJBException, RemoteException {
(156)   }
(157)
(158)   public void ejbStore() throws EJBException, RemoteException {
(159)           Statement stmt = getStatement();
(160)           String s =
(161)                   new String(
(162)                           "UPDATE PERSON SET Name='"
(163)                                   + name
(164)                                   + "', BDATE='"
(165)                                   + birthdate.get(Calendar.YEAR)
(166)                                   + "-"
(167)                                   + birthdate.get(Calendar.MONTH)
(168)                                   + "-"
(169)                                   + birthdate.get(Calendar.DATE)
(170)                                   + "', Street = '"
(171)                                   + street
(172)                                   + "' WHERE Name = '"
(173)                                   + name
(174)                                   + "';");
(175)           try {
(176)                   stmt.executeUpdate(s);
(177)           } catch (SQLException e) {
(178)                   e.printStackTrace();
(179)           }
(180)   }
(181)
(182)   public String ejbFindByPrimaryKey(String name) throws FinderException {
(183)           Statement stmt = getStatement();
(184)           String s =
(185)                   new String("SELECT Name FROM PERSON WHERE Name='" + name +
"';");
(186)           try {
(187)                   ResultSet rs = stmt.executeQuery(s);
(188)                   rs.first();
(189)                   return rs.getString("Name");
(190)           } catch (SQLException e) {
(191)                   e.printStackTrace();
(192)           }
(193)           return null; //never gets here
(194)   }
(195)}
```

Download des Beispiels

Zusätzlich enthält die Bean des Beispiels mit getAge eine zwar in der Remote-Schnittstelle veröffentlichte Operation, die keinen direkt abgespeicherten Wert liefert, sondern diesen dynamisch zur Ausführungszeit anhand der verfügbaren Daten berechnet.

Alle anderen in der Remote-Schnittstelle aufgeführten Operationen (etwa: `getStreet`, `setStreet`) modifizieren lediglich, den durch die Attribute repräsentierten Java-Objektzustand und greifen nicht direkt auf die Datenbank zu.

Innerhalb der Datenbankzugreifenden Methoden muß durch den Anwender die Abbildung der Java-Datentypen auf die des verwendeten Datenbankmanagementsystems erfolgen. Die mit dem Präfix `ejb` versehenden Methoden zeigen dies für die lesenden und schreibenden DB-Zugriffe. So kann die im Beispiel für `name` und `street` verwendete Java-Repräsentation `String` vergleichsweise leicht in den SQL-Typ `VARCHAR` abgebildet werden sofern durch alle Methoden, die Datenbankinhalte schreiben sicherstellen, daß nur zum Datenbankschema konforme Werte eingefügt werden. Die Beispielimplementierung zeigt dies examplarisch anhand der Methoden `ejbCreate` und `setStreet`. Für programmiersprachliche Typen, die nicht direkt in DB-Typen abbildbar sind muß im Falle der Bean Managed Persistence der Bean-Entwickler selbst Sorge für die adäquate Abbildung tragen. Das Beispiel illustriert dies anhand des Java-Datumstyps `GregorianCalendar`, der manuell in die durch das DBMS erwartete ISO 8601-konforme Darstellung zu überführen ist.

Einige der möglichen Interaktionen mit der Bean zeigt der Code des Clients aus Beispiel 30:

Beispiel 30: Client der auf eine Entity Bean zugreift

```
(1)import java.rmi.RemoteException;
(2)import javax.ejb.CreateException;
(3)import javax.ejb.FinderException;
(4)import javax.ejb.RemoveException;
(5)import javax.naming.Context;
(6)import javax.naming.InitialContext;
(7)import javax.naming.NamingException;
(8)import javax.rmi.PortableRemoteObject;
(9)
(10)public class CallPersonBean {
(11)    public static void main(String[] args) {
(12)            try {
(13)                    Context initial = new InitialContext();
(14)                    Object objRef = initial.lookup("personBean");
(15)
(16)                    PersonHome home =
(17)                            (PersonHome) PortableRemoteObject.narrow(
(18)                                    objRef,
(19)                                    PersonHome.class);
(20)                    Person p1 = home.create(1911, 1, 1, "Alice", "streetA");
(21)                    Person p2 = home.create(1922, 2, 2, "Bob", "streetB");
(22)                    System.out.println("Alice's Age: " + p1.getAge());
(23)
(24)                    System.out.println("Alice's primary key: "+p1.getPrimaryKey
());
(25)                    p1.remove();
(26)
(27)                    Person p3 = home.findByPrimaryKey("Bob");
(28)                    System.out.println(
(29)                            "Bob's modified street (before modification): "
(30)                                    + p3.getStreet());
(31)                    p3.setStreet("streetC");
(32)                    System.out.println(
(33)                            "Bob's modified street (after modification): "
(34)                                    + p3.getStreet());
(35)                    System.out.println("also the other reference: " + p2.
getStreet());
(36)
(37)                    System.out.println("Are both references the same Java
object: "+ (p2==p3));
(38)                    System.out.println("Are both references the same EJBObject
(calls isIdentical): "+p2.isIdentical(p3));
(39)
(40)            } catch (NamingException ne) {
(41)                    ne.printStackTrace();
(42)            } catch (RemoteException re) {
(43)                    re.printStackTrace();
(44)            } catch (CreateException ce) {
(45)                    ce.printStackTrace();
(46)            } catch (RemoveException e) {
(47)                    e.printStackTrace();
(48)            } catch (FinderException e) {
```

```
(49)                    e.printStackTrace();
(50)            }
(51)    }
(52)}
```

Download des Beispiels

Der Client ermittelt zunächst per JNDI eine Referenz auf die Bean, welche unter dem Namen *personBean* im Verzeichnisdienst registriert ist.
Die erhaltene generische Referenz wird durch Aufruf der statischen Methode `narrow` der Klasse `PortableRemoteObject` typsicher in eine Ausprägung der Home-Schnittstelle (`PersonHome`) konvertiert.
Der Aufruf der in dieser Schnittstelle durch den Anwender definierten `create`-Methode sorgt für die serverseitige Instanziierung der EJB, die als Ausprägung der Remote-Schnittstelle geliefert wird. Tatsächlich wird nicht das EJB-Objekt selbst durch den Methodenaufruf retourniert, sondern lediglich ein netzwerktransparenter Verweis darauf, der jedoch clientseitig einer lokalen Objektreferenz gleichgestellt verwendet werden kann.
Ferner wird serverseitig zur Kommunikation mit der EJB ein Home-Objekt erzeugt, welchem eine Stellvertreterrolle für den anfragenden Client zukommt.
Der Aufruf der durch die EJB zur Verfügung gestellten Methode erfolgt identisch zu dem einer Lokalen.

So dient der Aufruf der Methode `create` zur Erzeugung von serverseitig instanziiert und transparent persistierten EJB-Objekten sowie den lokalen Java-(Stellvertreter-)Objekten für den Zugriff darauf.
Der Aufruf von `getAge` zeigt die Nutzung einer in der Remote-Schnittstelle veröffentlichten Zugriffsmethode. Mit `getPrimaryKey` wird die, in der durch die Remote-Schnittstelle erweiterten Schnittstelle `EJBObject` angesiedelte, Operation zur Ermittlung des Primärschlüsselwertes eines EJB-Objektes aufgerufen.
Die Methode `remove` stellt dagegen eine durch die Home-Schnittstelle definierte Operation dar. Durch den Aufruf dieser Methode auf dem durch `p1` referenzierten Objekt wird werden durch durch Ausführung der Beanmethode `ejbRemove` die die Bean serverseitig repräsentierenden Datenbankeinträge entfernt sowie der durch die Bean belegte Speicherbereich als frei markiert. Alle Versuche nach Aufruf dieser Methode auf der clientseitigen hauptspeicherrepräsentation Wertänderungen durchzuführen führen daher zu einem Fehler.
Die Ermittlung von Referenzen auf existierende EJB-Objekte erfolgt durch die in der Remote-Schnittstelle definierte Methode `findByPrimaryKey`. Der EJB-Container stellt sicher, daß verschiedene Referenzen auf dasselbe EJB-Objekt synchronisiert in die Datenbank abgebildet werden, so daß keine Inkonsistenzen entstehen.

Für den Betrieb einer Enterprise Java Bean ist neben den bisher betrachteten Schnittstellen-Komponenten und der Realisierung der Bean selbst auch ein als *Deployment Deskriptor* bezeichnetes XML-Konfigurationsfile notwendig, welches verschiedene Einstellungsdaten sowie die Schnittstellendaten enthält.
Beispiel 31 zeigt ein Beispiel hierfür:

Beispiel 31: Deployment Deskriptor der Entity Bean

```
(1)<?xml version="1.0" encoding="UTF-8"?>
(2)
(3)<!DOCTYPE ejb-jar PUBLIC '-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans
2.0//EN' 'http://java.sun.com/dtd/ejb-jar_2_0.dtd'>
(4)
(5)<ejb-jar>
(6)  <display-name>Ejb1</display-name>
(7)  <enterprise-beans>
(8)    <entity>
(9)      <display-name>PersonBean</display-name>
(10)      <ejb-name>PersonBean</ejb-name>
(11)      <home>PersonHome</home>
(12)      <remote>Person</remote>
(13)      <ejb-class>PersonBean</ejb-class>
(14)      <persistence-type>Bean</persistence-type>
(15)      <prim-key-class>java.lang.String</prim-key-class>
(16)      <reentrant>False</reentrant>
(17)      <security-identity>
(18)        <description></description>
(19)        <use-caller-identity></use-caller-identity>
(20)      </security-identity>
(21)    </entity>
```

```
(22)  </enterprise-beans>
(23)  <assembly-descriptor>
(24)    <method-permission>
(25)      <unchecked />
(26)      <method>
(27)        <ejb-name>PersonBean</ejb-name>
(28)        <method-intf>Remote</method-intf>
(29)        <method-name>getAge</method-name>
(30)        <method-params />
(31)      </method>
(32)      <method>
(33)        <ejb-name>PersonBean</ejb-name>
(34)        <method-intf>Home</method-intf>
(35)        <method-name>remove</method-name>
(36)        <method-params>
(37)          <method-param>javax.ejb.Handle</method-param>
(38)        </method-params>
(39)      </method>
(40)      <method>
(41)        <ejb-name>PersonBean</ejb-name>
(42)        <method-intf>Home</method-intf>
(43)        <method-name>getHomeHandle</method-name>
(44)        <method-params />
(45)      </method>
(46)      <method>
(47)        <ejb-name>PersonBean</ejb-name>
(48)        <method-intf>Remote</method-intf>
(49)        <method-name>isIdentical</method-name>
(50)        <method-params>
(51)          <method-param>javax.ejb.EJBObject</method-param>
(52)        </method-params>
(53)      </method>
(54)      <method>
(55)        <ejb-name>PersonBean</ejb-name>
(56)        <method-intf>Home</method-intf>
(57)        <method-name>remove</method-name>
(58)        <method-params>
(59)          <method-param>java.lang.Object</method-param>
(60)        </method-params>
(61)      </method>
(62)      <method>
(63)        <ejb-name>PersonBean</ejb-name>
(64)        <method-intf>Remote</method-intf>
(65)        <method-name>getHandle</method-name>
(66)        <method-params />
(67)      </method>
(68)      <method>
(69)        <ejb-name>PersonBean</ejb-name>
(70)        <method-intf>Home</method-intf>
(71)        <method-name>findByPrimaryKey</method-name>
(72)        <method-params>
(73)          <method-param>java.lang.String</method-param>
(74)        </method-params>
(75)      </method>
(76)      <method>
(77)        <ejb-name>PersonBean</ejb-name>
(78)        <method-intf>Home</method-intf>
(79)        <method-name>getEJBMetaData</method-name>
(80)        <method-params />
(81)      </method>
(82)      <method>
(83)        <ejb-name>PersonBean</ejb-name>
(84)        <method-intf>Remote</method-intf>
(85)        <method-name>getPrimaryKey</method-name>
(86)        <method-params />
(87)      </method>
(88)      <method>
(89)        <ejb-name>PersonBean</ejb-name>
(90)        <method-intf>Remote</method-intf>
(91)        <method-name>remove</method-name>
(92)        <method-params />
(93)      </method>
```

```
(94)      <method>
(95)        <ejb-name>PersonBean</ejb-name>
(96)        <method-intf>Remote</method-intf>
(97)        <method-name>getEJBHome</method-name>
(98)        <method-params />
(99)      </method>
(100)   </method-permission>
(101)   <container-transaction>
(102)     <method>
(103)       <ejb-name>PersonBean</ejb-name>
(104)       <method-intf>Home</method-intf>
(105)       <method-name>create</method-name>
(106)       <method-params>
(107)         <method-param>int</method-param>
(108)         <method-param>int</method-param>
(109)         <method-param>int</method-param>
(110)         <method-param>java.lang.String</method-param>
(111)         <method-param>java.lang.String</method-param>
(112)       </method-params>
(113)     </method>
(114)     <trans-attribute>Never</trans-attribute>
(115)   </container-transaction>
(116)   <container-transaction>
(117)     <method>
(118)       <ejb-name>PersonBean</ejb-name>
(119)       <method-intf>Remote</method-intf>
(120)       <method-name>setStreet</method-name>
(121)       <method-params>
(122)         <method-param>java.lang.String</method-param>
(123)       </method-params>
(124)     </method>
(125)     <trans-attribute>Never</trans-attribute>
(126)   </container-transaction>
(127)   <container-transaction>
(128)     <method>
(129)       <ejb-name>PersonBean</ejb-name>
(130)       <method-intf>Remote</method-intf>
(131)       <method-name>getStreet</method-name>
(132)       <method-params />
(133)     </method>
(134)     <trans-attribute>Never</trans-attribute>
(135)   </container-transaction>
(136)   <container-transaction>
(137)     <method>
(138)       <ejb-name>PersonBean</ejb-name>
(139)       <method-intf>Home</method-intf>
(140)       <method-name>remove</method-name>
(141)       <method-params>
(142)         <method-param>javax.ejb.Handle</method-param>
(143)       </method-params>
(144)     </method>
(145)     <trans-attribute>Never</trans-attribute>
(146)   </container-transaction>
(147)   <container-transaction>
(148)     <method>
(149)       <ejb-name>PersonBean</ejb-name>
(150)       <method-intf>Home</method-intf>
(151)       <method-name>remove</method-name>
(152)       <method-params>
(153)         <method-param>java.lang.Object</method-param>
(154)       </method-params>
(155)     </method>
(156)     <trans-attribute>Never</trans-attribute>
(157)   </container-transaction>
(158)   <container-transaction>
(159)     <method>
(160)       <ejb-name>PersonBean</ejb-name>
(161)       <method-intf>Remote</method-intf>
(162)       <method-name>remove</method-name>
(163)       <method-params />
(164)     </method>
(165)     <trans-attribute>Never</trans-attribute>
```

```
(166)    </container-transaction>
(167)    <container-transaction>
(168)      <method>
(169)        <ejb-name>PersonBean</ejb-name>
(170)        <method-intf>Remote</method-intf>
(171)        <method-name>getAge</method-name>
(172)        <method-params />
(173)      </method>
(174)      <trans-attribute>Never</trans-attribute>
(175)    </container-transaction>
(176)    <container-transaction>
(177)      <method>
(178)        <ejb-name>PersonBean</ejb-name>
(179)        <method-intf>Home</method-intf>
(180)        <method-name>findByPrimaryKey</method-name>
(181)        <method-params>
(182)          <method-param>java.lang.String</method-param>
(183)        </method-params>
(184)      </method>
(185)      <trans-attribute>Never</trans-attribute>
(186)    </container-transaction>
(187)  </assembly-descriptor>
(188)</ejb-jar>
(189)
```

Download des Beispiels

1.3 Java Data Objects

Grundidee

Hintergrund des Ansatzes der *Java Data Objects* (JDO) ist es, die bestehenden Schnittstellenmechanismen dahingehend weiterzuentwickeln, daß die Persistenz von Objekten und Objektgraphen für den Programmierer vollständig transparent durch Komponenten der Laufzeitumgebung zur Verfügung gestellt werden.
Gleichzeitig etabliert JDO eine Abstraktion der verschiedenen Speicherungsmöglichkeiten und erlaubt es beispielsweise die dateibasierte Ablage innerhalb des Programmes identisch zur Objektspeicherung in einem Datenbankmanagementsystem zu handhaben. Auf dieser Basis läßt sich im Bedarfsfalle den Persistenzdienstleister auszutauschen ohne Änderungen am Programmcode zu erfordern.
Plakativ wird der Ansatz daher, in Anlehnung an die Zielsetzung der Programmiersprache Java des *write once -- run anywhere*, als *write once -- store anywhere* charakterisiert.

Technik

Um die weitestgehend transparente Handhabung der Objektpersistenz zu gewährleisten bedient sich JDO eines Ansatzes der über das alleinige Angebot einer Programmierschnittstelle hinausreicht. Die Zielsetzung der möglichst einfach handzuhabenden Interaktion mit den generischen Persistenzmechanismen läßt sich zwar durch das Angebot von durch den Programmierer zu implementierenden Schnittstellen und Persistenzklassen erreichen, jedoch ist der Einsatz signifikant komplexer als der bestehenden Persistenzschnittstellen. Darüberhinaus konterkariert der Zwang bei der Programmerstellung vorgegebene Schnittstellen zu berücksichtigen die Zielsetzung weitestgehender Transparenz der angebotenen Speichermechanismen.

Daher führt JDO die Technik der sog. *Bytecodeanreicherung* (engl. *bytecode enhancing*) ein.
Hierbei wird durch eine Programmkomponente vorübersetzer Bytecode so abgeändert, daß die notwendigen Persistenzanweisungen in den bereits erzeugten ausführbaren Bytecode eingewoben werden.
Die benötigte Übersetzerkomponente wird durch die jeweilige JDO-Implementierung zur Verfügung gestellt und muß durch den Programmierer im Bedarfsfalle lediglich geeignet parametrisiert werden.

Im Falle der Referenzimplementierung müssen daher alle Klassen, die Objekte ausprägen, welche persistiert werden sollen, mit dem Werkzeug entsprechend nachbearbeitet werden. Der notwendige Aufruf hat folgende Struktur: `java com.sun.jdori.enhancer.Main -d enhanced de/jeckle/jdotest/Employee.class de/jeckle/jdotest/Employee.jdo`.
Dieser Aufruf reichert die bereits übersetzte Klasse `Emplyoee` innerhalb der Pakethierarchie `de`.

`jeckle.jdotest` um Persistenzdaten an und legt das Ergebnis innerhalb des Dateisystemkatalogs `de/jeckle/jdotest` ab. Zur Anreicherung wird die Konfigurationsdatei `Employee.jdo` herangezogen, die im selben Pfad abgelegt ist wie die Quellcodedatei.

Alternativ zu diesem Ansatz steht auch die Möglichkeit zur Verfügung die benötigten Anweisungen bereits im Quellcode vorzusehen um so dasselbe Resultat zu erzielen, welches durch den Anreicherungsprozeß erzeugt wird. Diese Vorgehensweise hat jedoch wegen der damit verbundenen Aufwände kaum praktische Bedeutung erlangt und wird daher im folgenden nicht vertieft betrachtet.

Die Beispiele dieses Kapitels basieren auf der kostenfrei verfübaren JDO-Referenzimplementierung von SUN. Diese beschränkt zwar die unterstützten Persistenzmechanismen auf ausschließlich dateibasierte Speicherung und sieht keine Ablage in Datenbankmanagmenetsystemen vor. Konzeptionell und programmierseitig ist die Interaktion mit dieser Implementierung jedoch identisch zu kommerziell verfügbaren Lösungen und können daher ohne weiteres auf diese und damit beliebige Persistenzdienstleister übertragen werden.

Konfiguration des Persistenzdienstleisters

Die Abbildung der in der Programmiersprache formulierten Interaktionen auf den konkreten physischen Persistenzdienstleister erfolgt sinnvollerweise an einer für alle JDO-nutzenden Applikationen zugänglichen Stelle im Rahmen einer Property-Datei.
Die Inhalte dieser Datei unterscheiden naturgemäß bei den verschiedenen JDO-Herstellen und inhärent mit dem gewählten Persistenztyp. So benötigt die dateibasierte Objektablage offenkundig andere Festlegungen als der Zugriff auf ein relationales Datenbankmanagementsystem.
Beispiel 32 zeigt die notwendigen Einstellung zur Konfiguration der dateibasierten Speicherung mit der SUN-Referenzimplementierung. Dort wird mit der `PersistenceManagerFactoryClass` diejenige Klasse innerhalb des JDO-Rahmenwerkes benannt, welche dem Programmierer die Persistenzdienste zur Verfügung stellt. `ConnectionURL` bildet das Bindeglied der Abbildung auf die physische Datei und benennt daher den Speicherort aller persistierten Objekte. Die zusätzlichen Angaben dienen der Authentisierung und Zugriffssteuerung beim Zugriff auf die erstellte Datei.

Beispiel 32: Konfiguration einer JDO-Implementierung

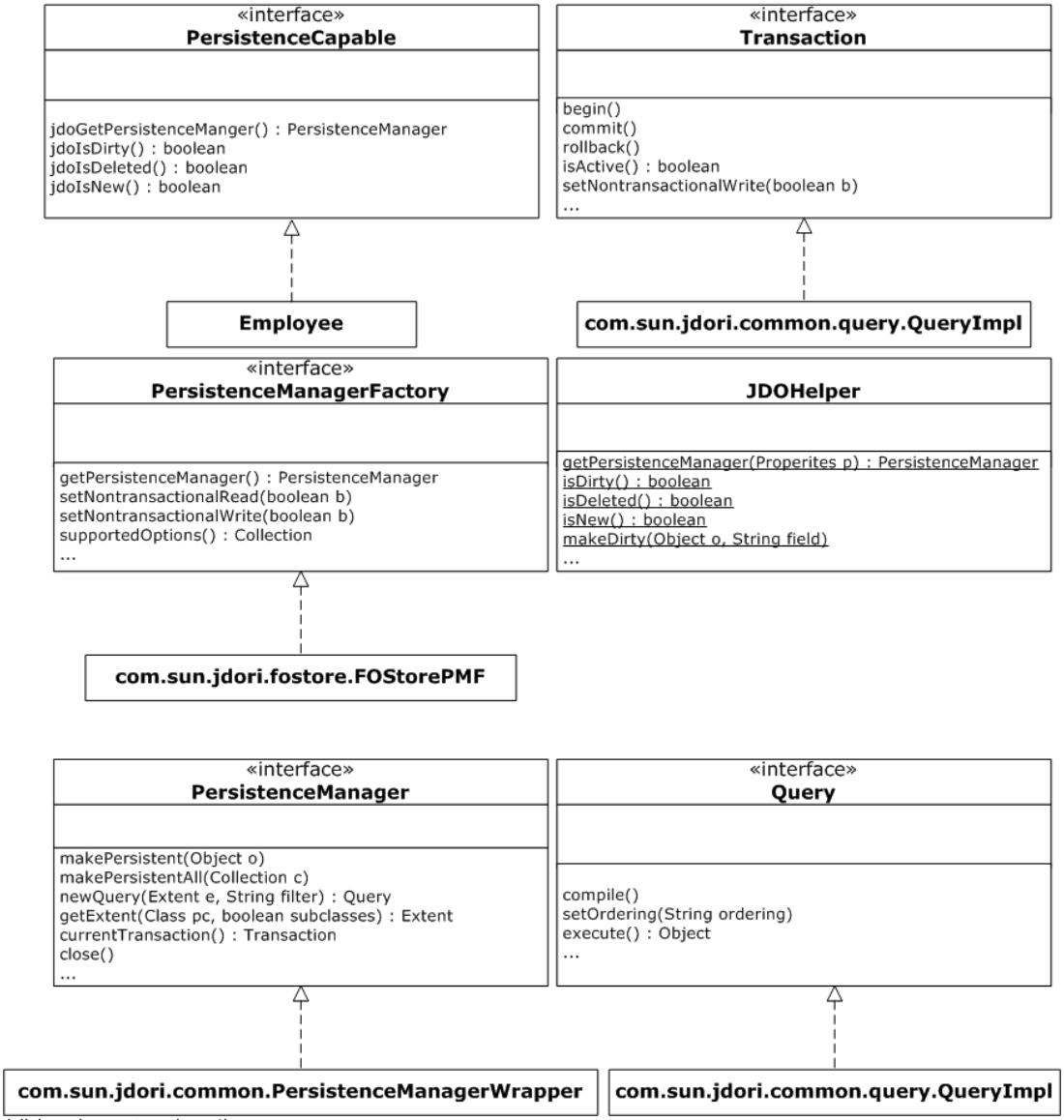
```
(1)javax.jdo.PersistenceManagerFactoryClass=com.sun.jdori.fostore.FOStorePMF
(2)javax.jdo.option.ConnectionURL=fostore:jdoriDB
(3)javax.jdo.option.ConnectionUserName=mario
(4)javax.jdo.option.ConnectionPassword=thePassword
```

Download des Beispiels

Struktur der JDO-API

Die JDO-API ist im Rahmen des Java Community Prozesses als Java-Schnittstellensammlung nebst zugehöriger Semantikdefinition spezifiziert. Die Implementierung der Schnittstellen erfolgt durch den Anbieter der jeweiligen JDO-Implementierung und erfolgt auf den jeweiligen Persistenztyp abgestimmt.
Abbildung 5 zeigt die grundlegenden Schnittstellen der JDO-API sowie die sie anbietenden Klassen der Referenzimplementierung.

Abbildung 5: *Grundlegende Struktur der JDO-API*



(click on image to enlarge!)

Die Schnittstelle `PersistenceCapable` bildet das Rückgrat der gesamten Persistenzbemühungen. Jede Klasse, deren Speicherung durch JDO verwaltet werden soll (in Beispiel die Klasse `Employee`) muß diese Schnittstelle zwingend implementieren.
Typischerweise erfolgt diese Implementierung jedoch nicht direkt durch den Applikationsprogrammierer, sondern wird im Rahmen der Bytecodeanreicherung nachträglich hinzugefügt.
Zur Interaktion mit Klassen, deren Implementierung der in `PersistenceCapable` deklarierten Methoden erst nach dem initialen Übersetzungsvorgang hinzugefügt werden kann der JDO-Anbieter die Hilfsklasse `JDOHelper` anbieten. Diese definiert verschiedene, ausschließlich als statisch deklarierte, Methoden um mit Objekten von Klassen zu operieren, als würden diese die Schnittstelle `PersistenceCapable` umsetzen, ohne deren Klassen zur tatsächlichen Schnittstellenimplementierung verpflichten.
Damit stellt `JDOHelper` die unabdingbare Voraussetzung zur Anwendungsentwicklung unter Verwendung der Bytecodeanreicherung dar, da diese erst nach dem Übersetzungsvorgang Implementierungen derjenigen Schnittstellen hinzufügt, die bereits im Code verwendeten wurden. Ferner bietet die Klasse die Möglichkeit den aktuellen Persistenzzustand eines JDO-verwalteten Objektes auszulesen.

Zur Erzeugung von Objekten, die später den Zugriff auf das physische Speichermedium regeln dienen die Umsetzungen der Schnittstelle `PersistenceManagerFactory`. Sie erlaubt die Parametrisierung und Verwaltung der Verbindung zum Persistenzmedium. Bereitgestellt wird die Implementierung, im Falle der Referenzimplementierung, durch die Klasse `com.sun.jdori.fostore.FOStorePMF`.
Die Verbindung zwischen Schnittstelle und tatsächlicher Implementierung wird im Rahmen der in Beispiel 32 gezeigten JDO-Konfiguration definiert. Zum Wechsel des Persistenzanbieters -- etwa von der durch die Referenzimplementierung angebotenen dateibasierten Speicherung auf eine datenbankgestützte Umsetzung -- genügt im die Abänderung dieses Eintrages in der Konfigurationsdatei.

Klassen, welche die Schnittstelle PersistenceManagerFactory implementieren, werden zur Erzeugung von sog. PersistenceMangern herangezogen. Umsetzungen dieser Schnittstelle (im Falle der Referenzimplementierung ist dies die Klasse com.sun.jdori.common.PersistenceManagerWrapper) dienen zur Interaktion mit der Persistenzveraltung innerhalb der JDO nutzenden Applikation. Alle Änderungen des Zustandes eines persistenten Objektes werden durch diese Klasse abgewickelt.

JDO wickelt sämtliche Zugriffe auf die persistenten Daten transaktionsgesichert ab. Dieser Mechanismus wird auf der abstrakten Ebene der API durch die Schnittstelle Transaction definiert und steht daher für alle Persistenzanbieter gleichermaßen zur Verfügung.
Die Schnittstelle definiert alle zur Transaktionssteuerung benötigten Operationen (darunter begin, commit und rollback) an.
Im Falle der Referenzimplementierung wird die Schnittstelle durch die Klasse com.sun.jdori.common.query.QueryImpl umgesetzt.

Zusätzlich sieht JDO eine abstrakte Möglichkeit zur Formulierung von Anfragen auf den verwalteten Datenbestand vor. Die notwendige Schnittstelle wird durch Query bereitgestellt.
Hierfür müssen die verschiedenen JDO-Implementierungen ebenfalls eigene Umsetzungen vorsehen.

Erzeugen eines persistenten Objektspeichers

Zur Erzeugung eines Objektspeichers ist bereits die Nutzung der Implementierungen der zentralen JDO-Schnittstellen sowie die der Transaktionssteuerung notwendig.
Das Beispiel zeigt die notwendigen Schritte zur Erzeugung eines persistenten Objektspeichers.
Zunächst lädt das Beispiel die Konfiguration aus der Eigenschaftsdatei des Beispiels 32.
Anschließend wird durch die mit true belegte implementierungsspezifische Eigenschaft com.sun.jdori.option.ConnectionCreate festgelegt, daß im Rahmen des Verbindungsaufbaus auch notwendigenfalls der Objektspeicher neu erzeugt wird.
Die Interaktion mit JDO beginnt durch die Erzeugung eines PersistenceManagerFactory konformen Objektes durch den Aufruf getPersistenceManagerFactory unter Auswertung der zuvor geladenen und ergänzten Konfigurationseigenschaften.
Nach der Erzeugung des Factory-Objektes kann mittels diesem durch den Aufruf getPersistenceManager ein Objekt erzeugt werden, das die Interaktion mit dem Objektspeicher bereitstellt. Durch die Ermittlung des Persistenzmanagers wird gleichzeitig eine Verbindung zum Persistenzanbieter aufgebaut.
Ausgehend von diesem Verwaltungsobjekt kann durch Definition einer „leeren“ Transaktion -- d.h. einer Transaktion, die jenseits der Erzeugung des transaktionalen Kontexts und seines Abschlusses mit *committ*, keine Operationen definiert -- der Objektspeicher erzeugt werden.
Den Abschluß der Interaktion mit dem Objektspeicher bildet die Beendigung der Verbindung durch Ausführung der Methode close des Verbindungsobjektes.

Beispiel 33: Erzeugung eines persistenten Objektspeichers

```
(1)import java.io.IOException;
(2)import java.io.InputStream;
(3)import java.util.Properties;
(4)
(5)import javax.jdo.JDOHelper;
(6)import javax.jdo.PersistenceManager;
(7)import javax.jdo.PersistenceManagerFactory;
(8)import javax.jdo.Transaction;
(9)
(10)public class JDOCreateDB {
(11)    public static void main(String args[]) {
(12)            Properties props = new Properties();
(13)            try {
(14)                    InputStream is = ClassLoader.getSystemResourceAsStream("jdo.
properties");
(15)                    props.load(is);
(16)                    props.put("com.sun.jdori.option.ConnectionCreate","true");
(17)            } catch (IOException ioe) {
(18)                    System.out.println("Error loading properties");
(19)                    System.exit(1);
(20)            }
(21)            PersistenceManagerFactory pmf = JDOHelper.
getPersistenceManagerFactory(props);
(22)            PersistenceManager pm = pmf.getPersistenceManager();
(23)
(24)            Transaction tx = pm.currentTransaction();
(25)            tx.begin();
```

```
(26)            tx.commit();
(27)            pm.close();
(28)    }
(29)}
```

Download des Beispiels

Parametrisierung der Persistenz

Grundsätzlich können Ausprägungen jeder beliebigen Javaklasse durch JDO persistiert werden, solange diese Klassen die Schnittstelle `PersistentCapable` explizit im Quellcode implementieren oder die benötigte Implementierung im Rahmen der Bytecodeanreicherung hinzugefügt wird.
Zur Steuerung des konkreten Persistenzverhaltens wird eine zusätzliche Konfigurationsdatei benötigt. Diese bedient sich der bekannten XML-Sytnax und definiert das Persistenzverhalten der durch JDO zu verwaltenden Klasseninstanzen näher.
Beispiel 34 zeigt zunächst die zu persistierende Klasse `Employee`.

Beispiel 34: Zu persistierende Javaklasse

```
(1)package de.jeckle.jdotest;
(2)
(3)import java.util.HashSet;
(4)import java.util.Iterator;
(5)
(6)public class Employee {
(7)     private String name;
(8)     private String department;
(9)     private HashSet projects = new HashSet();
(10)
(11)    public String getName() {
(12)            return name;
(13)    }
(14)    public String getDepartment() {
(15)            return department;
(16)    }
(17)    public void setName(String name) {
(18)            this.name = name;
(19)    }
(20)    public void setDepartment(String department) {
(21)            this.department = department;
(22)    }
(23)    public void addProject(String project) {
(24)            projects.add(project);
(25)    }
(26)
(27)    public String toString() {
(28)            String result="Employee named "+name+" works in department
"+department;
(29)            result+="\nworks in: ";
(30)            Iterator i = projects.iterator();
(31)            while (i.hasNext()) {
(32)                    result+=(String) i.next();
(33)                    result+=", ";
(34)            }
(35)            return (result);
(36)    }
(37)}
```

Download des Beispiels

Die Nutzung JDO-gestützter Objektpersistenz impliziert keinerlei Modifikationen oder Ergänzungen am Quellcode. Ebenso sind keinerlei Umsetzungskonventionen einzuhalten, die im Beispiel definierten `get-` und `set`-Methoden dienen lediglich der vereinfachten Interaktion.

Das Beispiel 35 illustriert eine Parameterdatei zur Definition des spezifischen Persistenzverhaltens von Objekten der Klasse `Employee`.

Beispiel 35: Parametrisierung der Objektpersistenz

```
(1)<?xml version="1.0"?>
(2)<!DOCTYPE jdo PUBLIC "-//Sun Microsystems, Inc.//DTD Java Data Objects Metadata
1.0//EN" "http://java.sun.com/dtd/jdo_1_0.dtd">
(3)<jdo>
(4)     <package name="de.jeckle.jdotest">
(5)             <class name="Employee">
(6)                     <field
(7)                             name="name"
(8)                             persistence-modifier="persistent"/>
(9)                     <field
(10)                            name="department"
(11)                            persistence-modifier="persistent"/>
(12)                    <field
(13)                            name="projects">
(14)                            <collection
(15)                                    element-type="java.lang.String"
(16)                                    embedded-element="true"/>
(17)                    </field>
(18)            </class>
(19)    </package>
(20)</jdo>
```

Download des Beispiels

Die XML-Datei definiert zunächst den Paket- und Klassennamen der zu persistierenden Klasse mittels des Attributs `name` der XML-Elemente `package` und `class`.
Innerhalb eines `class`-Elements kann für jedes Attribut der Javaklasse ein mit `field` benanntes Element zur näheren Charakterisierung des Speicherungsverhaltens angegeben werden.
Ein solches Element trägt zunächst im Attribut `name` den klassenweit eindeutigen Namen des Attributs und erlaubt die Festlegung des spezifischen Persistenzverhaltes mittels der Belegung des Attributs `persistence-modifier`. Ist dieses mit dem Wert `persistent` versehen, so wird ein so gekennzeichnetes Attribut durch JDO im Datenspeicher persistiert. Trägt das XML-Attribut den Wert none, so wird das Javaattribut bei der Abbildung in den JDO-Datenspeicher ignoriert.
Zusätzlich besteht die Möglichkeit durch die Belegung mit `transactional` die Zwischenspeicherung des Attributwertes während der Abarbeitung einer Transaktion zu erzwingen, um so eine spätere Wiederherstellung (nach einem Aufruf von `rollback`) zu gewährleisten. Jedoch werden Felder, die so gekennzeichnet sind, nicht persistent in den Datenspeicher übernommen, sondern stehen nur während der Programmlaufzeit zur Verfügung.
Fehlt diese Spezifikation zu einem Attribut in der XML-Datei, so wird vorgabegemäß die Belegung mit `persistent` angenommen, sofern es in der beherbergenden Javaklasse nicht als `static`, `transient` oder `final` ausgewiesen ist.

Attribute vom Typ einer Sammlungsklasse, wie sie durch die *Collection API* definiert werden müssen zusätzlich mit einem `collection`-Element, welches innerhalb des `field`-Elements plaziert ist, charakterisiert. Das `collection`-Element spezifiziert durch sein Attribut `element-type` den Typ der Elemente in der Sammlung festlegt. Zusätzlich kann durch das Boole'sche-Attribut `embedded-element` gesteuert werden, ob die Inhalte des Sammlungsobjektes zusammen mit dem die Sammlung referenzierenden Objekt persistiert werden sollen.

Das Beispiel legt für alle Attribute der Klasse `Employee` ihre persistente Speicherung fest (Belegung des XML-Attributs `persistence-modifier` für alle Attribute `persistent`); ebenso wird die in Objekten des Typs *Employee*, unter dem Namen `projects`, enthaltene Sammlungsinstanz einschließlich ihrer Inhaltsobjekte des Standard-API-Typs `String` dauerhaft abgespeichert.

Über diese Festlegungen hinaus gestattet das Parametrisierungsformat die Festlegung spezifischer Konsistenzsemantik in Gestalt der Auszeichnung eines *Primärschlüssels*. Dieses aus dem relationalen Modell bekannte Konstrukt fordert die Eindeutigkeit eines Attributs oder einer Kombination von Attributen über die gesamte Menge der Ausprägungen eines Typs.
Durch die Unterstützung als abstraktes JDO-Konstrukt steht dieses Konzept zur Konsistenzsicherung auch für Applikationen zur Verfügung, die sich nicht relationaler Datenbanken als Persistenzdienstleister bedienen.
Zur Realisierung des Primärschlüsselkonzeptes ist das als Schlüssel zu interpretierende Attribut in der XML-Beschreibung zusätzlich mit dem XML-Attribut `primary-key` zu versehen, welches den Wert `true` tragen muß. Zusätzlich ist innerhalb des Elements `class` diejenige Klasse anzugeben, welche das Attribut beherbergt, das als Schlüssel herangezogen werden soll.
36 zeigt die notwendigen Modifikationen an der Parameterdatei des Beispiels 35 um das Java-Attribut `name` als Primärschlüssel festzulegen. Die primärschlüsselanbietende Klasse ist in diesem Falle die Klasse `Employee` selbst, weshalb sich ihr Name auch im XML-Attribut `objectid-class` des

class-Elements findet.

Beispiel 36: Parametrisierung der Objektpersistenz und Definition eines Primärschlüssels

```
(1)<?xml version="1.0"?>
(2)<!DOCTYPE jdo PUBLIC "-//Sun Microsystems, Inc.//DTD Java Data Objects Metadata
1.0//EN" "http://java.sun.com/dtd/jdo_1_0.dtd">
(3)<jdo>
(4)     <package name="de.jeckle.jdotest">
(5)             <class name="Employee"
(6)                     objectid-class="Employee">
(7)                 <field
(8)                         primary-key="true"
(9)                         name="name"
(10)                        persistence-modifier="persistent"/>
(11)                <field
(12)                        name="department"
(13)                        persistence-modifier="persistent"/>
(14)                <field
(15)                        name="projects">
(16)                        <collection
(17)                                element-type="java.lang.String"
(18)                                embedded-element="true"/>
(19)                </field>
(20)            </class>
(21)    </package>
(22)</jdo>
```

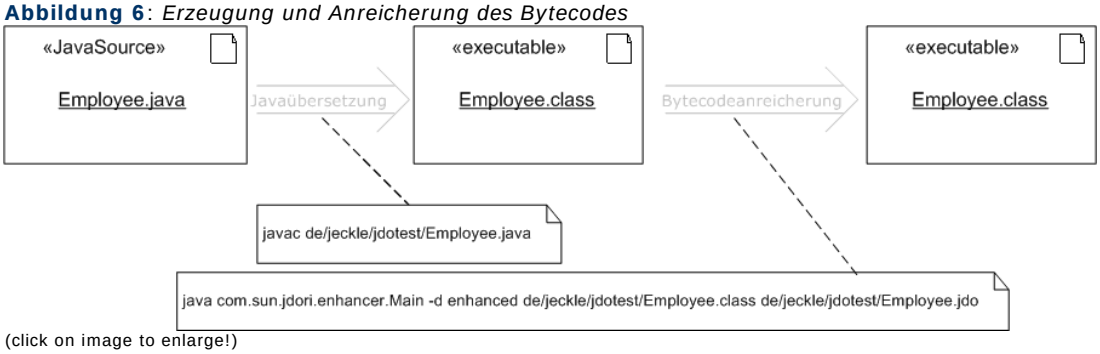
Download des Beispiels

Konsequenz der Einführung eines Primärschlüsselattributs ist die Überwachung der damit einhergehenden Konsistenzbedingungen durch das JDO-Laufzeitsystem. So führen Versuche zwei Objekte, die sich in der Belegung des als Primärschlüssel definierten Attributs nicht unterscheiden ebenso zu Fehlern wie schreibende Zugriffe auf dergestalt ausgezeichnete Attribute.

Anreicherung des Bytecodes

Voraussetzung der Persistenzverwaltung eines Objektes durch JDO ist die entsprechende Modifikation dieses Objektes, konkret die Implementierung der in `PersistenceCapable` festgelegten Operationen durch Methoden der objekterzeugenden Klasse.
Dies wird jedoch nur in Ausnahmefällen durch den Applikationsprogrammierer direkt vorgenommen. Häufigste Eionsatzform der JDO-API ist die Anwendung der Bytecodeanreicherung, welche die Implementierung der notwendigen Funktionalität automatisiert vornimmt und diese nach dem eigentlichen Übersetzungsvorgang in den erstellten Bytecode einbringt.
Abbildung 6 zeigt die daher notwendigen zwei Übersetzungsschritte.

Abbildung 6: *Erzeugung und Anreicherung des Bytecodes*



(click on image to enlarge!)

Die Illustration versammelt die zur Erzeugung und Anreicherung des Bytecodes der per JDO zu persistierenden Klasse `Employee` aus Beispiel 34. Zur Anreicherung des Bytecodes werden die in Beispiel 35 getroffenen Parametrisierungen herangezogen.
Zunächst wird der im Paket `de.jeckle.jdotest` abgelegte Quellcode `Employee.java` mit dem Javacompiler in (gewöhnlichen) Bytecode übersetzt.
Anschließend wird dieser vermöge des in der JDO-Referenzimplementierung vorhandenen Werkzeuges `Enhancer` um die Implementierung der in der Schnittstelle `PersistenceCapable` definierten Operationen angereichert. Hierzu wird dem Enhancer (bereitgestellt durch die Klasse `com.sun.jdori.enhancer.Main` zunächst das Zielverzeichnis des zu erzeugenden Bytecodes mittels des Parameters d übergeben. Naheliegernderweise kann der aus dem ursprünglichen Bytecode durch Erweiterung erzeugte nicht die Ausgangsdatei überschreiben, daher wird der angereicherte Bytecode im Verzeichnis enhanced gespeichert. Zusätzlich ist dem Enhancer der

vollqualifizierte Name der anzureichernden Klasse sowie der vollqualifizierte Pfad der Parameterdatei (im Beispiel: `de/jeckle/jdotest/Employee.jdo`) zu übergeben. Diese muß im Falle des Einsatzes der Referenzimplementierung zwingend die Extension `jdo` besitzen.

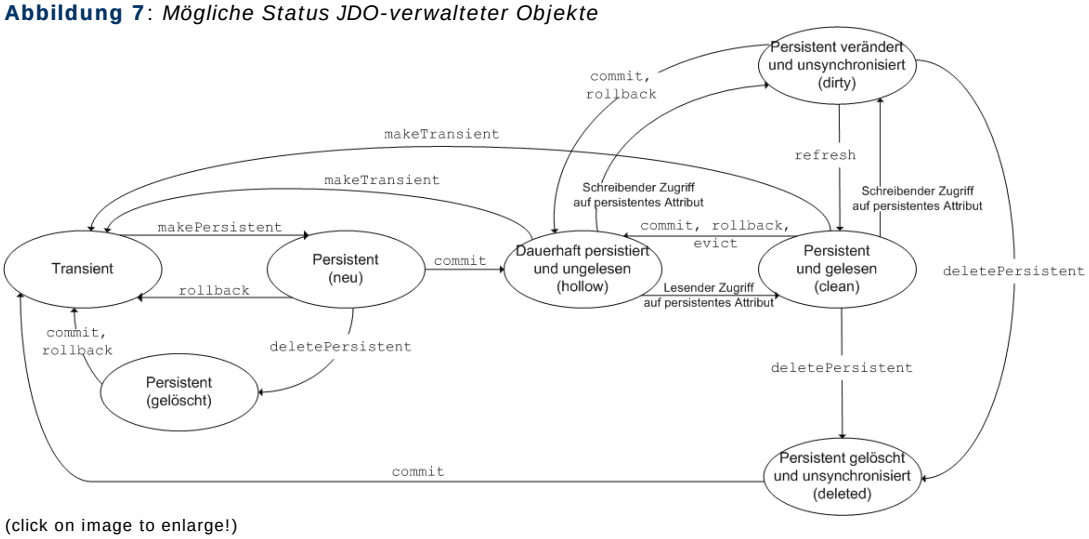
Status JDO-verwalteter Objekte

Im Zusammenspiel zwischen transienter Objektverwaltung durch die Applikation im Hauptspeicher und persistenter Objektverwaltung durch JDO im Hintergrundspeicher werden verschiedene Status eines verwalteten Objekts unterschieden zwischen denen explizite Übergänge durch API-Aufrufe vorgegeben sind bzw. implizit durch Operationen auf den involvierten Objekten bestehen.
Im Detail werden folgende Status unterschieden:

- **Transient**: Instantiierte Objekte im Hauptspeicher. Hierunter fallen alle noch nicht innerhalb von Transaktionen persistierten Objekte ebenso wie unangereicherte Javaobjekte und solche die ausschließlich über Attribut verfügen, die als in der XML-Parametrisierungsdatei als `transient` gekennzeichnet sind.
- **Persistent (neu)**: Objekte, die innerhalb einer laufenden (d.h. weder durch `commit` noch `rollback` abgeschlossenen) Transaktion erzeugt wurden.
  Endet eine Transaktion, etwa durch Programmabbruch, in diesem Zustand, so werden die Objekte mit diesem Status nicht dauerhaft gespeichert.
- **Persistent (gelöscht)**: Objekte, die in einer noch nicht abgeschossenen Transaktion persistiert und anschließend gelöscht wurden.
  Endet eine Transaktion, etwa durch Programmabbruch, in diesem Zustand, so werden die Objekte mit diesem Status nicht dauerhaft gespeichert, da sowohl der Persistierungs- als auch der anschließende Löschvorgang noch nicht durch `commit` bestätigt wurden.
- **Dauerhaft persistiert und ungelesen (hollow)**: Objekte, die durch Abschluß einer Transaktion mit `commit` dauerhaft gespeichert wurden und auf die noch kein Zugriff (weder lesend noch schreibend) erfolgte.
- **Persistent und gelesen (clean)**: Objekte, die persistent im Hintergrundspeicher abgelegt wurden und auf die bisher lediglich lesende Zugriffe erfolgten.
- **Persistent verändert und unsynchronisiert (dirty)**: Persistentes Objekt, dessen Inhalt im Rahmen einer noch nicht abgeschlossenen Transaktion verändert wurde.
  Wurden zwar Attributinhalte eines persistenten Objektes verändert, jedoch keine Wertänderungen vorgenommen, d.h. in ein Attribut wird mit demselben Wert belegt, den es bereits enthält, dann steht es dem JDO-Implementierer frei diese Schreiboperation so zu implementieren, daß nicht der Zustand *dirty* eingenommen wird.
  Zusätzlich kann jedes Objekt durch Aufruf der API-Methode `makeDirty` manuell in diesen Zustand versetzt werden.
  Als Folge der Schreiboperation im Hauptspeicher differieren dessen Inhalte von denen des persistenten Objektspeichers.
  Durch Aufruf der API-Methode `refresh` werden die Inhalte von Haupt- und Hintergrundspeicher synchronsiert, d.h. Inhalte des Hintergrundspeichers werden in den Hauptpeicher übernommen.
- **Persistent gelöscht und unsynchronisiert (deleted)**: Persistentes Objekt, das im Rahmen einer noch nicht abgeschlossenen Transaktion gelöscht wurde.
  Als Folge der Löschoperation im Hauptspeicher differieren dessen Inhalte von denen des persistenten Objektspeichers und alle Leseoperationen auf nicht-Primärschlüsselfelder führen zu Laufzeitfehlern.

Abbildung 7 zeigt die verschiedenen JDO-Status sowie die Ereignisse, die zu Zustandsübergängen führen, in der Übersicht.

Abbildung 7: *Mögliche Status JDO-verwalteter Objekte*



(click on image to enlarge!)

Speicherung von Objekten

Zur Speicherung von Objekten, deren Klassen durch den Bytecodeanreicherungsprozeß nachbearbeitet wurden, bietet die JDO-API die Aufrufe `makePersistent` und `makePersistentAll` an. Diese werden innerhalb eines Transaktionskontextes als Methoden eines `PersistenceManager`-Objekte ausgeführt.
Beispiel 37 zeigt die Speicherung von drei Objekten der Klasse `Emplyoee`, deren übersetzter Bytecode durch Anreicherung zur JDO-Kompatibilität modifiziert wurde.
Zunächst wird mit `empCol` vom Standardtyp `Vector` eine Sammlungsobjekt zur Aufnahme von Objektreferenzen definiert. Dieser Objektsammlung werden die Referenzen auf die erzeugten `Emplyoee`-Objekte (`emp1`, `emp2` und `emp3`) hinzugefügt.
Als Voraussetzung der Interaktion mit dem Objektspeicher muß zunächst eine Transaktion eröffnet werden. Hierzu muß zunächst durch Aufruf der Methode `currentTransaction` die der `PersistenceManager`-Instanz zugeordnete Transaktion ermittelt werden. Ausgehend vom gelieferten Ergebnisobjekt kann durch Ausführung der Methode `begin` eine neuer Transaktionskontext eröffnet werden.
Der Aufruf von `makePersistentAll` persistiert bei Übergabe der Objektsammlung alle in der Sammlung referenzierten Objekte. Alternativ können Einzelobjekte durch die Methode `makePersistent` in den Zustand dauerhafter Speicherung überführt werden.
Zur Übernahme in den Hintergrundspeicher muß der Transaktionskontext durch Aufruf von `commit` abgeschlossen werden. Der Aufruf von `rollback` würde stattdessen alle in der Transaktion vorgenommenen Änderungen verwerfen und auf den im Hintergrundspeicher verwalteten Datenzustand zurückgesetzt.

Beispiel 37: Speicherung von Objekten mit JDO

```
(1)import java.io.IOException;
(2)import java.io.InputStream;
(3)import java.util.Properties;
(4)import java.util.Vector;
(5)
(6)import javax.jdo.JDOHelper;
(7)import javax.jdo.PersistenceManager;
(8)import javax.jdo.PersistenceManagerFactory;
(9)
(10)import de.jeckle.jdotest.Employee;
(11)
(12)public class JDOStoreObj {
(13)    public static void main(String args[]) {
(14)            Properties props = new Properties();
(15)            try {
(16)                    InputStream is =
(17)                            ClassLoader.getSystemResourceAsStream("jdo.
properties");
(18)                    props.load(is);
(19)            } catch (IOException ioe) {
(20)                    System.out.println("Error loading properties");
(21)                    System.exit(1);
(22)            }
(23)            PersistenceManagerFactory pmf =
(24)                    JDOHelper.getPersistenceManagerFactory(props);
(25)            PersistenceManager pm = pmf.getPersistenceManager();
(26)
(27)            Vector empCol = new Vector();
(28)
(29)            Employee emp1 = new Employee();
(30)            emp1.setName("Mario Jeckle");
(31)            emp1.setDepartment("D001");
(32)            emp1.addProject("P001");
(33)            emp1.addProject("P002");
(34)            empCol.add(emp1);
(35)
(36)            Employee emp2 = new Employee();
(37)            emp2.setName("John DoeX");
(38)            emp2.setDepartment("D003");
(39)            emp2.addProject("P001");
(40)            emp2.addProject("P042");
(41)            empCol.add(emp2);
(42)
(43)            Employee emp3 = new Employee();
(44)            emp3.setName("John Doe");
(45)            emp3.setDepartment("B042");
```

```
(46)            empCol.add(emp3);
(47)
(48)            Employee emp4 = new Employee();
(49)            emp4.setName("Barnie Bar");
(50)            emp4.setDepartment("B042");
(51)
(52)            pm.currentTransaction().begin();
(53)            pm.makePersistentAll(empCol);
(54)            pm.makePersistent(emp4);
(55)
(56)            pm.currentTransaction().commit();
(57)            pm.close();
(58)    }
(59)}
```

Download des Beispiels

Würde wie im Beispiel 36 gezeigt das Attribut name der Klasse Employee als Primärschlüssel definiert sein, so würde der Persistierungsversuch des durch emp3 referenzierten Objektes einen Laufzeitfehler liefern, da mit emp2 bereits ein Objekt mit derselben Belegung des Attributs name persistiert wurde.

Rücksetzen von Transaktionen

Treten während der Interaktion mit dem Persistenzspeicher, d.h. während eines noch nicht mit commit abgeschlossenen Transaktionskontextes Fehler auf, so können durch Aufruf der Methode rollback alle im aktuellen Kontext vorgenommen Änderungen auf den Stand vor Beginn der Transaktion zurückgesetzt werden.
Beispiel 38 zeigt das Verhalten der Methode rollback am Beispiel. Durch die Schreiboperation innerhalb der geöffneten Transaktion wird der Wert des Attributs name zwar verändert, jedoch durch Aufruf von rollback wieder auf den ursprünglichen Wert zurückgesetzt.

Beispiel 38: Transaktionen mit JDO

```
(1)import java.io.IOException;
(2)import java.io.InputStream;
(3)import java.util.Iterator;
(4)import java.util.Properties;
(5)
(6)import javax.jdo.JDOHelper;
(7)import javax.jdo.PersistenceManager;
(8)import javax.jdo.PersistenceManagerFactory;
(9)
(10)import de.jeckle.jdotest.Employee;
(11)
(12)public class JDORollback {
(13)
(14)    public static void main(String args[]) {
(15)            Properties props = new Properties();
(16)            try {
(17)                    InputStream is =
(18)                            ClassLoader.getSystemResourceAsStream("jdo.
properties");
(19)                    props.load(is);
(20)            } catch (IOException ioe) {
(21)                    System.out.println("Error loading properties");
(22)                    System.exit(1);
(23)            }
(24)            PersistenceManagerFactory pmf =
(25)                    JDOHelper.getPersistenceManagerFactory(props);
(26)            PersistenceManager pm = pmf.getPersistenceManager();
(27)
(28)            Employee e = new Employee();
(29)            e.setName("Marta Mayer");
(30)
(31)            pm.currentTransaction().begin();
(32)            pm.makePersistent(e);
(33)            pm.currentTransaction().commit();
(34)            displayPersistedObjects(pm);
```

```
(35)
(36)            System.out.println("Martha gets married and changes her name");
(37)
(38)            pm.currentTransaction().begin();
(39)            e.setName("Marta Smith");
(40)            pm.makePersistent(e);
(41)            displayPersistedObjects(pm);
(42)            System.out.println("Suppose and error happens now ...\nRolling
back");
(43)            pm.currentTransaction().rollback();
(44)
(45)            displayPersistedObjects(pm);
(46)    }
(47)    private static void displayPersistedObjects(PersistenceManager pm) {
(48)            Iterator i = pm.getExtent(Employee.class, false).iterator();
(49)            while (i.hasNext()) {
(50)                    System.out.println((Employee) i.next());
(51)            }
(52)    }
(53)}
```

Download des Beispiels

Die Ausführung des Beispiels liefert folgende Ausgabe:

```
Employee named Marta Mayer works in department null
works in projects:
Martha gets married and changes her name
Employee named Marta Smith works in department null
works in projects:
Suppose and error happens now ...
Rolling back
Employee named Marta Mayer works in department null
works in projects:
```

Schreiboperationen ohne Transaktionsschutz

Ist in bestimmten Anwendungsfällen die Arbeit ohne Transaktionsschutz -- und damit ohne die Möglichkeit der expliziten Rücksetzung von Änderungen mittels `rollback` oder der impliziten Rücksetzung nach einem Systemausfall -- gewünscht, so kann dies durch Aktivierung der Schreibfunktionalität ohne Transaktionsschutz erreicht werden.
Hierzu muß de Methode `setNontransactionalWrite` mit dem Übergabeparameter `true` für eine Transaktion aufgerufen werden.
Das nachfolgende Beispiel zeigt als Modifikation von Beispiel 37 die persistente Übernahme einer Wertänderung ohne Transaktionsschutz.

Beispiel 39: Schreiboperation ohne Transaktionsschutz

```
(1)import java.io.IOException;
(2)import java.io.InputStream;
(3)import java.util.Iterator;
(4)import java.util.Properties;
(5)
(6)import javax.jdo.JDOHelper;
(7)import javax.jdo.PersistenceManager;
(8)import javax.jdo.PersistenceManagerFactory;
(9)
(10)import de.jeckle.jdotest.Employee;
(11)public class JDONonTransact {
(12)    public static void main(String args[]) {
(13)            Properties props = new Properties();
(14)            try {
(15)                    InputStream is =
(16)                            ClassLoader.getSystemResourceAsStream("jdo.
properties");
(17)                    props.load(is);
(18)            } catch (IOException ioe) {
(19)                    System.out.println("Error loading properties");
(20)                    System.exit(1);
```

```
(21)            }
(22)            PersistenceManagerFactory pmf =
(23)                    JDOHelper.getPersistenceManagerFactory(props);
(24)            PersistenceManager pm = pmf.getPersistenceManager();
(25)
(26)            Employee e = new Employee();
(27)            e.setName("Marta Mayer");
(28)            pm.currentTransaction().setNontransactionalWrite(true);
(29)
(30)            pm.currentTransaction().begin();
(31)            pm.makePersistent(e);
(32)            pm.currentTransaction().commit();
(33)            displayPersistedObjects(pm);
(34)
(35)            //martha gets married and changes her name
(36)
(37)            e.setName("Marta Smith");
(38)
(39)            displayPersistedObjects(pm);
(40)    }
(41)    private static void displayPersistedObjects(PersistenceManager pm) {
(42)            Iterator i = pm.getExtent(Employee.class, false).iterator();
(43)            while (i.hasNext()) {
(44)                    System.out.println((Employee) i.next());
(45)            }
(46)    }
(47)}
```

Download des Beispiels

Traversierung des persistenten Objektbestandes

Zugriffe auf alle im Hintergrundspeicher verwalteten Objekte werden ebenfalls einheitlich durch Methoden der Implementierung der Schnittstelle `PersistenceManager` abgewickelt. Zur Traversierung des vollständigen Bestandes aller Instanzen einer Klasse bietet diese Schnittstelle die Operation `getExtent` an. Sie liefert alle Elemente der Extension (d.h. der Gesamtheit von Ausprägungen) einer gegebenen Klasse.
Beispiel 40 zeigt die Verwendung der Methode. Als Parameter wird diejenige Klasse übergeben, deren Ausprägungen zu ermitteln sind. Zusätzlich kann durch einen Boole'schen Schalter gesteuert werden, ob auch Subklassen der übergebenen Klasse retourniert werden sollen.
Der Aufruf liefert eine Sammlung von Objekten des Typs, welcher der Methode `getExtent` übergeben wurde.

Beispiel 40: Traversierung des Objektbestandes

```
(1)import java.io.IOException;
(2)import java.io.InputStream;
(3)import java.util.Iterator;
(4)import java.util.Properties;
(5)
(6)import javax.jdo.JDOHelper;
(7)import javax.jdo.PersistenceManager;
(8)import javax.jdo.PersistenceManagerFactory;
(9)
(10)import de.jeckle.jdotest.Employee;
(11)
(12)public class JDOListObj {
(13)    public static void main(String args[]) {
(14)            Properties props = new Properties();
(15)            try {
(16)                    InputStream is =
(17)                            ClassLoader.getSystemResourceAsStream("jdo.
properties");
(18)                    props.load(is);
(19)            } catch (IOException ioe) {
(20)                    System.out.println("Error loading properties");
(21)                    System.exit(1);
(22)            }
(23)            PersistenceManagerFactory pmf =
(24)                    JDOHelper.getPersistenceManagerFactory(props);
```

```
(25)            PersistenceManager pm = pmf.getPersistenceManager();
(26)
(27)            Iterator i = pm.getExtent(Employee.class, false).iterator();
(28)            while (i.hasNext()) {
(29)                    System.out.println((Employee)i.next());
(30)            }
(31)    }
(32)}
```

Download des Beispiels

Anfragen an den persistenten Objektbestand

Als mächtige Alternative zur manuellen Traviersierung einer Objektextension spezifiziert JDO die Verwendung einer eigenen Anfragesprache auf Basis des Standards der *Object Query Language* (OQL) der Object Database Management Group (ODMG).
Diese -- als *JDO Object Query Language* (JDOQL) bezeichnete -- Anfragesprache ist direkt in die JDO-API integriert und wird über verschiedene Einzelmethoden genutzt. Aus diesem Grunde sind JDOQL-Anfragen nicht direkt mit den konsizsen SQL- oder OQL-Anfragen vergleichbar.
Beispiel 41 zeigt die Einbettung der Anfragesprache in die JDO-API.

Beispiel 41: Anfrage auf den persistenten Objektbestand mittels OQL

```
(1)import java.io.IOException;
(2)import java.io.InputStream;
(3)import java.util.Collection;
(4)import java.util.Iterator;
(5)import java.util.Properties;
(6)
(7)import javax.jdo.Extent;
(8)import javax.jdo.JDOHelper;
(9)import javax.jdo.PersistenceManager;
(10)import javax.jdo.PersistenceManagerFactory;
(11)import javax.jdo.Query;
(12)
(13)import de.jeckle.jdotest.Employee;
(14)public class JDOQuery {
(15)    public static void main(String args[]) {
(16)            Properties props = new Properties();
(17)            try {
(18)                    InputStream is =
(19)                            ClassLoader.getSystemResourceAsStream("jdo.
properties");
(20)                    props.load(is);
(21)            } catch (IOException ioe) {
(22)                    System.out.println("Error loading properties");
(23)                    System.exit(1);
(24)            }
(25)            PersistenceManagerFactory pmf =
(26)                    JDOHelper.getPersistenceManagerFactory(props);
(27)            PersistenceManager pm = pmf.getPersistenceManager();
(28)
(29)            Extent ext = pm.getExtent(Employee.class, false);
(30)            String filter = "department == \"B042\"";
(31)            Query qry = pm.newQuery(ext, filter);
(32)            qry.setOrdering("name ascending");
(33)            qry.compile();
(34)            Collection c = (Collection) qry.execute();
(35)
(36)            Iterator i = c.iterator();
(37)            while (i.hasNext()) {
(38)                    System.out.println(i.next());
(39)            }
(40)    }
(41)
(42)}
```

Download des Beispiels

Das Beispiel illustriert eine Anfrage, die alle Employee-Objekte liefert, deren department-Attribut mit dem Wert B042 belegt ist und liefert die nach dem Inhalt des Attributes name in aufsteigender Reihenfolge sortiert.
Hierzu wird zunächst die vollständige Extension der Klasse Employee ermittelt. Allerdings Extrahiert dieser Aufruf noch keine Werte aus dem persistenten Objektspeicher, sondern schafft nur die Grundlagen einer späteren manuellen Traversierung oder der Anfrage via JDOQL.
Zur Vorbereitung der tatsächlichen physischen Anfrage wird zunächst eine Zeichenkette geeignet belegt, um als Filterausdruck dienen zu können, der auf die vollständige Extension angewandt wird. Im Beispiel ist dieser Filterausdruck mit department == \"B042\" belegt. Aus Gründen der Zeichenkettenverarbeitung in Java muß hierzu der notwendige Einschluß des zu suchenden Wertes in Anführungszeichen geeignet maskiert werden.
Nach diesen Vorbereitungsschritten kann durch den Aufruf der durch das PersistenceManager-kompatible Objekt bereitgestellten Methode newQuery ein neues Anfrageobjekt (vom Typ Query) erzeugt werden.
Dieses Objekt erlaubt nach der gezeigten Festlegung des Anfrageumfanges die Parametrisierung der Anfrage. Das Beispiel illustriert dies am Aufruf der Methode setOrdering, die es erlaubt eine bestimmte Sortierreihenfolge der gelieferten Ergebnisse vorzugeben.
Zusätzlich kann durch die optionale Ausführung der Methode compile eine Prüfung der zusammengestellten Anfrage erfolgen, die zusätzlich auch interne implementierungsspezifische Optimierungen vornehmen kann.
Abschließend erfolgt die Ausführung der Anfrage durch Aufruf der Methode execute, welche die Anfrageergebnisse konform zur Standardschnittstelle Collection zurückliefert.

Löschen von Objekten

Zur Entfernung eines Objektes aus dem Objektspeicher stellt die Schnittstelle PersistenceManager die Methode deletePersistent zur Verfügung, welche ein einzelnes hauptspeicherresidentes Objekt aus dem persistenten Speicher löscht, bzw. mit deletePersistentAll eine Möglichkeit alle durch eine Sammlung referenzierten Objekte zu entfernen.
Da es sich hierbei um einen schreibenden Zugriff handelt, muß dieser in einen Transaktionskontext eingebettet werden oder explizit transaktionslos durchgeführt werden wie in Beispiel 39 gezeigt. Beispiel 42 zeigt die Löschung unter Verwendung eines Transaktionskontextes.

Beispiel 42: Löschen eines Objektes aus dem persistenten Objektbestand

```
(1)import java.io.IOException;
(2)import java.io.InputStream;
(3)import java.util.Collection;
(4)import java.util.Properties;
(5)
(6)import javax.jdo.Extent;
(7)import javax.jdo.JDOHelper;
(8)import javax.jdo.PersistenceManager;
(9)import javax.jdo.PersistenceManagerFactory;
(10)import javax.jdo.Query;
(11)
(12)import de.jeckle.jdotest.Employee;
(13)
(14)public class JDODeleteObj {
(15)    public static void main(String args[]) {
(16)            Properties props = new Properties();
(17)            try {
(18)                    InputStream is =
(19)                            ClassLoader.getSystemResourceAsStream("jdo.
properties");
(20)                    props.load(is);
(21)            } catch (IOException ioe) {
(22)                    System.out.println("Error loading properties");
(23)                    System.exit(1);
(24)            }
(25)            PersistenceManagerFactory pmf =
(26)                    JDOHelper.getPersistenceManagerFactory(props);
(27)            PersistenceManager pm = pmf.getPersistenceManager();
(28)
(29)            Extent ext = pm.getExtent(Employee.class, false);
(30)            String filter = "name == \"Marta Smith\"";
(31)            Query qry = pm.newQuery(ext, filter);
(32)            Collection c = (Collection) qry.execute();
(33)
```

```
(34)            pm.currentTransaction().begin();
(35)            pm.deletePersistentAll(c);
(36)            pm.currentTransaction().commit();
(37)            System.out.println("Object deleted");
(38)    }
(39)}
```

Download des Beispiels

Migration zu einem anderen Persistenzdienstleister

Der JDO-Ansatz tritt mit dem Versprechen auf vollständig sowohl unabhängig vom verwendeten Persistenzmedium (etwa: Datenbank, Dateisystem, etc.) als auch der eingesetzten JDO-Implementierung zu sein. Diese Zielsetzung wird nachfolgend auf Basis des im vorhergehenden diskutierten `Employee`-Beispiels untersucht. Hierzu wird die frei verfügbare JDO-Implementierung *TJDO* eingesetzt, welche verschiedene Datenbankmanagementsysteme zur Speicherung der Javaobjekte heranziehen kann. Im Beispiel wird das DBMS *MySQL* Persistierung der Applikationsobjekte genutzt.

Zur Portierung der bestehenden Applikation ist lediglich die Anpassung der JDO-Eigenschaften (Property-Datei) vorzunehmen, um den neuen Persistenzdienstleister sowie die verschiedenen DBMS-Spezifika zu berücksichtigen.
Beispiel 43 zeigt die neuen Inhalte.

Beispiel 43: Konfiguration der JDO-Implementierung TJDO

```
(1)javax.jdo.PersistenceManagerFactoryClass=com.triactive.jdo.
PersistenceManagerFactoryImpl
(2)javax.jdo.option.ConnectionURL=jdbc:mysql://localhost/jdotest/
(3)javax.jdo.option.ConnectionDriverName=com.mysql.jdbc.Driver
(4)javax.jdo.option.ConnectionUserName=mario
(5)javax.jdo.option.ConnectionPassword=thePassword
(6)com.triactive.jdo.autoCreateTables=true
```

Download des Beispiels

Zunächst werden die bereits in der Konfiguration der Referenzimplementierung durch Beispiel 32 genutzten Eigenschaften zur Identifikation derjenigen Klasse, welche die JDO-Schnittstelle `PersistenceManagerFactory` implementiert sowie zur Festlegung der Verbindungs-URL und des zu verwendenden Benutzernamens uns Passwortes an die neuen Gegebenheiten adaptiert. Konkret wird die durch TJDO bereitgestellte Klasse `com.triactive.jdo.PersistenceManagerFactoryImpl` als `PersistenceManagerFactory` konforme Implementierung sowie die Identifikation der zu verwendenden Datenbank nebst Benutzername und Anmeldekennwort bekanntgegeben. Zusätzlich wird mit `com.triactive.jdo.autoCreateTables` eine implementierungsspezifische Eigenschaft mit `true` belegt, die TJDO veranlaßt im Bedarfsfalle benötigte Tabellenstrukturen automatisiert zu erzeugen.

Zusätzlich erfordert die verwendete JDO-Implementierung die Adaption der im Rahmen des Bytecodeanreicherungsprozesses herangezogenen Konfigurationsdatei (Beispiel 44). Auf diesem Wege wird dem Programmierer die Möglichkeit eröffnet die Abbildung auf relationale Tabellenstrukturen beeinflussen. In der Konsequenz erfordert der Wechsel der JDO-Implementierung die Wiederholung des Anreicherungslaufes für den Bytecode der zu persistierenden Klassen.

Beispiel 44: Parametrisierung der Objektpersistenz

```
(1)<?xml version="1.0"?>
(2)  <!DOCTYPE jdo PUBLIC "-//Sun Microsystems, Inc.//DTD Java Data Objects Metadata
1.0//EN" "http://java.sun.com/dtd/jdo_1_0.dtd">
(3)  <jdo>
(4)      <package name="de.jeckle.jdotest">
(5)          <class name="Employee">
(6)              <field name="name">
(7)                  <extension vendor-name="triactive" key="length" value="max 32"/>
(8)              </field>
(9)              <field name="department">
(10)                  <extension vendor-name="triactive" key="length" value="max 32"/
>
(11)              </field>
```

```
(12)          </class>
(13)      </package>
(14)  </jdo>
```

Download des Beispiels

Weitere Änderungen an den zu persistierenden Klassen oder den mit deren Objekten operierenden Applikationen ist nicht notwendig, alle Zugriffe werden nach den oben beschriebenen Änderungen transparent und ohne Neuübersetzung datenbankbasiert abgewickelt.

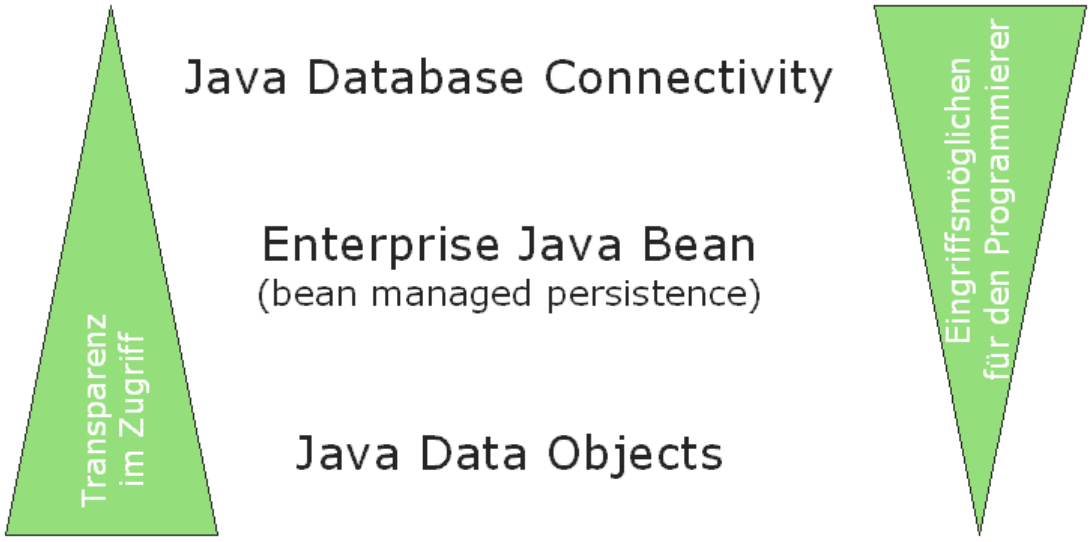
Vergleich der verschiedenen Persistenzansätze

Abschließend seien die charakteristischen Eigenschaften der drei diskutierten Persistenzansätze JDBC, EJB und JDO kurz vergleichend nebeneinandergestellt.

Merkmal	JDBC	EJB	JDO
Transaktionsunterstützung	✓	✓	✓
Anfragemöglichkeit	✓	✓	✓
Standardisiertes API	✓	✓	✓
Standardanfragesprache	✓ SQL	✓ SQL/EJBQL	✓ JDOQL
Unterstützte Hintergrundspeicher	RDBMS	RDMBS Integrationsmiddleware	RDBMS, ORDBMS Integrationsmiddleware, Dateisystem, bel. andere
Transparenter Zugriff auf persistierte Daten	⊘	✓	✓
Berücksichtigung existierender relationaler Strukturen	✓	✓ bei bean managed persistence	✓ allerdings nicht im Standard vorgesehen

Die Tabelle zeigt klar, daß alle drei Persistenzmechanismen grundlegende Eigenschaften teilen, sich jedoch auch in zentralen Charakteristika unterscheiden.
Während sowohl JDBC als auch EJBs die direkte Verwendung von SQL-Anfragen gestatten bietet JDO mit JDOQL eine eigenständige Anfragesprache, die direkt in die Sprach-API eingebettet ist. Für EJBs existiert neben den in Kapitel 1.2 gezeigten Mechanismen auch die Möglichkeit der Verwendung der EJB-spezifischen Anfragesprache *EJBQL*, die jedoch hier nicht betrachtet wurde.
Hinsichtlich der jeweils unterstützten Hintergrundspeicherarchitekturen zur Realisierung der Persistenz treten jedoch deutliche Unterschiede zu Tage. So ist der Einsatz der JDBC-API auf relationale Datenquellen, bzw. Datenquellen die eine relationale Sicht anbieten, beschränkt. Innerhalb der EJB-Architektur können hingegen neben den -- hier diskutierten JDBC-basierten Mechanismen -- auch die Dienste einer Integrationsmiddleware zu Speicherung herangezogen werden und so eine gewisse Unabhängigkeit vom physischen Speichermedium erreicht werden. Einzig JDO bietet durch seine starke Abstraktion die Möglichkeit beliebige Persistenzdienstleister zu nutzen.
Zur effizienten Abwicklung dieses speicherformunabhängigen Zugriffs etabliert JDO notwendigerweise eine stark abstrahierte API, deren Funktionen keinerlei Rückschlüsse auf den verwendeten Persistenzmechanismus zulassen. Für EJB läßt sich dies prinzipiell auch realisieren, allerdings müssen für die Variante der bean managed persistence innerhalb der Entity Bean die Interaktionen mit dem Persistenzdienstleister expliziert werden, beispielsweise durch JDBC. Daher verhält sich dieser Ansatz intern ähnlich zur direkten Verwendung der JDBC-API, die inhärent jeden angebundenen Persistenzmechanismus mit relationaler Zugriffssemantik belegt.
Aufgrund des vorherrschenden relationalen Speicherparadigmas kann die Einbindung bestehender Tabellenstrukturen in den API-Mechanismus gewünscht sein. Dies ist ausschließlich mit Ansätzen möglich, welche die anwenderdefinierte Strukturierung der Zugriffsausdrücke -- etwa durch die Verwendung von SQL -- gestatten. Dies ist ausschließlich für JDBC und EJB (sofern bean managed persistence verwendet wird) möglich; JDO sieht dies generell nicht vor.

Abbildung 8: *Vergleich zwischen den diskutierten Persistenztechniken*



(click on image to enlarge!)

Abschließend lassen sich die vorgestellten Schnittstellen hinsichtlich ihrer Möglichkeiten zur Bereitstellung eines transparenten Zugriffs auf den Hintergrundspeicher und der manuellen Eingriffsmöglichkeiten zur Kontrolle der Persistenz durch den Programmierer kategorisieren. Prinzipiell läßt sich festhalten, daß diese Eigenschaftstypen konkurrierende Zielsetzungen darstellen. So bietet JDBC zweifelsohne die größten Möglichkeiten zum steuernden Eingriff durch den Programmierer, wobei dieser Ansatz in der Interaktion auch die größte Menge Wissen des Programmierers über die etablierten Speicherstrukturen erfordert. Daher realisiert JDBC generell die geringste Transparenz im Zugriff auf den Objektspeicher.
Auf der anderen Seite realisiert JDO die größtmögliche Transparenz im Objektzugriff, wobei dieser Freiheitsgrad zu generell zu Lasten der Eingriffsmöglichkeiten durch den Programmierer umgesetzt werden.

Web-Referenzen 2: Weiterführende Links

- TJDO -- eine freie JDO-Implementierung
- JDO @ SUN
- JDOCentral.com -- Die Anlaufstelle der JDO-Entwickler
- JDO-Spezifikation

Architekturmuster and Umsetzungstechniken

Neben den Basistechniken und bisher vorgestellten Schnittstellen zur Realisierung von Persistenz finden gegenwärtig eine Reihe von Architekturmustern und Handreichungen zur Umsetzung der Verbindung von dauerhafter Datenspeicherung und Anwendungsprogrammierung Einsatz.
Ziel dieses Kapitels ist es, ausgewählte Muster an Beispielen vorzustellen und ihren Einsatz in den Kontext der im Abschnitt eins eingeführten Schnittstellentechniken zu stellen. Hierzu werden die Muster zunächst in drei Abschnitte gegliedert und gleichzeitig im Hinblick auf die durch sie angesprochene Problemdomäne kategorisiert.

- Domänenlogik: Muster dieser Klasse dienen dazu, die Verbindung zwischen Applikationslogik und persistenter Datenspeicherung zu strukturieren und die beiden Systemebenen wirkungsvoll zu entkoppeln.
- Datenzugriff: Muster dieser Klasse dienen dazu, den Zugriff auf die verwalteten Daten zu abstrahieren und diese in einheitlicher Weise zugreifbar werden zu lassen.
- Objekt-Relational-Abbildung und -Interoperabilität: Muster dieser Klasse dienen dazu, eine Brücke zwischen objektorientierter Applikationsdatenhaltung und heute (noch) vorherrschender relationaler Speicherung zu schlagen.

Die vorgestellten Muster und Beispiele orientieren sich an den im Buch *Patterns of Enterprise Application Architecture* von M. Fowler vorgestellten.

Domänenlogik

Die nachfolgend eingeführten Muster zur Abbildung von Datenbank-gestützt operierender Applikationslogik in Programmstrukturen. Durch die Anwendung der diskutierten Mechanismen wird

eine Entkopplung zwischen Datenbankoperationen und Domänen-induzierten Operationen der Applikationsebene angestrebt, um diese voneinander separiert entwickeln und modifizieren zu können.

Transaction Script

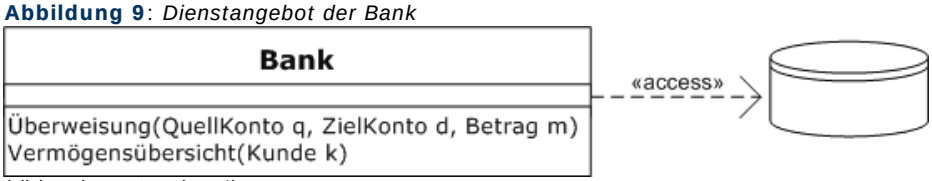
Motivation und Grundidee: Zur Abbildung komplexer Geschäftsoperationen sind in der Regel eine Reihe von eigenständigen Datenbankinteraktionen notwendig. Werden einzelne dieser Interaktionen außerhalb ihres logischen Kontexts ausgeführt, so kann dies zu Inkonsistenzen des verwalteten Datenbestandes führen.
Gleichzeitig setzt die korrekte Ausführung der einzelnen Datenbankoperationen die Kenntnis der Abbildung des Geschäftsprozesses auf die technischen Strukturen voraus um gültige Abläufe konstruieren zu können.

Ziel der Anwendung der *Transaktionsskripte* (engl. *transaction script*) ist es jeden Geschäftsablauf durch genau eine Applikationsmethode abzubilden, welche die notwendigen Datenbankinteraktionen zuverlässig kapselt.
Hierbei impliziert der Terminus der *Transaktion* nicht zwingend die Nutzung von Datenbanktransaktionen im Sinne der ACID-Prinzipien.

Struktur: Typischerweise werden die Geschäftsoperationen durch eine oder mehrere Domänenklassen, d.h. Klassen deren Struktur und Logik nicht auf Basis technischer Erwägungen und Notwendigkeiten gebildet wurde, zur Verfügung gestellt. Diese Klassen treten als Dienstleister gegenüber dem Applikationsprogramm auf.

Abbildung 9 zeigt die Grundstruktur des Dienstangebotes einer Bank. Sie bietet Überweisungen eines Geldbetrages von einem Konto zum anderen sowie die Möglichkeit der Erstellung einer Vermögensübersicht für einen Kunden an.

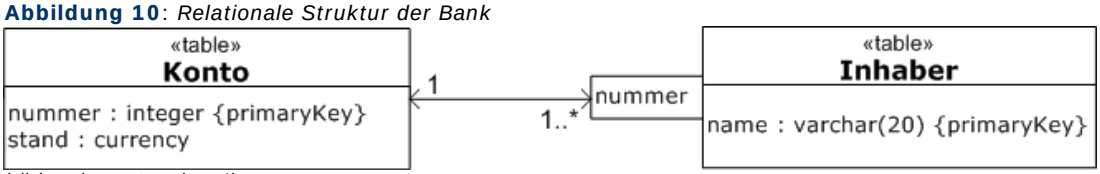
Abbildung 9: *Dienstangebot der Bank*



(click on image to enlarge!)

Die verwalteten Daten seien in diesem Beispiel in einer relationalen Datenbank abgelegt, die aus zwei Tabellen (`Konto` und `Inhaber`) besteht. Abbildung 10 zeigt den Datenbankaufbau.

Abbildung 10: *Relationale Struktur der Bank*



(click on image to enlarge!)

Der Aufbau der Datenbank und ihre Befüllung mit Beispieldaten geschieht durch das folgende SQL-Script:

Beispiel 45: Erzeugung und Befüllung der Datenbank mit SQL

```
(1)CREATE TABLE konto(nummer INTEGER PRIMARY KEY, stand DECIMAL(5,2) NOT NULL);
(2)CREATE TABLE inhaber(name VARCHAR(20) NOT NULL, nummer INTEGER NOT NULL, PRIMARY
KEY(name, nummer));
(3)
(4)INSERT INTO konto VALUES(12345678, 5000);
(5)INSERT INTO konto VALUES(11111111, 100);
(6)
(7)INSERT INTO inhaber VALUES("John Doe", 12345678);
(8)INSERT INTO inhaber VALUES("John Doe", 11111111);
(9)INSERT INTO inhaber VALUES("Barnie Bar", 11111111);
```

Download des Beispiels

Das Transaktionsskript: Beispiel 46 zeigt die das Transaktionsskript realisierende Klasse. Sie kapselt die Datenbankinteraktion vollständig und stellt die beiden in Abbildung 9 gezeigten Methoden zur Verfügung.

Beispiel 46: Transaktionsskript zur Interaktion mit der Bank

```
(1)import java.sql.Connection;
(2)import java.sql.DriverManager;
(3)import java.sql.ResultSet;
(4)import java.sql.SQLException;
(5)import java.sql.Statement;
(6)
(7)public class Bank {
(8)     public static void Überweisung(
(9)             int srcAccount,
(10)            int dstAccount,
(11)            double amount) {
(12)            Connection con = connectDB();
(13)            try {
(14)                    con.setAutoCommit(false);
(15)                    con.setTransactionIsolation(Connection.
TRANSACTION_SERIALIZABLE);
(16)                    Statement stmt = con.createStatement();
(17)                    stmt.executeUpdate("UPDATE konto set stand=stand-"+amount+"
where nummer='"+srcAccount+"';");
(18)                    stmt.executeUpdate("UPDATE konto set stand=stand+"+amount+"
where nummer='"+dstAccount+"';");
(19)                    con.commit();
(20)            } catch (SQLException e) {
(21)                    e.printStackTrace();
(22)            }
(23)    }
(24)    public static String Vermögensübersicht(String kunde) {
(25)            String result=null;
(26)            try {
(27)                    result= "Vermögensübersicht für: "+kunde+"\n";
(28)                    result+="-----------------------------------------\n";
(29)                    Statement stmt = connectDB().createStatement();
(30)                    stmt.executeUpdate("LOCK TABLES konto READ, inhaber READ;");
(31)                    ResultSet rs = stmt.executeQuery("SELECT k.nummer, stand
FROM konto AS k, inhaber AS i WHERE i.name='"+kunde+"' AND i.nummer=k.nummer;");
(32)                    while(!rs.isLast()) {
(33)                            rs.next();
(34)                            result+=rs.getInt("nummer")+"\t"+rs.getInt("stand")
+"\n";
(35)                    }
(36)                    result+="-----------------------------------------\n";
(37)                    rs = stmt.executeQuery("SELECT SUM(stand) AS s from konto as
k,inhaber as i where i.name='"+kunde+"' and i.nummer=k.nummer;");
(38)                    stmt.executeUpdate("UNLOCK TABLES;");
(39)                    rs.next();
(40)                    result+="Saldo aller Konten: "+rs.getInt("s")+"\n";
(41)            } catch (SQLException e) {
(42)                    e.printStackTrace();
(43)            }
(44)            return result;
(45)    }
(46)    private static Connection connectDB() {
(47)            try {
(48)                    Class.forName("com.mysql.jdbc.Driver");
(49)            } catch (ClassNotFoundException e) {
(50)                    System.err.println("Driver class not found");
(51)                    e.printStackTrace();
(52)            }
(53)            Connection con = null;
(54)            try {
(55)                    con =
(56)                            (Connection) DriverManager.getConnection(
(57)                                    "jdbc:mysql://localhost/bank/",
(58)                                    "root",
(59)                                    "");
(60)            } catch (SQLException e1) {
(61)                    System.err.println("Error establishing database connection");
(62)                    e1.printStackTrace();
(63)            }
(64)            return con;
(65)    }
(66)
```

```
(67)}
```

Download des Beispiels

Die gesamte Interaktion mit der Persistenzlogik geschieht durch die beiden Geschäftsmethoden und bedarf keiner Kenntnis und Berücksichtigung der technischen Datenbankcharakteristika. Beispiel 47 zeigt die Nutzung der beiden angebotenen Methoden.

Beispiel 47: Nutzung des Transaktionsskriptes

```
(1)public class Driver {
(2)     public static void main(String args[]) {
(3)             Bank.Überweisung(12345678, 11111111, 500);
(4)             System.out.println(Bank.Vermögensübersicht("John Doe"));
(5)     }
(6)}
```

Download des Beispiels

Umsetzung unter Einsatz des *Command Musters*:
Die Nutzung von Transaktionsskripten führt zur Bildung von Methoden, die mit Namen aus der Geschäftsdomäne belegt sind. So treten im Beispiel 46 die Methoden `Überweisung` und `Vermögensübersicht` auf.
Diese Benennungseigenschaft ist jedoch nicht immer gewünscht. Vielmehr strebt man bei der Implementierung häufig eine gleichartige Aufrufschnittstelle verschiedener Sachverhalte an. Daher findet sich häufig Transaktionsskripte mithilfe des *Command Musters* umgesetzt.

Dieses Muster definiert für alle aufrufbaren Domänenmethoden des Transaktionsskriptes eine einheitliche Schnittstelle (im Beispiel durch die Methode `run` verkörpert).
Der Einsatz des Musters hat jedoch keinen Einfluß auf die verwirklichte Domänenlogik, sondern ändert nur die Aufrufmimik.

Die Beispiele 48 mit 50 zeigen die modifizierte Umsetzung der einzelnen Methoden des Transaktionsskripts, die nun durch separate Klassen repräsentiert werden. Die Kontroll-Logik bleibt jedoch gegenüber der vorhergehenden Lösung unverändert.

Beispiel 48: TransactionScript.java

```
(1)public abstract class TransactionScript {
(2)     public void run(){
(3)     }
(4)}
```

Download des Beispiels

Beispiel 49: Überweisung.java

```
(1)import java.sql.Connection;
(2)import java.sql.DriverManager;
(3)import java.sql.SQLException;
(4)import java.sql.Statement;
(5)
(6)public class Überweisung extends TransactionScript {
(7)     private int srcAccount;
(8)     private int dstAccount;
(9)     private double amount;
(10)
(11)    public Überweisung(int srcAccount, int dstAccount, double amount) {
(12)            this.srcAccount = srcAccount;
(13)            this.dstAccount = dstAccount;
(14)            this.amount = amount;
(15)    }
(16)    public void run() {
(17)            Connection con = connectDB();
(18)            try {
(19)                    con.setAutoCommit(false);
(20)                    con.setTransactionIsolation(Connection.
TRANSACTION_SERIALIZABLE);
(21)                    Statement stmt = con.createStatement();
(22)                    stmt.executeUpdate(
```

```
(23)                            "UPDATE konto set stand=stand-"
(24)                                    + amount
(25)                                    + " where nummer='"
(26)                                    + srcAccount
(27)                                    + "';");
(28)                    stmt.executeUpdate(
(29)                            "UPDATE konto set stand=stand+"
(30)                                    + amount
(31)                                    + " where nummer='"
(32)                                    + dstAccount
(33)                                    + "';");
(34)                    con.commit();
(35)            } catch (SQLException e) {
(36)                    e.printStackTrace();
(37)            }
(38)    }
(39)    private static Connection connectDB() {
(40)            try {
(41)                    Class.forName("com.mysql.jdbc.Driver");
(42)            } catch (ClassNotFoundException e) {
(43)                    System.err.println("Driver class not found");
(44)                    e.printStackTrace();
(45)            }
(46)            Connection con = null;
(47)            try {
(48)                    con =
(49)                            (Connection) DriverManager.getConnection(
(50)                                    "jdbc:mysql://localhost/bank/",
(51)                                    "root",
(52)                                    "");
(53)            } catch (SQLException e1) {
(54)                    System.err.println("Error establishing database connection");
(55)                    e1.printStackTrace();
(56)            }
(57)            return con;
(58)    }
(59)}
```

Download des Beispiels

Beispiel 50: Vermögensübersicht.java

```
(1)import java.sql.Connection;
(2)import java.sql.DriverManager;
(3)import java.sql.ResultSet;
(4)import java.sql.SQLException;
(5)import java.sql.Statement;
(6)
(7)public class Vermögensübersicht extends TransactionScript {
(8)     private String kunde;
(9)     public String result;
(10)
(11)    public Vermögensübersicht(String kunde) {
(12)            this.kunde = kunde;
(13)    }
(14)
(15)    public void run() {
(16)            try {
(17)                    result = "Vermögensübersicht für: " + kunde + "\n";
(18)                    result += "----------------------------------------\n";
(19)                    Statement stmt = connectDB().createStatement();
(20)                    ResultSet rs =
(21)                            stmt.executeQuery(
(22)                                    "SELECT k.nummer, stand FROM konto AS k,
inhaber AS i WHERE i.name='"
(23)                                            + kunde
(24)                                            + "' AND i.nummer=k.nummer;");
(25)                    while (!rs.isLast()) {
(26)                            rs.next();
(27)                            result += rs.getInt("nummer")
(28)                                    + "\t"
```

```
(29)                                    + rs.getInt("stand")
(30)                                    + "\n";
(31)                    }
(32)                    result += "----------------------------------------\n";
(33)                    rs =
(34)                            stmt.executeQuery(
(35)                                    "SELECT SUM(stand) AS s from konto as k,
inhaber as i where i.name='"
(36)                                            + kunde
(37)                                            + "' and i.nummer=k.nummer;");
(38)                    rs.next();
(39)                    result += "Saldo aller Konten: " + rs.getInt("s") + "\n";
(40)            } catch (SQLException e) {
(41)                    e.printStackTrace();
(42)            }
(43)    }
(44)    private static Connection connectDB() {
(45)            try {
(46)                    Class.forName("com.mysql.jdbc.Driver");
(47)            } catch (ClassNotFoundException e) {
(48)                    System.err.println("Driver class not found");
(49)                    e.printStackTrace();
(50)            }
(51)            Connection con = null;
(52)            try {
(53)                    con =
(54)                            (Connection) DriverManager.getConnection(
(55)                                    "jdbc:mysql://localhost/bank/",
(56)                                    "root",
(57)                                    "");
(58)            } catch (SQLException e1) {
(59)                    System.err.println("Error establishing database connection");
(60)                    e1.printStackTrace();
(61)            }
(62)            return con;
(63)    }
(64)
(65)}
```

Download des Beispiels

Bei der Erstellung von Applikationen, welche die vereinheitlichten Schnittstellen nutzen zeigt sich das Resultat in Form einer gleichartigen Aufrufschnittstelle (im Beispiel die Methode run) für die verschiedenen Domänenmethoden:

Beispiel 51: Driver2.java

```
(1)public class Driver2 {
(2)     public static void main(String args[]) {
(3)             Überweisung ü = new Überweisung(12345678, 11111111, 500);
(4)             ü.run();
(5)
(6)             Vermögensübersicht v = new Vermögensübersicht("John Doe");
(7)             v.run();
(8)             System.out.println(v.result);
(9)     }
(10)}
```

Download des Beispiels

Abschließende Würdigung:
Das Transaktionsskript-Muster bietet eine vergleichsweise einfach nachvollziehbare Möglichkeit zur Entkopplung von Persistenz- und Geschäftslogik an. Jedoch tritt sehr schnell (wie in den Beispielen 49 und 50 anhand der Methode `connectDB` gezeigt) die Gefahr auf, daß gleichartiger Code in verschiedene Transaktionsskripte zu integrieren ist.
Überdies führen komplexe Geschäftslogiken, die sich partiell überlappen und gegenseitig enthalten zu aufwendigen Entwürfen in denen Coderedundanz nicht immer zu vermeiden ist.

Abhilfe kann hier die Verwendung eines *Domänenmodells* bieten.

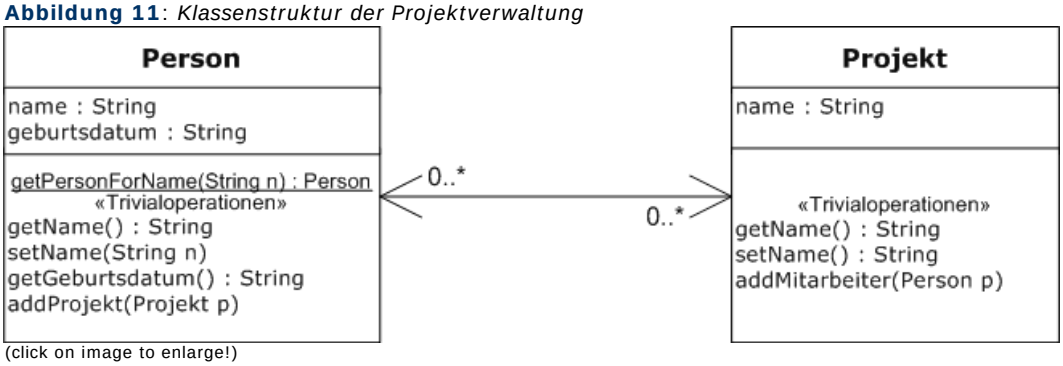
Domain Model

Motivation und Grundidee: Hintergrund der Strukturform des *Domain Model*s ist der Versuch die in der Analysephase vorgefundenen Objekte des betrachteten Problembereichs möglichst unverändert durch die Applikation zur Verfügung zu stellen und in die Datenbank zu übernehmen. Im Gegensatz zur Strukturierungsform des *Transaction Script*s erfolgt der Anwendungsaufbau hierbei nicht an den Geschäftsprozessen orientiert, sondern rein Daten-getrieben.

Struktur: Im Idealfall entsprechen sich die Struktur der applikationsimmanenten Klassen und die der datebankresidenten Tabellen eineindeutig. Abweichungen davon können sich lediglich durch Tabellen ergeben, welche dieselbe Realweltentität abbilden. Diese können durch den Normalisierungsprozeß gebildet worden sein.
Das Beispiel der Abbildung 11 zeigt die Klassenstruktur einer Projektverwaltung, in der die Zuordnungen zwischen `Personen` und den `Projekten` in denen diese eingesetzt sind verwaltet werden.

Jeder Ausprägung von `Person` können hierbei mehrere Objekte des Typs `Projekt` zugeordnet sein und umgekehrt.
Zusätzlich sind die für die jeweiligen Klassen definierten Operation dargestellt. Hierbei kann es sich um triviale Operationen zum Setzen und Auslesen einzelner Attributwerte oder beliebig aufwendige Vorgänge handeln. Gemeinsames Kennzeichen aller Operationen ist jedoch, daß sie nur dasjenige Objekt betreffen in dessen Kontext sie definiert sind.

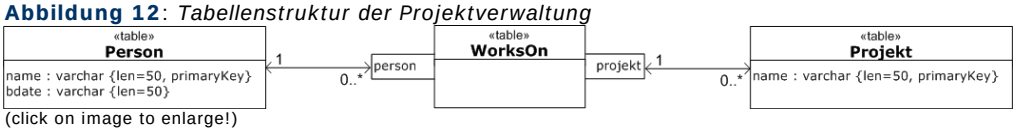
Abbildung 11: *Klassenstruktur der Projektverwaltung*



(click on image to enlarge!)

Die verwalteten Daten sind in drei Datenbanktabellen abgelegt. Abbildung 12 zeigt die Struktur der Tabellen einschließlich der definierten Fremdschlüsselbeziehungen.
Aus Gründen der Normalisierung (die Relation befindet sich in vierter Normalform) wird zusätzlich die Tabelle `WorksOn` eingeführt, welche die Daten über die Zuordnung zwischen `Projekten` und den sie bearbeitenden `Personen` enthält.

Abbildung 12: *Tabellenstruktur der Projektverwaltung*



(click on image to enlarge!)

Der Aufbau der Datenbank geschieht durch das folgende SQL-Script:

Beispiel 52: DB-Aufbau

```
(1)CREATE TABLE Person(
(2)     name VARCHAR(50) PRIMARY KEY,
(3)     bdate DATE);
(4)
(5)CREATE TABLE Project(
(6)     name VARCHAR(50) PRIMARY KEY);
(7)
(8)CREATE TABLE WorksOn(
(9)     person VARCHAR(50) NOT NULL,
(10)    project VARCHAR(50) NOT NULL,
(11)    PRIMARY KEY (person, project));
(12)
(13)ALTER TABLE WorksOn ADD INDEX WO_person_IDX(person);
(14)ALTER TABLE WorksOn ADD CONSTRAINT WO_person_FK FOREIGN KEY (person) REFERENCES
Person(name);
(15)ALTER TABLE WorksOn ADD INDEX WO_project_IDX(project);
(16)ALTER TABLE WorksOn ADD CONSTRAINT WO_project_FK FOREIGN KEY (project)
REFERENCES Project(name);
```

Download des Beispiels

Das Domänenmodell: Die Beispiele 53 und 54 zeigen die beiden fachlichen Domänenklassen `Person` und `Projekt`.

Beispiel 53: Die Domänenklasse Person

```
(1)import java.sql.ResultSet;
(2)import java.sql.SQLException;
(3)import java.sql.Statement;
(4)import java.util.Vector;
(5)
(6)public class Person {
(7)     private String name;
(8)     private Vector projekte = new Vector();
(9)
(10)    public void setName(String name) {
(11)            Statement stmt = DBConnector.getConnectedStatement();
(12)            try {
(13)                    ResultSet rs =
(14)                            stmt.executeQuery(
(15)                                    "SELECT count(*) FROM Person WHERE name='"
(16)                                            + this.name
(17)                                            + "';");
(18)                    rs.next();
(19)                    if (rs.getDouble(1) < 1) {
(20)                            System.out.println(
(21)                                    "Database entry for person does not exist
and will be created");
(22)                            stmt.executeUpdate(
(23)                                    "INSERT INTO Person (name) VALUES('" + name
+ "')");
(24)                    } else {
(25)                            System.out.println(
(26)                                    "Database entry for person exists and will
be updated");
(27)                            stmt.executeUpdate(
(28)                                    "UPDATE Person SET name='"
(29)                                            + name
(30)                                            + "' where name='"
(31)                                            + this.name
(32)                                            + "';");
(33)                    }
(34)            } catch (SQLException e) {
(35)                    System.out.println("Cannot access database");
(36)                    e.printStackTrace();
(37)            }
(38)            this.name = name;
(39)    }
(40)    public void setGeburtsdatum(String gebDat) {
(41)            Statement stmt = DBConnector.getConnectedStatement();
(42)            try {
(43)                    System.out.println("Database entry (bdate) will be updated");
(44)                    stmt.executeUpdate(
(45)                            "UPDATE Person SET bdate='"
(46)                                    + gebDat
(47)                                    + "' WHERE name='"
(48)                                    + this.name
(49)                                    + "';");
(50)            } catch (SQLException e) {
(51)                    System.out.println("Cannot access database");
(52)                    e.printStackTrace();
(53)            }
(54)    }
(55)    public String getGeburtsdatum() {
(56)            Statement stmt = DBConnector.getConnectedStatement();
(57)            ResultSet rs;
(58)            String result = "";
(59)            try {
(60)                    rs =
(61)                            stmt.executeQuery(
(62)                                    "SELECT bdate FROM Person WHERE name='" +
name + "';");
(63)                    rs.next();
```

```
(64)                    result = rs.getString(1);
(65)            } catch (SQLException e) {
(66)                    // TODO Auto-generated catch block
(67)                    e.printStackTrace();
(68)            }
(69)            return result;
(70)    }
(71)    public String getName() {
(72)            return name;
(73)    }
(74)    public void addProjekt(Projekt p) {
(75)            Statement stmt = DBConnector.getConnectedStatement();
(76)            try {
(77)                    stmt.executeUpdate(
(78)                            "INSERT INTO WorksOn VALUES('"
(79)                                    + this.name
(80)                                    + "','"
(81)                                    + p.getName()
(82)                                    + "');");
(83)            } catch (SQLException e) {
(84)                    System.out.println("Cannot access database");
(85)                    e.printStackTrace();
(86)            }
(87)    }
(88)    public Vector getAllProjekt() {
(89)            Vector projekte = new Vector();
(90)
(91)            return projekte;
(92)    }
(93)    public static Person getPersonForName(String name) {
(94)            Statement stmt = DBConnector.getConnectedStatement();
(95)            ResultSet rs;
(96)            Person p = new Person();
(97)            try {
(98)                    rs =
(99)                            stmt.executeQuery(
(100)                                   "SELECT bdate FROM Person WHERE name='" +
name + "';");
(101)                   rs.next();
(102)                   p.name = name;
(103)           } catch (SQLException e) {
(104)                   System.out.println("Cannot access database");
(105)                   e.printStackTrace();
(106)           }
(107)           return p;
(108)   }
(109)   public String toString() {
(110)           return ("(name: "+name+", bdate: "+this.getGeburtsdatum()+")");
(111)   }
(112)}
```

Download des Beispiels

Beispiel 54: Die Domänenklasse Projekt

```
(1)import java.sql.ResultSet;
(2)import java.sql.SQLException;
(3)import java.sql.Statement;
(4)import java.util.Vector;
(5)
(6)public class Projekt {
(7)     private String name;
(8)     private Vector mitarbeiter = new Vector();
(9)
(10)    public void setName(String name) {
(11)            Statement stmt = DBConnector.getConnectedStatement();
(12)            try {
(13)                    ResultSet rs =
(14)                            stmt.executeQuery(
(15)                                    "SELECT count(*) FROM Project WHERE name='"
(16)                                            + this.name
```

```
(17)                                            + "';");
(18)                    rs.next();
(19)                    if (rs.getDouble(1) < 1) {
(20)                            System.out.println(
(21)                                    "Database entry for project does not exist
and  will be created");
(22)                            stmt.executeUpdate(
(23)                                    "INSERT INTO Project (name) VALUES('" + name
+ "')");
(24)                    } else {
(25)                            System.out.println(
(26)                                    "Database entry for project exists and will
be updated");
(27)                            stmt.executeUpdate(
(28)                                    "UPDATE Project SET name='"
(29)                                            + name
(30)                                            + "' where name='"
(31)                                            + this.name
(32)                                            + "';");
(33)                    }
(34)            } catch (SQLException e) {
(35)                    System.out.println("Cannot access database");
(36)                    e.printStackTrace();
(37)            }
(38)            this.name = name;
(39)    }
(40)    public String getName() {
(41)            return name;
(42)    }
(43)    public void addMitarbeiter(Person p) {
(44)            Statement stmt = DBConnector.getConnectedStatement();
(45)            try {
(46)                    stmt.executeUpdate(
(47)                            "INSERT INTO WorksOn VALUES('"
(48)                                    + p.getName()
(49)                                    + "','"
(50)                                    + this.name
(51)                                    + "');");
(52)            } catch (SQLException e) {
(53)                    System.out.println("Cannot access database");
(54)                    e.printStackTrace();
(55)            }
(56)    }
(57)    public Vector getAllMitarbeiter() {
(58)            Vector mitarbeiter = new Vector();
(59)            Statement stmt = DBConnector.getConnectedStatement();
(60)            Person p;
(61)            try {
(62)                    ResultSet rs = stmt.executeQuery("SELECT name FROM Person as
p, WorksOn as w WHERE w.project='"+this.name+"' and w.person=p.name;");
(63)                    while (!rs.isLast()) {
(64)                            rs.next();
(65)                            p = Person.getPersonForName(rs.getString(1));
(66)                            mitarbeiter.add(p);
(67)                    }
(68)            } catch (SQLException e) {
(69)                    System.out.println("Cannot access database");
(70)                    e.printStackTrace();
(71)            }
(72)            return mitarbeiter;
(73)    }
(74)
(75)}
```

Download des Beispiels

Zusätzlich ist aus Gründen der vereinfachten Interaktion mit dem Datenbankmanagementsystem die Klasse `DBConnector` umgesetzt.

Beispiel 55: Die Klasse DBConnector

```
(1)import java.sql.Connection;
(2)import java.sql.DriverManager;
(3)import java.sql.SQLException;
(4)import java.sql.Statement;
(5)
(6)public class DBConnector {
(7)     public static Statement getConnectedStatement() {
(8)             try {
(9)                     Class.forName("com.mysql.jdbc.Driver");
(10)            } catch (ClassNotFoundException e) {
(11)                    System.err.println("Driver class not found");
(12)                    e.printStackTrace();
(13)            }
(14)            Connection con = null;
(15)
(16)            try {
(17)                    con =
(18)                            (Connection) DriverManager.getConnection(
(19)                                    "jdbc:mysql://dbServerMachine/mcjtest/",
(20)                                    "user",
(21)                                    "thePassword");
(22)                    return (con.createStatement());
(23)            } catch (SQLException e1) {
(24)                    System.err.println("Error establishing database connection");
(25)                    e1.printStackTrace();
(26)            }
(27)            //never gets here
(28)            return null;
(29)    }
(30)}
```

Download des Beispiels

Die beiden Domänenklassen kapseln die Interaktion mit der Datenbank vollständig vor dem Aufrufer. Alle Persistenzoperationen werden in Form einfacher („low-level“) Operationen zur Verfügung gestellt. Die durch eine Klasse angesprochenen Datenbanktabellen sind dabei streng auf diejenigen beschränkt, welche die durch die Klasse verwaltenden Daten aufnehmen.
Auffallend ist, daß die Domänenobjekte außer den Attributen, die den Primärschlüssel repräsentieren, keine durch Attribute ausgedrückte Eigenschaften besitzen. Diese werden ausschließlich in der Datenbank repräsentiert und im Bedarfsfalle angefragt (Beispiel: Realisierung der Methode `getGeburtsdatum` der Klasse `Person`).
Die Begründung hierfür wird bei der Analyse der Zugriffe auf die Tabelle `WorksOn` offenkundig. Da diese Tabelle durch die beiden Domänenobjekte unabhängig voneinander zugegriffen werden kann, würde eine (redundante) Datenverwaltung im Hauptspeicher tendenziell zu Konsistenzproblemen mit durch das Datenbankmanagementsystem verwalteten Daten führen.

Einen solchen Fall, der durch die gewählte Umsetzung korrekt behandelt wird, zeigt der in Beispiel 56 wiedergegebene Code:

Beispiel 56: Domänenmodell verwendende Applikation

```
(1)public class TestDriver {
(2)     public static void main(String args[]) {
(3)             Person pers1 = new Person();
(4)             pers1.setName("Max Mustermann");
(5)             pers1.setName("XX");
(6)             pers1.setName("Max Mustermann");
(7)             pers1.setGeburtsdatum("1970-11-12");
(8)
(9)             Person pers2 = new Person();
(10)            pers2.setName("John Doe");
(11)
(12)            Projekt prj1 = new Projekt();
(13)            prj1.setName("Reorganisation");
(14)
(15)            pers1.addProjekt(prj1);
(16)            prj1.addMitarbeiter(pers2);
(17)
(18)            System.out.println(prj1.getAllMitarbeiter());
(19)    }
(20)
```

```
(21)}
```

Download des Beispiels

Im Beispiel werden zunächst Ausprägungen des Typs `Person` und `Projekt` erzeugt und manipuliert. Hierbei wird für dasselbe `Personen`-Objekt eine Zuordnung dieses Objekts zu einem Projekt vorgenommen und anschließend diesem Projekt eine andere Person zugeordnet. Diese beiden Interaktionen entsprechen der Instanziierung der die beiden Domänenklassen verbindenden Assoziation mit jeweils unterschiedlichen Ausgangspunkten der Beziehungsetablierung.
Nur durch den Rückgriff auf die Datenbankinhalte liefert der Aufruf von `getAllMitarbeiter` konsistente Daten, da sowohl innerhalb der `Person`- als auch der `Projekt`-Ausprägung nur unvollständige (d.h. diejenigen durch den jeweiligen `add...`-Aufruf erzeugten) Daten vorliegen.

Abschließende Würdigung: Das Domänenmodell ermöglicht eine vergleichsweise einfache Abbildung der Objektstrukturen des Hauptspeichers in relationale Datenbankstrukturen.
Allerdings ist die Abbildung komplexer Abhängigkeitsstrukturen der objektorientierten Applikationsdaten in relationale Tabellenstrukturen mitunter schwierig; insbesondere wenn hinsichtlich der Güte der entstehenden DB-Strukturen zusätzliche Qualitätskriterien (wie Redundanzfreiheit durch Normalisierung) angelegt werden.
Grundsätzlich bietet dieses Umsetzungsmuster den Vorteil aus Sicht des Fachanwenders „naheliegende“ Entitäten bereitstellen zu können.
Die Interaktion auf der Basis der durch Domänenklassen angebotenen Operationen kann, abhängig vom Komplexitätsgrad der zu realisierenden Anwendung, -- im Vergleich zum Ansatz des *Transaction Script*s -- aufwendig sein. Insbesondere offenbart sich die vermöge der Domänenoperationen etablierte Abstraktionsschicht als Hemmnis, wenn eine direkte Interaktion mit den wertrepräsentierenden Tabellen intendiert ist.
In diesen Fällen eignet sich ein *Table Module* besser zur Realisierung.
