Universität Stuttgart

Institut für Parallele und Verteilte Systeme
Anwendersoftware

Datenbankanbindungen

Seminararbeit

SIMPL (SS 2009)

Betreuer: Marko Vrhovnik

Michael Hahn

Stuttgart, 29. Juni 2009

Datenbankanbindungen

Michael Hahn

Zusammenfassung In dieser Arbeit werden verschiedene Konzepte zur Datenbankanbindung vorgestellt und erläutert. Dabei werden ausgehend von der JDBC API, die die physikalische Verbindung verwaltet, über die JNDI API, die Namen und Objekte verbindet, über das SDO Konzept, das die Daten von ihrer spezifischen Quelle unabhängig macht, bis zur "inline" SQL Unterstützung in Workflow-Prozessen mit BPEL/SQL alle Konzepte erklärt. Für jedes dieser Konzepte wird sowohl deren zugrunde liegende Architektur als auch deren konkrete Verwendung erläutert, sodass ein grundlegendes Verständnis für Datenbankanbindungen geschaffen wird.

1 Einleitung

Die Anbindung von Datenbanken an Anwendungen verläuft über mehrere Abstraktionsstufen und Konzepte. Wie in Abbildung 1 zu sehen ist soll ein Datenbanksystem an einen Business Process angebunden werden, dazu werden 4 Konzepte in dieser Arbeit vorgestellt und erläutert. In der untersten Schicht befindet sich die Java Database Connectivity (JDBC) API. Diese realisiert die Anbindung von relationalen Datenbanksystemen an Anwendungen und liefert die nötige Funktionalität um mit diesen und den enthaltenen Daten zu arbeiten. Darüber befindet sich die Java Naming and Directory Interface (JNDI) API. Sie dient dem Umgang mit naming and directory services, das sind Dienste mit denen Bindungen zwischen Objekten und logischen Namen dynamisch verwaltet werden können. Das darüber liegende Service Data Objects (SDO) Konzept sorgt dafür, das Daten von ihrer Quelle entkoppelt werden und über einheitliche Schnittstellen zur Verfügung stehen, bearbeitet und innerhalb von Systemen oder Netzwerken transportiert werden können. Als oberste Schicht folgt das Business Process Execution Language/Structured Query Language (BPEL/SQL) Konzept, mit dem die Ausführung von SQL-Befehlen direkt aus einem Prozess eines Workflows ermöglicht wird. Jede dieser 4 Schichten leistet dabei ihren Beitrag um die Datenbankanbindung zu realisieren. So wird mit BPEL/SQL ein SQL-Befehl abgesetzt und das Ziel-Datenbanksystem als logischer Namen übergeben. Die JNDI API löst diesen logischen Namen auf und gibt die Informationen an die JDBC API weiter. Die JDBC API führt dann den SQL-Befehl auf dem entsprechenden Datenbanksystem aus und hält die Ergebnisse bereit. Diese werden dann von der SDO API gekapselt und können so in den Business Process geladen werden.

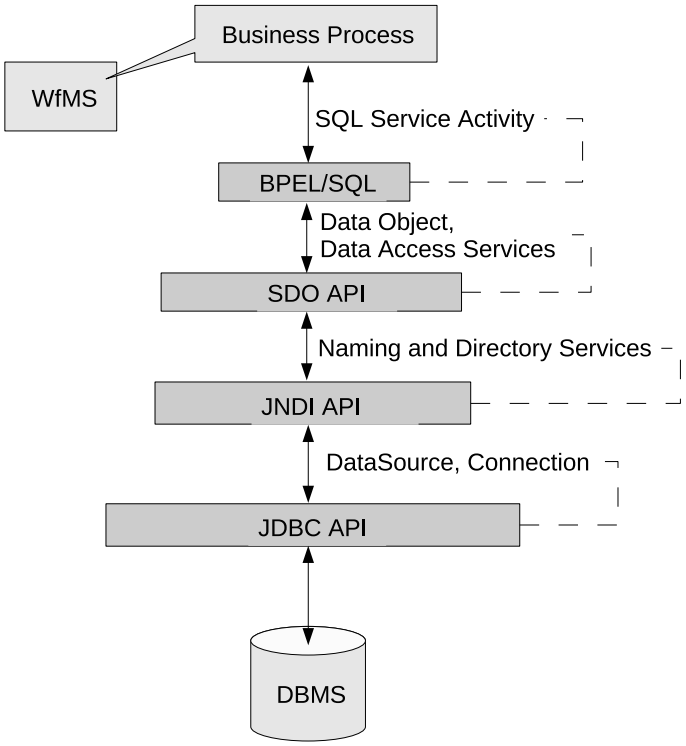

Abbildung 1. Übersicht der 4 Ebenen und ihrer Zusammenhänge

2 Java Database Connectivity (JDBC)

Die JDBC API (siehe [2] und [3]) stellt Klassen und Schnittstellen für den Zugriff auf relationale Datenbanksysteme in Java bereit. Ihr spezielles Treiberkonzept ermöglicht es auf Datenbanken verschiedenster Hersteller plattformunabhängig zuzugreifen. Um die JDBC API kennenzulernen wird ihre Architektur vorgestellt und ihre wichtigsten Klassen und Schnittstellen sowie von ihr unterstützte Konzepte näher beschrieben. Am Ende des Kapitels folgen noch Beispiele anhand

derer der Umgang und die Verwendung der JDBC API demonstriert werden sollen.

2.1 Architektur

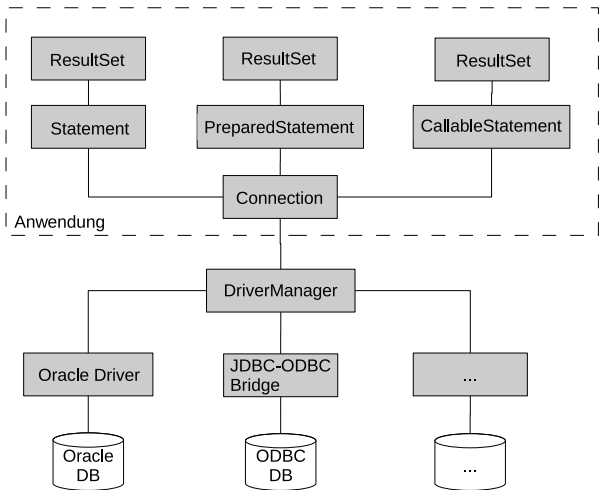

Abbildung 2. Architektur der JDBC API aus [1]

Wie in Abbildung 2 dargestellt, besteht die Architektur der JDBC API aus zwei großen Blöcken. Aus einem von der Datenquelle unabhängigen Teil, der in der späteren Anwendung integriert ist und die Verbindung zur Datenquelle über die *Connection* Schnittstelle, das Senden von SQL-Queries, die durch die Schnittstellen *Statement*, *PreparedStatement* und *CallableStatement* repräsentiert werden und die Ergebnisverarbeitung durch die *ResultSet* Schnittstelle realisiert. Sowie aus einem von der Datenquelle abhängigen Teil, der die physikalische Verbindung zur Datenquelle, über ein Treiberkonzept, realisiert. Dabei werden einfach herstellerspezifische Treiberklassen über den DriverManager registriert. Diese Treiberklassen implementieren dafür vorgegebene Schnittstellen und bilden so den Übergang zwischen der allgemeinen und der herstellerspezifischen Ebene.

Durch diese Aufteilung entstehen die wichtigsten Vorteile der JDBC API: Dynamisches Binden von Datenquellen zur Laufzeit, Unabhängigkeit des Programmcodes vom Hersteller der Datenquelle und die Plattform- und Systemunabhängigkeit der Anwendung.

2.2 API-Overview

In diesem Abschnitt werden die wichtigsten Klassen und Schnittstellen der JDBC API und einige unterstützte Konzepte vorgestellt.

2.2.1 Wichtige Klassen und Schnittstellen

DataSource Durch die Implementierung der *DataSource* Schnittstelle können Datenquellen mit Eigenschaften spezifiziert werden. Ein Objekt dieser Implementierung wird dann zum Modell einer bestimmten Datenquelle, die eindeutig über die im Objekt hinterlegten Eigenschaften bestimmt ist (z.B. Servername, Port, usw.). Dadurch wird es auch möglich dem *DataSource* Objekt einen logischen Namen mithilfe der JNDI API zuzuweisen und somit unabhängig von spezifischen Treiberinformationen zu sein. Außerdem können zur Laufzeit auftretende Änderungen, wie z.B. der Serverwechsel einer Datenbank, einfach über die Änderung der jeweiligen Eigenschaften des verwendeten *DataSource* Objekts durchgeführt werden. Mithilfe von *[DataSource Objekt].getConnection()* kann dann eine Verbindung zur Datenquelle, die durch das Objekt spezifiziert ist, hergestellt werden.

Connection *Connection* Objekte spiegeln die physikalische Verbindung zwischen der Anwendung und der Datenquelle. Eine Anwendung kann dabei beliebig viele Verbindungen haben, die entweder auf verschiedene Datenquellen oder aber auch nur auf eine verweisen. Die *Connection* Schnittstelle hält eine Vielzahl von Methoden bereit, um eine Verbindung zu parametrieren, so kann z.B. die Verbindung "nur lesend" gesetzt oder "auto-commit", das ist eine automatische Bestätigung das eine Transaktion ohne Fehler beendet wurde, an- und abgeschaltet werden. Ebenso können für ein *Connection* Objekt *Savepoints* (siehe Kapitel 2.2.2) innerhalb der Transaktionen mit *setSavepoint()* gesetzt, die *Database Metadata* (siehe Kapitel 2.2.2) einer Datenquelle mit *getMetaData()* abgefragt und Änderungen bis zu einem *Savepoint* mit *rollback()* rückgängig gemacht *werden*. Die Hauptaufgabe der *Connection* Schnittstelle ist es, SQL-Queries zu erzeugen. Dies geschieht je nach Art des SQL-Befehls mit folgenden drei Methoden: *createStatement()*, *prepareStatement()* und *prepareCall()*.

Statement Die *Statement* Schnittstelle ist die Basis des *PreparedStatement* und des *Callable Statement* Schnittstellen und hält Methoden für die Ausführung von SQL-Anweisungen, die keine Parameter enthalten, bereit. Durch die Angabe von zusätzlichen Parametern, bei der Erzeugung des *Statement* Objekts über *createStatement()*, können verschiedene Eigenschaften der Ergebnismenge der SQL-Queries definiert werden, wie z.B. deren Typ oder Änderbarkeit. Um ein *Statement* Objekt auszuführen bzw. den SQL-Befehl des Objekts, gibt es je nachdem, ob eine SQL-Query oder ein DDL bzw. DML Befehl ausgeführt wird, zwei verschiedene Methoden: *executeQuery()* oder *executeUpdate()*.

Für alle drei Statementarten gilt, dass falls das *Connection* Objekt, das sie erzeugt hat, geschlossen wird, auch die Statements geschlossen werden. Explizit ist dies über *[Statement Objekt].close()* möglich. Es empfiehlt sich allerdings ein abgearbeitetes Statement sofort explizit zu schließen, um fehlerhafte Zugriffe zu vermeiden. Ebenso sollten auch ResultSets, die auch automatisch mit dem Statement, das sie erzeugt hat, geschlossen und gelöscht

werden, explizit beendet werden, da auch hier die Gefahr von fehlerhaften Zugriffen besteht.

PreparedStatement Die *PreparedStatement* Schnittstelle erweitert *Statement*, um die Eigenschaft SQL-Befehle mit Parametern zu verwenden, die zur Laufzeit mit Werten gefüllt werden. So kann ein "vorgefertigter" SQL-Befehl mehrere Male zur Laufzeit mit unterschiedlichen Werten ausgeführt werden, wie z.B. *preparedStatement("INSERT INTO KUNDENLIST" + "(NAME, VORNAME) VALUES (?,?)")*. Die einzelnen Parametermarken müssen mit *set<Type>(Index, Wert)* Methoden gesetzt werden, d.h. um einen String für den ersten Marker zuzuweisen muss *setString(1,"Meier")* aufgerufen werden. Ausgeführt und beendet werden *PreparedStatement*-Objekte genau wie *Statement*-Objekte.

SEHR WICHTIG sind bei PreparedStatements folgende zwei Punkte:

- ALLEN Parametern MUSS ein Wert zugewiesen werden, ansonsten wird eine SQLException geworfen.
- Nachdem ein PreparedStatement ausgeführt wurde BEHALTEN ALLE Parameter ihre Werte, d.h. man muss entweder *clearParameters()* aufrufen oder die alten Werte überschreiben.

CallableStatement Die *CallableStatement* Schnittstelle erweitert *PreparedStatement* und liefert die Möglichkeit stored procedures aufzurufen und deren Resultate abzufragen. Es gibt zwei Arten von CallableStatements, mit oder ohne Rückgabewert. Falls ein Rückgabewert benötigt wird muss dieser mit *registerOutParameter()* als solcher registriert werden. Weiterhin müssen alle Variablen der stored procedure wie bei PreparedStatements mit Werten belegt werden, bevor der Aufruf ausgeführt werden darf. Ausgeführt und beendet werden *CallableStatement*-Objekte genau wie *Statement*-Objekte.

ResultSet Die *ResultSet* Schnittstelle stellt Methoden für die Abfrage und die Bearbeitung von durch Queries erzeugten Ergebnismengen bereit. Die Schnittstelle ermöglicht es *ResultSet* Objekten in Bezug auf Funktionalität und Verhalten besondere Eigenschaften aus drei verschiedenen Bereichen zu verleihen.

- *ResultSet Types*: z.B. Bewegungsrichtung durch Daten, Sichtbarkeit von Datenquellenupdates
- *ResultSet Concurrency*: z.B. Updatefähigkeit des ResultSets
- *ResultSet Holdability*: z.B. Reaktion auf Commit

In einem *ResultSet* Objekt bewegt man sich innerhalb der einzelnen Einträge, die den Zeilen entsprechen, mithilfe eines Cursors. Für die Bewegung des Cursors stehen entsprechende Methoden bereit, wie z.B. *next()*, *first()*, usw. Die jeweils zur Verfügung stehenden Cursor-Methoden werden durch den gesetzten *ResultSet* Typen bestimmt. Um auf einen expliziten Wert der gerade fokusierten Zeile zugreifen zu können, wird die Methode findColumn("Spaltenname") verwendet. Es können auch Zeilen aus *ResultSets* gelöscht werden, indem die Zeile ausgewählt und dann mit *deleteRow()* gelöscht wird. Ein *ResultSet* Objekt kann aber auch zur Änderung der Daten auf der Datenquelle genutzt werden, falls dies mit dem entsprechenden *ResultSet Concurrency* Parameter eingestellt wurde. Um nun die Daten auf der

Datenquelle zu ändern, werden die Daten im *ResultSet* selbst geändert und dann auf die Datenquelle übertragen. Es kann jedoch sein, dass das DBMS oder der JDBC-Treiber diese Art des Updates nicht unterstützen, dann müssen eben die Daten aus dem *ResultSet* gelesen und mittels SQL-Queries auf die Datenquelle übertragen werden.

2.2.2 Unterstützte Konzepte

Die JDBC API unterstützt eine Reihe von Konzepten, die im Umgang mit Datenbanken sehr hilfreich sind. Im folgenden werden diese aufgezählt und kurz erläutert:

- Two-tier und Three-tier Modell,
- Transaktionskonzept,
- Savepoints,
- Database Metadata,
- Connection Pooling und
- Batch Updates.

Durch die Unterstützung des Two-tier und Three-tier Modells kann die JDBC API sowohl direkt an eine Java Anwendung angebunden werden oder aber auch auf einem Anwendungs-Server, der per Applet angesteuert wird, integriert sein. Das Transaktionskonzept sorgt für Datenintegrität und eine konstante Sicht auf die Daten, während konkurrierende Zugriffe auf diese stattfinden. Mit Savepoints ist es möglich Zwischenschritte in Transaktionen zu markieren und zu deren Zustand zurückzukehren ohne das vorherige Änderungen betroffen sind. Die Database Metadata liefert die Möglichkeit eine Reihe von Informationen über die zugrunde liegende Datenquelle abzurufen. Dadurch kann im Vorfeld festgestellt werden, ob eine Datenquelle von der Anwendung genutzt werden kann oder nicht. Connection Pooling sorgt dafür, dass die physikalischen Verbindungen zu Datenquellen gepuffert werden und so nicht mit jedem Connection Objekt erzeugt bzw. zerstört werden. Mit Batch Updates erhält man die Möglichkeit in ein Statement Objekt nicht nur einen SQL-Befehl einzubetten, sondern beliebig viele. So wird z.B. der Datentransport zwischen Anwendung und Datenbank reduziert, da nur noch ein Statement gesendet werden muss.

2.3 Arbeiten mit der JDBC-API

In diesem Abschnitt soll eine kleine praktische Einführung in einige der im Abschnitt 2.2 vorgestellten Inhalte anhand verschiedener Beispiele gegeben werden.

Für die Beispiele wird die nachfolgende Datenbanktabelle KUNDEN verwendet:

Tabelle 1. KUNDEN Datenbanktabelle

Id	Name	Vorname	Umsatz
1	Meyer	Hans	345,20
2	Müller	Lisa	147,89

In diesem Beispiel wird in Zeile 1 ein neues *DataSource* Objekt instanziert, das dann zur Verfügung steht. In den Zeilen 3 bis 6 werden die Eigenschaften des *DataSource* Objekts über Set-Methoden mit Werten belegt. Mit der Anwendung von *getConnection()* auf das *DataSource* Objekt in Zeile 10 wird dann die physikalische Verbindung zur Datenbank hergestellt und ein *Connection* Objekt erzeugt, das diese Verbindung repräsentiert. Über das *Connection* Objekt wird in Zeile 12 ein Statement mittels createStatement() erzeugt. Die Parameter die dabei übergeben werden definieren gewisse Eigenschaften der ResultSets, die dieses Statement liefert. Im Beispiel geben wir an, dass das ResultSet nur vorwärts durchlaufen werden kann (*TYPE_FORWARD_ONLY*) und Updates auf der Datenquelle durchführen darf (CONCUR_UPDATABLE).

```
1  MyDataSource ds = new MyDataSource();
2  */Setzt die Eigenschaften des DataSource Objekts/*
3  ds.setPort(1527);
4  ds.setHost("localhost");
5  ds.setUser("User")
6  ds.setPassword("User");
7  */Liefert das Connection Objekt und stellt
8  * die physikalische Verbindung her
9  */
10 Connection con = ds.getConnection();
11 */Erzeugt ein Statement Objekt
12 Statement stmt = con.createStatement(ResultSet.TYPE_FORWARD_ONLY,
13     ResultSet.CONCUR_UPDATABLE);
```

Hier führen wir nun in Zeile 3 eine SQL-Query aus und erhalten so ein ResultSet, das Namen und Umsätze enthält. Mit der while-Schleife in Zeile 5 durchlaufen wir nun dieses ResultSet, um bestimmte Werte zu ändern. So wird in Zeile 6 überprüft, ob der Cursor momentan auf der Zeile steht, in der der Name Meyer vorkommt. Ist dies der Fall wird in Zeile 7 der Umsatzwert ausgelesen dann erhöht und in Zeile 11 zurück geschrieben. Zeile 13 sorgt dafür das die Änderung im ResultSet in der Datenbank persistent wird. Ebenso wird die Namensänderung in den Zeilen 15 bis 18 durchgeführt. Mit der *next()* Methode in Zeile 20 wird in jedem Durchlauf der Cursor vorwärts bewegt, bis es kein nächstes Element mehr gibt und die while-Schleife terminiert. Die Zeilen 23 und 25 sorgen dafür, dass das *ResultSet* und das *Statement* Objekt ordentlich beendet werden.

```
1  */Werte werden zuerst im ResultSet aktuallisiert und dann
2  */in die DB übertragen.*/
3  ResultSet rSet = stmt.executeQuery("SELECT NAME,UMSATZ FROM KUNDEN");
4  */Durchläuft das ResultSet*/
5  while (rSet.next()) {
6      if (rSet.getName == "Meyer") {
7          float umsatz = rSet.getFloat(2);
8          */auch rSet.getFloat("Umsatz") wäre möglich*/
```

```
9           umsatz += 60.0;
10          */Aktualisiert das ResultSet*/
11          rSet.updateFloat(2, umsatz);
12          */Übernimmt die Änderung in die Datenbank*/
13          rSet.updateRow();
14      }
15      if (rSet.getName == "Müller") {
16          */Nächste Zeile wird ausgewählt*/
17          rSet.updateString("Name", "Schmidt");
18          rSet.updateRow();
19      }
20      rSet.next();
21  }
22  */Schließt das ResultSet-Objekt*/
23  rSet.close();
24  */Schließt das Statement-Objekt*/
25  stmt.close();
```

Dieses Beispiel stellt eine alternative Möglichkeit zum vorherigen dar, denn Änderungen können auch direkt auf der Datenbanktabelle durchgeführt werden ohne ein ResultSet zu nutzen. Dazu wird einfach wie in Zeile 4 und 8 gezeigt, die *executeUpdate("SQL-Befehl")* Methode über ein *Statement* Objekt ausgeführt. Hier ist allerdings zu beachten, dass alle Datensätze deren Inhalt z.B. die Abfrage *"WHERE NAME LIKE '...'"* positiv beantworten auch geändert werden, d.h. hier im Beispiel müsste ein Namen ein eindeutiges Merkmal eines Datensatzes sein und dürfte nur einmal in der Tabelle auftauchen, ansonsten empfielt sich die Bearbeitung über ein ResultSet.

```
1   */Werte werden direkt in der DB über executeUpdate("") geändert
2
3   */Umsatz von Hans Meyer wird erhöht
4   stmt.executeUpdate("UPDATE KUNDEN SET UMSATZ = 405,20" +
5       "WHERE NAME LIKE 'Meyer'");
6
7   */Nachname von Lisa Müller wird geändert
8   stmt.executeUpdate("UPDATE KUNDEN SET NAME = 'Schmidt'" +
9       "WHERE NAME LIKE 'Müller'");
10
11  */Schließt das Statement-Objekt*/
12  stmt.close();
```

Nachdem wir nun die Änderungen über das ResultSet oder auch direkt in die Datenbank übernommen haben, sieht die Datenbanktabelle KUNDEN folgendermaßen aus:

Tabelle 2. KUNDEN Datenbanktabelle nach Update

Id	Name	Vorname	Umsatz
1	Meyer	Hans	405,20
2	Schmidt	Lisa	147,89

3 Java Naming and Directory Interface (JNDI)

Die JNDI API (siehe [5]) stellt Klassen und Schnittstellen für die Verwendung und Implementierung von Naming und Directory Services bereit. Naming und Directory Services werden benutzt, um Daten und Objekte anhand eines Namens abzulegen und auch wieder über diesen aufzurufen. Dadurch ist ein Zugriff auf Daten oder Objekte nicht mehr nur über die physikalischen Adressen möglich, sondern auch über dynamisch veränderbare logische Namen.

3.1 Architektur

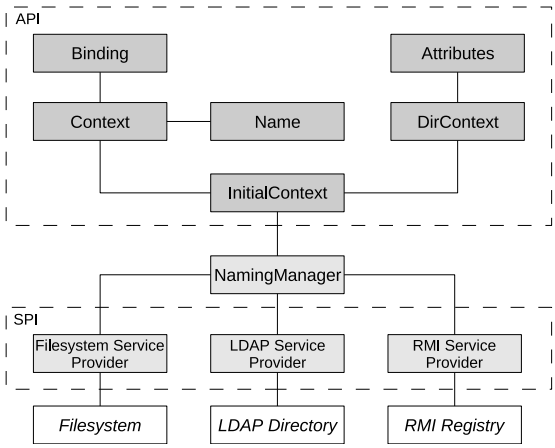

Abbildung 3. Architektur der JNDI API aus [4]

Die JNDI Architektur besteht aus der JNDI API und dem JNDI Service Provider Interface (SPI), siehe Abbildung 3. Die JNDI API stellt eine Vielzahl von *naming and directory Services,* sowie Funktionalität für die Nutzung dieser Services, für Java Anwendungen bereit. In Abbildung 3 sind die wichtigsten Schnittstellen der API enthalten. Die Schnittstelle *Binding* realisiert die

Bindung von Namen und Objekten, diese Bindungen werden dann in contexts, die von der Schnittstelle *Context* abgeleitet werden, abgelegt. Die *InitialContext* Schnittstelle ist eine Erweiterung von *Context*, der einzige Unterschied besteht darin das *InitialContext* Objekte als Einstiegspunkte bei der Namensauflösung benötigt werden und dafür noch spezielle Funktionen besitzen. Die *DirContext* Schnittstelle stellt ebenso eine Erweiterung der *Context* Schnittstelle dar, da die *DirContext* Schnittstelle für die Modellierung von Verzeichnisstrukturen erstellt wurde und zu diesem Zweck Attribut-Wert Paare über die Schnittstelle *Attribute* aufnehmen kann. Mit Hilfe der SPI können über den *NamingManager* einfach weitere Services eingebunden werden, die dann ebenfalls durch Java Anwendungen nutzbar sind. Die Aufteilung in 2 Ebenen und die Verbindung über den *NamingManager* ist der JDBC API Architektur sehr ähnlich, denn auch hier wird so die Plattformunabhängigkeit bewahrt und die Anbindung von herstellerspezifischen Diensten vereinfacht.

3.2 API-Overview

Dieser Abschnitt gibt einen tieferen Einblick in die wichtigsten Klassen und Schnittstellen der API, die in Kapitel 3.1 eingeführt wurden.

3.2.1 javax.naming Paket

Dieses Paket stellt Funktionalität für den Zugriff auf Namensdienste bereit.

Context Die *Context* Schnittstelle bildet die Basis für die Verknüpfung von Namen mit Objekten. Es hält Methoden um Objekte an- und abzubinden, Namen zu ändern, subcontexts zu erzeugen oder zu löschen und alle bindings des *Context* Objekts auszugeben.

Name Es gibt zwei Möglichkeiten in einem context Namen anzugeben, als String oder als *Name* Objekt. Die *Name* Schnittstelle ist vorallem dazu gedacht, den Umgang mit compound oder composite names zu erleichtern. Dazu hält die Schnittstelle Methoden bereit, mit denen ein Name gezielt geändert werden kann und auch einzelne Komponenten des Namens gelesen werden können. Ein compound name ist eine Sequenzen von mehreren nicht mehr teilbaren Namen (atomic names), z.B. ein Ordnername in einem Dateisystem, die nach bestimmten Richtlinien (naming conventions) verbunden sind, so z.B. Pfadnamen: usr/local/bin. Ein composite name ist ein Name, der über mehrere naming systems verteilt ist, d.h. er besteht aus einer geordneten Liste mehrerer Komponenten, von denen jede einzelne ein Namen aus einem namespace eines naming systems ist. Als Beispiel: "www.eclipse.org/abc/index.html" besteht aus 2 Komponenten, einem DNS-Namen "www.eclipse.org" und einem compound name "abc/index.html" aus einem Dateinamensraum. Ein naming system ist dabei eine verbundene Menge von contexts, die alle den selben Typ (die gleichen naming conventions) und die gleiche Menge von Operationen haben. LDAP und DNS sind z.B. naming systems. Ein namespace ist die Menge aller Namen eines naming

systems, z.B. in einem Dateisystem die Menge aller Verzeichnisnamen und Dateinamen.

Binding Die Klasse *Binding* modelliert die Bindung eines Namens an ein Objekt innerhalb eines contexts. Mit dieser Klasse werden die *Binding* Objekte erzeugt und es stehen Methoden bereit, mit denen das Objekt des bindings gelesen und gesetzt werden kann.

InitialContext Die Klasse *InitialContext* implementiert die *Context* Schnittstelle und dient als Wurzelcontext bei der Auflösung von Namen. Um einen *InitialContext* Objekt zu erzeugen, muss über sogenannte *environment properties,* die in einer Hashtabelle übergeben werden, dem Konstruktor der Name der Service Provider Klasse, des Services den man nutzen will (z.B. LDAP), mitgeteilt werden. Bei einigen Service Providern ist es auch nötig, weitere Informationen, wie z.B. die URL des Providers, mithilfe der *environment properties* anzugeben. Danach kann der *InitialContext* erzeugt und wie ein normales *Context* Objekt genutzt werden.

3.2.2 javax.naming.directory Paket

Dieses Paket stellt Funktionalität für den Zugriff auf Verzeichnisdienste bereit und bildet den Directory-Teil des JNDI. Es ist speziell für die Bindung von sogenannten Directory-Objekten an Namen ausgelegt. Als Directory-Objekte gelten z.B. Objekte die Dateien oder Drucker repräsentieren, die spezifische Eigenschaften wie z.B. "Besitzer", "Schreibschutz" und "Druckberechtigung" haben. Diese Eigenschaften werden modelliert, indem die Bindings eines *DirContext* Objekts mit Attributen verknüpft werden. So besteht die Möglichkeit eine Vielzahl von Informationen in den Attributen der Bindings zu speichern. Die Klassen und Schnittstellen des *javax.naming.directory* Pakets sind genau auf diese Art von Objekt-Namen-Bindungen ausgelegt.

DirContext Die *DirContext* Schnittstelle enthält dieselbe Funktionalität wie die *Context* Schnittstelle. Sie hat jedoch ein paar Erweiterungen für die Verwaltung der Attribute der Directory-Objekte. Die Attribute selbst werden durch die Schnittstelle *Attribute* bzw. Attributlisten durch die Schnittstelle *Attributes* modelliert. Um ein Directory-Objekt nun an einen Namen zu binden und eine Attributliste anzuhängen, wird die Methode *bind([Name], [Object], [Attributes])* verwendet. Etwas wirklich Neues ist die Möglichkeit, nach Directory-Objekten innerhalb eines contexts anhand ihrer Attribute zu suchen. So könnten beispielsweise alle Dateien eines bestimmten Benutzers ganz einfach über ein Direktory-Objekt Attribut der Form *String owner* gefunden und bereitgestellt werden.

Attribute Die *Attribute* Schnittstelle dient dazu Attribute für Directory-Objekte zu erzeugen. Jedes Attribut kann dabei null oder beliebig viele Werte besitzen. Werte können mit *add()* zu einem Attribut hinzugefügt, mit *get()* gelesen und mit *remove()* wieder entfernt werden, mit *clear()* löscht man alle Werte eines Attributs.

3.2.3 javax.naming.event Paket

Um Änderungen innerhalb des Namespaces oder der verbundenen Objekte besser greifbar zu machen, wurde für die JNDI ein Event-Konzept eingeführt. Mit Hilfe der Events und der entsprechenden Listener ist es möglich Änderungen einer Bindung zu publizieren, d.h. alle beim Listener registrierten Klassen werden sofort über Änderungen informiert. Das ist sehr wichtig wenn sich z.B. der Namen einer Bindung ändert oder das Objekt einer Bindung ausgetauscht wird, damit weiterhin auf das referenzierte Objekt über seinen Namen zugegriffen werden kann und keine fehlerhaften Referenzen benutzt werden.

NamingEvent Die Klasse *NamingEvent* representiert Events, die von Naming oder Directory Services ausgelöst werden. Es gibt zwei Arten von NamingEvents: Zum einen NamingEvents, die durch Änderungen im Namespace ausgelöst werden und zum anderen NamingEvents, die durch Änderungen der angebundenen Objekte ausgelöst werden. Für beide Arten gibt es entsprechende Listener-Schnittstellen. Jedes NamingEvent besitzt Informationen über die Quelle, die das Event gefeuert hat (EventContext), was für ein Eventtyp vorliegt, das neue binding des Objekts bzw. der neue Objektzustand und das alte binding des Objekts bzw. der alte Objektzustand. Diese Informationen sind über entsprechende Methoden zugänglich.

EventContext Die *EventContext* Schnittstelle hält Methoden um gezielt einzelne Objekte, innerhalb eines Contexts, bei einem Listener zu (de)registrieren. Mittels der Angabe eines Scope-Wertes kann die Art und Weise der Objektüberwachung ausgewählt werden.

3.2.4 javax.naming.spi Paket

Dieses Paket bildet die Implementierung der JNDI SPI und stellt Klassen und Schnittstellen für die dynamische Einbindung und die Implementierung von weiteren *naming and directory services* bereit.

3.3 Arbeiten mit der JNDI-API

Die nachfolgenden Beispiele zeigen die grundlegende Verwendung der JNDI API.

In Zeile 2 wird eine Hashtabelle erstellt, in die die environment properties abgelegt werden. Als Service Provider Klasse, diese definiert den Namensraum und die Funktionalität, die der Context später haben wird, wird in Zeile 5 eine Klasse für Dateisysteme ausgewählt. In Zeile 8 wird der Standardpfad auf dem gearbeitet wird gesetzt. Der InitialContext wird dann in Zeile 11 mit der Hashtabelle erzeugt. Mit der Methode *lookup("Name")* wird dann in Zeile 14 nach einem Dateinamen im gesetzten Pfad gesucht und dieser auf ein File-Objekt gecastet. In Zeile 17 wird der Context mittels *close()* geschlossen.

```
1 // Erzeugt die environment für den initial context
2 Hashtable env = new Hashtable();
3
```

```
4  // Nutzt eine Service Provider Klasse für Filesysteme
5  env.put(Context.INITIAL_CONTEXT_FACTORY,
6      "com.sun.jndi.fscontext.RefFSContextFactory");
7  // Setzt den Pfad
8  env.put(Context.PROVIDER_URL, "file:/usr/");
9
10 // Erzeugt den initial context
11 Context ctx = new InitialContext(env);
12
13 // Sucht das Objekt und castet es auf ein File
14 File f = (File) ctx.lookup("report.txt");
15
16 // Schließt den context wieder
17 ctx.close();
```

In diesem Beispiel wird die Bindung von Objekten und Namen vorgestellt. Mit *bind("Name", Objekt)* in Zeile 3 wird das in Zeile 2 erstellte Personenobjekt unter dem logischen Namen "chef" im Context abgelegt. Führt man nun *lookup("chef")* aus wie in Zeile 7 erhählt man die Referenz auf das zuvor angebundene Personenobjekt zurück. In Zeile 11 wird ein neu erstelltes Personenobjekt unter dem selben Namen im Context abgelegt, dafür wird deshalb die Methode *rebind("Name", Objekt)* verwendet. Führt man nun nochmal *lookup("chef")* aus wie in Zeile 15 erhählt man nun die Referenz auf das neu angebundene Personenobjekt zurück. Mit *unbind("Name")* wird in Zeile 18 die Bindung aufgelöst und aus dem Context entfernt.

```
1  // Bindet ein Objekt an einen Namen
2  Person pers = new Person("Meyer", "Hans");
3  ctx.bind("chef", pers);
4
5  // Führt ein lookup aus und
6  // liefert das Personenobjekt Hans Meyer
7  Object obj = ctx.lookup("chef");
8
9  // Ändert die Bindung
10 Person pers2 = new Person("Müller", "Lisa");
11 ctx.rebind("chef", pers2);
12
13 // Führt ein lookup aus und
14 // liefert das Personenobjekt Lisa Müller
15 Object obj = ctx.lookup("chef");
16
17 // Löst die Bindung
18 ctx.unbind("chef");
19
```

Im letzten Beispiel wird das Erstellen und Löschen eines Subcontexts und das Auflisten aller Bindungen eines Contexts erläutert. In Zeile 2 wird dazu

ausgehend vom InitialContext ein Subcontext mit dem Namen "neu" erzeugt. Nun werden in Zeile 5 alle Bindungen des InitialContext mit *list()* ausgelesen und in den Zeilen 6 bis 10 ausgegeben. Da wir nur den Subcontext erzeugt haben und keine weiteren Bindungen hinzugefügt wurden, enthält der InitialContext nur eine Bindung in der der Name "neu" an ein Context Objekt gebunden wurde. In Zeile 18 wird der oben erzeugte Subcontext mit *destroySubcontext("Name")* gelöscht.

```
1  // Erzeugt einen neuen context
2  Context cont = ctx.createSubcontext("neu");
3
4  // Liefert eine Liste aller Verknüpfungen
5  NamingEnumeration list = ctx.list();
6  while (list.hasMore()) {
7      NameClassPair ncp = (NameClassPair)list.next();
8      // Geben die Liste der verknüpften Objekte aus
9      System.out.println(ncp);
10 }
11 /* Erzeugt folgende Ausgabe:
12  *
13  * neu: javax.naming.Context
14  *
15  */
16
17 // Löscht den neuen context
18 ctx.destroySubcontext("neu");
```

4 Service Data Objects (SDO)

Die SDO API (siehe [6] und [7]) liefert ein einheitliches Konzept für den Zugriff auf und die Manipulation von heterogenen Daten bzw. Datenquellen.

4.1 Architektur

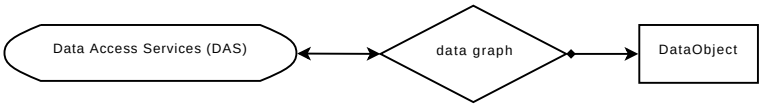

Abbildung 4. SDO Architektur aus 6

Die SDO-Architektur besteht aus drei zentralen Komponenten, die in Abbildung 4 dargestellt sind: Data Access Service, data graph und DataObject. Ein DataObject ist ein Container in den Daten mit benannten Eigenschaften, ähnlich

dem Aufbau von XML-Dateien, abgelegt werden können. Data graphs werden dazu benötigt, um Verbindungen oder Abhängigkeiten zwischen mehreren DataObjects zu modellieren und so komplexere Datengebilde erfassen zu können. Ein Beispiel für einen data graph wäre z.B. ein Modell für ein Lager mit mehreren Posten, wie ihn Abbildung 6 zeigt. Ein Data Access Service sorgt dafür das data graphs an eine oder mehrere heterogene Datenquellen übergeben und dort gespeichert werden und das in die entgegengesetzte Richtung aus einer Datenquelle gelesene Daten geladen und in DataObjects abgelegt werden. Die dadurch erzeugten DataObjects werden gegebenenfalls wieder zu einem data graph organisiert, der dann dem Client zur Verfügung steht. Abbildung 5 zeigt die Verbindung zwischen einer Anwendung und verschiedenen Datenquellen über Data Access Services. Der Data Access Service (DAS) wird manchmal auch als Data Mediator Service (DMS) bezeichnet. Durch das SDO-Konzept werden heterogene Daten so abstrahiert, das sie von ihrer Datenquelle unabhängig und somit deutlich einfacher und universeller zu verarbeiten sind. Mit der Verwendung des DAS wird weiterhin der Datenzugriff auf heterogene Datenquellen und die Datenmanipulation stark vereinfacht, da EIN zentraler Dienst alle Zugriffe ausführen kann und die Manipulation der Daten so vereinheitlicht wird. Die Gesamtheit dieser Eigenschaften wurde so mit keinem bisherigen Konzept erfasst, lediglich Teilaspekte wurden bisher durch andere Ansätze abgedeckt. Mit der SDO API steht nun ein zentrales Konzept zur Verfügung, das für alle Arten von Daten und Datenquellen genutzt werden kann. Dies stellt eine enorme Erleichterung dar, da nun bei der Entwicklung von Systemen nicht mehr viele verschiedene API's erlernt und verwendet werden müssen, sondern eine zentrale universelle API zur Verfügung steht.

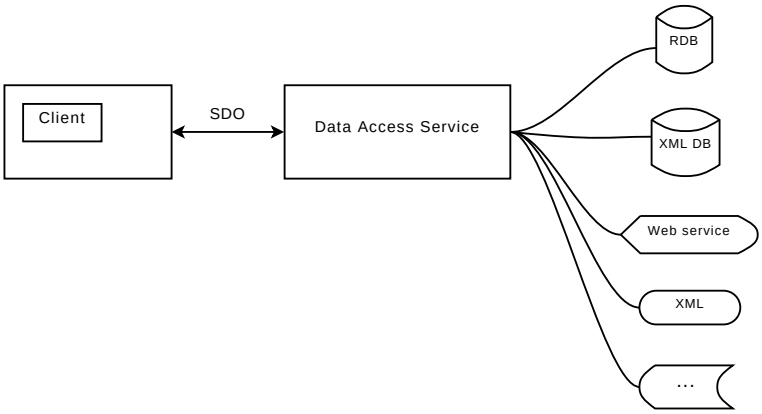

Abbildung 5. Umgang mit heterogenen Datenquellen mithilfe von SDO's

4.2 API-Overview

In diesem Abschnitt wird ein Überblick über die API gegeben und die wichtigsten Klassen, Schnittstellen und Konzepte vorgestellt.

Property Ein *Property* Objekt steht für eine benannte Eigenschaft eines DataObjects, die einen Datentyp hat und einen oder beliebig viele Werte (many-valued Property). Als Datentyp tauchen nicht nur die Standardtypen auf, sondern auch Referenzen auf andere DataObjects und die ChangeSummary eines data graphs wird ebenfalls als Property des Wurzelknotens gehalten. Eine Property sieht z.B. so aus: <phone-number, ["1234-5678", "5432-2321", "8796-9432"]>

Containment Mithilfe des *containment* werden die data graphs aufgebaut. Ein DataObject kann ein contained DataObject erzeugen, das dann das Kind des Container DataObjects ist. So lassen sich Graphen von DataObjects implizit erstellen ohne dafür eine extra Klasse zu verwenden. Das Containment ist gesteuert und jedes DataObject kann nur einen Container besitzen, d.h. wird ein bereits referenziertes DataObject bei einem neuen Container in den Properties registriert, dann wird die Referenz aus dem alten Container DataObject entfernt. Falls Zyklen bei der Referenzierung auftreten wird eine Exception geworfen.

DataObject Die Schnittstelle *DataObject* modelliert business data objects und besitzt Methoden um DataObjects zu erzeugen und ihre Properties und contained DataObjects zu verwalten. Ein DataObject hält eine Menge von Properties, von denen jede mit einem einfachen Datentypwert oder einer Referenz auf ein anderes DataObject belegt ist.

data graph Ein data graph ist die baumartige Strukturierung von mehreren DataObjects durch Referenzen. Immer wenn DataObjects transportiert oder verarbeitet werden, sind sie als data graph strukturiert und werden so auch dem DAS übergeben bzw. von ihm geliefert. Ebenso wie in Bäumen gibt es auch in data graphs genau einen Wurzelknoten, von dem aus alle anderen DataObjects erreichbar sind. Jeder Knoten, ausser der Wurzel, hat wiederum genau einen Vorgänger (Container). Über die Methoden *getContainer()* kann der Baum nach oben und über *getContainmentProperty()* nach unten durchlaufen werden. Ein Beispiel zeigt Abbildung 6, das einen data graph mit 3 DataObjects zeigt. Das DataObject vom Typ Lager modelliert ein Lager und bildet die Wurzel des data graph. Es besitzt 2 contained DataObjects (Kindobjekte) vom Typ Posten, die den Inhalt des Lagers darstellen und deshalb mit diesem in einem data graph organisiert sind. Alle DataObjects haben benannte Eigenschaften, die mit Werten belegt sind, wie z.B. das Lager eine Eigenschaft *ort* mit dem Wert "Stuttgart" und ein Posten eine Eigenschaft *name* mit dem Wert "Glühbirne" besitzt. Über die oben genannten Methoden *getContainer()* und *getContainmentProperty()* kann ein solcher data graph nun durchlaufen werden.

ChangeSummary Die *ChangeSummary* Schnittstelle stellt Methoden für die Änderungsverfolgung in data graphs und ihrer Elemente bereit. Wird eine

ChangeSummary für einen data graph angelegt, können mittels Change Logging alle Änderungen des data graphs und seiner DataObjects aufgezeichnet werden. Durch die ChangeSummary stehen viele Funktionen bereit, mit denen ein älterer Zustand des Graphen wiederhergestellt, alle Änderungen angezeigt und alte Werte abgerufen werden können. So kann bei Fehlern die Konsistenz der Daten problemlos wieder hergestellt werden. Eine ChangeSummary wird angelegt, indem man dem Wurzel DataObject des Graphen eine Property vom Typ ChangeSummaryType hinzufügt. Das Logging muss im Normalfall dann explizit gestartet werden, bevor es die Änderungen aufzeichnet.

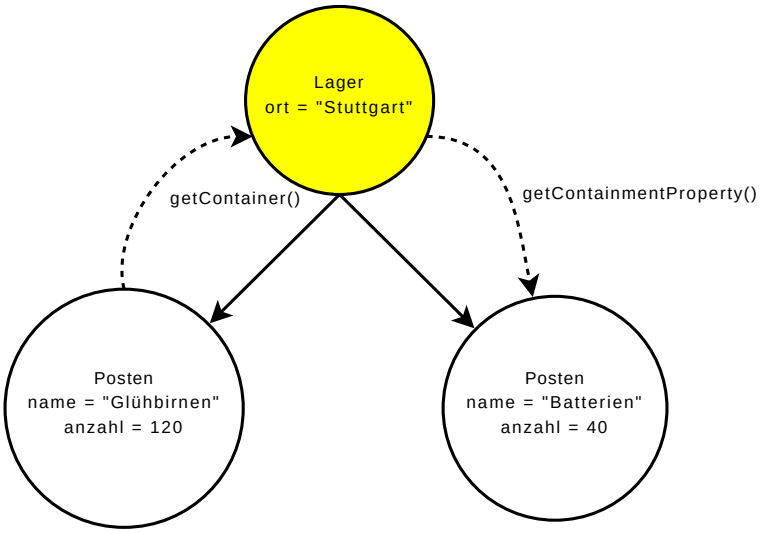

Abbildung 6. Beispiel eines data graph

4.3 Arbeiten mit der SDO-API

Das nachfolgende Beispiel zeigt die grundlegende Verwendung der SDO API. In den Zeilen 1 bis 24 werden dazu zwei DataObject Typen definiert: Ein Typ *Kunde* mit den Eigenschaften KundenID, Vorname, Nachname, Adresse und einer Referenz auf eine Liste von DataObjects vom Typ Konto (siehe Containment in Kapitel 4.2) und ein Typ *Konto* mit den Eigenschaften KontoID, Kontostand und der Referenz auf EIN DataObject vom Typ Kunde. Für alle Eigenschaften werden Getter- und Setter-Methoden definiert mit denen die Werte der Eigenschaften gelesen und gesetzt werden können. In Zeile 27 wird dann das Wurzel DataObject vom Typ *Kunde* erzeugt, indem einer DataFactory die Klasse in der sich die Typdefinition befindet übergeben wird. Ebenso erzeugen wir ein DataObject vom Typ *Konto* in Zeile 30. Um nun die Verbindung zwischen Kunde

und Konto zu erstellen, referenzieren wir in Zeile 31 zuerst im Konto DataObject den entsprechenden Kunden über setBesitzer(kunde) als Besitzer dieses Kontos, damit ist die Referenz nach oben erstellt. In Zeile 32 wird nun noch das Konto dem Kunden zugeordnet, indem setKonto(*Liste von Konten*) aufgerufen wird. Nun sind beide DataObjects in einem data graph organisiert. In den Zeilen 35 bis 40 werden nun noch die einzelnen Eigenschaften der DataObjects mit Werten belegt. Dies kann entweder mit set<Eigenschaft>(Wert) statisch geschehen wie in Zeile 35 oder auch dynamisch mit set<Datentyp>(Name der Eigenschaft, Wert) wie in Zeile 36. Am Ende des Beispiels haben wir den data graph aus Abbildung 7 erzeugt.

```
1  // Modellierung eines DataObjectTypes Kunde
2  public interface Kunde {
3          String getKundenID();
4          void setKundenID(String kundenID);
5          String getVorname();
6          void setVorname(String vorname);
7          String getNachname();
8          void setNachname(String nachname);
9          String getAdresse();
10         void setAdresse(String adresse);
11         // Setzt eine containment reference als Property
12         List<Konto> getKonten();
13         void setKonten(List<Konto> konten);
14 }
15 // Modellierung eines DataObjectTypes Konto
16 public interface Konto {
17         String getKontoID();
18         void setKontoID(String kontoID);
19         float getKontostand();
20         void setKontostand(float kontostand)
21         // Setzt eine Referenz auf ein Kunden DataObject
22         Kunde getBesitzer();
23         void setBesitzer(Kunde besitzer);
24 }
25
26 // Erzeugt ein DataObject Kunde ohne Referenz (Wurzel des data graphs)
27 DataObject kunde = DataFactory.INSTANCE.create(Kunde.class);
28
29 // Erzeugt ein DataObject Konto und setzt es als Kindknoten von kunde
30 DataObject konto = DataFactory.INSTANCE.create(Konto.class);
31 konto.setBesitzer(kunde);
32 kunde.setKonto(new List<Konto>(konto));
33
34 // Die DataObjects kunde und konto werden mit Werten gefüllt
35 kunde.setKundenID("meyerhs"); //statischer Zugriff auf die Variable
36 kunde.setString("Vorname", "Hans"); //dynamsicher Zugriff auf die Variable
37 kunde.setNachname("Meyer");
```

```
38 kunde.setAdresse("Haupstrasse 1");
39 konto.setKontoID("meyerhsK001");
40 konto.setKontostand(2000);
41
```

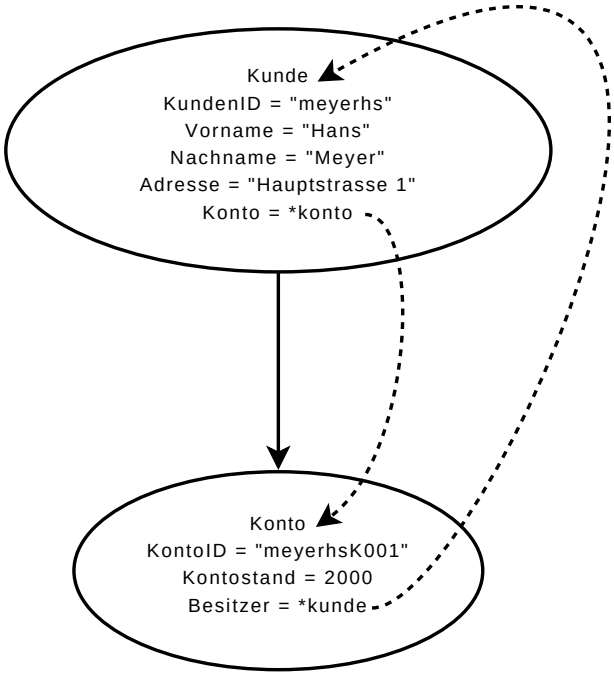

Abbildung 7. Der entstandene data graph

5 Business Process Execution Language/Structured Query Language (BPEL/SQL)

Mit BPEL/SQL (siehe [8] und [9]) soll eine inline SQL Unterstützung für Workflows realisiert werden, d.h. SQL-Befehle sollen direkt aus einem Prozess abgesetzt werden, Daten direkt in einen Prozess geladen werden können und Updates auf der Datenquelle mit Prozessdaten ausgeführt werden. Es gibt bereits eine Vielzahl von Ansätzen, die sich mit BPEL/SQL und der Umsetzung des Konzepts in Workflows beschäftigen. In diesem Kapitel werden die Ansätze von Microsoft und Oracle kurz vorgestellt und die Realisierung von IBM etwas näher betrachet, da dieses Konzept die größte SQL Unterstützung und Funktionalität bereithält.

5.1 Herstellerspezifische Ansätze

Microsoft stellt mit der *Workflow Foundation* eine Code-Bibliothek bereit, die eine Basis für die Enwicklung und Ausführung von Workflow-basierten Anwendungen darstellt. Die *Workflow Foundation* bietet eine *Custom Activity Library*, in der neue Aktivitäten für ein Problem definiert werden können. Mit Hilfe von ADO.NET, das Methoden für die Verarbeitung von relationalen Daten bereithält, können benutzerspezifische SQL Aktivitäten realisiert werden.

Oracle verwendet mit seiner SOA Suite *XPath Extension Functions* und keine Service Activities für die Realisierung der inline SQL Unterstützung. Diese werden in *BPEL Assign Activities* gekapselt und so in den Prozess eingebunden und ausgeführt.

IBM verwendet ebenfalls Service Activities, die als Container für SQL-Befehle verwendet werden. Auf den Ansatz von IBM wird in Kapitel 5.2 ausführlicher eingegangen.

Eine Übersicht der drei Ansätze in Kurzform liefert Tabelle 3, in der die Eigenschaften der Ansätze anhand von einigen Punkten nochmal gegenübergestellt werden.

Tabelle 3. Übersicht der herstellerspezifischen Ansätze aus [9]

	IBM Business Execution Suite	**Microsoft Workflow Foundation**	**Oracle SOA Suite**
Generelle Informationen			
Workflow Sprache	BPEL	C#, VB, XOML (BPEL)	BPEL
Level der Prozessmodellierung	graphisch, (markup)	graphisch, Code, markup	graphisch, (markup)
Workflow Entwicklungstool	WebSphere Integration Developer	Workflow Designer	Process Designer
Daten Management Fähigkeiten			
SQL Inline Support	SQL Activity, Retrieve Set Activity, Atomic SQL Sequence	customized SQL Activity	XPath Extension Functions
Referenz zu externen Datenmengen	Set Reference, statischer Text	statischer Text	statischer Text
Verwirklichung der Mengenrepräsentation	proprietäres XML RowSet	DataSet Objekt	proprietäres XML RowSet
Referenz zu externen Datenquellen	dynamisch, statisch	statisch	statisch
Weitere Funktionen	Lifecycle Management für DB Entities		

5.2 BPEL/SQL-Ansatz von IBM

Im IBM Ansatz werden die SQL-Befehle in so genannte *Information Service Activities* eingebunden und können so in einen Business Process integriert werden. Für die Bereitstellung der inline SQL Unterstützung existieren drei verschiedene Container:

SQL Snippet Erlaubt die Ausführung von SQL-Befehlen gegen Datenbanktabellen. Es werden sowohl DML- wie auch DDL-Befehle unterstützt. Alle Operationen arbeiten mit Set-Referenzen, auf deren Beschreibung später eingegangen wird, nicht mit den Daten selbst. Durch die Verwendung von Referenzen kann so das aufwändige Laden der Daten in den Prozesscache umgangen werden und somit die Performance erhöht und die Prozessbelastung verringert werden.

Retrieve Set Erlaubt das Laden von referenzierten Daten in Variablen innerhalb des Prozesses. Die Daten werden dabei in Sets, deren Beschreibung im folgenden Text nachgeliefert wird, gekapselt und sind so im Prozess sichtbar.

Atomic SQL Sequence Erlaubt die Angabe beliebig vieler *SQL Snippets* oder *Retrieve Sets* in einer *Information Service Activity*. Die Sequenz der Anweisungen wird dabei innerhalb einer Transaktion verarbeitet und zwar genau in der Reihenfolge, in der die Befehle in der *Atomic SQL sequence* angegeben sind.

Es gibt zwei Arten von Referenzen: Referenzen auf Datenquellen und Referenzen auf Sets. Ein Set ist ein Container für den Inhalt von Datenbanktabellen. Eine entsprechende Set-Variable vom Typ *tSet* dient dazu, die von einer Set-Referenz referenzierten Daten aufzunehmen. Referenzen auf Datenquellen sind vom Typ *tDataSource*, wobei die entsprechenden Datenquellen über ihren JNDI Namen identifiziert werden. Eine Set-Referenz dagegen zeigt auf eine konkrete Datenbanktabelle einer Datenquelle. Sie enthält eine Referenz zur Datenquelle (*tDataSource*), den Tabellennamen und optional auch noch das Schema der Tabelle sowie Befehle um Tabellen zu Erzeugen oder zu Löschen. Set-Referenzen sind vom Typ *tSetReference*. Durch die Verwendung von Set-Referenzen anstelle von statischen Tabellennamen kann dynamisch zur Laufzeit die Tabelle ausgewählt werden und Daten per Referenz, über mehrere Aktivitäten und Prozesse, bereit gehalten werden.

Werden *SQL-Snippets* zu einer *Information Service Activity* hinzugefügt müssen zwei Variablen definiert werden: eine tDataSource-Variable für die Empfängerdatenquelle und eine *tSetReference*-Variable für die Referenz, die durch die Ausführung des SQL-Befehls, zurückgegeben wird. Werden *Retrieve Sets* zu einer *Information Service Activity* hinzugefügt muss im Normalfall eine neue *tSet*-Variable angelegt werden, die die Daten, auf die die *tSetReference* zeigt, aufnehmen kann. Beim hinzufügen von *Atomic SQL Sequences* ist im Grunde nichts weiteres zu beachten, da auch hier für die einzelnen Bestandteile (*SQL Snippets, Retrieve Sets*) dasselbe wie gerade besprochen gilt.

Die Lebenszeit von Tabellen, die innerhalb eines Prozesses erzeugt werden, ist normalerweise mit der Lebenszeit des Prozesses verbunden, der sie erzeugt

hat. Es können aber auch eigene sogenannte preparation- und cleanup-Befehle definiert werden, mit denen die Lebenszeit von Tabellen explizit im Prozess gesteuert werden kann. Diese Befehle werden dazu direkt in SQL-Snippets eingefügt und dann entweder bei der Installation (preparation) und dem Entfernen (cleanup) des Prozesses oder beim Starten (preparation) und Beenden (cleanup) der Prozessinstanz ausgeführt. So können Tabellen innerhalb des Prozessrahmens explizit erstellt und gelöscht werden.

6 Zusammenfassung

Durch die Verwendung der hier vorgestellten Konzepte und API's lassen sich immer komplexere und verständlichere Datenbankanwendungen realisieren. Auch hier ist die Abstraktion der Schlüssel zum Erfolg, denn durch den Aufbau von immer leistungsfähigeren Hierarchien wird die Komplexität an der sichtbaren Schicht deutlich reduziert und es wird möglich komplexe Anwendungen von "Laien" implementieren zu lassen. Gerade im Bereich der SOA stellen diese Vereinfachungen ein zentrales Ziel dar, denn durch die graphische Programmierung von Workflows und die Ausweitung deren Funktionalität, z.B. durch die Unterstützung von inline SQL, können immer leistungsfähigere Systeme mit immer weniger Aufwand erstellt werden. Auch der Umgang mit Daten wird deutlich durch die Abstraktion erleichtert. Durch die Verwendung des SDO Konzepts spielt es im Grunde keine Rolle mehr welcher Datentyp vorliegt oder wo und wie er hinterlegt ist. Alle typspezifischen Unterschiede werden aus dem Sichtfeld verbannt und in den unteren Schichten geregelt.

Literatur

1. Java Enterprise in a Nutshell,
   http://docstore.mik.ua/orelly/java-ent/jenut/ch02_01.htm, 08.06.2009
2. JDBC Homepage,
   http://java.sun.com/javase/technologies/database/index.jsp, 08.06.2009
3. JDBC 4.0 API Specification,
   http://jcp.org/aboutJava/communityprocess/final/jsr221/index.html, 08.06.2009
4. Java Enterprise in a Nutshell,
   http://docstore.mik.ua/orelly/java-ent/jenut/ch06_01.htm, 08.06.2009
5. JNDI Homepage,
   http://java.sun.com/products/jndi/, 08.06.2009
6. Service Data Objects Homepage,
   http://jcp.org/aboutJava/communityprocess/final/jsr235/index.html, 08.06.2009
7. Simplify and unify data with a Service Data Objects architecture,
   http://www.ibm.com/developerworks/library/ws-sdoarch/, 08.06.2009
8. WebSphere Integration Developer Hilfe-Bibliothek,
   http://publib.boulder.ibm.com/infocenter/dmndhelp/v6rxmx/index.jsp?topic=
   /com.ibm.is.bpel.help.doc/topics/accessdata.htm, 08.06.2009
9. Vrhovnik, M.; Schwarz, H.; Radeschütz, S.; Mitschang, B.: An Overview of SQL Support in Workflow Products. In: Proc. of the 24th International Conference on Data Engineering (ICDE 2008), Cancún, México, April 7-12, 2008