iBATIS

Data Mapper

(SQL Maps)

Version 2.0

Handbuch

1. November, 2007

Übersetzung Axel Leucht (Axel.Leucht@gmx.net)

Inhaltsverzeichnis

Einführung

Das iBATIS Data Mapper Framework hilft Ihnen Ihre Menge an Java Code zu verringern, die Sie benötigen um auf eine relationale Datenbank zuzugreifen.. iBATIS bildet JavaBeans auf SQL Befehle mit Hilfe einer einfachen XML Beschreibung ab. *Einfachheit* ist der Hauptvorteil von iBATIS gegenüber anderen objektrelationalen Werkzeugen. Um iBATIS Data Mapper verwenden zu können, benötigen Sie nur Kenntnisse von JavaBeans, XML und SQL. Daneben müssen sie wenig Zusätzliches wissen. Es gibt kein komplexes Verfahren um Tabellen zu joinen oder komplexe Abfragen (Queries) auszuführen. Mit Data Mapper haben sie die volle Kontrolle über das eingesetzte SQL.

Data Mapper (com.ibatis.sqlmap.*)

Konzept

Mit der iBATIS Data Mapper API können Entwickler JavaBean-Objekte auf PreparedStatement Parameter und ResultSets abbilden. Die Philosophie hinter Data Mapper ist einfach: stelle ein einfaches Framework für ungefähr 80% der üblich notwendigen JDBC Funktionalität mit nur ca. 20% Code zur Verfügung.

Wie funktioniert das Ganze?

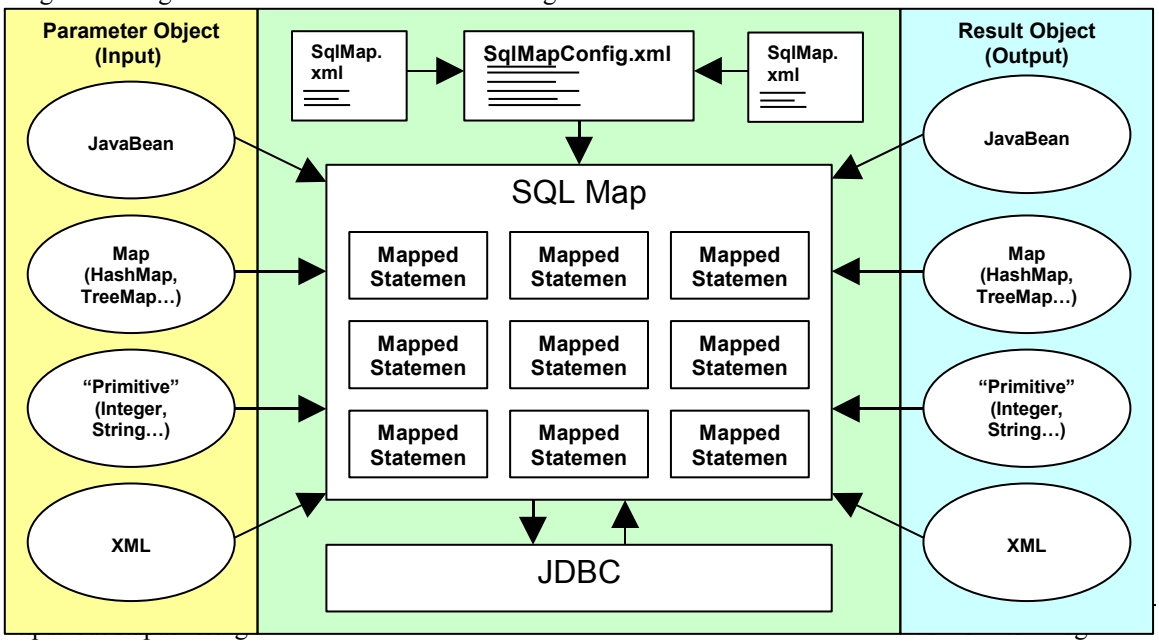
Data Mapper stellt ein einfaches Framework bereit, um in einer XML-Beschreibungsdatei JavaBeans, Map Implementierungen, primitive Wrapper-Typen (String, Integer…) und sogar XML Dokumente auf einen SQL Befehl abzubilden. Die folgende Beschreibung kann als Schablone der Arbeit verstanden werden:

1. Erstelle ein Objekt als Parameter (entweder eine JavaBean, Map oder primitiven Wrappertyp). Dieses Parameterobjekt wird für die Eingabewerte eines update Befehls oder die Werte einer where Klausel einer Abfrage benutzt, ...

2. Führe das Mapped-Statement aus. Hinter dieses Schritt verbirgt sich die ganze Magie. Data Mapper Framework erzeugt ein PreparedStatement, setzt alle notwendigen Parameter, führt den SQL Befehl aus und baut das Resultat-Objekt aus dem JDBC-ResultSet.

3. Im Falle eines SQL-Update gibt es die Anzahl der veränderten Zeilen zurück. Im Falle eines SQL-Select wird ein einziges Objekt oder eine Kollektion von Objekten zurückgegeben. Wie Parameterobjekte kann es sich bei Resultat Objekten um einfache JavaBeans, eine Map, einen primitiven Typ-Wrapper oder XML handeln.

Folgendes Diagramm illustriert den Zusammenhang.



Installation

Die Installation des iBATIS Data Mapper Frameworks besteht darin, die notwendigen JAR Dateien im Klassenpfad aufzunehmen. Dieses kann dadurch geschehen den Klassenpfad beim Start der JVM (*java* -cp Argument) anzugeben, oder im Falle einer Webapplikation sie in das /WEB-INF/lib Verzeichnis zu kopieren. Eine vollständige Beschreibung über den Java Classpath ist nicht Teil dieses Dokumentes. Wenn sie mehr darüber kennen lernen möchten, können sie an folgenden Stellen mehr darüber erfahren:

http://java.sun.com/j2se/1.4/docs/tooldocs/win32/classpath.html
http://java.sun.com/j2se/1.4.2/docs/api/java/lang/ClassLoader.html
http://java.sun.com/j2se/1.4.2/docs/

iBATIS wird in einer einzigen JAR Datei ausgeliefert. Der Name dieser Datei ist von der Form:

ibatis-*version*.*build*.jar

zum Beispiel, **ibatis-2.3.0.677.jar**.

Üblicherweise genügt es, diese eine JAR Datei in den Applikations-Klassenpfad aufzunehmen.

JAR Dateien und ihre Abhängigkeiten

Wenn ein Framework zu viele Abhängigkeiten besitzt, ist es schwierig es in eine Applikation oder mit anderen Frameworks zu integrieren. Ein Hauptaugenmerk von 2.0 war die Reduktion der Abhängigkeiten zu anderen Frameworks. Deshalb hat iBATIS unter JDK 1.4 keine weiteren Abhängigkeiten. Optionale JAR Dateien können von den angegebener Website geladen werden. Folgende Übersicht zeigt eine kurze Zusammenfassung des Zwecks der optionalen Pakete und warum sie diese benötigen könnten:

Beschreibung	Zu benutzen wenn...	Abhängigkeit
Altsystem JDK Support	Wenn die Applikation mit Java-Version vor JDK 1.4 läuft oder ihr Applikationsserver diese Jar Dateien nicht bereits mitbringt.	**JDBC 2.0 Erweiterungen** **http://java.sun.com/products/jdbc/download.html** **JTA 1.0.1a** **http://java.sun.com/products/jta/** **Xerces 2.4.0** **http://xml.apache.org/xerces2-j/**
iBATIS Kompatibilität mit älteren Versionen	Wenn sie das vorherige iBATIS (1.x) DAO Framework, oder das vorherige Data Mapper (1.x) Framework verwenden.	**iBATIS DAO 1.3.1** **http://sourceforge.net/projects/ibatisdb/**
Bytecode Erweiterung zur Laufzeit	Wenn sie durch die Laufzeit-Erweiterung des CGLIB 2.0 Bytecodes das späte Laden (lazy loading) oder die Reflection Performance steigern möchten.	**CGLIB 2.0** **http://cglib.sf.net**
DataSource Implementation	Wenn sie den Jakarta DBCP Connection Pool verwenden möchten.	**DBCP 1.1** **http://jakarta.apache.org/commons/dbcp/**
Verteiltes Caching	Wenn sie OSCache für einen zentralen oder verteilte Cachelösung verwenden möchten.	**OSCache 2.0.1** **http://www.opensymphony.com/oscache/**
Logging	Wenn sie Log4J einsetzen möchten.	**Log4J 1.2.8** **http://logging.apache.org/log4j/docs/**
Logging	Wenn sie Jakarta Commons Logging verwenden möchten	**Jakarta Commons Logging** **http://jakarta.apache.org/commons/logging**

Upgraden von Version 1.x

Sollten Sie upgraden?

Der einfachste Weg zu bestimmen, ob sie upgraden sollten, ist: Versuchen sie es! Es gibt aber einige Hinweise zu beachten.

1. Version 2.0 ist fast vollständig kompatibel zur Version 1.x, Für Einige reicht daher der Tausch von wenigen JAR Dateien bereits aus. Zwar machen sie dadurch keinen Gebrauch neuer Features, ist dafür aber einfach umzusetzen. Sie müssen ihre XML Dateien oder Java Code nicht ändern. In seltenen Fällen kann es aber zu Inkompatibilitäten kommen.

2. Bei der zweite Option konvertieren sie zwar ihre XML Dateien auf die 2.0 Spezifikation, verwenden aber weiterhin die 1.x Java API. Durch diesen Ansatz gibt es weniger Kompatibilitätsunterschiede. Eine Ant-Task ist dem Framework beigefügt, um ihre XML Dateien zu konvertieren (Beschreibung unten).

3. Bei der dritten Option konvertieren sie ihre XML Dateien (wie bei #2) und ihren Java Code. Es gibt leider keine Unterstützung für die Anpassung des Java Codes und sie müssen dies daher selber tun.

4. Die letzte Option ist es, überhaupt nicht upzugraden. Wenn sie Schwierigkeiten bei der Umstellung haben, scheuen sie sich nicht und lassen das laufende System auf einem 1.x Release. Es ist nicht die schlechteste Vorgehensweise, Altprojekte auf 1.x zu lassen und nur neue Projekte auf Version 2.0 zu beginnen. Wenn sie natürlich ihre Alt-Applikation umbauen, bis sie es kaum noch wieder erkennen, können sie auch auf den neuen Data Mapper upgraden.

XML Konfigurationsdateien von 1.x nach 2.x upgraden

Im 2.0 Framework ist ein Konverter für ihre XML Dokumente enthalten. Dieser Konverter läuft als eine Ant-Task. Allerdings ist diese Konvertierung komplett optional, da 1.x Code ihre alten XML Dokumente zur Laufzeit umwandelt. Es kann aber trotzdem eine gute Idee sein, upzugraden. Erstens werden sie weniger Kompatibilitätsunterschiede bemerken und sie können einige neue Merkmale verwenden (selbst wenn sie noch die alte 1.x Java API verwenden).

Die Ant-Task in ihrer build.xml Datei sieht wie folgt aus:

```
<taskdef name="convertSqlMaps"
      classname="com.ibatis.db.sqlmap.upgrade.ConvertTask"
      classpathref="classpath"/>

<target name="convert">
  <convertSqlMaps todir="D:/targetDirectory/" overwrite="true">
    <fileset dir="D/sourceDirectory/">
      <include name="**/maps/*.xml"/>
    </fileset>
  </convertSqlMaps>
</target>
```

Wie sie sehen, sieht das genauso aus wie eine Ant Copy-Task, und tatsächlich erweitert diese die Ant Copy-Task. Alles was Copy kann, kann somit auch diese Task. (schauen sie in die Dokumentation der Ant Copy-Task für eine genauere Beschreibung).

JAR Dateien: Raus mit den Alten, rein mit den Neuen

Wenn sie upgraden möchten, ist es gute Praxis ihre vorhandenen (alten) iBATIS Dateien und Abhängigkeiten zu löschen und durch die neuen Dateien zu ersetzen. Stellen sie sicher, Komponenten nicht zu löschen, die sie noch benötigen. Die meisten JAR Dateien sind optional und sie benötigen diese nur bei gewissen Gegebenheiten. Obige Diskussion gibt ihnen Hinweise bei ihrer Entscheidung.

In folgender Tabelle werden alte und neue Dateien beschrieben.

Alte Dateien	**Neue Dateien**
ibatis-db.jar *Nach dem 1.2.9b Release wurde diese Datei in 3 Dateien geteilt:* ibatis-common.jar ibatis-dao.jar ibatis-sqlmap.jar	Ibatis-*version*.*build*.jar ***(erforderlich)***
commons-logging.jar commons-logging-api.jar commons-collections.jar commons-dbcp.jar commons-pool.jar oscache.jar jta.jar jdbc2_0-stdext.jar xercesImpl.jar xmlParserAPIs.jar jdom.jar	commons-logging-1-0-3.jar ***(optional)*** commons-collections-2-1.jar ***(optional)*** commons-dbcp-1-1.jar ***(optional)*** commons-pool-1-1.jar ***(optional)*** oscache-2-0-1.jar ***(optional)*** jta-1-0-1a.jar ***(optional)*** jdbc2_0-stdext.jar ***(optional)*** xercesImpl-2-4-0.jar ***(optional)*** xmlParserAPIs-2-4-0.jar ***(optional)*** xalan-2-5-2.jar ***(optional)*** log4j-1.2.8.jar ***(optional)*** cglib-full-2-0-rc2.jar ***(optional)***

Der Rest dieses Handbuchs führt sie in die Verwendung des Data Mapper Frameworks ein.

Die SQL Map XML Konfiguatonsdatei (http://ibatis.apache.org/dtd/sql-map-config-2.dtd)

Data Mapper wird durch eine zentrale XML Datei konfiguriert. In dieser werden Details wie zum Beispiel DataSources, Data Mapper und andere Optionen wie Thread Management festgelegt. Folgendes Beispiel zeigt eine SQL Map Konfigurationsdatei:

SqlMapConfig.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE sqlMapConfig
    PUBLIC "-//ibatis.apache.org//DTD SQL Map Config 2.0//EN"
    "http://ibatis.apache.org/dtd/sql-map-config-2.dtd">

<!-- Stellen sie sicher immer obigen korrekten XML Header zu nehmen! -->
<sqlMapConfig>

  <!-- Eigenschaften in der properties-Datei sind in der Form name=value und können als Platzhalter
in der Konfigurationsdatei verwendet werden (z.B. "${driver}". Die Dateiangabe ist relativ zum
Klassenpfad und komplett optional. -->
  <properties resource=" examples/sqlmap/maps/SqlMapConfigExample.properties " />

  <!-- Diese Einstellungen betreffen Details der SqlMapClient Konfiguration hauptsächlich zum
Transaktion Management. Sie sind alle optional (nähere Beschreibung folgt im Dokument). -->
  <settings
    cacheModelsEnabled="true"
    enhancementEnabled="true"
    lazyLoadingEnabled="true"
    maxRequests="128"
    maxSessions="10"
    maxTransactions="5"
    useStatementNamespaces="false"
    defaultStatementTimeout="5"
    statementCachingEnabled="true"
    classInfoCacheEnabled="true"
  />

  <!-- Diese Element definiert eine Factory, die von iBATIS benutzt wird, um Resultat Objekte zu
erzeugen. Dieses Element ist optional (nähere Beschreibung folgt im Dokument). -->
  <resultObjectFactory type="com.mydomain.MyResultObjectFactory" >
      <property name="someProperty" value="someValue"/>
  </resultObjectFactory>

  <!-- Mit Type Aliase können sie kürzere Namen statt langer, voll qualifizierten Klassennamen
definieren. -->
  <typeAlias alias="order" type="testdomain.Order"/>

  <!-- Definiere eine DataSource für diese SQL Map und verwende hierzu eine SimpleDataSource.
       Beachten sie die Verwendung von Eigenschaften aus der oben angegebenen properties-Datei--
>
  <transactionManager type="JDBC" >
    <dataSource type="SIMPLE">
      <property name="JDBC.Driver" value="${driver}"/>
      <property name="JDBC.ConnectionURL" value="${url}"/>
      <property name="JDBC.Username" value="${username}"/>
      <property name="JDBC.Password" value="${password}"/>
      <property name="JDBC.DefaultAutoCommit" value="true" />
      <property name="Pool.MaximumActiveConnections" value="10"/>
      <property name="Pool.MaximumIdleConnections" value="5"/>
      <property name="Pool.MaximumCheckoutTime" value="120000"/>
```

```
    <property name="Pool.TimeToWait" value="500"/>
    <property name="Pool.PingQuery" value="select 1 from ACCOUNT"/>
    <property name="Pool.PingEnabled" value="false"/>
    <property name="Pool.PingConnectionsOlderThan" value="1"/>
    <property name="Pool.PingConnectionsNotUsedFor" value="1"/>
  </dataSource>
 </transactionManager>

  <!-- Laden sie alle SQL Map XML Dateien für diese SQL Map.  Pfade sind relativ zum Klassenpfad.
Zur Zeit haben wir nur eine Datei -->
  <sqlMap resource="examples/sqlmap/maps/Person.xml" />
</sqlMapConfig>
```

Die folgenden Beschreibung erläutert die einzelnen Abschnitte in der SQL Map Konfigurationsdatei.

Das <properties> Element

Zu einer SQL Map Konfigurationsdatei können sie Angaben mit einer Standard-Java Propertiesdatei verbinden. Angaben der Form (name=value) innerhalb der Konfigurationsdatei können durch eine Variable referenziert werden. Wenn zum Beispiel die Propertiesdatei folgende Zeile enthält:

driver=org.hsqldb.jdbcDriver

kann in der Konfigurationsdatei durch die Variable ${driver} der entsprechende Wert referenziert werden. Also zum Beispiel:

<property name="JDBC.Driver" value="**${driver}**"/>

Dieses ist bequem beim Entwickeln, Testen oder Deployment der Applikation. Es macht es einfacher die Applikation auf unterschiedliche Umgebungen anzupassen. Die Propertiesdatei muss entweder im Classpath (bei Angabe des *resource* Attributs) enthalten sein oder durch eine gültig URL (bei Angabe des *url* Attributs) geladen werden. Zum Beispiel lädt die folgende Anweisung die Propertiesdatei von einem absoluten Dateipfad:

<properties url="file:///c:/config/my.properties" />

Das <settings> Element

Mit dem <settings> Element können sie verschiedene Optionen oder Optimierungen für die SqlMapClient Instanz, die durch die XML Datei beschrieben wird, festlegen. Dieses settings Element und alle ihre Attribute sind vollständig optional. Die möglichen Attribute und ihr Verhalten beschreibt folgende Tabelle:

maxRequests	Dies legt die maximale Anzahl von Threads, die gleichzeitig SQL Statements ausführen möchten, fest. Threads jenseits dieser Grenze werden blockiert bis ein anderer Thread seine Ausführung beendet hat. Verschiedene DBMS haben verschiedenen Grenzen, aber jede Datenbank hat doch irgendeine Grenze. Werte sollten mindestens 10 mal so groß sein wie maxTransactions (siehe unten) und mindestens so groß wie maxSessions zusammen mit maxTransactions. Eine Reduktion von parallelen Anfragen führt häufig zu einer *besseren* Performance. *Beispiel: maxRequests="256"* *Voreinstellung: 512*

maxSessions	Dies legt die maximale Anzahl gleichzeitig offener Sessions (oder Clients) fest. Werte sollten größer oder gleich der Anzahl von maxTransactions und kleiner als maxRequests sein. Eine kleinere Anzahl von parallel offenen Sessions führt zu einem geringeren Speicherverbrauch. *Beispiel: maxSessions=”64”* *Voreinstellung: 128*
maxTransactions	Dieses legt die maximale Anzahl von Threads fest, die SqlMapClient.startTransaction() zeitgleich aufrufen dürfen. Threads jenseits dieser Grenze werden blockiert bis ein anderer Thread die Transaktion beendet. Verschiedene DBMS haben verschiedenen Grenzen, aber jede Datenbank hat doch irgendeine Grenze. Werte sollten kleiner gleich zu maxSessions und wesentlich kleiner als maxRequests sein. Eine kleinere Anzahl von gleichzeitig offenen Transaktionen führt oft zu besserer Performance. *Beispiel: maxTransactions=”16”* *Voreinstellung: 32*
cacheModelsEnabled	Diese Einstellung aktiviert oder deaktiviert alle Cache Modelle, Dieses ist hilfreich beim Debuggen. *Beispiel: cacheModelsEnabled=”true”* *Voreinstellung: true (aktiviert)*
lazyLoadingEnabled	Diese Einstellung aktiviert oder deaktiviert lazy loading. Dieses ist hilfreich beim Debuggen. *Beispiel: lazyLoadingEnabled=”true”* *Voreinstellung: true (aktiviert)*
enhancementEnabled	Diese Einstellung aktiviert die Erweiterung des Laufzeit Bytecodes um verbessert auf JavaBean Eigenschaft zuzugreifen und lazy loading ausführen zu können. *Beispiel: enhancementEnabled=”true”* *Voreinstellung: false (deaktiviert)*
useStatementNamespaces	Wird diese Einstellung aktiviert, müssen sie jedes Mapped-Statement mit seinem voll qualifizierteren Namen ansprechen. Dieser besteht aus der Kombination des sqlMap-Namens und des Statement-Namens. Zum Beispiel: queryForObject(“sqlMapName.statementName”); *Beispiel: useStatementNamespaces=”false”* *Voreinstellung: false (deaktiviert)*

defaultStatementTimeout	(ab iBATIS Version 2.2.0) Diese Einstellung ist ein Ganzzahl-Wert der für alle JDBC Abfragen angewendet wird. Dieser Wert kann bei jedem "statement" Attribut überschrieben werden. Wird kein Wert angegeben, wird kein Abfrage-Timeout gesetzt, es sei denn, im spezifischen "statement" Attribute ist doch ein Wert definiert. Der Wert gibt die Zeit in Sekunden an, die der Treiber auf das Ende des Statements wartet. Beachten sie, das nicht alle Treiber dies unterstützen.
classInfoCacheEnabled	Wird diese Einstellung aktiviert, verwaltet iBATIS einen Cache für bereits überprüfte Klassen. Dies führt zu einer deutlichen Verbesserung der startup Zeiten wenn viele Klassen mehrfach verwendet werden. *Beispiel: classInfoCacheEnabled="true"* *Voreinstellung: true (aktiviert)*
statementCachingEnabled	(ab iBATIS Version 2.3.0) Wird diese Einstellung aktiviert, verwaltet iBATIS einen Cache für Prepared-Statements. Dies führt zu einer deutlichen Verbesserung der Performance. *Beispiel: statementCachingEnabled="true"* *Voreinstellung: true (aktviert)*

Das <resultObjectFactory> Element

Wichtig: Dieses Feature gibt es ab iBATIS Versionen 2.2.0.

Mit dem *resultObjectFactory* Element können sie eine Factory Klasse angeben, die benutzt wird, wenn Objekte bei der Ausführung von SQL Befehlen erzeugt werden. Dieses Element ist optional – wenn sie es nicht angeben nutzt iBATIS ein internes Verfahren um Objekte zu erzeugen (class.newInstance()).

iBATIS erzeugt in folgenden Fällen neue Objekte:

1. Beim Mappen von Zeilen aus einem ResultSet (der gebräuchlichste Fall)
2. Bei einem verschachtelten Select Befehl eines *result* Elements in einer *resultMap*. Wenn das verschachtelte Select Statement eine *parameterClass* deklariert, erzeugt und instantiiert iBATIS das Objekt bevor es den verschachtelten Select Befehl ausführt.
3. Beim Ausführen von Stored Procedures – iBATIS erzeugt Objekte für die OUTPUT Parameter.
4. Bei der Abarbeitung von verschachtelten Result-Maps. Wird die verschachtelte Result-Map in Verbindung der groupBy Angabe, um das N+1 Problem bei Queries zu verhindern, benutzt, handelt es sich typischerweise um den Typ Collection, List oder Set. In diesen Fällen können sie eine spezifische Factory angeben, Bei einem 1:1 Join in einer Result-Map instantiiert iBATIS ein Domäennobjekt durch die Factory.

Wenn sie eine eigene Factory implementieren möchten, muss diese das Interface **com.ibatis.sqlmap.engine.mapping.result.ResultObjectFactory** implementieren und ihre Klasse benötigt einen public default Konstruktor. Das ResultObjectFactory Interface besitzt zwei Methoden – eine erzeugt das Objekt, die andere akzeptiert alle Werte aus der Konfigurationsdatei..

Haben sie zum Beispiel folgendes *resultObjectFactory* Element angegeben:

```
<resultObjectFactory type="com.mydomain.MyResultObjectFactory" >
  <property name="someProperty" value="someValue"/>
</resultObjectFactory>
```

Dann sollte eine mögliche Implementiert so aussehen:

```
package com.mydomain;

import com.ibatis.sqlmap.engine.mapping.result.ResultObjectFactory;

public class MyResultObjectFactory implements ResultObjectFactory {

  public MyResultObjectFactory() {
    super();
  }

  public Object createInstance(String statementId, Class clazz)
      throws InstantiationException, IllegalAccessException {

    // create and return instances of clazz here...

  }

  public void setProperty(String name, String value) {
    // save property values here...
  }
}
```

iBATIS ruft die *setProperty* Methode für jedes Property-Element, die in der Konfigurationsdatei angegeben sind, auf. Alle Properties werden gesetzt bevor die *createInstance* Method aufgerufen wird.

iBATIS ruft *createInstance* jedes Mal auf, wenn ein Objekt für eine der oben beschriebenen Fälle benötigt wird.. Geben sie `null` von ihrer *createInstance* Methode zurück, versucht iBATIS das Objekt mit seiner

normalen Behandlungsroutine (class.newInstance()) zu erzeugen. Geben sie `null` für die Anforderung eine `java.util.Collection` oder `java.util.List` zu erzeugen zurück, wird von iBATIS eine `java.util.ArrayList` erzeugt. Geben sie `null` für die Anforderung `java.util.Set` zu erzeugen zurück, wird von iBATIS eine `java.util.HashSet` erzeugt. iBATIS übergibt die aktuelle Statement ID, damit sie wissen in welchem Kontext ein Objekt angefordert wird..

Das <typeAlias> Element

Mit dem *typeAlias* Element können sie einen kürzeren Namen statt eines langen voll qualifizierten Klassennamen angeben. Zum Beispiel:

```
<typeAlias alias="shortname" type="com.long.class.path.Class"/>
```

Es gibt eine Reihe von vordefinierten Alias-Angaben in der Konfiguration:

Transaction Manager Aliases	
JDBC	com.ibatis.sqlmap.engine.transaction.jdbc.JdbcTransactionConfig
JTA	com.ibatis.sqlmap.engine.transaction.jta.JtaTransactionConfig
EXTERNAL	com.ibatis.sqlmap.engine.transaction.external.ExternalTransactionConfig
Data Source Factory Aliases	
SIMPLE	com.ibatis.sqlmap.engine.datasource.SimpleDataSourceFactory
DBCP	com.ibatis.sqlmap.engine.datasource.DbcpDataSourceFactory
JNDI	com.ibatis.sqlmap.engine.datasource.JndiDataSourceFactory

Das <transactionManager> Element

1.0 Konversionshinweise: Data Mapper 1.0 erlaubte die Definition von mehreren DataSources. Dieses war aber schwerfällig und führte oftmals zu schlechten Praktiken. Daher erlaubt 2.0 nur noch eine einzige DataSource. Benötigen Sie einen mehrfachen Einsatz der Konfiguration so sollten sie mehrere Properties-Dateien, die für die unterschiedlichen Umgebungen definiert sind, einsetzen. Sie können auch als Parameter beim Bau der SQL Map verwendet werden (Details im Java API Abschnitt).

Mit dem <transactionManager> Element können sie das Transaktionsmanagement festlegen. Das *type* Attribut legt die Art des Transaktionsmanagements fest. Werte können entweder ein Klassenname oder ein Typ-Alias sein. Die drei möglichen Transaktionsmanagement-Angaben sind: JDBC, JTA oder EXTERNAL.

JDBC – Sie verwenden JDBC um eine Transaktion mit den üblichen Connection commit() oder rollback() Methoden zu kontrollieren.

JTA – Mit diesem Transaktion Manager nimmt SQL Map an einer globalen JTA teil und ist daher oftmals Teil eines größeren Gültigkeitsbereichs der sich zum Beispiel über mehrere Datenbanken oder andere transaktionale Ressourcen erstrecken kann. Die Konfiguration erfordert eine UserTransaction, die eine JNDI Ressource referenziert. Das JNDI DataSource Beispiel unten zeigt ein Beispiel für eine derartige Konfiguration.

EXTERNAL – Hiermit geben sie an, sich selbst um das Transaktionsmanagement zu kümmern. Sie können immer noch eine DataSource definieren, Transaktionen aber werden vom Framework weder committed noch zurück gerollt. Dieses impliziert, das ein externer Teil der Applikation sich um Transaktionen kümmern muss. Diese Einstellung ist auch sinnvoll bei nicht-transaktionalen Datenbanken (zum Beispiel Read-only).

Normalerweise committed iBATIS Transaktionen nur bei einem Insert, Update oder Delete. **Dieses ist selbst dann der Fall wenn sie commitTransaction() aufrufen,** Dieses Verhalten kann in manchen Fällen

zu Problemen führen. Wenn sie möchten, das iBATIS immer eine Transaktion committed, können sie das commitRequired Attribut auf true setzen. Beispiele für einen sinnvollen Einsatz sind:

1. Sie rufen eine stored procedure auf, die neben Updates auch Zeilen zurück liefert. Wenn sie die Prozedur durch die queryForList() Methode aufrufen – normalerweise commited iBATIS die Transaktion nicht. Aber in einem solchen Fall würden die Updates verloren gehen (rolled back).
2. Wenn sie in In WebSphere Connection Pooling und eine JNDI <dataSource> mit einem JDBC oder JTA Transaktion-Manager benutzen. WebSphere verlangt, das alle pooled Verbindungen committed werden oder die Verbindung wird nicht wieder dem Pool zur Verfügung gestellt,

Beachten sie das die Angabe von *commitRequired* keinen Effekt bei der Angabe eines EXTERNAL Transaktion-Manager besitzt.

Einige Transaktion-Manager erlauben zusätzliche Parameter. Folgende Tabelle zeigt die möglichen zusätzlichen Angaben:

<table>
<tr><th>Transaction Manager</th><th>Properties</th></tr>
<tr><td>EXTERNAL</td><td>
<table>
<tr><th>Property</th><th>Description</th></tr>
<tr><td>DefaultAutoCommit</td><td>Bei der Angabe von “true” für diesen Parameter wird setAutoCommit(true) auf der zugrunde liegenden Connection für jede Transaktion aufgerufen wenn dieses nicht der Wert der zugrunde liegenden DataSource sein sollte.

Bei der Angabe von “false” (oder keine Angabe) für diesen Parameter wird setAutoCommit(false) auf der zugrunde liegenden Connection für jede Transaktion aufgerufen wenn dieses nicht der Wert der zugrunde liegenden DataSource sein sollte.

Das Verhalten kann durch eine Angabe des “SetAutoCommitAllowed” Parameters überschrieben werden.</td></tr>
<tr><td>SetAutoCommitAllowed</td><td>Bei der Angabe von “true” (oder keine Angabe) für diesen Parameter, wird das Verhalten durch die Angaben des “DefaultAutoCommit” Paramters bestimmt.

Bei der Angabe von “false” für diesen Parameter ruft iBATIS in keinem Fall setAutoCommit auf– dieses ist in Umgebungen, wie zum Beispiel WebSphere, hilfreich, in denen unter keinen Umständen setAutoCommit aufgerufen werden darf.</td></tr>
</table>
</td></tr>
<tr><td>JTA</td><td>
<table>
<tr><th>Property</th><th>Description</th></tr>
<tr><td>UserTransaction</td><td>Dieser Parameter ist erforderlich.

Der Wert der user transaction. Beachtem sie, dass in vielen Fällen dies zu “java:comp/UserTransaction” gesetzt sein sollte.</td></tr>
</table>
</td></tr>
</table>

Das <dataSource> Element

Teil der Transaktion Manager Konfiguration ist ein dataSource Element mit einem Satz an Parametern um eine DataSource einzurichten, um diese in SQL Map zu verwenden. Zur Zeit existieren drei Datasource Factories im Framework, aber sie können jederzeit eine eigene schreiben. Die zur Verfügung stehenden DataSourceFactory Implementierungen werden im folgenden beschrieben.

SimpleDataSourceFactory

Durch eine SimpleDataSource Factory geben sie eine rudimentäre, gepoolte Datasource an, die ideal in Fällen ist, in denen ein Container keine DataSource zur Verfügung stellt

```
<transactionManager type="JDBC">
  <dataSource  type="SIMPLE">
      <property name="JDBC.Driver" value="org.postgresql.Driver"/>
      <property name="JDBC.ConnectionURL"
                  value="jdbc:postgresql://server:5432/dbname"/>
      <property name="JDBC.Username" value="user"/>
      <property name="JDBC.Password" value="password"/>
      <!-- OPTIONAL PROPERTIES BELOW -->
      <property name="JDBC.DefaultAutoCommit" value="false"/>
      <property name="Pool.MaximumActiveConnections" value="10"/>
      <property name="Pool.MaximumIdleConnections" value="5"/>
      <property name="Pool.MaximumCheckoutTime" value="120000"/>
      <property name="Pool.TimeToWait" value="10000"/>
      <property name="Pool.PingQuery" value="select * from dual"/>
      <property name="Pool.PingEnabled" value="false"/>
      <property name="Pool.PingConnectionsOlderThan" value="0"/>
      <property name="Pool.PingConnectionsNotUsedFor" value="0"/>
      <property name="Driver.DriverSpecificProperty" value="SomeValue"/>
  </dataSource>
</transactionManager>
```

Beachten sie, das Eigenschaften die mit "*Driver.*" beginnen, als Eigenschaftswerte an den JDBC Treiber weitergereicht werden.

DbcpDataSourceFactory

Diese Implementierung verwendet Jakarta DBCP (Database Connection Pool) . Eine derart konfigurierte DataSource ist dann ideal wenn der Applikation/Web Container keine geeignete Implementierung bereitstellt oder wenn sie eine standalone Applikation laufen lassen. Mit IBATIS können sie direkt Eigenschaften der DBCP DataSource setzen. Zum Beispiel:

```
<transactionManager type="JDBC">
  <dataSource type="DBCP">
    <property name="driverClassName" value="${driver}"/>
    <property name="url" value="${url}"/>
    <property name="username" value="${username}"/>
    <property name="password" value="${password}"/>
    <!-- OPTIONAL PROPERTIES BELOW -->
    <property name="maxActive" value="10"/>
    <property name="maxIdle" value="5"/>
    <property name="maxWait" value="60000"/>
    <!-- Use of the validation query can be problematic.
         If you have difficulty, try without it.  -->
    <property name="validationQuery" value="select * from ACCOUNT"/>
    <property name="logAbandoned" value="false"/>
    <property name="removeAbandoned" value="false"/>
    <property name="removeAbandonedTimeout" value="50000"/>
    <property name="Driver.DriverSpecificProperty" value="SomeValue"/>
```

```
  </datasource>
</transactionManager>
```

Sie können alle möglichen Einstellmöglichkeiten hier nachlesen:

http://jakarta.apache.org/commons/dbcp/configuration.html

Beachten sie das Eigenschaften die mit "*Driver.*" beginnen, als Eigenschaftswerte an den JDBC Treiber weitergereicht werden.

iBATIS unterstützt auch eine weniger flexible alte Möglichkeit der Konfiguration. Wir empfehlen aber, das oben beschriebene Verfahren, wenn möglich, einzusetzen.

```
<transactionManager type="JDBC">  <!-- Legacy DBCP Configuration -->
  <dataSource type="DBCP">
    <property name="JDBC.Driver" value="${driver}"/>
    <property name="JDBC.ConnectionURL" value="${url}"/>
    <property name="JDBC.Username" value="${username}"/>
    <property name="JDBC.Password" value="${password}"/>
    <!-- OPTIONAL PROPERTIES BELOW -->
    <property name="Pool.MaximumActiveConnections" value="10"/>
    <property name="Pool.MaximumIdleConnections" value="5"/>
    <property name="Pool.MaximumWait" value="60000"/>
    <!-- Use of the validation query can be problematic.
         If you have difficulty, try without it.  -->
    <property name="Pool.ValidationQuery" value="select * from ACCOUNT"/>
    <property name="Driver.DriverSpecificProperty" value="SomeValue"/>
  </datasource>
</transactionManager>
```

Diese Eigenschaften sind die einzigen von iBATIS unterstützen Werte bei der Altdaten-Konfiguration. Beachten sie, das Eigenschaften die mit "*Driver.*" beginnen, als Eigenschaftswerte an den JDBC Treiber weitergereicht werden.

JndiDataSourceFactory

Diese Implementierung erhält eine DataSource von einem JNDI Kontext, Diese wird üblicherweise von einem Applikationsserver bereitgestellt. Üblicherweise werden derartige Verbindungen vom Container verwaltet und gepooled. Das Standardverfahren eine JDBC DataSource zu erhalten ist über einen JNDI Kontext. JndiDataSourceFactory stellt ein probates Mittel bereit, auf eine DataSource über JNDI zuzugreifen. Die Konfigurationsparameters müssen folgende Angaben enthalten:

```
<transactionManager type="JDBC" >
  <dataSource type="JNDI">
    <property name="DataSource" value="java:comp/env/jdbc/jpetstore"/>
  </dataSource>
</transactionManager>
```

Obige Konfiguration verwendet ein normales JDBC Transaktionsmanagement. Aber zusammen mit einer durch einen Container verwalteten Ressource, kann diese auch wie folgt konfiguriert werden:

```
<transactionManager type="JTA" >
  <property name="UserTransaction" value="java:/comp/UserTransaction"/>
  <dataSource type="JNDI">
    <property name="DataSource" value="java:comp/env/jdbc/jpetstore"/>
  </dataSource>
</transactionManager>
```

Beachten sie, das über die *UserTransaction* Angabe auf einen JNDI Pfad verweist, über den die UserTransaction Instanz gefunden werden kann. Dieses ist bei JTA Transaktionsmanagement notwendig, damit die SQL Map in einem größeren Rahmen (z.B. mehrere Datenbanken, andere transaktionale Ressourcen) teilnehmen kann.

JNDI Kontext Werte können vor dem Lookup angegeben werden, indem solche Werte mit dem Präfix "*context.*" spezifiziert werden. Zum Beispiel:

<property name="context.java.naming.provider.url" value= "ldap://somehost:389"/>

Das <sqlMap> Element

Im sqlMap Element werden explizit einzelne SQL Map Dateien oder weitere SQL Map Konfigurationsdateien angegeben.. Jede SQL Map XML Datei, die sie verwenden möchten, muss in diesem Abschnitt definiert werden. Die SQL Map XML Dateien werden entweder als stream Ressource über den Classpath oder von einer URL geladen. Hier sind einige Beispiele:

```
<!-- CLASSPATH RESOURCES -->
<sqlMap resource="com/ibatis/examples/sql/Customer.xml" />
<sqlMap resource="com/ibatis/examples/sql/Account.xml" />
<sqlMap resource="com/ibatis/examples/sql/Product.xml" />

<!-- URL RESOURCES -->
<sqlMap url="file:///c:/config/Customer.xml " />
<sqlMap url="file:///c:/config/Account.xml " />
<sqlMap url="file:///c:/config/Product.xml" />
```

Die nächsten Abschnitte beschreiben Details von SQL Map XML Dateien.

Die SQL Map XML Datei (http://ibatis.apache.org/dtd/sql-map-config-2.dtd)

In den bisherigen Beispielen haben wir die einfachste Formt von Data Mapper kennen gelernt. Es gibt weitere Optionen innerhalb eines SQL Map Dokuments. Hier ein Beispiel eines Mapped-Statement das von einiges weiteren Features Gebrauch macht.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE sqlMap
   PUBLIC "-//ibatis.apache.org//DTD SQL Map 2.0//EN"
   "http://ibatis.apache.org/dtd/sql-map-2.dtd">
<sqlMap namespace="Product">

        <cacheModel   id="productCache"   type="LRU">
                <flushInterval hours="24"/>
                <property name="size" value="1000" />
        </cacheModel>

        <typeAlias alias="product" type="com.ibatis.example.Product" />

        <parameterMap   id="productParam" class="product">
                <parameter property="id"/>
        </parameterMap>

        <resultMap    id="productResult" class="product">
                <result property="id" column="PRD_ID"/>
                <result property="description" column="PRD_DESCRIPTION"/>
        </resultMap>

        <select  id="getProduct"  parameterMap="productParam"
                 resultMap="productResult"  cacheModel="product-cache">
            select * from PRODUCT where PRD_ID = ?
        </select>

</sqlMap>
```

ZU VIEL? Obwohl das Framework einiges an Arbeit für sie übernimmt, sieht das trotzdem nach viel Arbeit (XML) nur für einen einzigen einfachen Select-Befehl aus. Keine Angst. Hier ist eine kürzere Version.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE sqlMap
   PUBLIC "-//ibatis.apache.org//DTD SQL Map 2.0//EN"
   "http://ibatis.apache.org/dtd/sql-map-2.dtd">
<sqlMap   namespace="Product">

    <select  id="getProduct"  parameterClass=" com.ibatis.example.Product"
                              resultClass="com.ibatis.example.Product">
        select
           PRD_ID as id,
           PRD_DESCRIPTION as description
        from PRODUCT
        where PRD_ID = #id#
    </select>

</sqlMap>
```

Nun, beide SQL-Befehle sind funktional völlig gleich - aber es gibt subtile Unterschiede. In der zweiten Version wird kein Cache definiert, so dass jede Verwendung dieses Mapped-Statements dazu führt auf die Datenbank zuzugreifen. Zweitens macht das zweite Beispiel von auto-mapping von iBATIS Gebrauch, das

einen kleinen Performance Nachteil mit sich bringt.. Trotzdem würden beide Statements genau *gleich* aus dem Java Code verwendet werden. Daher können sie mit einer simplen und einfachen Lösung starten und später auf eine erweitertes Mapping wechseln, wenn sie das für erforderlich halten. Eine einfache, schnell umsetzbare Lösung ist gute Praxis in der modernen Softwareentwicklung.

In einer SQL Map XML Datei können mehrere cache-Modelle, Parameter Maps, Result Maps und Statements angegeben werden. Organisieren sie ihre Statements nach ihren Gutdünken (gruppieren sie diese zum Beispiel nach logischen Gesichtspunkten).

Mapped Statements

Das zentrale Konzept von Data Mapper sind Mapped Statements. Mapped Statements kann jeder SQL Befehls sein. Diese können Parameter Maps (Eingabe) und Result Maps (Ausgabe) beinhalten. Ist der Fall sehr einfach, kann das Mapped-Statement auch direkt mit Parameter und Resultaten angegeben werden. Zu einem Mapped-Statement kann auch ein cache-Modell definiert werden um häufig genutzte Resultate im Speicher vorrätig zu halten.

```
<statement    id="statementName"
              [parameterClass="some.class.Name"]
              [resultClass="some.class.Name"]
              [parameterMap="nameOfParameterMap"]
              [resultMap="nameOfResultMap"]
              [cacheModel="nameOfCache"]
              [timeout="5"]>
      select * from PRODUCT where PRD_ID = [?|#propertyName#]
      order by [$simpleDynamic$]
</statement>
```

Bei *statement* kann es sich um jedes *insert*, *update*, *delete*, *select*, *procedure* handeln. Alle Angaben in [**Klammern**] sind optional und in einigen Fällen sind nur bestimmte Angaben zulässig. So kann ein wirklich einfaches Beispiel aussehen:

```
<insert id="insertTestProduct" >
      insert into PRODUCT (PRD_ID, PRD_DESCRIPTION) values (1, "Shih Tzu")
</insert>
```

Das Beispiel wird wahrscheinlich nicht in einer solch einfachen Form vorkommen. Aber es kann nützlich sein, wenn sie mit dem Framework einen beliebigen SQL Befehl ausführen möchten. Oftmals werden sie aber mit den JavaBean Mapping Merkmalen, Parameter Maps und Result Maps, arbeiten, da darin die wirklichen Fähigkeiten liegen. Die folgenden Abschnitte beschreiben Struktur, Attribute und ihre Arbeitsweise.

Statement Typen

Das <statement> Element kann als Obermenge aller SQL Befehle verstanden werden. Häufig aber ist es eine bessere Idee die mehr spezialisierten Formen zu benutzen. Diese sind über ihre XML DTD intuitiver zu benutzen und stellen manchmal Merkmale bereit, die das normale <statement> nicht kann. Folgende Tabelle fasst diesen Zusammenhang zusammen:

Statement Element	Attribute	Kind Element	Methoden
<statement>	id parameterClass resultClass parameterMap resultMap cacheModel resultSetType fetchSize xmlResultName remapResults timeout	Alle dynamischen Element	insert update delete Alle Abfragen (query)-Methoden
<insert>	id parameterClass parameterMap timeout	Alle dynamischen Elemente <selectKey>	insert update delete
<update>	id parameterClass parameterMap timeout	Alle dynamischen Elemente	insert update delete
<delete>	id parameterClass parameterMap timeout	Alle dynamischen Elemente	insert update delete
<select>	id parameterClass resultClass parameterMap resultMap cacheModel resultSetType fetchSize xmlResultName remapResults timeout	Alle dynamischen Elemente	Alle Abfragen (query)-Methoden
<procedure>	id parameterClass resultClass parameterMap resultMap cacheModel fetchSize xmlResultName remapResults timeout	Alle dynamischen Elemente	insert update delete Alle Abfragen (query)-Methoden

Das SQL

Das SQL ist natürlich der interessanteste Teil einer Map. Es kann sich dabei um jeden gültigen Befehl für ihre Datenbank bzw. JDBC Treiber handeln. Sie können jede Funktion nutzen und sogar mehrere Befehle senden wenn ihr Treiber dies unterstützt. Da sie SQL und XML in einer Datei kombinieren, können Probleme mit Sonderzeichen auftreten. Das häufigste Problem ist das größer oder kleiner Zeichen (<>). Dies sind oft in SQL erforderlich, stellen aber reservierte Symbole in XML dar. Es gibt eine einfache Lösung für XML Sonderzeichen, die sie in SQL verwenden möchten. Betten sie den SQL-Teil in einen CDATA Abschnitt, wird keines der Sonderzeichen geparsed und das Problem ist behoben. Beispielsweise:

```
<select id="getPersonsByAge" parameterClass=”int” resultClass="examples.domain.Person">
        SELECT *
        FROM PERSON
        WHERE AGE <![CDATA[  >  ]]>  #value#
</select>
```

Wiederverwenden von SQL Fragmenten

Beim Schreiben von SqlMaps werden sie oft doppelte Fragmente im SQL-Teil schreiben müssen, zum Beispiel die FROM-Klausel oder bei Nebenbedingungen (constraint-statement). iBATIS bietet hier ein einfaches aber wirkungsvolle Verfahren an, mit dem derartig Teile wiederverwendet werden können.. Nehmen wir als einfaches Beispiel an, das wir Items bekommen und ihre Anzahl zählen möchten. Normalerweise würden wir so etwas schreiben:

```
<select id="selectItemCount" resultClass="int">
        SELECT COUNT(*) AS total
        FROM items
        WHERE parentid = 6
</select>

<select id="selectItems" resultClass="Item">
        SELECT id, name
        FROM items
        WHERE parentid = 6
</select>
```

Um die Verdoppelung zu vermeiden, können wir die Tags <sql> und <include> verwenden. Mit dem <sql> Tag definieren wir ein SQL-Fragment und mit dem <include> Tag fügen wir ein solches Fragment in einem Statement ein. Also:

```
<sql id="selectItem_fragment">
        FROM items
        WHERE parentid = 6
</sql>

<select id="selectItemCount" resultClass="int">
        SELECT COUNT(*) AS total
        <include refid="selectItem_fragment"/>
</select>

<select id="selectItems" resultClass="Item">
        SELECT id, name
        <include refid="selectItem_fragment"/>
</select>
```

Das <include> Tag ist sich des Namensraums bewusst (namespace aware) in dem es definiert wurde und sie können daher Fragmente aus anderen SqlMaps referenzieren. (in der Art und Weise wie iBATIS die SqlMaps lädt, sollten die Fragmente aber geladen bevor sie in einem Statement eingefügt werden).

Fragmente sind in ihrer Ausführungsumgebung eingeschlossen. Daher können sie darin alle Parameter verwenden.

```
<sql id="selectItem_fragment">
        FROM items
        WHERE parentid = #value#
</sql>

<select id="selectItemCount" parameterClass="int" resultClass="int">
        SELECT COUNT(*) AS total
        <include refid="selectItem_fragment"/>
</select>

<select id="selectItems" parameterClass="int" resultClass="Item">
        SELECT id, name
        <include refid="selectItem_fragment"/>
</select>
```

Auto-Generated Keys

Viele relationale Datenbank unterstützen die automatische Erzeugung von Primärschlüsseln. Leider ist dieses Merkmal oft proprietär. Data Mapper unterstützt dieses durch einen <selectKey> Abschnitt innerhalb des <insert> Elements. Sowohl pre-generated keys (z.B.. Oracle) als auch post-generated (MS-SQL Server) Verfahren werden unterstützt. Hier einige Beispiele:

```
<!—Oracle SEQUENCE Example -->
<insert id="insertProduct-ORACLE" parameterClass="com.domain.Product">
    <selectKey resultClass="int" >
        SELECT STOCKIDSEQUENCE.NEXTVAL AS ID FROM DUAL
    </selectKey>
     insert into PRODUCT (PRD_ID,PRD_DESCRIPTION)
     values  (#id#,#description#)
</insert>

<!— Microsoft SQL Server IDENTITY Column Example -->
<insert id="insertProduct-MS-SQL" parameterClass="com.domain.Product">
     insert into PRODUCT (PRD_DESCRIPTION)
     values  (#description#)
    <selectKey resultClass="int" >
        SELECT @@IDENTITY AS ID
    </selectKey>
</insert>
```

Das selectKey Statement wird vor dem insert Statement ausgeführt, wenn es vor der insert SQL Amweisung steht, steht es danach wird es auch danach ausgeführt. Im obigen Oracle Beispiel wird selectKey ausgeführt **bevor** der insert Befehl ausgeführt wird (sinnvoll bei einer Sequence). Das SQL Server Beispiel dagegen zeigt, das das selectKey Statement ausgeführt **nachdem** das insert Sstatement ausgeführt wird (sinnvoll bei identity Spalten).

Ab iBATIS Version 2.2.0 können sie auch die Reihenfolge der Befehle definieren. Das *selectKey* Element unterstützt mit dem Attribute *type* die Angabe dieser Reihenfolge. Werte können entweder “pre” oder “post” sein - das Statement wird entweder vor (pre) oder nach (post) dem insert-Befehl ausgeführt. Wenn sie das *type* Attribut angeben wird dieser Wert genommen und die Angabe im *selectKey* Element ignorert. Im folgenden Beispiel wird das *selectKey* Statement vor dem insert Befehl ausgeführt obwohl es nach dem insert-Befehl geschrieben wird.

```
<insert id="insertProduct-ORACLE-type-specified" parameterClass="com.domain.Product">
      insert into PRODUCT (PRD_ID,PRD_DESCRIPTION)
      values  (#id#,#description#)
     <selectKey resultClass="int" type="pre" >
           SELECT STOCKIDSEQUENCE.NEXTVAL AS ID FROM DUAL
     </selectKey>
</insert>
```

<selectKey> Attribute Referenz:

<selectKey> Attribute	*Description*
resultClass	Die Java Klasse die beim Aufruf des selectKey> Statement, erzeugt werden soll. Üblicherweise ein Integer oder Long.
keyProperty	Die Eigenschaft die im Parameter Objekt bei der Ausführung des <selectKey> Statement gefüllt wird. Wird diese Eigenschaft nicht angegeben, versucht iBATIS die Eigenschaft aus dem Spaltennamen der Datenbank zu finden. Wurde keine Eigenschaft gefunden, wird auch keine Eigenschaft gesetzt aber iBATIS liefert immer noch den erzeugten Key als Ergebnis des <insert> Statement.
type	"pre" or "post". Wenn angegeben, bestimmt dieses, ob das select key Statement bevor (pre) oder nach (post) dem entsprechenden insert Statement ausgeführt wird . Wird nichts angegeben, wird die Reihenfolge von der Platzierung innerhalb des insert Statement abhängig gemacht. Ist es vor dem SQL platziert, dann wird auch das selectKey vor dem insert-Statement ausgeführt). Dieses Attribut gibt es ab iBATIS Version 2.2.0..

Stored Procedures

Gespeicherte Prozeduren (stored procedure) werden mit dem <procedure> Statement Element unterstützt. Folgendes Beispiel zeigt eine stored procedure mit output Parametern.

```
<parameterMap id="swapParameters" class="map" >
   <parameter property="email1" jdbcType="VARCHAR" javaType="java.lang.String" mode="INOUT"/>
   <parameter property="email2" jdbcType="VARCHAR" javaType="java.lang.String" mode="INOUT"/>
</parameterMap>

<procedure id="swapEmailAddresses" parameterMap="swapParameters" >
   {call swap_email_address (?, ?)}
</procedure>
```

Der Aufruf vertauscht zwei email Adressen, die in zwei Spalten (Datenbank-Tabelle) und auch im Parameter Objekt (Map) vorliegen. Das Parameter Objekt wird ebenfalls geändert, sofern das mapping `mode` Attribut "INOUT" oder "OUT" ist. Sonst bleibt es unverändert. Natürlich können unveränderliche (immutable) Objekte nicht geändert werden.

Achtung! Verwenden sie nur die Standard JDBC stored procedure Syntax. Schauen sie in die JDBC CallableStatement Dokumentation für weitere Informationen.

parameterClass

Der Wert eines parameterClass Attributs ist ein voll qualifizierter Klassenname (einschließlich Package). Zwar ist das parameterClass Attribut optional, wird aber ausdrücklich empfohlen. Es dient dazu Eingabe-Parameter an das Statement zu begrenzen als auch die Performance des Frameworks zu optimieren. Wenn sie eine Parameter Map angeben, brauchen sie kein parameterClass Attribut mehr angeben. Wenn sie nur Objekt des Typs (also instanceof) "examples.domain.Product" zulassen möchten, können sie das so tun:

```
<insert id="statementName"  parameterClass=" examples.domain.Product">
        insert into PRODUCT values (#id#, #description#, #price#)
</insert>
```

WICHTIG: Obwohl optional aus Gründen der Rücklwärtskompatibiltät, empfehlen wir hochgradig immer eine parameter class anzugeben (es sein denn natürlich es gibt keine Parameter). So erhalten sie eine bessere Performance, da das Framework gewisse Optimierungen durchführen kann wenn es den Typen im Voraus bereits kennt.

Ohne die Angabe einer parameterClass, kann jede JavaBean mit entsprechen Properties (get/set Methoden) verwendet werden, was in manchen Situationen hilfreich sein kann.

parameterMap

Der Wert eines Parameter Map Attributs ist der Name eines definierten parameterMap Elements (siehe unten). Das parameterMap Attribut wird allerdings selten benutzt. Stattdessen wird häufiger das parameterClass Attribute (oben) oder inline Parameter (unten) verwendet. Dieses ist aber ein guter Ansatz wenn XML Reinheit und Konsistenz ihre Anliegen ist, oder wenn sie eine mehr beschreibende parameterMap (z.B. für stored procedures) möchten.

Achtung! Dynamische Mapped Statements (siehe unten) unterstützen ausschließlich inline Parameter und funktionieren nicht mit Parameter Maps.

Mit einer Parameter Map definieren sie eine geordnete Liste von Parametern, die Zeichen im JDBC PreparedStatement darstellen und abbilden sollen. Zum Beispiel:

```
<parameterMap id="insert-product-param" class="com.domain.Product">
        <parameter property="id"/>
        <parameter property="description"/>
</parameterMap>

<insert id="insertProduct"  parameterMap="insert-product-param">
        insert into PRODUCT (PRD_ID, PRD_DESCRIPTION) values (?,?)
</insert>
```

Das obige Beispiel beschreibt zwei Parameter die in dieser Reihenfolge die Angaben ("?") im SQL Statement darstellen. Das erste "?" wird somit durch die "id" Eigenschaft und das zweite durch die description" Eigenschaft ersetzt. Parameter Maps und ihre Optionen werden später im Dokument näher beschrieben.

Kurzer Blick auf Inline Parameter

Obwohl Details noch fehlen, hier schon einmal eine kurze Einführung in inline Parameter. Inline Parameter werde innerhalb eines Mapped Statement verwendet. Zum Beispiel:

```
<insert id="insertProduct" >
        insert into PRODUCT (PRD_ID, PRD_DESCRIPTION)
        values (#id#, #description#)
</insert>
```

In diesem Beispiel sind *#id#* und *#description#* inline Parameter. Jeder Parameter ist eine JavaBean Eigenschaft, die benutzt wird, um das Statement zu definieren.. Im Beispiel mit der Product Klasse gab es die Eigenschaften *id* and *description.* Diese werden verwendet um im Statement deren Werte einzusetzen. Wenn also zu einem Statement ein Product mit *id=5* und *description=”dog”*angegeben wird, wird letztendlich folgender SQL-Befehl ausgeführt:

```
insert into PRODUCT (PRD_ID, PRD_DESCRIPTION)
values (5, ‘dog’)
```

resultClass

Der Wert eines resultClass Attributs ist ein voll qualifizierter Klassenname (einschließlich Package). Mit dem resultClass Attribut können JDBC ResultSets (aus den ResultSetMetaData) automatisch auf Klassen abgebildet werden. Sobald Eigenschaft einer Klasse und Spalte eines ResultSet sich entsprechen wird die Eigenschaft mit dem entsprechenden Spaltenwert gefüllt. das macht query Mapped Statements kurz und bündig! Zum Beispiel:

```
<select id="getPerson" parameterClass=”int” resultClass="examples.domain.Person">
        SELECT
                PER_ID                  as id,
                PER_FIRST_NAME          as firstName,
                PER_LAST_NAME           as lastName,
                PER_BIRTH_DATE          as birthDate,
                PER_WEIGHT_KG           as weightInKilograms,
                PER_HEIGHT_M as heightInMeters
        FROM PERSON
        WHERE PER_ID = #value#
</select>
```

Nehmen wir an, die Person Klasse habe die Eigenschaften *id, firstName, lastName, birthDate, weightInKilograms* und *heightInMeters*. Jede stimmt mit den Spaltenpseudonymen im SQL select-Statement überein (durch das “as” Schlüsselwort – einem Standard-SQL Merkmal). Spaltenpseudonyme sind nur notwendig, wenn Datenbankspaltennamen nicht mit Eigenschaftsnamen übereinstimmen, dies tun sie allerdings nur selten. Bei der Ausführung wird ein Person Objekt instantiiert und die Werte aus dem Result Set werden auf Eigenschaften mit entsprechenden Spaltennamen abgebildet..

Wie bereits bemerkt gibt es einige Einschränkungen beim automatischen Mapping durch eine resultClass. Es gibt keinen Weg den Typen einer Ausgabespalte zu spezifizieren (sofern notwendig) und es gibt auch keinen Weg, abhängige Objekte automatisch zu laden. Zudem gibt es einen kleinen Performance-Nachteil weil auf die ResultSetMetaData zugegriffen werden muss. Diese Beschränkungen können alle dadurch vermieden werden, indem sie explizit eine Result Map definieren. Result Maps werden noch genauer behandelt.

resultMap

Das resultMap Element ist eines der am häufigsten eingesetzten Attribute und es ist daher wichtig es richtig zu verstehen. Der Wert eines resultMap Attributs ist der Name eines definierten Result Map Elements. Mit einem resultMap Attribut können sie kontrollieren wie Daten aus einem Result Set extrahiert und auf Eigenschaften von Datenbankspalten abgebildet werden sollen. Anders als der Ansatz des automatischen Mappings mit dem resultClass Attribut (oben), können sie mit einer Result Map den Spaltentyp, einen Ersatz für null-Werte oder die Abbildung komplexer Objektgraphen (einschließlich andere JavaBeans, Collections oder primitive Typwrapper) vornehmen.

Zwar folgt noch eine vollständige Beschreibung, aber das folgende Beispiel soll Result Map im Zusammenhang eines Mapped Statement verdeutlichen.

```
<resultMap id=”get-product-result” class=”com.ibatis.example.Product”>
        <result property=”id” column=”PRD_ID”/>
        <result property=”description” column=”PRD_DESCRIPTION”/>
```

```
</resultMap>

<select id="getProduct" resultMap="get-product-result">
        select * from PRODUCT
</select>
```

In diesem Beispiel wird das ResultSet von der SQL Abfrage auf Instanzen der Product Klasse abgebildet, Die Result Map definiert, dass die "id" Eigenschaft durch die "PRD_ID" Spalte und die "description" Eigenschaft durch die "PRD_DESCRIPTION" Spalte gefüllt wird. Beachten sie, das "select *" verwendet wird. Sie müssen nicht alle Spalten aus einem ResultSet abbilden.

cacheModel

Der Wert eines cacheModel Attributs ist der Name eines definierten cacheModel Elements (siehe unten). Ein cacheModel beschreibt einen Cache für Abfrage-Statements. Jede Abfrage kann ihr eigenes cacheModel definieren oder ein bestehendes verwenden.. Zwar folgt noch eine vollständige Beschreibung, aber das folgende Beispiel soll cacheModel im Zusammenhang eines Mapped Statement verdeutlichen.

```
<cacheModel id="product-cache" type="LRU">
        <flushInterval hours="24"/>
        <flushOnExecute statement="insertProduct"/>
        <flushOnExecute statement="updateProduct"/>
        <flushOnExecute statement="deleteProduct"/>
        <property name="size" value="1000" />
</cacheModel>

<select id="getProductList"  parameterClass="int"  cacheModel="product-cache">
        select * from PRODUCT where PRD_CAT_ID = #value#
</select>
```

Hier wird ein Cache definiert der schwache (WEAK) Referenzen auf products enthält. Die Einträge werden automatisch alle 24 Stunden verworfen oder wenn die verknüpften update Statements ausgeführt werden.

xmlResultName

Beim Mapping von Resultaten direkt in ein XML Dokument, gibt xmlResultName den Namen des root Elementes im XML Dokument an. Zum Beispiel:

```
<select id="getPerson" parameterClass="int"  resultClass="xml" xmlResultName="person">
        SELECT
                PER_ID                  as id,
                PER_FIRST_NAME          as firstName,
                PER_LAST_NAME           as lastName,
                PER_BIRTH_DATE          as birthDate,
                PER_WEIGHT_KG           as weightInKilograms,
                PER_HEIGHT_M as heightInMeters
        FROM PERSON
        WHERE PER_ID = #value#
</select>
```

Obiges Statement würden ein XML Objekt folgender Struktur erzeugen:

```
<person>
  <id>1</id>
  <firstName>Clinton</firstName>
  <lastName>Begin</lastName>
  <birthDate>1900-01-01</birthDate>
  <weightInKilograms>89</weightInKilograms>
  <heightInMeters>1.77</heightInMeters>
</person>
```

remapResults

Das remapResults Attribut ist bei <statement>, <select>, und <procedure> Mapped Statements anwendbar. Es ist ein optionales Attribut und seine Voreinstellung ist false.

Das remapResults Attribut sollte auf true gesetzt wertem. wenn eine Abfrage eine variable Anzahl von Spalten liefern kann. Nehmen wir beispielsweise folgende Abfragen:

```
SELECT  $fieldList$
FROM    table
```

In diesem Beispiel ist die Anzahl der Spalten variabel, aber der Tabellenname ist konstant.

```
SELECT  *
FROM    $sometable$
```

In diesem Beispiel kann die Tabelle variabel sein. Weil ein * im select gemacht wird, können dann auch die Spaltennamen sich ändern. Dynamische Element könnten ebenfalls die Spaltennamen von Fall zu Fall ändern.

Da es eine nicht triviale Aufgabe ist die resultset Metadaten zu überprüfen, merkt sich iBATIS diese Einstellungen der letzten Abfrage.. Diese kann in Situationen, wie oben beschrieben, zu Problemen führen. Daher gibt es die Möglichkeit die Metadata Überprüfung bei jeder Abfrage zu erzwingen.

Können sich also Spaltennamen ändern, setzen sie remapResults auf true, sonst lassen sie remapResults auf false um den Overhead bei der Prüfung der Metdadaten zu vermeiden.

resultSetType

Um den resultSetType eines SQL Statement zu setzen. Mögliche Werte:

- **FORWARD_ONLY**: der Cursor darf sich nur vorwärts bewegen
- **SCROLL_INSENSITIVE**: der Cursor kann scrollend navigiert werden ist aber üblicherweise nicht sensitiv auf Änderungen von anderen
- **SCROLL_SENSITIVE**: der Cursor kann scrollend navigiert werden ist aber üblicherweise sensitiv auf Änderungen von anderen

Beachten sie, das die Angaben von resultSetType normalerweise nicht notwendig sind und verschiedene JDBC Treiber unterschiedlich reagieren können. (beispielsweise unterstützt Oracle **SCROLL_SENSITIVE nicht**).

fetchSize

Setzt die fetchSize eines SQL Befehls. Dieses gibt dem JDBC Treiber einen Hinweis, damit dieser eventuell Daten vorzeitig lädt um Anfragen an den Datenbankserver zu minimieren.

timeout (iBATIS ab Version 2.2.0)

Setzt das JDBC Abfrage-timeout für dieses Statement. Jeder hier spezifizierte Wert überschreibt den globalen Wert von “defaultStatementTimeout” in der SQLMapConfig.xml Datei. Haben sie ein globales Timeout angegeben und möchten kein spezifisches Timeout für einen bestimmten Befehl, setzen sie es auf 0. Der angegebene Wert ist die Zeit in Sekunden, die der Treiber auf ein Ergebnis warten darf. Beachten sie, das nicht alle Treiber diese Angabe unterstützen

Parameter Maps und Inline Parameter

Wie sie gesehen haben, sind Parameter Map dafür verantwortlich, JavaBean Eigenschaften auf Parameter eines SQL-Befehls abzubilden. Obwohl Parameter Maps in ihrer externen Form sehr selten vorkommen, hilft ihr Verständnis doch sehr bei inline Parametern. Inline Parameter werden direkt im Anschluss erläutert. Wir erwähnen die Informationen hier der Vollständigkeit halber, empfehlen aber den Einsatz von inline Parameter statt Parameter Maps.

```
<parameterMap id="parameterMapName" [class="com.domain.Product"]>
        <parameter property ="propertyName" [jdbcType="VARCHAR"] [javaType="string"]
                                            [nullValue="-9999"]
                                            [typeName="{REF or user-defined type}"]
                                            [resultMap=someResultMap]
                                            [mode=IN|OUT|INOUT]
                                            [typeHandler=someTypeHandler]
                                            [numericScale=2]/>
        <parameter …… />
        <parameter …… />
</parameterMap>
```

Teile in **[Klammern]** sind optional. Das Parameter Map selbst benötigt nur ein ***id*** *A*ttribut. Dieses dient als Bezeichner, damit andere Statements es referenzieren können. Das ***class*** Attribut ist zwar optional aber hochgradig empfohlen. Ähnlich wie beim parameterClass Attribut eines Statements erlaubt das *class* Attribut dem Framework Parameter zu validieren und zu optimieren.

<parameter> Elemente

Die Parameter Map kann eine Zahl von parameter Abbildungen enthalten. Hiermit werden Parameter auf ein Statement abgebildet. Die nächsten Abschnitte beschreiben die Attribute des *property* Element:

property

Das property Attribut einer Parameter Map ist der Name einer JavaBean Eigenschaft (get Methode) um den Wert aus einem Parameter Objekt für ein Mapped Statement zu bekommen. Der Name kann mehrmals vorkommen, abhängig wie oft es im Statement gebraucht wird (z.B. die gleiche Eigenschaft in der where-Klausel und gleichzeitig in der set Klausel bei einem SQL update Statement).

jdbcType

Das *jdbcType* Attribut gibt explizit den Typen einer Datenbankspalte an, die durch diese Eigenschaft gesetzt wird. Einige JDBC Treiber können nicht den Typ einer Spalte bei bestimmten Operationen identifizieren ohne ihn explizit dem Treiber mitzuteilen. Ein schönen Beispiel hierfür ist die PreparedStatement.setNull(`int parameterIndex, int sqlType`) Methode. Diese Methode macht es erforderlich den Typen zu spezifizieren. Einige Treiber nutzen einen impliziten Typ indem sie Types.OTHER oder Types.NULL senden. Aber dieses Verhalten ist leider nicht konsistent und anderen muss der exakte Typ angegeben werden. Für solche Situationen erlaubt die Data Mapper API den *jdbcType* Attribute in einem parameterMap property-Element anzugeben.

Dieses Attribut ist üblicherweise notwendig bei Spalten, die Null-Werte enthalten können. Ein weiterer Grund für das Typ-Attribut ist bei der genauen Angabe von Datum-Typen.. Wo Java nur ein Date Wertobjekt besitzt (java.util.Date), haben SQL Datenbanken mehrere – meistens wenigstens 3 verschiedene. Daher können sie explizit den Typen der Spalte angeben (DATE oder DATETIME (...).

Das *jdbcType* Attribut kann jede Zeichenfolge aus der JDBC Types Klasse sein. Zwar kann es jede Zeichenfolge aus der Klasse sein, aber nicht alle werden auch wirklich unterstützt. (z.B.. blobs). Ein Abschnitt beschreibt den Umgang mit diesem Typ.

Achtung! Die meisten Treiber benötigen die Typangabe nur für Spalten, in denen Null-Werte gespeichert werden können.

Achtung! Mit dem Oracle Treiber erhalten sie ein "Invalid column type" Fehler wenn sie versuchen eine Spalte zu null zu setzen ohne ihren Typ anzugeben.

javaType

Das *javaType* Attribut wird benutzt um explizit den Java Property Typ eines Parameter zu setzen. Normalerweise kann dies über Reflection der JavaBean Eigenschaft abgeleitet werden, aber bei bestimmten Abbildungen wie Map oder XML kann der Typ dem Framework mitgeteilt werden. Ist javaType nicht gesetzt und das Framework kann es nicht bestimmen wird Object angenommen

typeName

Das *typeName* Attribut spezifiziert einen REF Typ oder einen benutzerdefinierten Typ.

Aus der javadoc-Dolumentation:

The typeName attribute... "should be used for a user-defined or `REF` output parameter. Examples of user-defined types include: `STRUCT`, `DISTINCT`, `JAVA_OBJECT`, and named array types. ... For a user-defined parameter, the fully-qualified SQL type name of the parameter should also be given, while a `REF` parameter requires that the fully-qualified type name of the referenced type be given. A JDBC driver that does not need the type code and type name information may ignore it. To be portable, however, applications should always provide these values for user-defined and `REF` parameters. Although it is intended for user-defined and `REF` parameters, this *attribute* may be used to register a parameter of any JDBC type. If the parameter does not have a user-defined or `REF` type, the *typeName* parameter is ignored."

* *italicized* words were substituted to put the explanation in context of this documentation

nullValue

Das nullValue Attribut kann auf jeden gültigen Wert des Eigenschaftstypen gesetzt werden. Dieses null Attribut kann abgehende Werte durch einen null Wert ersetzen.. Dies bedeutet, wird dieser Wert erkannt wird stattdessen eine NULL in die Datenbank geschrieben Dies erlaubt es ihnen einen speziellen, magischen Wert als Null in ihrer Applikation zu verwenden, wenn der Typ null-Werte nicht zulässt (z.B. . int, double, float ...). Wenn die Eigenschaft einen entsprechenden Wert enthält (z.B.. –9999), wird stattdessen eine NULL in die Datenbank geschrieben.

resultMap

Geben sie ein *resultMap* Element an, wenn sie eine Instanz vom Typ java.sql.ResultSet als Output Parameter einer stored procedure erwarten. Dies erlaubt iBATIS das normale Result Set Mapping zu Objekt Mapping durchzuführen.

mode

Das *mode* Attribut gibt den Modus für stored procedure Parameter an. Gültige Werte sind IN, OUT, oder INOUT.

typeHandler

Das *typeHandler* Attribut spezifiziert einen benutzerdefinierten Typhandler der statt des iBATIS Typ-Systems genommen wird. Der Wert ist ein voll qualifizierter Klassenname (einschließlich Package), die entweder dasInterface `com.ibatis.sqlmap.engine.type.TypeHandler` oder `com.ibatis.sqlmap.client.extensions.TypeHandlerCallback` implementieren muss.

Dieser Wert überschreibt jeden global definierten Typhandler für dieses Eigenschaft. Mehr Details hierzu folgen.

numericScale

(numericScale ist ab iBATIS Version 2.2.0 verfügbar)

Das *numericScale* Attribut wird benutzt um den Maßstab (Ziffern rechts vom Dezimalpunkt) für NUMERIC oder DECIMAL stored procedure Output-Parameter festzulegen. Wenn sie OUT oder INOUT für das *mode* Attribut angeben und der *jdbcType* ist DECIMAL oder NUMERIC, dann sollten sie auch einen Wert für *numericScale* angeben. Der Wert muss eine Ganzzahl größer oder gleich Null sein.

Ein <parameterMap> Beispiel

Ein Beispiel für eine Parameter Map die alle Möglichkeiten darstellt:

```
<parameterMap   id="insert-product-param" class="com.domain.Product">
        <parameter property="id" jdbcType="NUMERIC" javaType="int" nullValue="-9999999"/>
        <parameter property="description" jdbcType="VARCHAR" nullValue="NO_ENTRY"/>
</parameterMap>

<insert   id="insertProduct"  parameterMap="insert-product-param">
        insert into PRODUCT (PRD_ID, PRD_DESCRIPTION) values (?,?)
</insert>
```

In diesem Beispiel werden die JavaBean Eigenschaften *id* and *description* auf das Mapped Statemnent *insertProduct* genau in der angegebenen Reihenfolge angewendet. Also wird *id* der erste Parameter (?) und *description* der zweite. Würde die Reihenfolge vertauscht, sähe das XML aus wie folgt:

```
<parameterMap id="insert-product-param" class="com.domain.Product">
        <parameter property="description" />
        <parameter property="id"/>
</parameterMap>

<insert id="insertProduct"  parameterMap="insert-product-param">
        insert into PRODUCT (PRD_DESCRIPTION, PRD_ID) values (?,?)
</insert>
```

Achtung! Parameter Map Namen sind immer lokal zur SQL Map XML Datei in der sie definiert sind. Sie können andere Parameter Map in einer anderen SQL Map XML Datei referenzieren, indem sie der *id* der Parameter Map die *id* der anderen SQL Map voranstellen (im <sqlMap> Wurzel-Tag). Um beispielsweise obige Parameter-Map aus einer anderen Datei anzusprechen, lautet der vollständige Name "Product.insert-product-param".

Inline Parameter Maps

Zwar ist das Ganze sehr interessant, aber auch sehr langatmig zu beschreiben. Inline Parameter Maps sind kürzer, selbst erklärender und wir raten dazu, dieses Verfahren statt expliziter Parameter Maps zu verwenden. Mit inline Parameter Maps schreiben sie die Parameter Definitionen inline in die SQL. Voreingestellt werden alle Mapped Statement ohne explizite Parameter Map nach inline Parameter durchsucht. Oberes Beispiel (product), sähe mit inline Parameter Map folgendermaßen aus:

```
<insert id="insertProduct" parameterClass="com.domain.Product">
          insert into PRODUCT (PRD_ID, PRD_DESCRIPTION)
          values (#id#, #description#)
</insert>
```

Typen in inline Parametern können so definiert werden:

```
<insert id="insertProduct" parameterClass="com.domain.Product">
        insert into PRODUCT (PRD_ID, PRD_DESCRIPTION)
        values (#id:NUMERIC#, #description:VARCHAR#)
</insert>
```

Typen und null-Werte Ersetzung können folgendermaßen mit inline Parametern realisiert werden:

```
<insert id="insertProduct" parameterClass="com.domain.Product">
        insert into PRODUCT (PRD_ID, PRD_DESCRIPTION)
        values (#id:NUMERIC:-999999#, #description:VARCHAR:NO_ENTRY#)
</insert>
```

Achtung! **Mit** inline Parametern können sie keinen null-Werte Ersatz angeben ohne den Typ zu spezifizieren. Sie müssen beide in der richtigen Reihenfolge angeben..

Achtung! Wenn null-Werte völlig transparent sein sollen, müssen sie Ersatzwerte auch bei Result Maps angeben.

Achtung! Wenn sie eine Vielzahl von Typ-Deskriptoren und null Ersatzwerte benötigen, sollten sie über den Einsatz einer externen Parameter Map für saubere Darstellungen nachdenken.

Inline Parameter Map Syntax

iBATIS bietet zwei Arten der Syntax von inline Parameter Maps an– eine einfache Syntax, und eine fortgeschrittenere vollständigere Syntax..

Die einfache Syntax:

#propertyName# - OR -
#propertyName:jdbcType# - OR -
#propertyName:jdbcType:nullValue#

Beispiele für diese Syntax sehen sie oben. Das *propertyName* Element ist der Name einer Eigenschaft im Parameter Objekt (oder das Parameter Objekt selbst als einfacher Wert zum Beispiel String, Integer, ...). Das *jdbcType* Element beschreibt den JDBC Typen eines Parameters. Der Wert kann einer aus der Liste aus `java.sql.Types` (VARCHAR, INTEGER, ...) sein. Generell wird dieses Element benötigt, wenn der Wert NULL sein kann oder um festzulegen, DATE oder TIME Felder zu nutzen (statt TIMESTAMP Feldern). Das *nullValue* Element spezifiziert einen NULL Ersatzwert wie oben bereits beschrieben. Sie können *nullValue* Werte nur angeben wenn sie auch den *jdbcType* definieren.

Diese Syntax reicht für die meisten Situationen völlig aus. Die erweiterte Form benötigen sie nur, wenn sie fortgeschrittene Optionen verwenden oder die Parameter Map formaler (z.B. beim Aufruf einer stored procedure) beschreiben möchten.

Die Syntax der erweiterten Form ist:

#propertyName,javaType=?,jdbcType=?,mode=?,nullValue=?,handler=?,numericScale=?#

Wobei sie für "?" einen geeigneten Wert für das Attribut einsetzen.

Mit der erweiterten Syntax können sie praktisch alle Werte, die sie auch formal definieren können, angeben, Das *propertyName* Element ist notwendig, alle anderen Angaben sind optional. Die Reihenfolge der Werte ist beliebig, außer das das *propertyName* Element als Erstes definiert werden muss. Zulässiges Werte für die einzelnen Attribute sind die gleichen wie bei der formalen Parameter Map. Beachten sie dabei, das beim *handler* Attribut ein Alias benutzt werden kann, sofern der Alias zuvor registriert wurde. Ein Beispiel bei dem eine stored procedure aufgerufen wird:

```
<procedure id="callProcedure" parameterClass="com.mydomain.MyParameter">
        {call MyProcedure
        (#parm1,jdbcType=INTEGER,mode=IN#, #parm2,jdbcType=INTEGER,mode=IN#,
         #parm3,jdbcType=DECIMAL,mode=OUT,numericScale=2#)}
</procedure>
```

Primitive Type Parameter

Sie müssen nicht für jeden Parameter gleich eine eigene JavaBean schreiben.. In manchen Fällen können sie stattdessen ein primitives Typ-Wrapper Objekt (String, Integer, Date etc.) als Parameter direkt nehmen. Beispielsweise:

```
<select id="insertProduct" parameter="java.lang.Integer">
        select * from PRODUCT where PRD_ID = #value#
</select>
```

Angenommen PRD_ID sei ein *numerischer* Typ. Dann kann beim Aufruf ein java.lang.Integer als Parameter Objekt angegeben werden. Die **#value#** Angabe wird beim Aufruf durch den Wert der Integer Instanz ersetzt. Der Name **"value"** ist dabei nur ein Platzhalter und kann beliebig sein.. Result Maps (siehe unten) unterstützen derartige primitive Typen bei Resultaten genauso.

Für primitive Typen existieren Alias-Namen für einen ausagekräftigeren Code. Beispielswiese kann "int" an Stelle von "java.lang.Integer" geschrieben werden. Eine vollständige Liste finden sie unter: "Unterstützte Typen von Parameter Maps und Result Maps".

Map Type Parameter

Sind sie mal nicht in der Lage oder ist es zu aufwändig eine JavaBean zu schreiben oder reicht ein primitive Typ-Wrapper nicht aus (z.B. weil sie mehrere Parameter benötigen), können sie auch eine Map (z.B. HashMap, TreeMap) als Parameter Objekt benutzen. Zum Beispiel:

```
<select  id="insertProduct"  parameterClass="java.util.Map">
        select * from PRODUCT
        where PRD_CAT_ID = #catId#
        and PRD_CODE = #code#
</select>
```

Beachten sie dass es keine gravierenden Unterschiede im Mapped Statement gibt! Wird das obige Beispiel mit einer Map Instanz aufgerufen, muss die Map die Schlüssel "catId" und "code" enthalten. Die Werte dieser Schlüssel müssen vom entsprechenden Typ (wie Integer und String (für dieses Beispiel)) sein. Result Maps (siehe unten erlauben auch Map Typen als Resultate.

Für Map Typen existieren ebenfalls Alias-Namen für ausagekräftigeren Code . Beispielsweise kann "map" anstelle von "java.util.Map"geschrieben werden. Eine vollständige Liste finden sie unter: "Unterstütze Typen von Parameter Maps und Result Maps".

Substitution Strings

iBATIS verwendet immer JDBC vorbereitete Anweisungen (prepared statements) zur Ausführung von SQL. Ein JDBC prepared statement unterstützt Parameter durch "markierte Parameter". Sowohl Parameter Maps als auch inline Parameter veranlassen iBATIS, SQL mit entsprechenden Parameter Markierungen zu verwenden. Zum Beispiel bei diesem Statement:

```
select * from PRODUCT where PRD_ID = #value#
```

iBATIS erzeugt ein prepared statement mit dieser SQL Zeichenkette:

```
select * from PRODUCT where PRD_ID = ?
```

Datenbanken erlauben Parameter bei fast allen Teilen im SQL Statement. Leider gibt es hier Ausnahmen. Dieses Statement zum Beispiel ist nicht erlaubt:

```
select * from ?
```

Die Datenbank kann dieses Statement nicht kompilieren, da nicht bekannt ist, um welche Tabelle es sich hier handelt.. Wenn sie also Statement wie folgt verwenden:

```
select * from #tableName#
```

werden sie typischerweise eine SQLException bekommen.

Um einige der Schwierigkeiten zu vermeiden, ermöglicht iBATIS Zeichenketten zu ersetzen bevor das Statement vorbereitet wird. Sie können dies verwenden um dynamische SQL Befehle zu erzeugen. Ein Beispiel für eine solchen Fall:

```
select * from $tableName$
```

In dieser Form ersetzt iBATIS die “tableName” Eigenschaft im SQL bevor das Statement vorbereitet wird.. Dadurch kann jede Zeichenkette in einem SQL-Befehl ersetzt werden.

Wichtiger Hinweis 1: Dies geht nur mit Zeichenketten. Komplexe andere Typen wie zum Beispiel Date oder Timestamp funktionieren damit nicht.

Wichtiger Hinweis 2: Wenn sie dies verwenden um den Tabellen- oder Spaltennamen zu ändern müssen sie auch remapResults=“true” angeben.

Result Maps

Result Maps sind ein extrem wichtiger Bestandteil von Data Mapper. Mit Result Maps werden JavaBean Eigenschaften auf Spalten eines ResultSet bei der Ausführung eines Abfrage-Statements abgebildet.. Die Struktur einer Result Map ist folgende:

```
<resultMap  id="resultMapName"  class="some.domain.Class"
     [extends="parent-resultMap"]
     [groupBy="some property list"]>
          <result property="propertyName"  column="COLUMN_NAME"
                    [columnIndex="1"]  [javaType="int"]  [jdbcType="NUMERIC"]
                    [nullValue="-999999"] [select="someOtherStatement"]
                    [resultMap="someOtherResultMap"]
                    [typeHandler="com.mydomain.MyTypehandler"]
                    />
          <result ……/>
          <result ……/>
          <result ……/>
</resultMap>
```

Teile in **[Klammer]** sind optional. Die Result Map selbst besitzt ein ***id*** Attribut über das andere Statements dieses referenzieren können. Result Map hat ein ***class*** Attribut das den voll qualifzierten Klassenname (einschließlich Package) oder einen Typ Alias enthält. Diese Klasse wird mit den Angaben aus den Abbildungen dieser Result Map instantiiert und gefüllt. Das ***extends*** Attribute kann optional eine andere Result Map angeben auf dem diese basiert. Diese ist ähnlich der Vererbung in Java, alle Eigenschaften der Ober-Result Map (Vater) sind auch Bestandteil der Kind Result Map. Eigenschaften der Vater Result Map werden immer vor Eigenschaften der Kind Result Map hinzugefügt und die Vater Result Map muss vor der Kind Result Map definiert sein. Die Klassen für Vater/Kind müssen nicht gleich sein oder in irgendeiner Form voneinander abhängen (es kann sich also um irgendwelche Klassen handeln).

Das *resultMap* Element unterstützt das Attribut *groupBy*. Hiermit werden in einer Liste von Eigenschaften angegeben, die Zeilen im Result Set eindeutig kennzeichnet. Zeilen mit gleichen Werten mit diesen Eigenschaften führen dazu, das nur ein Resultat Objekt erzeugt wird. Verwenden sie *groupBy* in Verbindung mit geschachtelten Result Maps um das N+1 Abfrage-Problem zu lösen (siehe unten stehende Erläuterung).

Eine Result Map kann eine Vielzahl von Abbildungen von JavaBean Eigenschaften auf Spalten des Result Set enthalten. Diese Abbildungen werden in der Reihenfolge in der sie im Dokument stehen, angewendet. Die angegebene Klasse muss der JavaBean Konvention genügen und entsprechende getter/setter Methoden für jede Eigenschaft, Map oder XML besitzen.

Achtung! Spalten werden genau in der angegeben Reihenfolge gelesen (dieses ist ganz bequem mit einigen schlecht geschriebenen JDBC Treibern).

Die nächsten Abschnitte erläutern die Attribute des *result* Elements:

property

Das *property* Attribut einer Result Map Eigenschaft ist der Name einer JavaBean Eigenschaft (get method) von einem Resultat Objekt. Der Name kann mehrmals, abhängig von der Anzahl im Resultat, vorkommen.

column

Das column Attribut ist der Name einer Spalte aus dem ResultSet. Der Wert dieser Spalte wird in die Eigenschaft der JavaBean gefüllt.

columnIndex

Durch die Angabe des columnIndex Attributs, welches den Index der Spalte im ResultSet angibt, kann ein wenig die Performance gesteigert werden um die Eigenschaft der JavaBean zu füllen. Das werden sie zu 99% ihrer Applikation nicht benötigen und geht zu Lasten der Wartbarkeit und Lesbarkeit. Bei einigen JDBC Treibern geht dies nicht mit einer Performance-Steigerung einher während andere dramatisch davon profitieren

jdbcType

Mit dem jdbcType Attribut geben sie explizit den Typ der Datenbankspalte im ResultSet an. Zwar haben Result Maps nicht mit den gleichen Schwierigkeiten mit null Werten zu kämpfen, trotzdem kann es sinnvoll sein, diese für gewissen Typen wie zum Beispiel Date Eigenschaften anzugeben. Da Java nur einen Date Typ enthält, während SQL Datenbanken mehrere besitzen (üblicherweise mindestens 3), kann es sinnvoll und wichtig sein, diesen bei einigen Fällen anzugeben. Ähnlich kann der String Typ durch VARCHAR, CHAR oder CLOB Spalten gefüllt werden, aber die Angabe mag auch hier in einigen Fällen sinnvoll sein (Treiber abhängig).

javaType

Mit dem *javaType* Attribut geben sie explizit den Typ der Java Eigenschaft an. Dies kann normalerweise durch Reflection der JavaBean Eigenschaft abgeleitet werden, aber gewisse Abbildungen wie Map oder XML können diese Informationen nicht dem Framework übergeben. Ist javaType nicht gesetzt und das Framework kann es nicht bestimmen, wird Object als Typ angenommen

nullValue

Mit dem *nullValue* Attribut geben sie einen Ersatzwert an, wenn NULL Werte aus der Datenbank gelesen werden. Steht also NULL im ResultSet, wird die JavaBean Eigenschaft auf den angegeben Wert (des nullValue Attributs) statt NULL gesetzt. Es kann jeder gültige Wert für den Typ der Eigenschaft sein.

Wenn in der Datenbank in einer Spalte NULL-Werte auftreten können und sie in ihrer Applikation diesen Zustand mit einem konstanten Wert abbilden möchten, können sie dies in der Result Map wie folgt tun:

```
<resultMap id="get-product-result" class="com.ibatis.example.Product">
        <result property="id" column="PRD_ID"/>
        <result property="description" column="PRD_DESCRIPTION"/>
        <result property="subCode" column="PRD_SUB_CODE" nullValue="-999"/>
</resultMap>
```

Wenn die Datenbankspalte PRD_SUB_CODE einen NULL-Wert enthält, wird die subCode Eigenschaft auf den Wert 999 gesetzt. Somit können sie in ihrer Java Klasse auch einen primitiven Typ verwenden.. Beachten sie, das sie, damit dies bei Abfragen (queries) ebenso funktioniert wie bei updates/inserts, müssen sie auch einen nullValue in der Parameter Map angeben (oben bereits beschrieben).

select

Mit dem *select* Attribut können sie Beziehungen zwischen Objekten angeben und damit automatisch komplexe Objektgraphen laden (z.B. bei benutzerdefinierten Typen). Die Angabe referenziert ein weiteres Mapped Statement, das benutzt wird um diese Eigenschaft zu füllen. Der Datenbankspalte wird dem aufgerufenen Mapped Statement als Parameter übergeben. Bei der Spalte muss es sich daher um einen primitiven Typ handeln. Mehr Informationen zur Unterstützung von primitiven Typen und komplexe Abbildungen zwischen Eigenschaften/Beziehungen werden später genauer beschrieben.

resultMap

Mit dem *resultMap* Attribut können verschachtelte, wiederverwendbare Result Maps aufgebaut werden. Dies kann für 1:1 oder 1:N Beziehungen verwendet werden. Modellieren sie eine 1:N Beziehung, muss die zugehörige Eigenschaft eine Kollektion (List, Set, Collection, ….) sein und sie sollten mit dem *groupBy* Attribut im *resultMap* Element iBATIS angeben, wie es Zeilen zusammenfassen soll. Bei einer 1:1 Beziehung kann die zugehörige Eigenschaft irgendein Typ sein und das groupBy Attribute kann, muss aber nicht, angegeben werden. Das *groupBy* Attribut kann auch bei Joins, von denen einige 1:N andere aber 1:1 sind, angewendet werden.

typeHandler

Mit dem *typeHandler* Attribut geben sie einen benutzerdefinierten Typ-Handler an, der anstatt des iBATIS Typ-Systems, für diese Eigenschaft verwendet wird. Bei der Angabe sollte es sich um den voll qualifizierten Klassennamen handeln der entweder das Interface `com.ibatis.sqlmap.engine.type.TypeHandler` oder `com.ibatis.sqlmap.client.extensions.TypeHandlerCallback` implementiert. Eine derartige Angabe überschreibt jeden anderen global definierten Typ-Handler für diese Eigenschaft. Es folgt eine detailliertere Erläuterung von Benutzer-definierten Type Handler später im Dokument.

Implizite Result Maps

Sind ihre Anforderungen recht einfach und sie benötigen keine explizite definierte Rresult Map, gibt es auch die Möglichkeit die Result Map durch Angabe des resultClass Attributs zu definieren. Dieses erfordert aber, dass die Spaltennamen (oder Pseudonyme) mit den Namen der (schreibbaren) JavaBean Eigenschaften übereinstimmen. Für die obige Product Klasse könnten wir die Result Map auch implizit abbilden:

```
<select id="getProduct" resultClass="com.ibatis.example.Product">
        select
           PRD_ID as id,
           PRD_DESCRIPTION as description
        from PRODUCT
        where PRD_ID = #value#
</select>
```

Obiges Statement legt die resultClass fest und stellt über Spaltenpseudonyme sicher, das die Namen zwischen Datenbankspalte und JavaBean Eigenschaft übereinstimmen. Ist dies der Fall benötigen sie keine Result Map. Nachteile dieser einfachen Möglichkeit sind, dass sie nicht den Typ der Spalte oder einen Ersatzwert für null-Werte oder andere Eigenschafts-Attribute angeben können. Da vielen Datenbanken nicht zwischen Groß-/Kleinschreibung unterscheiden, sind implizite Result Maps dies auch nicht. Wenn es also in ihrer JavaBean zwei Eigenschaften mit den Namen firstName und firstname gibt, würden diese als gleich betrachtet und könnten nicht in einer impliziten Result Map (es wäre auch ein potenzielles Design-Problem der JavaBean Klasse) verwendet werden. Außerdem gibt es einen kleinen Performance-Nachteil mit der automatischen Abbildung durch eine resultClass. Der Zugriff auf die ResultSetMetaData kann bei schlechten JDBC Treibern sehr lange dauern.

Primitive Resultate (z.B. String, Integer, Boolean)

Neben JavaBean konformen Klassen, können Result Maps zusätzlich auch Java Type-Wrapper wie String, Integer, Boolean,..enthalten . Kollektionen von Objekts können durch die weiter unten beschriebene API (z.B. queryForList()) abgefragt werden. Primitive Typen werden genauso wie JavaBeans abgebildet. Es muss nur eine Sache beachtet werden. Ein primitiver Typ kann nur eine Eigenschaft besitzen und der Name ist bedeutungslos (üblicherweise "value" oder "val"). Wenn sie also zum Beispiel eine Liste der product Beschreibungen (Strings) statt der ganzen Product Klasse benötigen, sähe ihre Map so aus:

```
<resultMap id="get-product-result" class="java.lang.String">
        <result property="value" column="PRD_DESCRIPTION"/>
</resultMap>
```

Ein einfacheres Verfahren ist, einfach result class im Mapped Statement anzugeben. (beachten sie die Verwendung des Spaltenpseudonyms "value" durch das "as" Schlüsselwort):

```
<select  id="getProductCount" resultClass="java.lang.Integer">
        select count(1) as value
        from PRODUCT
</select>
```

Map Resultate

Result Maps können auch bequem Instanzen einer Map wie HashMap oder TreeMap füllen. Kollektionen derartiger Objekte (z.B. Listen von Maps) können durch unten beschrieben API abgefragt werden (siehe queryForList()). Map Typen werden exakt in gleicher Weise wie JavaBeans abgebildet. Aber, statt Eigenschaften von JavaBeans zu setzen, werden die Schlüssel der Map auf Werte der entsprechenden Spalten abgebildet.. Wenn sie somit Produkte schnell in eine Map laden möchten:

```
<resultMap id="get-product-result" class="java.util.HashMap">
        <result property="id" column="PRD_ID"/>
        <result property="code" column="PRD_CODE"/>
        <result property="description" column="PRD_DESCRIPTION"/>
        <result property="suggestedPrice" column="PRD_SUGGESTED_PRICE"/>
</resultMap>
```

In diesem Beispiel wird eine HashMap instantiiert und mit den Produkt-Daten gefüllt.. Die Eigenschaftsnamen (z.B.. "id") wären die Schlüssel der HashMap und die Werte wären die Werte der zugehörigen Spalten.

Dieses ginge natürlich durch eine implizite Result Map mit einem Map Typ. Zum Beispiel:

```
<select id="getProductCount" resultClass="java.util.HashMap">
        select * from PRODUCT
</select>
```

Das gäbe im Wesentlichen eine Map Darstellung vom zurückgelieferten ResultSet.

Komplexe Eigenschaften(oder Eigenschaften in benutzerdefinierten Klassen)

Komplexe Typen, also benutzerdefinierte Klassen, können ebenfalls automatisch gefüllt werden, indem man die Result Map Eigenschaft mit einem weiteren Mapped Statement verknüpft, welches weiß wie die entsprechenden Daten der Klasse geladen werden können. In der Datenbank wird dies häufig durch eine 1:1 Beziehung, oder einer 1:M Beziehung, in der die (benutzerdefinierte) Klasse die "many side" der Beziehung und die Eigenschaft selbst die "one side" der Beziehung darstellt. Betrachten wir folgendes Beispiel

```
<resultMap id="get-product-result" class="com.ibatis.example.Product">
        <result property="id" column="PRD_ID"/>
        <result property="description" column="PRD_DESCRIPTION"/>
        <result property="category" column="PRD_CAT_ID" select="getCategory"/>
</resultMap>

<resultMap id="get-category-result" class="com.ibatis.example.Category">
        <result property="id" column="CAT_ID"/>
        <result property="description" column="CAT_DESCRIPTION"/>
</resultMap>

<select id="getProduct" parameterClass="int" resultMap="get-product-result">
        select * from PRODUCT where PRD_ID = #value#
</select>

<select id="getCategory" parameterClass="int" resultMap="get-category-result">
        select * from CATEGORY where CAT_ID = #value#
</select>
```

In diesem Beispiel hat jede *Product*-Instanz die Eigenschaft *category* vom Typ *Category*. category ist ein komplexer Typ, da benutzerdefiniert. JDBC weiß also nicht wie es Klassen dieses Typs füllen kann. Wir verknüpfen diese Eigenschaft nun mit einem weiteren Mapped Statement um genügend Informationen

bereitzustellen damit SQL Map dieses bewerkstelligen kann.. Wird *getProduct* ausgeführt, ruft die *get-product-result* Result Map das Statement *getCategory* auf und übergibt den Wert der *PRD_CAT_ID* Spalte. Die *get-category-result* Result Map instantiiert ein Category Objekt und füllt es entsprechend. Das ganze Category-Objekt wird dann als category-Eigenschaft im Product-Objekt gesetzt.

Vermeiden von N+1 Selects (1:1)

Das Problem obiger Lösung ist, das immer zwei SQL-Befehle ausgeführt werden, möchte man ein Product laden (eins für das Product und eins für Category). Dieses Problem sieht nicht besonders kompliziert aus, wenn man nur ein einziges Product lädt, Möchte man aber zum Beispiel zehn Products laden, wird für jede Product Abfrage eine weitere für die category abgesetzt. Dieses ergibt dann insgesamt elf Abfragen: eine für die Liste der Products und eines für jedes der zurückgelieferten Products um die zugehörigen Category zu laden (N+1 oder in diesem Fall 10+1=11).

Die Lösung hier wäre einen Join mit verschachtelten Eigenschaft Abbildungen statt getrennter Select-Statements zu verwenden. Hier ist das gleiche Beispiel (Products und Categories):

```
<resultMap id="get-product-result" class="com.ibatis.example.Product">
        <result property="id" column="PRD_ID"/>
        <result property="description" column="PRD_DESCRIPTION"/>
        <result property="category.id" column="CAT_ID" />
        <result property="category.description" column="CAT_DESCRIPTION" />
</resultMap>

<select  id="getProduct"  parameterClass="int" resultMap="get-product-result">
        select *
        from PRODUCT, CATEGORY
        where PRD_CAT_ID=CAT_ID
        and PRD_ID = #value#
</select>
```

Ab iBATIS Version 2.2.0, können sie Result Maps in einer 1:1 Abfrage auch wiederverwenden statt die Spalten zu wiederholen. Dieses Beispiel sähe also wie folgt aus

```
<resultMap id="get-product-result" class="com.ibatis.example.Product">
        <result property="id" column="PRD_ID"/>
        <result property="description" column="PRD_DESCRIPTION"/>
        <result property="category" resultMap="get-category-result"  />
</resultMap>

<resultMap id="get-category-result" class="com.ibatis.example.Category">
        <result property="id" column="CAT_ID" />
        <result property="description" column="CAT_DESCRIPTION" />
</resultMap>

<select  id="getProduct"  parameterClass="int" resultMap="get-product-result">
        select *
        from PRODUCT, CATEGORY
        where PRD_CAT_ID=CAT_ID
        and PRD_ID = #value#
</select>
```

Lazy Loading gegenüber Joins (1:1)

Beachten sie, dass es nicht *immer* besser ist einen Join zu benutzen. Greifen sie beispielsweise nur selten auf das abhängige Objekt zu (also die *category* Eigenschaft der Product Kasse), mag es manchmal schneller sein, auf das unnötige Laden der category Eigenschaft zu verzichten. Dies gilt insbesondere bei Datenbank-Design, die einen outer join oder nicht indizierte Spalten beinhalten. In derartigen Situationen mag es besser sein, stattdessen den Sub-Select mit der lazy loading und bytecode Option (siehe SQL Map Konfigurations-

Einstellung) zu verwenden. Daumenregel ist: nehmen sie einen Join, wenn sie wahrscheinlicher auf die verbundene Eigenschaft zugreifen. Sonst nehmen sie diese Lösung nur wenn sie lazy loading nicht benutzen können.

Können sie sich nicht einfach zwischen den beiden Wegen entscheiden: Keine Panik! Sie können später sich anders entscheiden und müssen noch nicht einmal ihren Java Code anpassen. Obige beide Beispiele würden mit den gleichen Methodenaufrufen zum gleichen Objektgraphen führen. Die einzige Abwägung beträfe den Fall, wenn sie caching aktiviert haben. Im Fall einer separaten Abfrage(nicht der Join) kann es zur Rückgabe einer bereits gecachten Instanz führen. Aber dies wird häufig nicht zu einem Problem führen. (Ihre Applikation sollte auch nicht davon abhängig, dass sie die Gleichheit von Instanz-Objekte durch "==" beachten müssen).

Komplexe Kollektion Properties

Es ist möglich, auch Eigenschaften von Listen komplexer (benutzerdefinierter) Typen zu laden. In der Datenbank wird dies durch eine M:M Beziehung oder eine 1:M Beziehung, in der die Klasse mit der Liste die "one side" der Beziehung und die Objekte in der Liste die "many side" Beziehung darstellen. Um eine Liste von Objekten zu laden , ist keine Änderung der SQL-Befehle notwendig (siehe oben). Der einzige notwendige Unterschied, damit das SQL Map Framework eine derartige Eigenschaft im Business-Objekt laden kann, ist, dass die Eigenschaft vom Typ java.util.List oder java.util.Collection sein muss. Wenn also eine Category eine Liste von Product Instanzen besitzen kann, sähe die Abbildung wie folgt aus (nehmen wir an, Category besäße eine Eigenschaft "productList" vom Typ type java.util.List):

```
<resultMap id="get-category-result" class="com.ibatis.example.Category">
        <result property="id" column="CAT_ID"/>
        <result property="description" column="CAT_DESCRIPTION"/>
        <result property="productList" column="CAT_ID" select=" getProductsByCatId"/>
</resultMap>

<resultMap id="get-product-result" class="com.ibatis.example.Product">
        <result property="id" column="PRD_ID"/>
        <result property="description" column="PRD_DESCRIPTION"/>
</resultMap>

<select id="getCategory"  parameterClass="int" resultMap="get-category-result">
        select * from CATEGORY where CAT_ID = #value#
</select>

<select  id="getProductsByCatId"  parameterClass="int" resultMap="get-product-result">
        select * from PRODUCT where PRD_CAT_ID = #value#
</select>
```

Vermeiden von N+1 Selects (1:M und M:N)

Die Situation ist ähnlich zum oben beschriebenen 1:1 Problem, ist aber hier von noch größerem Belang, da sehr viel größere Datenmengen involviert sind. Das Problem dieser Lösung besteht darin, das sie für jede Category, die sie laden möchten, zwei SQL Statements gegen die Datenbank absenden (eines für die Category und eines für die Liste der Products). Möchte man aber zum Beispiel zehn Products laden, wird für jede Product Abfrage eine weitere für die Category abgesetzt. Dieses ergibt dann insgesamt elf Abfragen: eine für die Liste der Products und eines für jedes der zurückgelieferten Product um die zugehörigen Category zu laden (N+1 oder in diesem Fall 10+1=11).

1:N & M:N Lösung

iBATIS löst dieses N+1 Abfrage-Problem vollständig. Hier ein Beispiel:

```
<sqlMap namespace="ProductCategory">

<resultMap id="categoryResult" class="com.ibatis.example.Category" groupBy="id">
        <result property="id" column="CAT_ID"/>
        <result property="description" column="CAT_DESCRIPTION"/>
        <result property="productList" resultMap="ProductCategory.productResult"/>
</resultMap>

<resultMap id="productResult" class="com.ibatis.example.Product">
        <result property="id" column="PRD_ID"/>
        <result property="description" column="PRD_DESCRIPTION"/>
</resultMap>

<select id="getCategory"  parameterClass="int" resultMap="categoryResult">
        select C.CAT_ID,  C.CAT_DESCRIPTION, P.PRD_ID, P.PRD_DESCRIPTION
        from CATEGORY C
        left outer join PRODUCT P
        on C.CAT_ID = P.PRD_CAT_ID
        where CAT_ID = #value#
</select>
</sqlMap>
```

Wenn sie...

```
List myList = queryForList("ProductCategory.getCategory", new Integer(1002));
```

...aufrufen wird die Hauptabfrage ausgeführt, die Resultate werden in der `myList` Variable gespeichert, wobei die Listenelemente Beans vom Typ "com.ibatis.example.Category" sind. Jedes Objekt in der Liste hat eine "productList" Eigenschft, die ihrerseits wieder eine Liste, gefüllt durch die gleiche eine Abfrage,enthält. Die "productResult" wird verwendet, um die Beans in der Kind-Liste zu füllen. Wir haben also eine Liste, die ihrerseits Sub-Listen enthält und dies mit nur einer Datenbank-Abfrage.

Das wichtige Element hier ist das...

```
groupBy="id"
```

...Attribut und die..

```
<result property="productList" resultMap="ProductCategory.productResult"/>
```

...Abbildung in der "categoryResult" Result Map. Ein weiteres wichtiges Detail ist, das die Abbildung in der Result Map für die productList Eigenschaft sich des Namensraums bewusst sein muss – wäre es nur einfach "productResult" reichte dies nicht aus und würde nicht funktionieren.

Mit diesem Ansatz können sie jedes N+1 Problem jedweder Tiefe oder Breite lösen.

Wichtiger Hinweis: Eine Kombination von groupBy zusammen mit der queryForPaginatedList() API ist nicht definiert und sehr wahrscheinlich erhalten sie andere Ergebnisse als erwartet. Daher sollten sie beide Ansätze nicht gemeinsam verwenden. Verwenden sie groupBy, sollten sie auch immer queryForList oder queryForObject Methoden aufrufen.

Die verschachtelte Eigenschaft kann irgendeine Implementierung von java.util.Collection sein, und die zugehörigen Getter und Setter dafür sollten einfach den Zugriff auf das interne Attribut ermöglichen. iBATIS ruft intern mehrmals die add()-Methode der Eigenschaft. während es das Result Set abarbeitet, auf. Versuchen sie daher nicht, etwas anderes in ihren Getter und Setter (beispielsweise ein internes Array in einer List zu verpacken) auszuführen – dies führt sehr wahrscheinlich zu Fehlern mit iBATIS. Es gibt die weit verbreitete Auffassung, das iBATIS irgendwie Aufrufe für die Objekte zusammenfasst und die set Methode nur einmal aufruft. Dies ist nicht der Fall – iBATIS ruft die set Methode nur auf, wenn die get Methode null zurück liefert– in diesem Fall ruft iBATIS die vorbelegte Implementierung für die Eigenschaft auf und füllt das neue Objekt in das Result Objekt. Das neu erzeugte Objekt wird zunächst leer sein – iBATIS ruft anschließend die get Methode für die Eigenschaft auf und dann die add Methode.

Lazy Loading gegenüber Joins (1:M und M:N)

Wie bei der 1:1 Situation beschrieben, ist es wichtig anzumerken, dass es nicht *immer* besser sein muss einen Join zu benutzen Dieses gilt für Collection-Eigenschaften umso mehr, da es hierbei um wesentlich größere Datenmengen als bei individuellen Eigenschaften handelt. Wenn sie also wahrscheinlicher nicht auf das verbundene Objekt zugreifen (die *productList* Eigenschaft von der Category Klasse) mag es schneller sein, den Join und damit das unnötige Laden der Liste von products zu vermeiden. Dies gilt insbesondere bei Datenbank-Design, die einen Outer Join oder nicht indizierte Spalten beinhalten In derartigen Situationen mag es besser sein, stattdessen den Sub-Select mit der lazy loading und bytecode Option (siehe SQL Map Konfigurationseinstellungen) zu verwenden Daumenregel ist: nehmen sie einen Join, wenn sie wahrscheinlicher auf die verbundene Eigenschaft zugreifen. Sonst nehmen sie diese Lösung nur wenn sie lazy loading nicht benutzen können.

Wie bereits erwähnt, wenn sie sich nicht zwischen den beiden Wegen entscheiden können: Keine Panik! Sie können später sich anders entscheiden und müssen noch nicht einmal ihren Java Code anpassen. Obige beide Beispiele würden mit den gleichen Methodenaufrufen zum gleichen Objektgraphen führen. Die einzige Abwägung beträfe den Fall, wenn sie caching aktiviert haben. Im Fall einer separaten Abfrage(nicht der Join) kann zur Rückgabe einer bereits gecachten Instanz führen. Aber dies wird häufig nicht zu einem Problem führen. (Ihre Applikation sollte auch nicht davon abhängig sein, dass sie die Gleichheit von Instanz-Objekte durch "==" beachten müssen).

Zusammengesetzte Schlüssel oder multiple komplexe Parameter-Properties

Sie haben vielleicht bemerkt, dass in den Beispielen nur ein einziger Schlüssel in der Result Map (im *column* Attribut) angegeben wurde. Diese suggeriert das nur eine Spalte für ein verbundenes Mapped-Statement benutzt werden kann. Allerdings gibt es eine weitere Syntax, das es erlaubt, mehrere Spalten anzugeben. Dieses ist recht hilfreich wenn sie zusammengesetzte Schlüssel für die Beziehung haben oder einen anderen Parameter als *#value#* angeben möchten. Bei der alternativen Syntax für das column Attribut geben sie einfach {param1=column1, param2=column2, …, paramN=columnN} an. Bei dem Beispiel der PAYMENT Tabelle sind sowohl Customer ID und Order ID der zusammengesetzte Schlüssel:

```
<resultMap id="get-order-result" class="com.ibatis.example.Order">
        <result property="id" column="ORD_ID"/>
        <result property="customerId" column="ORD_CST_ID"/>
        …
        <result property="payments" column="{itemId=ORD_ID, custId=ORD_CST_ID}"
            select="getOrderPayments"/>
</resultMap>

<select id="getOrderPayments" resultMap="get-payment-result">
        select * from PAYMENT
```

```
        where PAY_ORD_ID  = #itemId#
        and PAY_CST_ID  = #custId#
</select>
```

Sie können auch optional nur die Spaltennamen angeben, sofern sie in der gleichen Reihenfolge wie die Parameter lauten. Zum Beispiel:

```
{ORD_ID, ORD_CST_ID}
```

Wie üblich, gibt diese Art der Angabe einen kleinen Performance-Vorteil und ist vielleicht etwas besser zu lesen und zu verstehen.

Wichtig! Das SQL Map Framework löst zyklische Beziehung nicht automatisch auf. Seien sie sich dessen bewusst wenn sie Vater/Kind Beziehungen abbilden möchten. Sie können ersatzweise eine zweite Result-Map definieren, für den Fall, das sie das Vater-Objekt nicht laden möchten (oder umgedreht), oder einen Join verwenden um das "N+1 Problem" zu umgehen.

Achtung! Einige JDBC Treiber (z:b: PointBase Embedded) unterstützen nicht mehrere gleichzeitig offene ResultSets (pro Connection). Derartige Treiber arbeiten mir komplexen, benutzerdefinierten Abbildungen nicht zusammen, da das SQL Map Framework mehrfache Connections erfordert. Mit einem Join könnten sie dies umgehen.

Achtung! Result-Map Namen sind immer lokal zur SQL Map XML Datei, in der sie definiert wurden. Sie können eine Result-Map in einer anderen SQL Map XML Datei referenzieren, indem sie der Result Map den Namen der SQL Map (definiert im <sqlMap> Wurzel Abschnitt) voranstellen.

Benutzen sie Microsoft SQL Server 2000 Treiber müssen sie wahrscheinlich auch SelectMethod=Cursor zur Verbindungs-URL hinzufügen, damit mehrere Statements im manuellen Transaktionsmodus ausgeführt werden kann.(lesen sie hierzu den MS Knowledge Base Article 313181: http://support.microsoft.com/default.aspx?scid=kb%3Ben-us%3B313181).

Unterstützte Typen von Parameter Maps und Result Maps

Folgende Java Typen werden vom iBATIS Framework als Parameter und Result unterstützt:

Java Type	JavaBean/Map Property Abbildung	Result Class / Parameter Class***	Type Alias**
boolean	JA	NEIN	boolean
java.lang.Boolean	JA	JA	boolean
byte	JA	NEIN	byte
java.lang.Byte	JA	JA	byte
short	JA	NEIN	short
java.lang.Short	JA	JA	short
int	JA	NEIN	int/integer
java.lang.Integer	JA	JA	int/integer
long	JA	NEIN	long
java.lang.Long	JA	JA	long
float	JA	NEIN	float
java.lang.Float	JA	JA	float
double	JA	NEIN	double
java.lang.Double	JA	JA	double
java.lang.String	JA	JA	string
java.util.Date	JA	JA	date
java.math.BigDecimal	JA	JA	decimal
* java.sql.Date	JA	JA	N/A
* java.sql.Time	JA	JA	N/A
* java.sql.Timestamp	JA	JA	N/A

Beachten sie, dass Typ Aliase seit Version 2.2.0.nicht zwischen Groß- und Kleinschreibung unterscheiden.. So werden Typ Aliase “string”, “String”, “StrinG” alle auf den Java Typ “java.lang.String” abgebildet.

** Von der Verwendung von java.sql. date types wird abgeraten. Gute Praxis ist es stattdessen java.util.Date zu verwenden.*

*** .Type Aliase können auch anstelle der voll qualifizierten Klassenname bei der Angabe von Parametern oder der Result-Class angegeben werden.*

**** Primitive Typen, wie int, boolean oder float werden vom iBATIS Database Layer nicht unterstützt, da es sich um einen vollständigen Objekt-orientierten Ansatz handelt. Daher müssen alle Parameter und Resultate letztendlich ein Objekt darstellen. Im Übrigen erlaubt das autoboxing-Feature von JDK 1.5 die Verwendung dieser primitiven Typen.*

Anwendungsspezifische Type Handler

Die Typ-Unterstützung in iBATIS kann durch Type-Handler oder das TypeHandlerCallback Interface erweitert werden. Da das TypeHandlerCallback Interface einfacher zu implementieren ist als das TypeHandler Interface, empfehlen wir ihnen, dieses zu nutzen. Für ihren eigenen Typ Handler müssen sie eine eigene Klasse schreiben, die TypeHandlerCallback implementiert. Mit derartigen benutzerdefinierten Typ-Handlern können sie das Framework um Typen erweitern, die es sonst nicht unterstützen würde. Beispielsweise könnten sie einen eigenen Typ-Handler implementieren, um proprietäre BLOB Unterstützung (z.B. bei Oracle) zu ermöglichen. Oder sie verwenden dies, um boolesche Werte durch "Y" und "N" statt 0/1 darzustellen

Hier ein einfaches Beispiel eines boolean Handler der "Yes" and "No" verwendet:

```
public class YesNoBoolTypeHandlerCallback implements TypeHandlerCallback {

  private static final String YES = "Y";
  private static final String NO = "N";

  public Object getResult(ResultGetter getter)
    throws SQLException {
    String s = getter.getString();
    if (YES.equalsIgnoreCase(s)) {
      return new Boolean (true);
    } else if (NO.equalsIgnoreCase(s)) {
      return new Boolean (false);
    } else {
      throw new SQLException (
        "Unexpected value " + s + " found where " + YES + " or " + NO + " was expected.");
    }
  }

  public void setParameter(ParameterSetter setter, Object parameter)
    throws SQLException {
    boolean b = ((Boolean)parameter).booleanValue();
    if (b) {
      setter.setString(YES);
    } else {
      setter.setString(NO);
    }
  }

  public Object valueOf(String s) {
    if (YES.equalsIgnoreCase(s)) {
      return new Boolean (true);
    } else {
      return new Boolean (false);
    }
  }
```

Um diesen Typen in iBATIS verwenden zu können, müssen sie ihn in ihrer sqlMapConfig.xml definieren:

```
<typeHandler
  javaType="boolean"
  jdbcType="VARCHAR"
  callback="org.apache.ibatis.sqlmap.extensions.YesNoBoolTypeHandlerCallback"/>
```

Damit weiß iBATIS, wie es zwischen dem angegebenen Java Typen und jdbc Typen mit dem angegebenen Type-Handler transformieren soll. Sie können einen spezifischen Type-Handler für einzelne Parameter beim <result> Mapping oder bei expliziten oder inline Parameter Maps angeben.

Cachen von Mapped Statement Resultaten

Die Resultate eines Abfrage Mapped Statement können durch Angabe eines cacheModel Parameter im statement Abschnitt gecached werden (siehe oben). Ein cache-Modell wird in ihrer SQL Map konfiguriert. Cache Modelle werden durch das cacheModel Element wie folgt definiert:

```
<cacheModel id="product-cache" type ="LRU" readOnly=”true” serialize=”false”>
   <flushInterval hours="24"/>
   <flushOnExecute statement="insertProduct"/>
   <flushOnExecute statement="updateProduct"/>
   <flushOnExecute statement="deleteProduct"/>
   <property name=”cache-size” value=”1000” />
</cacheModel>
```

Dieses cache Modell definiert einen Cache namens “product-cache” der eine Least Recently Used (LRU) Implementierung verwendet. Werte für das *type* Attribut sind entweder der voll qualifiziertet Klassenname oder ein Alias für die vorhandenen Implementierungen (siehe unten. Im angegebenen flush Element wird der Cache alle 24 Stunden geleert. Es darf nur eine Definition eines flush interval Elements geben und Werte dürfen in *hours, minutes, seconds* oder *milliseconds* angegeben werden. Jedes mal, wenn eines der Statements *insertProduct*, *updateProduct*, oder *deleteProduct* ausgeführt werden, wird der Cache ebenfalls geleert. Von derartigen “flush on execute” Elementen kann es eine beliebige Anzahl geben. Weitere Cache Implementierungen können zusätzliche Eigenschaften erfordern, wie zum Beispiel die cache-size’ Eigenschaft im Beispiel. Im Falle eines LRU Cache, gibt size die Anzahl von Einträgen, die im Cache gespeichert werden können, an. Sobald ein cache-Modell definiert wurde, kann es in Mapped-Statements verwendet werden, zum Beispiel:

```
<select id=”getProductList” cacheModel=”product-cache”>
      select * from PRODUCT where PRD_CAT_ID = #value#
</select>
```

Read-Only gegen Read/Write

Das Framework unterstützt sowohl Nur-Lese-Caches als auch Schreib/Lese-Caches. Nur-Lese-Caches werden von allen Benutzern geteilt und erlauben so eine bessere Performance. Daher sollten zurückgelieferte Objekte aus einem Nur-Lese-Cache auch nicht modifiziert werden. Stattdessen sollte ein neues Objekt aus der Datenbank für die Änderung gelesen oder ein Schreib/Lese-Cache verwendet werden. Wenn aber Objekte nicht nur gelesen sondern auch modifiziert werden müssen, empfehlen (notwendig) wir einen Schreib/Lese-Cache. Für einen Nur-Lese-Cache setzen sie *readOnly=”true”* im cache-Modell Element. Für einen Schreib/Lese-Cache entsprechend *readOnly=”false”*. Die Voreinstellung ist read-only (*true*).

Serializable Read/Write Caches

Sie stimmen hoffentlich überein, dass Caches pro Session wie bisher beschrieben nur eine geringe Leistungssteigerung für eine Applikation verspricht. Ein anderer Typ von Schreib/Lese-Cache verspricht einen Leistungsschub für die gesamte Applikation indem ein serialisierbarer Schreib/Lese-Cache definiert wird. Ein derartiger Cache liefert verschiedene Instanzen (Kopien) der gecachten Objekte für jede Session zurück. Daher kann das Instanz-Objekt von jeder Session bedenkenlos geändert werden. Beachten sie den Semantik-Unterschied. Normalerweise würden sie erwarten immer die gleiche Instanz vom Cache geliefert zu bekommen, aber hier in diesem Fall bekommt die Applikation bei jedem Aufruf ein anderes Objekt. Alle Objekte in einem serialisierbaren Cache müssen dazu auch *serializable* implementieren. Die bedeutet ‚dass sie lazy loading Merkmale zusammen mit einem serialisierbaren Cache nicht verwenden können, da lazy proxies nicht serialisierbar sind. Der beste Weg einer geeigneten Kombination von caching, lazy loading und table Joins ist: Probieren sie es einfach aus! Für einen serialisierbaren Cache setzen sie *readOnly=”false”* und *serialize=”true”*. Die Voreinstellung von Cache Modelle ist ‚das sie read-only und nicht serialisierbar sind. Nur-Leses-Caches werden nicht serialisiert, es gäbe auch keinen Vorteil.

Cache Typen

Das Cache Modell verwendet ein erweiterbares Framework um verschiedene Typen von Caches zu unterstützen. Die zu verwendende Implementierung wird durch das *type* Attribut im cacheModel-Element angegeben (wie oben bereits beschrieben). Die angegebene Klasse muss das CacheController Interface implementieren oder eine der vier vordefinierten Aliase benennen. Weitere Konfigurations-Parameter können mit dem property-Element an die Implementierung weitergegeben werden. Zur Zeit existieren vier Implementierungen in der Distribution. Es sind:

“MEMORY” (com.ibatis.db.sqlmap.cache.memory.MemoryCacheController)

Eine MEMORY Cache Implementierung verwendet Referenz Typen zur Verwaltung des Caches. Dies bedeutet, das der Garbage Collector bestimmt, ob ein Objekt im Cache bleibt oder verdrängt wird. Ein MEMORY Cache ist eine gute Wahl bei Applikation die keine erkennbare Struktur der Wiederverwendung von Objekten haben oder solchen bei denen Speicher eine knappe Ressource darstellt.

Die MEMORY Implementierung wird wie folgt konfiguriert:

```
<cacheModel id="product-cache" type="MEMORY">
        <flushInterval hours="24"/>
        <flushOnExecute statement="insertProduct"/>
        <flushOnExecute statement="updateProduct"/>
        <flushOnExecute statement="deleteProduct"/>
        <property name=”reference-type” value=”WEAK” />
</cacheModel>
```

Es gibt nur eine Eigenschaft die bei der MEMORY Cache Implementierung einstellbar ist. Diese Eigenschaft namens‘reference-type’ kann auf STRONG, SOFT oder WEAK gesetzt werden . Diese Werte entsprechen dem memory reference Typ für die JVM.

Die folgende Tabelle beschreibt die verschiedenen Referenz Typen für einen MEMORY Cache. Um dieses noch besser zu verstehen, verweisen wir auf die JDK Dokumentation zu java.lang.ref und hier insbesondere zum Thema “reachability”.

WEAK *(Voreinstellung)*	Dieser Referenz Typ ist wahrscheinlich die beste Wahl und die Voreinstellung wenn kein anderer Typ angegeben wird, Es verbessert die Performance für häufig verwendete Resultate, gibt den Speicher wieder frei, wenn andere Objekte Speicher anfordern, aber nur wenn die Resultate nicht mehr verwendet werden.
SOFT	Dieser Referenz Typ reduziert die Wahrscheinlichkeit des Auftretens von „out of memory“ in Fällen in denen Resultate derzeit nicht verwendet werden aber Speicher für andere Objekte benötigt wird. Es ist aber für diesen Zweck nicht die aggressivste Angabe für einen Referenz Typen und Speicher kann trotzdem angefordert werden und somit für vielleicht wichtigere Objekte entzogen werden.
STRONG	Dieser Referenz Typ garantiert, das Resultate im Speicher verbleiben, bis dieser explizit geleert wird (zum Beispiel durch eine Zeitangabe oder einem flush on execute). Dies ist ideal bei Resultaten die: 1) sehr klein sind, 2) absolut statisch sind und 3) oft verwendet werden. Vorteil ist, das die Performance sehr gut für diese Abfrage ist. Nachteil ist, dass der Speicher für die Resultate verwendet und nicht freigegeben wird auch wenn andere Objekte (vielleicht wichtigere Objekte) diesen Speicher benötigen könnten.

“LRU” (com.ibatis.db.sqlmap.cache.lru.LruCacheController)

Die LRU Cache Implementierung verwendet einen Least Recently Used ("Am längsten nicht verwendet") Algorithmus um zu bestimmen, welche Objekte automatisch aus dem Cache verdrängt werden können. Läuft der Cache voll, werden Objekte die lange nicht verwendet wurden, aus dem Cache verdrängt. Dadurch bleiben Objekte auf die häufig zugegriffen wird, im Cache und haben eine geringere Chance verdrängt zu werden. Ein LRU Cache ist eine gute Wahl bei Applikationen die auf Daten von mehreren Benutzern über eines längeren Zeitraum zugreifen (zum Beispiel vorwärts/rückwärts navigieren in Listen, häufige Suchbegriffe, etc.).

Die LRU Implementierung wird wie folgt konfiguriert:

```
<cacheModel id="product-cache" type="LRU">
        <flushInterval hours="24"/>
        <flushOnExecute statement="insertProduct"/>
        <flushOnExecute statement="updateProduct"/>
        <flushOnExecute statement="deleteProduct"/>
        <property name=”size” value=”1000” />
</cacheModel>
```

Es gibt nur eine Eigenschaft die bei der LRU Cache Implementierung einstellbar ist. Diese Eigenschaft, namens ‘size’, gibt als Ganzzahl (integer) die maximale Anzahl von im Cache speicherbaren Objekten an. Ein Objekt im Cache kann jedes Objekt sein, zum Beispiel eine String Instanz oder eine ArrayList von JavaBeans. Seien sie also vorsichtig und speichern nicht zu große Objekte im Cache oder sie bekommen ein Problem mit dem Hauptspeicher!

“FIFO” (com.ibatis.db.sqlmap.cache.fifo.FifoCacheController)

Die FIFO Cache Implementierung verwendet einen First In First Out Algorithmus um zu bestimmen, welche Objekte automatisch aus dem Cache verdrängt werden können. Läuft der Cache voll, werden die ältesten Objekte aus dem Cache verdrängt. Ein FIFO Cache ist eine gute Wahl, wenn eine bestimmte Abfrage mehrmals, schnell nacheinander, aber wahrscheinlich später nicht wieder, ausgeführt wird.

Die FIFO Implementierung wird wie folgt konfiguriert:

```
<cacheModel id="product-cache" type="FIFO">
        <flushInterval hours="24"/>
        <flushOnExecute statement="insertProduct"/>
        <flushOnExecute statement="updateProduct"/>
        <flushOnExecute statement="deleteProduct"/>
        <property name=”size” value=”1000” />
</cacheModel>
```

Es gibt nur eine Eigenschaft die bei der FIFO Cache Implementierung einstellbar ist. Diese Eigenschaft, namens ‘size’, gibt als Ganzzahl (integer) die maximale Anzahl - von im Cache speicherbaren - Objekten an. Ein Objekt im Cache kann jedes Objekt sein, zum Beispiel eine String Instanz oder eine ArrayList von JavaBeans. Seien sie also vorsichtig und speichern nicht zu große Objekte im Cache oder sie bekommen ein Problem mit dem Hauptspeicher

“OSCACHE” (com.ibatis.db.sqlmap.cache.oscache.OSCacheController)

Die OSCACHE Cache Implementierung ist ein PlugIn für das OSCache 2.0 Cache Framework. Es ist höchst konfigurierbar, verteilbar und flexibel.

Die OSCACHE Implementierung wird wie folgt konfiguriert:

```
<cacheModel id="product-cache" type="OSCACHE">
        <flushInterval hours="24"/>
        <flushOnExecute statement="insertProduct"/>
        <flushOnExecute statement="updateProduct"/>
        <flushOnExecute statement="deleteProduct"/>
</cacheModel>
```

Für die OSCACHE Implementierung gibt es keine weiteren einstellbaren Eigenschaften. Stattdessen verwendet die OSCache Instanz die Standard *oscache.properties* Datei zur Konfiguration, die im Classpath liegen muss. Darin können sie Dinge wie Algorithmen , Cache Größen, Persistenzansatz (Speicher, Datei, ...), und Clustering einstellen.

Schauen sie bitte in die OSCache Dokumentation für weitere Informationen. OSCache und die Dokumentation können sie auf folgender Open Symphony Website finden:

http://www.opensymphony.com/oscache/

Dynamische Mapped Statements

Ein häufiges Problem beim Arbeiten mit JDBC ist dynamisches SQL. Es ist nicht so leicht, mit SQL Statements zu arbeiten bei denen sich nicht nur die Werte von Parametern, sondern Parameter-Anzahl und Spalten, ändern können.. Dies führt typischerweise zu einem Durcheinander von if-else Statements und vielen Zeichenketten Operationen. Häufig findet man dies in bei einer query-by-example, bei der eine Abfrage gebaut wird ein Objekt zu finden, das ähnlich zu einem Schablonen-Objekt ist. Die Data Mapper API stellt eine elegante Lösung bereit, die für jedes Mapped-Statement angewendet werden kann. Hier ein einfache Beispiel:

```
<select id="dynamicGetAccountList"
                    cacheModel="account-cache"
                    resultMap="account-result" >

    select * from ACCOUNT

        <isGreaterThan prepend="and" property="id" compareValue="0">
            where  ACC_ID = #id#
        </isGreaterThan>

    order by ACC_LAST_NAME

</select>
```

In diesem Beispiel könne zwei verschiedene SQL-Befehle erzeugt werden, abhängig von der “id” Eigenschaft der Parameter Bean. Ist der id Parameter größer als 0, wird folgender SQL-Befehl erzeugt:

```
select * from ACCOUNT where ACC_ID = ?
```

Ist der id Parameter kleiner oder gleich 0, sieht der Befehl so aus.

```
select * from ACCOUNT
```

Die Nützlichkeit hiervon wird mit einem komplexeren Beispiel sichtbar:

```
<select id="dynamicGetAccountList"
                    resultMap="account-result" >
  select * from ACCOUNT
  <dynamic prepend="WHERE">
    <isNotNull prepend="AND" property="firstName"
      open="”(“ close="”)”>
      ACC_FIRST_NAME = #firstName#
      <isNotNull prepend="OR" property="lastName">
        ACC_LAST_NAME = #lastName#
      </isNotNull>
    </isNotNull>
    <isNotNull prepend="AND" property="emailAddress">
      ACC_EMAIL like #emailAddress#
    </isNotNull>
    <isGreaterThan prepend="AND" property="id" compareValue="0">
      ACC_ID = #id#
    </isGreaterThan>
  </dynamic>
  order by ACC_LAST_NAME
</select>
```

Je nach Situation erzeugt dieser Abschnitt bis zu 16 verschiedene SQL Abfragen. Um dies mit einer if-else Struktur und Zeichenkettenmanipulation abzubilden, wären hunderte Zeilen unordentlicher Code notwendig.

Für dynamische Statements müssen sie einfach bedingte Marken um die dynamische Teile ihrer SQL einfügen. Zum Beispiel:

```
<select id="someName"
                resultMap="account-result" >
  select * from ACCOUNT
  <dynamic prepend="where">
    <isGreaterThan prepend="and" property="id" compareValue="0">
      ACC_ID = #id#
    </isGreaterThan>
    <isNotNull prepend=”and" property="lastName">
      ACC_LAST_NAME = #lastName#
    </isNotNull>
  </dynamic>
  order by ACC_LAST_NAME
</select>
```

In diesem Statement grenzt das <dynamic> Element einen SQL-Teil ab der dynamisch ist. Das dynamische Element ist optional und stellt eine Möglichkeit dar, einen SQL-Teil (z.B. “WHERE”) nur dann hinzuzufügen, wenn gewisse Bedingungen im Statement dazuführen eine Zeichenkette dem Statement wirklich hinzuzufügen. Dieser Abschnitt kann eine beliebige Anzahl von konditionalen Elementen enthalten, die ihrerseits festlegen ob ein SQL-Teil zum Gesamt-SQL-Befehl hinzugenommen werden soll. Alle Konditional-Elemente arbeiten auf dem Parameter Objekt der Abfrage. Sowohl das dynamic Element als auch die Konditionalelemente verfügen über das “prepend” Attribut. Das prepend Attribut wird durch den prepend-Angabe des Vater-Elementes überschrieben soweit erforderlich. Im obigen Beispiel überschreibt der “where”-prepend das erste konditionale prepend. Das ist notwendig damit der SQL Befehl korrekt erzeugt wird. Im Falle der ersten Bedingung gibt es keinen Grund AND hinzuzufügen und es würde in der Tat auch das Statement syntaktisch falsch erzeugen..Die folgenden Abschnitte erläutern die verschiedenen Elemente, wie binäre und unäre Konditionalelemente und Iterate.

Dynamic Element

Mt der dynamic Marke werden andere dynamische SQL-Elemente umhüllt. Damit kann ein SQL-Fragment dem Gesamtinhalt für die Einleitung oder das Schließen definiert werden. Durch diese Marke wird die Funktionalität des removeFirstPrepend Attributs erzwungen. Die erste Konditionalmarke die also Zeichenketten zum Gesamt-SQL Befehl hinzufügt, dessen prepend-Attribut wird entfernt

Binary Konditional Attribute:
prepend – SQL Teil, der dem Statement vorangestellt wird (optional)
open – die Zeichenkette, die als Einleitung für den gesamten Text, dient
close – die Zeichenkette, die als Schluss für den gesamten Text, dient (optional)

<dynamic>	Marker der erlaubt ein gesamtes prepend, open oder close-Segement zu definieren

Binary Konditional Elemente

Binäre Konditionalelemente vergleichen eine Eigenschaft mit einem statischen Wert oder mit dem Wert einer anderen Eigenschaft. Liefert der Vergleich true, wird der Inhalt der SQL Abfrage hinzugefügt.

Binary Konditional-Attribute:
prepend – SQL Teil, der dem Statement vorangestellt wird (optional)
property – Eigenschaft die verglichen werden soll (erforderlich)
compareProperty – andere Eigenschaft, mit der verglichen werden soll (erforderlich oder compareValue)
compareValue – Wert mit dem verglichen werden soll (erforderlich oder compareProperty)
removeFirstPrepend – entfernt den prepend-Teil von der ersten Text erzeugenden Marke (true|false, optional)

open – die Zeichenkette, die als Einleitung für den gesamten Text, dient (optional)
close – die Zeichenkette, die als Schluss für den gesamten Text, dient (optional)

<isEqual>	Prüft auf Gleichheit einer Eigenschaft mit einem Wert oder einer anderen Eigenschaft.
<isNotEqual>	Prüft auf Ungleichheit einer Eigenschaft mit einem Wert oder einer anderen Eigenschaft.
<isGreaterThan>	Prüft ob eine Eigenschaft größer als ein Wert oder eine anderen Eigenschaft ist.
<isGreaterEqual>	Prüft ob eine Eigenschaft größer oder gleich als ein Wert oder eine andere Eigenschaft ist.
<isLessThan>	Prüft ob eine Eigenschaft kleiner als ein Wert oder eine andere Eigenschaft ist.
<isLessEqual>	Prüft ob eine Eigenschaft kleiner oder gleich als ein Wert oder eine andere Eigenschaft ist Beispiel: **<isLessEqual prepend=”AND” property=”age” compareValue=”18”> ADOLESCENT = ‘TRUE’ </isLessEqual>**

Unäre Konditionalelemente

Unäre Konditionalelemente überprüfen den Status einer Eigenschaft auf eine bestimmte Bedingung.

Unäre Konditional-Attribute:
prepend – SQL Teil der dem Statement vorangestellt wird (optional)
property – die zu prüfende Eigenschaft (erforderlich)
removeFirstPrepend – entfernt den prepend Teil von der ersten Text produzierenden Marke (true|false, optional)
open – die Zeichenkette, die als Einleitung für den gesamten Text, dient (optional)
close – die Zeichenkette, die als Schluss für den gesamten Text, dient (optional)

<isPropertyAvailable>	Prüft ob eine Eigenschaft vorhanden ist (ist eine Eigenschaft der Parameter Bean)
<isNotPropertyAvailable>	Prüft ob eine Eigenschaft nicht vorhanden ist (ist keine Eigenschaft der Parameter Bean)
<isNull>	Prüft ob eine Eigenschaft null ist
<isNotNull>	Prüft ob eine Eigenschaft nicht null ist
<isEmpty>	Prüft ob der Wert einer Collection, String oder String.valueOf() Eigenschaft null oder leer ist (“” oder size() < 1).
<isNotEmpty>	Prüft ob der Wert einer Collection, String oder String.valueOf() Eigenschaft nicht null oder leer ist (“” oder size() < 1). Beispiel: **<isNotEmpty prepend=”AND” property=”firstName” > FIRST_NAME=#firstName# </isNotEmpty>**

Andere Elemente

Parameter Present: Dieses Element prüft auf Existenz eines Parameter Objektes.
Parameter Present Attribute:
prepend – SQL Teil der dem Statement vorangestellt wird (optional)
removeFirstPrepend – entfernt den prepend Teil von der ersten Text produzierenden Marke (true|false, optional)
open – die Zeichenkette, die als Einleitung für den gesamten Text, dient (optional)
close – die Zeichenkette, die als Schluss für den gesamten Text, dient (optional)

<isParameterPresent>	Prüft ob ein Parameter Objekt vorhanden ist (nicht null).
<isNotParameterPresent>	Prüft ob ein Parameter Objekt nicht vorhanden ist (null). Beispiel: **<isNotParameterPresent prepend="AND"> EMPLOYEE_TYPE = 'DEFAULT' </isNotParameterPresent>**

Iterate: Dieses Marke iteriert über eine Collection und wiederholt den Inhalt für jeden Eintrag der Liste
Iterate Attribute:
prepend – SQL Teil der dem Statement vorangestellt wird (optional)
property – eine Eigenschaft vom Typ java.util.Collection, java.util.Iterator, oder ein Array über das iteriert werden soll.
over (optional – das Parameter Objekt wird als die Collection angenommen wenn keine property spezifiziert wurde. (Mehr Informationen weiter unten)
open – die Zeichenkette, die den ganzen Block aller Iterationen startet. Hilfreich bei Klammern (optional)
close – die Zeichenkette, die den ganzen Block aller Iterationen beendet. Hilfreich bei Klammern (optional)
conjunction – die Zeichenkette, die zwischen den Iteration eingefügt wird. Hilfreich bei AND und OR (optional)
removeFirstPrepend – entfernt den prepend Teil von der ersten Text produzierenden Marke (true|false|iterate, optional – mehr Information weiter unten)

<iterate>	Iteriert über eine Eigenschaft vom Typ java.util.Collection, java.util.Iterator, oder ein Array . Beispiel **<iterate prepend="AND" property="userNameList"** **open="(" close=")" conjunction="OR">** **username=#userNameList[]#** **</iterate>** Es ist auch möglich über eine Collection zu iterieren, wenn bereits der Parameter vom Mapped Statement eine Collection ist. Beispiel: **<iterate prepend="AND" open="(" close=")" conjunction="OR">** **username=#[]#** **</iterate>** Hinweis: Die eckigen Klammern beim Namen der Eigenschaft sind sehr wichtig. Damit weiß der Parser, das es sich um eine Collection handelt und gibt nicht einfach die Collection als eine Zeichenkette aus..

Weitere <iterate> Verwendungen:

Beachten sie, dass im ersten Beispiel, "userNameList[]" zu einem Operator wird der das aktuelle Element der Liste referenziert. Daher können sie diesen Operator auch verwenden um auf Eigenschaften der Liste zuzugreifen::

```
<iterate prepend="AND" property="userList"
        open="(" close=")" conjunction="OR">

  firstname=#userList[].firstName#  and
  lastname=#userList[].lastName#

</iterate>
```

Seit iBATIS Version 2.2.0 können iterate Marken auch verschachtelt werden, um komplexe Bedingungen abzubilden. Hier ein Beispiel:

```
<dynamic prepend="where">
 <iterate property="orConditions" conjunction="or">
 (
  <iterate property="orConditions[].conditions"
        conjunction="and">
      $orConditions[].conditions[].condition$
      #orConditions[].conditions[].value#
  </iterate>
 )
 </iterate>
</dynamic>
```

Dieses unterstellt, das Parameter Objekt habe die Eigenschaft "orConditions" die eine Liste von Objekten repräsentiert. Jedes Objekt dieser Liste enthält wieder eine List Eigenschaft namens "conditions". So haben wir Listen innerhalb von Listen im Parameter Objekt.

Beachten sie, das der Passus “orConditions[].conditions[].condition” bedeutet: “Hole die Eigenschaft vom aktuellen Element in der inneren Schleife, die die Eigenschaft vom aktuellen Durchlauf der äußeren Schleife darstellt. Es gibt keine Einschränkungen wie tief iterate Marken verschachtelt werden dürfen. Somit kann der “aktuelles Element” Operator als Eingabe in jeder dynamischen Marke verwendet werden.

Das *removeFirstPrepend* Attribut bei der <iterate> Marke funktioniert etwas anders als bei den anderen Marken. Geben sie hier *true* für *removeFirstPrepend* an, wird der Text des prepend-Attributs von der ersten Text-produzierenden Marke entfernt. Dieses gilt einmal für die ganze Schleife.. Dieses ist in den allermeisten Fällen das korrekte Verhalten.

In einigen wenigen Fällen aber mag es richtig erscheinen die *removeFirstPrepend* Funktion für jeden Durchlauf der Schleife auszuführen und nicht nur einmal. In diesem Fall müssen sie der *iterate* Marke das removeFirstPrepend Attribut zusätzlich angeben. Diese Funktion gibt es erst seit iBATIS Version 2.2.0.

Einfache dynamische SQL Elemente

Manchmal benötigen sie nicht die ganze Mächtigkeit der kompletten Dynamic Mapped Statement API wie oben beschrieben, sondern nur ein kleiner Teil ihrer SQL der dynamisch sein muss. Dafür können einfache dynamische Elemente angegeben werden, um dynamische *order by* Klauseln, dynamische select-Spalten oder irgendeinen Teil des SQL Statements zu ändern. Dieses Konzept funktioniert ähnlich wie inline Parameter Maps, verwendet aber eine leicht abgewandelte Syntax. Nehmen wir folgendes Beispiel:

```
<select  id=”getProduct” resultMap=”get-product-result”>
        select * from PRODUCT order by $preferredOrder$
</select>
```

In diesem Beispiel ist *preferredOrder* ein dynamisches Element das durch den Wert der Eigenschaft *preferredOrder* aus dem Parameter Objekt (genau wie bei einer Parameter Map) ersetzt wird. Unterschied hier aber ist, dass dies zu einem fundamentalen anderen SQL Statement führt, als nur einen Parameterwert zu setzen. Ein Fehler mit einem dynamischen Element kann leicht zu Sicherheits-, Performance- oder Stabilitätsproblemen führen. Testen sie daher ob dynamische Befehle immer entsprechend wie erwartet ausgeführt werden. Beachten sie auch, dass die Gefahr besteht, das Datenbank-Interna in ihre Business-Objekte eindringen. Beispielsweise möchten sie einen Spaltennamen nicht als Eigenschaft in ihrer Business-Welt oder als Feld in ihrer JSP Seite wiederfinden.

Einfache dynamische Element können innerhalb von Statements eingefügt werden und sind manchmal hilfreich, wenn es darum geht, das SQL Statement selbst zu ändern. Beispielsweise:

```
<select  id=”getProduct” resultMap=”get-product-result”>
        SELECT * FROM PRODUCT
        <dynamic prepend=”WHERE”>
          <isNotEmpty property=”description”>
            PRD_DESCRIPTION   $operator$   #description#
          </isNotEmpty>
        </dynamic>
</select>
```

In diesem Beispiel ersetzt die *operator* Eigenschaft aus dem Parameter Objekt das $operator$ Kürzel Wenn also die *operator* Eigenschaft ‘like’ ist und die *description* Eigenschaft ‘%dog%’, lautet das Gesamt SQL Statement :

```
SELECT * FROM PRODUCT WHERE PRD_DESCRIPTION LIKE ‘%dog%’
```

Entwickeln mit dem Data Mapper: die API

Die SqlMapClient API ist einfach und minimalistisch. Sie unterstützt den Entwickler mit im wesentlichen vier Funktionen: konfiguriere ein SQL Map, führe einen SQL update (kann auch insert und delete) Befehl aus, führe eine Abfrage nach einem einzigen Objekt aus, und führe eine Abfrage nach einer Liste von Objekten aus.

Konfiguration

Die Konfiguration der SQL Map ist eigentlich trivial, wenn man bereits die SQL Map XML Definitionsdateien und die SQL Map Konfigurationsdatei erzeugt hat. (Beschreibung oben). SqlMapClient Instanzen werden durch einen SqlMapClientBuilder erzeugt. Diese Klasse bietet statische Methoden namens buildSqlMap() an, die eine SqlMapClient Instanz bauen und zurück liefern. Die buildSqlMap() Methode kann aus einem Reader oder einem InputStream die Werte aus einer Konfigurationsdatei lesen. Optional können sie diese Werte auch in einem Properties Objekt übergeben.. Hier einige Beispiele zum Gebrauch:

```
String resource = "com/ibatis/example/sqlMap-config.xml";
Reader reader = Resources.getResourceAsReader (resource);
SqlMapClient sqlMap =  SqlMapClientBuilder.buildSqlMap(reader);

String resource = "com/ibatis/example/sqlMap-config.xml";
InputStream inputStream = Resources.getResourceAsStream (resource);
SqlMapClient sqlMap =  SqlMapClientBuilder.buildSqlMap(inputStream);
```

Die Unterschiede zwischen beiden Versionen betrifft hauptsächlich die Zeichenkodierung und Internationalisierung. Lesen sie im Abschnitt über Internationalisierung hierzu weitere Details.

Transaktionen

Voreinstellung ist, dass bei der Ausführung von Methoden der SqlMapClient Instanz (also z.B. queryForObject() oder insert()) auto-commit oder auto-rollback verwendet wird. Dies bedeutet, dass nach der Ausführung jeder SQL-Anweisung automatisch die Änderungen in der Datenbank durch einen COMMIT festgeschrieben oder bei einem Fehler rückgängig gemacht werden. Somit wird jede SQL-Anweisung mit einer eigenen Transaktion assoziiert In der Tat ist dies einfach und reicht nicht aus, wenn sie mehrere Befehle als zusammengehörend betrachten müssen, die entweder erfolgreich sein oder scheitern können. Und hier kommen Transaktionen ins Spiel.

Wenn sie Global Transactions (eingestellt in der SQL Map Konfigurationsdatei) benutzen, können sie auto-commit verwenden und trotzdem transaktionales Verhalten erhalten. Dennoch mag es insbesondere aus Performance-Gründen sinnvoll sein, Transaktions-Grenzen zu kennzeichnen

Das SqlMapClient Interface bietet Methoden an um transaktionale Grenzen zu kennzeichnen. Eine Transaktion kann gestartet, committed oder zurückgerollt werden:

```
public void startTransaction () throws SQLException
public void commitTransaction () throws SQLException
public void endTransaction () throws SQLException
```

Das Starten einer Transaktion geschieht, indem sie eine Connection vom Verbindungspool holen, die Transaktion öffnen und SQL Anfragen und Updates abzuschicken.

Ein Beispiel für die Verwendung von Transaktionen ist:

```
private Reader reader = new Resources.getResourceAsReader(
                                        "com/ibatis/example/sqlMap-config.xml");
private SqlMapClient sqlMap = SqlMapClientBuilder.buildSqlMap(reader);

public updateItemDescription (String itemId, String newDescription)
      throws SQLException {
   try {
      sqlMap.startTransaction ();
      Item item =  (Item) sqlMap.queryForObject ("getItem", itemId);
      item.setDescription (newDescription);
      sqlMap.update ("updateItem", item);
      sqlMap.commitTransaction ();
   } finally {
      sqlMap.endTransaction ();
   }
}
```

Beachten sie das endTransaction() aufgerufen wird ungeachtet ob ein Fehler aufgetreten ist. Dieser wichtiger Schritt ist wichtig für Aufräumarbeiten.. Regel ist: Rufen sie startTransaction() aus müssen sie auch sicherstellen, dass endTransaction() aufgerufen wird (egal ob sie committen oder nicht).

Achtung! Transaktionen können nicht ineinander verschachtelt werden. Rufen sie startTransaction() innerhalb eines Threads mehrmals auf , bevor sie commit() oder rollback() aufrufen, wird eine Exception geworfen. Mit anderen Worten kann jeder Thread höchstens eine offene Transaktion pro SqlMapClient Instanz haben.

Achtung! SqlMapClient Transaktionen verwenden Java's ThreadLocal Speicher um transaktionale Objekte zu speichern. Dies bedeutet, dass jeder Thread bei Aufruf von startTransaction() ein eindeutiges Connection Objekt für dessen Transaktion bekommt. Der einzige Weg diese Verbindung zurückzugeben (oder die Verbindung zu schließen) geschieht entweder durch commitTransaction() oder endTransaction(). Vergisst man dieses, wird der Verbindungspool leer laufen und keine Verbindungen mehr herstellen können.

Automatische Transaktionen

Obwohl wir explizite Transaktionssteuerung hochgradig empfehlen, gibt es eine vereinfachte Semantik für anspruchslose Anforderungen (normalerweise Nur-Lese-Anforderung).Wenn sie eine Transaktion nicht explizit durch startTransaction(), commitTransaction() und endTransaction() kennzeichnen, werden sie automatisch für sie aufgerufen, wenn ihre Statements außerhalb eines transaktionalen Blocks stehen. Zum Beispiel:

```
private Reader reader = new Resources.getResourceAsReader(
                                        "com/ibatis/example/sqlMap-config.xml");
private SqlMapClient sqlMap = SqlMapClientBuilder.buildSqlMap(reader);

public updateItemDescription (String itemId, String newDescription)
      throws SQLException {
   try {
      Item item =  (Item) sqlMap.queryForObject ("getItem", itemId);
      item.setDescription ("TX1");
      // Keine Transaktion markiert, also ist die Transaktion automatisch (impliziert)
      sqlMap.update ("updateItem", item);
      item.setDescription (newDescription);
      item.setDescription ("TX2");
      // Keine Transaktion markiert, also ist die Transaktion automatisch (impliziert)
      sqlMap.update("updateItem", item);
   } catch (SQLException e) {
      throw (SQLException) e.fillInStackTrace();
   }
}
```

Achtung! Seien sie vorsichtig bei automatischen Transaktionen. Sie sind zwar reizvoll, sie riskieren aber Probleme, sollte ihre Arbeit aus mehreren Befehlen gegen die Datenbank bestehen. Würde der zweite Aufruf von "updateItem" fehlschlagen, wäre die item description ("TX1") trotzdem geändert worden. Beide Updates zusammen wären somit nicht transaktional.

Globale (DISTRIBUTED) Transaktionen

Das Data Mapper Framework unterstützt ebenfalls globale Transaktionen. Globale Transaktionen, auch bekannt unter verteilten Transaktionen (distributed transactions), erlauben die Änderung von mehreren Datenbanken (oder anderen JTA konformen Ressourcen) in einem atomaren Arbeitsschritt (Änderungen auf mehreren Datasources können entweder als Ganzes erfolgreich sein oder scheitern).

External/Programmatic Global Transactions

Sie können globale Transaktionen extern, entweder programmatisch (manuelle Codierung) oder durch ein weiteres Framework wie zum Beispiel das weit verbreitete EJB Framework, steuern. Mit EJBs können sie Transaktionen deklarativ im EJB Deployment Descriptor kennzeichnen. Weitere Erläuterung hierzu ist nicht Bestandteil dieses Dokuments. Um die Unterstützung für externe oder programmatische globale Transaktionen zu aktivieren, müssen sie das <transactionManager> *type* Attribute auf "EXTERNAL" in ihrer SQL Map Konfigurationsdatei setzen (siehe oben). Werden Transaktion extern gesteuert, sind die SQL Map Transaktion-Methoden überflüssig, da Start, Commit und Rollback der Transaktion von einem externen Transaktions-Manager gesteuert werden. Trotzdem kann es einen kleinen Performance-Vorteil geben , Transaktionen mit den SqlMapClient Methoden startTransaction(), commitTransaction() und endTransaction() zu kennzeichnen (statt über eine automatische Transaktion diese zu starten, committen oder zurückzurollen). Hält man sich an diese Methoden, wird einerseits das Design konsistenter und sie vermindern auch die Anzahl von notwendigen Verbindungen im Pool. Ein weiterer Vorteil kann darin bestehen, wenn sie die Reihenfolge der Freigabe von Ressourcen (commitTransaction() oder endTransaction()), gegenüber dem wann die globale Transaktion committed wird, ändern möchten. Verschiedene Applikationsserver haben (leider) unterschiedliche Verhalten. Außer diesen einfachen Überlegungen gibt es keine weiteren Änderungen an ihrem SQL Map Code um globale Transaktionen verwenden zu können.

Managed Global Transactions

Das SQL Map Framework kann auch mit verwalteten globalen Transaktionen auskommen. Für diese Art der Unterstützung müssen sie das <transactionManager> *type* Attribut auf "JTA" in ihrer SQL Map Konfigurationsdatei setzen und die "UserTransaction" Eigenschaft auf den vollen JNDI Namen setzen unter dem die SqlMapClient Instanz die UserTransaction Instanz finden kann. Schauen sie in die Beschreibung zum <transactionManager> für weitere Details.

Entwickeln für globale Transaktionen unterscheidet sich kaum vom bisher beschriebenen, es gibt aber einige kleine Überlegungen, die beachtet werden sollten. Hier ein Beispiel:

```
try {
  orderSqlMap.startTransaction();
  storeSqlMap.startTransaction();

  orderSqlMap.insertOrder(…);
  orderSqlMap.updateQuantity(…);

  storeSqlMap.commitTransaction();
  orderSqlMap.commitTransaction();
} finally {
  try {
    storeSqlMap.endTransaction()
  } finally {
    orderSqlMap.endTransaction()
  }
}
```

In diesem Beispiel gibt es zwei SqlMapClient Instanzen von denen wir annehmen, dass diese auf zwei verschiedenen Datenbanken zugreifen. Die erste SqlMapClient (orderSqlMap)auf der wir eine Transaktion starten, startet auch die globale Transaktion. Danach betrachten wir alle Aktivitäten als Bestandteil der globalen Transaktion bis der gleiche SqlMapClient (orderSqlMap) commitTransaction() bzw. endTransaction() aufruft, zu diesem Zeitpunkt wird die global transaction committed und die ganze Arbeit ist abgeschlossen.

Warnung! Obwohl das Verfahren einfach aussieht, sollte es auch nicht überstrapaziert werden. Es gibt neben Performance-Verwicklungen auch eine komplexere Konfiguration für den Applikationsserver und die Datenbank zu beachten. Zwar gibt es große Unterstützung seitens der Industrie für EJBs, aber vielleicht sollten sie doch Session EJBs für ihre Arbeit mir verteilten Transaktionen verwenden. Die JPetStore Beispiel-Applikation unter ibatis.apache.org ist ein Beispiel für SQL Map globale Transaktionen.

Multi Threaded Programmierung

iBATIS unterstützt multi threaded Programmierung, es gibt aber einige Dinge, dessen man sich bewusst sein sollte.

Zuerst müssen Transaktionen vollständig in einem Thread enthalten sein. Anders gesagt, dürfen Transaktionen nicht über Thread Grenzen laufen. Daher ist ein guter Denkansatz, wenn man den Start eines Threads als ein vollständiges Beispiel von transaktionaler Arbeit betrachtet. Es ist keine gute Idee einen Pool von Threads bereitzustellen, die darauf warten, Transaktionen zu starten und auszuführen, es sei denn, sie stellen sicher, dass Threads mit transaktionaler Arbeit übereinstimmen.

Ein weiterer Gesichtspunkt, der zu beachten ist, bedeutet, dass es nur eine aktive Transaktion in einem Thread zu jedem Zeitpunkt geben darf. Zwar können sie Code schreiben der mehrere Transaktionen in einem Thread ausführt, diese Transaktionen müssen aber sequentiell, nacheinander ausgeführt werden und können nicht parallel zur selben Zeit offen sein. Dies ist ein Beispiel für mehrere, nacheinander ausgeführte Transaktionen in einem Thread:

```
try {
  sqlMap.startTransaction();
  // führe statements in der ersten Transaktion aus
  sqlMap.commitTransaction();
} finally {
  sqlMap.endTransaction();
}

try {
  sqlMap.startTransaction();
  // führe statements in einer zweiten Transaktion aus
  sqlMap.commitTransaction();
} finally {
  sqlMap.endTransaction();
}
```

Hier ist wichtig, dass zu jedem Zeitpunkt nur eine Transaktion aktiv in einem Thread sein darf. Bei automatischen Transaktionen ist jedes Statement bereits eine Transaktion.

iBATIS Classloading

(Die Information in diesem Abschnitt beschreibt das Verhalten ab iBATIS Version 2.2.0)

iBATIS verwendet Methoden aus der *com.ibatis.common.resources.Resources* Klasse um andere Klassen zu laden. Die wichtigste Methode hierbei ist *classForName(String)* für die Überlegungen zum Classloading und ist die Wurzel für das ganze class loading von iBATIS. Die Methode arbeitet wie folgt:

- Versuche die Klasse durch den context class loader vom aktuellen Thread zu laden
- Wenn dabei ein Fehler auftritt, versuche die Klasse über *Class.forName(String)* zu laden

In den meisten Umgebungen funktioniert dieses Verfahren gut. Wenn dies aus irgendwelchen Gründen in ihrer Umgebung nicht läuft, können sie einen anderen Klassenlader für alle Operationen mit der statischen Methode *Resources.setDefaultClassLoader(ClassLoader)* angeben. Wenn sie einen Klassenlader über diese Methode bereitstellen, versucht iBATIS alle Klassen über diesen Klassenlader zu laden (und bei Fehler auf *Class.forName(String)* zurückzufallen). Wenn sie einen eigenen, benutzerdefinierten Klassenlader angeben möchten, sollten sie dies tun, bevor irgendeine Operation von iBATIS ausgeführt wurde.

Stapelverarbeitung

Haben sie eine große Anzahl von Statements (insert/update/delete) auszuführen, möchten sie diese vielleicht in einem Rutsch (Stapel) ausführen, um den Netzwerkverkehr zu minimieren oder um dem JDBC Treiber Gelegenheit zu geben weitere Optimierungen (z.B. Komprimierung) vorzunehmen. Mit der SQL Map API können sie einfach eine Stapelverarbeitung anstoßen und eine Gruppe von Befehlen kennzeichnen:

```
try {
  sqlMap.startTransaction();
  sqlMap.startBatch();
  // … führe einige Befehle aus
  int rowsUpdated = sqlMap.executeBatch();  //optional
  sqlMap.commitTransaction();
} finally {
  sqlMap.endTransaction();
}
```

Beim Aufruf von executeBatch() wird der gesamte Stapel von Statements durch den JDBC Treiber ausgeführt. executeBatch() ist dabei optional, da die commit Operation einen Stapel (Batch) automatisch ausführen wird, wenn noch ein nicht ausgeführter Stapel offen ist. Sie können executeBatch() aufrufen, wenn sie die Anzahl der betroffenen Zeilen wissen möchten oder sie können darauf verzichten und rufen nurcommitTransaction() auf.

Haben sie eine große Anzahl von Operationen in ihrer Stapelverarbeitung, möchten sie vielleicht periodische Commits während der Ausführung einstreuen. Wenn sie zum Beispiel 1000 Zeilen einfügen möchten, möchten aber alle 100 ein Commit machen, damit die Transaktion nicht riesig wird. Wenn sie periodisch commits ausführen möchten, müssen sie nach jedem periodischen commit auch wieder startBatch() aufrufen – der commit wird ausgeführt und beendet auch die Stapelverarbeitung. Hier ein Beispiel:

```
try {
  int totalRows = 0;
  sqlMap.startTransaction();

  sqlMap.startBatch();
  // … füge 100 Zeilen ein
  totalRows += sqlMap.executeBatch();  //optional
  sqlMap.commitTransaction();

  sqlMap.startBatch();
  // … füge 100 Zeilen ein
  totalRows += sqlMap.executeBatch();  //optional
  sqlMap.commitTransaction();

  sqlMap.startBatch();
  // … insert 100 rows
  totalRows += sqlMap.executeBatch();  //optional
  sqlMap.commitTransaction();
```

```
  // etc.

} finally {
  sqlMap.endTransaction();
}
```

Wichtige Hinweise zur Stapelverarbeitung:

- Eine Stapelverarbeitung sollte IMMER in einer expliziten Transaktion geschachtelt sein..Wenn sie dies nicht tun, ruft iBATIS jedes Statement individuell auf, so, als ob sie überhaupt keine Stapelverarbeitung gestartet haben.
- Sie können jedes Mapped-Statement in ihrer Stapelverarbeitung aufrufen. Wenn sie unterschiedliche Mapped-Statements (z.B. inserts, danach updates) ausführen, teilt iBATIS dieses in kleinere "Sub-Stapel" abhängig vom letzten generierten SQL-Befehl. Nehmen wir beispielsweise folgenden Code:

```
try {
  sqlMap.startTransaction();
  sqlMap.startBatch();

  sqlMap.insert("myInsert", parameterObject1);
  sqlMap.insert("myInsert", parameterObject2);
  sqlMap.insert("myInsert", parameterObject3);
  sqlMap.insert("myInsert", parameterObject4);

  sqlMap.update("myUpdate", parameterObject5);
  sqlMap.update("myUpdate", parameterObject6);

  sqlMap.insert("myInsert", parameterObject7);
  sqlMap.insert("myInsert", parameterObject8);
  sqlMap.insert("myInsert", parameterObject9);

  sqlMap.executeBatch();
  sqlMap.commitTransaction();
} finally {
  sqlMap.endTransaction();
}
```

  iBATIS führt die gesamte Stapelverarbeitung in drei Sub-Stapeln aus– einer für die ersten vier insert Statements, ein weiterer für die nächsten zwei update Statements, und der dritte für die letzten drei insert Statements. Zwar sind die letzten drei insert Statements gleich zu den ersten vier, iBATIS aber führt trotzdem einen anderen Sub-Stapel aus, da zwischenzeitlich update Statements aufgerufen wurden.

- Die executeBatch() Methode liefert einen int – die Gesamtzahl der betroffenen Zeilen. Gibt es Sub-Stapel, addiert iBATIS die Anzahl der Zeilen der Sub-Stapel zur Gesamtzahl hinzu. Beachten sie bitte, das es erlaubt ist, das der JDBC Treiber scheitern kann die Anzahl der betroffenen Zeilen in einer Stapelverarbeitung zu liefern- in diesem Fall liefert executeBatch() 0 zurück, obwohl einige Zeilen geändert worden sein können.. Der Oracle Treiber ist dafür bekannt, das er sich so verhält..
- Seit iBATIS Version 2.2.0 , gibt es eine weitere Methode zur Stapelverarbeitung - *executeBatchDetailed*. Diese funktioniert ähnlich wie die reguläre Methode executeBatch (erfordert eine explizite Transaktion, verwendet Sub-Stapel, usw.), aber es gibt detailliertere Informationen über die Zeilenzahl. Die *executeBatchDetailed* Methode liefert eine Liste von BatchResult Objekten – eines für jeden Sub-Stapel. Jedes BatchResult Objekt enthält Informationen zum Statement des Sub-Stapels als auch ein int[] vom JDBC Treiber. Wenn eine java.sql.BatchUpdateException auftritt, wirft die Methode eine BatchException mit der

Information welches Statement die Exception verursacht hat und eine Liste BatchResult Objekte der erfolgreichen Sub-Stapel.

Ausführung von Statements mit der SqlMapClient API

SqlMapClient stellt eine API zur Verfügung um Mapped-Statements auszuführen. Diese Methoden sind:

```
        public Object insert(String statementName, Object parameterObject)
        throws SQLException

        public Object insert(String statementName) throws SQLException

        public int update(String statementName, Object parameterObject)
        throws SQLException

        public int update(String statementName) throws SQLException

        public int delete(String statementName, Object parameterObject)
        throws SQLException

        public int delete(String statementName) throws SQLException

        public Object queryForObject(String statementName,
        Object parameterObject)
        throws SQLException

        public Object queryForObject(String statementName) throws SQLException

        public Object queryForObject(String statementName,
        Object parameterObject, Object resultObject)
        throws SQLException

        public List queryForList(String statementName, Object parameterObject)
        throws SQLException

        public List queryForList(String statementName) throws SQLException

        public List queryForList(String statementName, Object parameterObject,
        int skipResults, int maxResults)
        throws SQLException

        public List queryForList(String statementName, int skipResults, int
maxResults)
        throws SQLException

        void queryWithRowHandler (String statementName,
        Object parameterObject, RowHandler rowHandler)
        throws SQLException

        void queryWithRowHandler (String statementName, RowHandler rowHandler)
        throws SQLException

        public PaginatedList queryForPaginatedList(String statementName,
        Object parameterObject, int pageSize)
        throws SQLException

        public PaginatedList queryForPaginatedList(String statementName,
        int pageSize) throws SQLException

        public Map queryForMap (String statementName, Object parameterObject,
               String keyProperty)
               throws SQLException
```

```
public Map queryForMap (String statementName, Object parameterObject,
       String keyProperty, String valueProperty)
       throws SQLException

public void flushDataCache()

public void flushDataCache(String cacheId)
```

In jedem Fall ist der *name* eines Mapped Statement der erste Parameter. Dieser Name entspricht dem name Attribut des Statement-Elements wie bereits beschrieben (<insert>, <update>, <select>, etc.). Weiterhin kann immer optional ein Parameter Objekt angegeben werden. Es kann auch ein null Parameter Objekt angegeben werden, wenn kein Parameter erwartet wird, ansonsten ist es erforderlich. Ab iBATIS 2.2.0 sind viele Methoden entsprechend überladen, wenn kein Parameter Objekt erwartet wird. Dies sind die Gemeinsamkeiten. Die Unterschiede werden nachfolgend beschrieben.

insert(), update(), delete(): Diese Methoden sind spezifisch für update Statements (keine Select-Abfragen). Daher ist es nicht möglich ein update Statement mit einer Abfrage-;Methode auszuführen., Dies ist allerdings Semantik- und Treiber abhängig. Im Fall von executeUpdate(), wird das Statement einfach ausgeführt und die Anzahl der betroffenen Zeilen zurückgeliefert.

queryForObject(): Es gibt zwei Arten von executeQueryForObject(), eine liefert ein neu erzeugtes Objekt, die andere verwendet ein zuvor instantiiertes Objekt das als Parameter beim Aufruf mitgeliefert wird. Letzteres ist hilfreich bei Objekten die durch mehr als ein Statement gefüllt werden.

queryForList(): Es gibt vier Arten von queryForList(). Beim ersten wird eine Abfrage ausgeführt und die Resultate der Abfrage zurückgeliefert. Die zweite ist wie die erste, erwartet aber kein Parameter Objekt. Die dritte erlaubt die Angabe einer Anzahl von Elementen die übersprungen werden sollen(Startpunkt) und die maximale Anzahl von zu liefernden Datensätzen. Dies ist wertvoll wenn sie extrem große Datensätze haben und diese nicht in der Gesamtheit bearbeiten möchten. Die vierte ist wie die dritte, erwartet aber kein Parameter Objekt.

queryWithRowHandler(): Diese Methode erlaubt ihnen das Result-Set Zeile für Zeile zu verarbeiten. Diese Methode erwartet typischerweise den Namen und ein Parameter Objekt, und zusätzlich einen RowHandler. Ein RowHandler ist die Instanz einer Klasse die das RowHandler Interface implementiert. Das RowHandler Interface besitzt nur eine Methode:

```
public void handleRow (Object valueObject);
```

Diese Methode wird für jeden Datensatz (Zeile) der Datenbank aufgerufen. Das valueObjekt ist das aufgelöste Java Objekt der aktuellen Zeile. Dies ist ein sauberer, einfacher und skalierbarer Weg um Resultate einer Abfrage zu verarbeiten. Damit können sie auf jedes Objekt individuell bei einer Abfrage reagieren ohne dass iBATIS eine Liste füllen und diese in Gänze zurück liefern muss. Dies ist somit der effizienteste Weg um mit sehr großen Result-Sets umzugehen und Speicher zu sparen.

Ein Beispiel für einen RowHandler folgt.

queryForPaginatedList(): Diese wertvolle Methode liefert eine Liste in der komfortabel vorwärts und rückwärts navigiert werden kann.. Dieses findet häufig Einsatz bei User Interfaces die nur einen Ausschnitt einer viel größeren Anzahl von Datensätzen anzeigen möchten. Ein Beispiel das wahrscheinlich jeder kennt, ist eine Web Suchmaschine die insgesamt 10,000 Datensätze findet, aber nur 100 zur Zeit anzeigen möchte. Das PaginatedList interface verfügt über Methoden zum seitenweisen Navigieren (nextPage(), previousPage(), gotoPage()) und zur Prüfung des Status einer Seite (isFirstPage(), isMiddlePage(), isLastPage(), isNextPageAvailable(), isPreviousPageAvailable(), getPageIndex(), getPageSize()). Zwar ist die Gesamtzahl der Datensätze per Interface nicht verfügbar, kann aber leicht durch ein zweites Statement abgefragt werden.

queryForMap(): Diese Methode stellt eine Alternative zur Verfügung um eine Menge von Datensätzen zu liefern. Es lädt alle Datensätze in eine Map. Der Schlüssel für die Map wird im Parameter Objekt in der

keyProperty Eigenschaft mitgeliefert. Beispielsweise könnten sie eine Sammlung von Employee Objekten laden, sie können diese aber auch in eine Map laden mit der employeeNumber Eigenschaft als Schlüssel .Die Werte in der Map können entweder das ganze employee Objekt, oder eine weitere Eigenschaft des employee Objekts, spezifiziert durch einen optionalen zweiten Parameter namens valueProperty. Beispielsweise könnten sie eine Map von employee Namen mit den employee number als Schlüssel laden. Verwechseln sie diese Methode nicht mit dem Map Type bei einem Resultat-Objekt. Diese Methode kann bei jeder JavaBean oder einer Map (oder ein primitiver Wrapper, aber das wäre nutzlos) für ein Resultat-Objekt angewendet werden.

flushDataCache(): Diese Methode erlaubt ihnen den Datencache programmatisch zu löschen. Der Aufruf ohne Argumente leert alle Datencaches. Die zweite Methode erwartet eine cacheId als Parameter und leert nur den benannten Datencache. Sie müssen die cacheId mit dem Namespace-Namen angeben (selbst wenn **useStatementNamespaces** auf **false** gesetzt ist).

Beispiel 1: Ausführung von Update (insert, update, delete)

```
sqlMap.startTransaction();

Product product = new Product();
product.setId (1);
product.setDescription ("Shih Tzu");

Integer primKey = (Integer)sqlMap.insert ("insertProduct", product);

sqlMap.commitTransaction();
```

Beispiel 2: Ausführung von Query for Object (select)

```
sqlMap.startTransaction();

Integer key = new Integer (1);

Product product = (Product)sqlMap.queryForObject ("getProduct", key);

sqlMap.commitTransaction();
```

Beispiel 3: Ausführung von Query for Object (select) mit zuvor allozierten Result Objekt

```
sqlMap.startTransaction();

Customer customer = new Customer();

sqlMap.queryForObject("getCust", parameterObject, customer);
sqlMap.queryForObject("getAddr", parameterObject, customer);

sqlMap.commitTransaction();
```

Beispiel 4: Ausführung Query for List (select)

```
sqlMap.startTransaction();

List list = sqlMap.queryForList ("getProductList");

sqlMap.commitTransaction();
```

Beispiel 5: Auto-commit

```
  // Wird startTransaction nicht aufgerufen, wird für die statements
  // auto-commit angenommen.  Aufruf von commit/rollback ist nicht
notwendig.
  Integer primKey = (Integer)sqlMap.insert ("insertProduct", product);
```

Beispiel 6: Ausführung Query for List (select) mit Resultat Grenzen

```
  sqlMap.startTransaction();

  List list = sqlMap.queryForList ("getProductList", 0, 40);

  sqlMap.commitTransaction();
```

Beispiel 7: Ausführung Query mit einem RowHandler (select)

```
  public class MyRowHandler implements RowHandler {
    private SqlMapClient sqlMap;
    public MyRowHandler(SqlMapClient sqlMap) {
      this.sqlMap = sqlMap;
    }

    public void handleRow (Object valueObject)
                  throws SQLException {
      Product product = (Product) valueObject;
      product.setQuantity (10000);
      sqlMap.update ("updateProduct", product);
    }

  }

  sqlMap.startTransaction();

  RowHandler rowHandler = new MyRowHandler(sqlMap);
  sqlMap.queryWithRowHandler ("getProductList", rowHandler);

  sqlMap.commitTransaction();
```

Beispiel 8: Ausführung Query for Paginated List (select)

```
  PaginatedList list =
            sqlMap.queryForPaginatedList ("getProductList", 10);

  list.nextPage();
  list.previousPage();
```

Beispiel 9: Ausführung Query for Map

```
sqlMap.startTransaction();

Map map = sqlMap.queryForMap ("getProductList", null, "productCode");

sqlMap.commitTransaction();

Product p = (Product) map.get("EST-93");
```

Loggen von SqlMap Aktivitäten

Das SqlMap Framework unterstützt Log-Informationen durch eine interne Log Fabrik. Die Log-Fabrik reicht Log-Informationen an einer der folgenden Implementierungen weiter:

- Jakarta Commons Logging (JCL – *NICHT Job Control Language!*)
- Log4J
- JDK logging (JRE 1.4 oder grö0er erforderlich)

Die iBATIS Log Fabrik wählt zur Laufzeit die erste Implementierung, die es findet (in oben angegebener Reihenfolge). Findet iBATIS keine der oben genannten Implementierung ist Logging nicht verfügbar.

In vielen Umgebungen wird JCL als Teil der Applikation im Server Klassenpfad bereits mitgeliefert (Beispiele hier sind zum Beispiel Tomcat oder WebSphere). In derartigen Umgebungen verwendet iBATIS daher JCL als Logging-Implementierung. In Umgebungen wie WebSphere wird also ihre Log4J Konfiguration nicht genommen, weil WebSphere seine eigene, proprietäre Implementierung von JCL benutzt. Dies kann zur frustrierenden Suche führen, weil es so aussieht als ob iBATIS die Log4J Konfiguration ignorieren würde (in der Tat *ignoriert* iBATIS tatsächlich die Log4J Konfiguration weil iBATIS stattdessen die JCL Umgebung nutzt).

Läuft ihre Applikation in einer Umgebung in der JCL bereits im Klassenpfad enthalten ist, möchten aber eine andere Log-Implementierung verwenden, so können sie, seit iBATIS Version 2.2.0, eine andere Implementierung durch folgende Aufrufe anfordern

```
com.ibatis.common.logging.LogFactory.selectLog4JLogging();
com.ibatis.common.logging.LogFactory.selectJavaLogging();
```

Diese Methoden sollten aufgerufen werden, bevor irgendeine andere iBATIS Methode benutzt wird. Diese Methoden wechseln nur dann auf die angegebene Log-Implementierung wenn diese auch im Klassenpfad enthalten ist. Wenn sie also Log4J als Logging-Framework wählen, dies aber zur Laufzeit nicht verfügbar ist, ignoriert iBATIS die Anforderung und nutzt das übliche Verfahren (siehe oben) um eine geeignete Logging-Implementierung zu finden.

Die Besonderheiten von Jakarta Commons Logging, Log4J oder JDK 1.4 Logging API sind nicht Bestandteil dieser Beschreibung. Der folgende kurze Abriss einer Beispiel-Konfiguration sollte allerdings helfen. Wenn sie mehr über diese Frameworks wissen möchten, verweisen wir auf die Informationen an folgenden Stellen:

Jakarta Commons Logging
- http://jakarta.apache.org/commons/logging/index.html

Log4J
- http://jakarta.apache.org/log4j/docs/index.html

JDK 1.4 Logging API
- http://java.sun.com/j2se/1.4.1/docs/guide/util/logging/

Log Konfiguration

iBATIS logged die meisten Aktivitäten von Klassen, die nicht innerhalb der iBATIS Packages liegen. Um iBATIS Logging Statements zu sehen, müssen sie das Logging für Klassen aus dem java.sql package einschalten – genauer:

- java.sql.Connection
- java.sql.PreparedStatement
- java.sql.Resultset
- java.sql.Statement

Wie dies genau gemacht wird hängt von der Log-Implementierung ab. Wir zeigen ihnen aber wie das mit Log4J gemacht werden kann.

Das Einrichten der Log-Implementierung besteht darin eine oder mehrere Konfigurationsdatei(en) (z.B. log4j.properties) und eine JAR Datei (z.B. log4j.jar) einzurichten. Das folgende Beispiel konfiguriert das vollständige Logging mit Log4J . Es besteht aus 2 Schritten.

Schritt 1: Die Log4J JAR Datei aufnehmen

Um Log4J verwenden zu können, müssen sie die entsprechende JAR Datei in ihrem Applikation-Klassenpfad aufnehmen. Sie können Log4J von der oben angegebenen URL herunterladen. Bei Web- oder Enterprise-Applikationen können sie log4j.jar in das WEB-INF/lib Verzeichnis kopieren, oder bei eigenständigen Applikationen diese in den JVM -Klassenpfad beim Start angeben.

Schritt 2: Einrichten von Log4J

Das Einrichten von Log4J ist einfach – sie erzeugen eine Datei mit dem Namen log4j.properties, die folgendermaßen aussieht:

log4j.properties

```
1  # Globale Logging Konfiguration
2  log4j.rootLogger=ERROR, stdout
3
4  # SqlMap Logging Konfiguration...
5  #log4j.logger.com.ibatis=DEBUG
6  #log4j.logger.com.ibatis.common.jdbc.SimpleDataSource=DEBUG
7  #log4j.logger.com.ibatis.sqlmap.engine.cache.CacheModel=DEBUG
8  #log4j.logger.com.ibatis.sqlmap.engine.impl.SqlMapClientImpl=DEBUG
9  #log4j.logger.com.ibatis.sqlmap.engine.builder.xml.SqlMapParser=DEBUG
10 #log4j.logger.com.ibatis.common.util.StopWatch=DEBUG
11 #log4j.logger.java.sql.Connection=DEBUG
12 #log4j.logger.java.sql.Statement=DEBUG
13 #log4j.logger.java.sql.PreparedStatement=DEBUG
14 #log4j.logger.java.sql.ResultSet=DEBUG
15
16 # Konsole Ausgabe...
17 log4j.appender.stdout=org.apache.log4j.ConsoleAppender
18 log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
19 log4j.appender.stdout.layout.ConversionPattern=%5p [%t] - %m%n
```

Diese Datei ist dahingehend minimal weil nur Fehler gemeldet werden. Zeile 2 veranlasst Log4J lediglich Fehler über den Standardausgabe-Appender auszugeben. Ein Appender nimmt Ausgaben für ein bestimmtes Ausgabegerät (z.B.. Console, Datei, Datenbank, etc..) entgegen. Um den Level der Ausgabemenge zu beeinflussen, können sie die Zeile 2 wie folgt ändern:

```
log4j.rootLogger=DEBUG, stdout
```

Durch diese Änderung, veranlassen sie, das Log4J nun alle Ereignisse auf den Standardausgabe-Appender (console) ausgibt. Sie können dass noch feiner einstellen und jede Klasse aus dem 'SqlMap logging Konfiguration' Abschnitt aus der Datei auskommentieren (Zeilen 5 bis 14). Wenn sie also Aktivitäten mit einem PreparedStatement auf der Konsole im DEBUG Level sehen möchten (SQL Befehle) , müssen sie einfach Zeile 14 wieder scharf schalten (beachten sie dass die Zeile nicht mehr kommentiert ist):

```
log4j.logger.java.sql.PreparedStatement=DEBUG
```

Der Rest der log4j.properties Konfigurationsdatei richtet Appender ein, die nicht Bestandteil dieser Dokumentation ist. Sie können aber weitere Informationen von der Log4J Website (URL oben) erhalten.

Alles über JavaBeans auf EINER SEITE

Das Data Mapper Framework erfordert ein grundlegendes Verständig über JavaBeans. Glücklicherweise ist das nicht viel. Hier also eine kurze Einführung in JavaBeans, wenn sie dieses nicht bereits wissen.

Was ist eine JavaBean? Ein JavaBean ist eine Klasse die sich strikt an eine Namenskonvention hält um den Status dieser Klasse zu ändern. Eine andere Art dies auszudrücken, ist es zu sagen, dass JavaBeans einer bestimmten Konvention folgen um Eigenschaften zu setzen oder zu laden (Getter/Setter). Auf Eigenschaften von JavaBeans wird über Methoden zugegriffen und nicht direkt über die Felder. Methoden, die mit "set" beginnen, definieren schreibbare Eigenschaften (z.B. setEngine), während Methoden die mit "get" beginnen, lesbare Eigenschaften definieren (z.B. getEngine). Bei booleschen Eigenschaften kann die Methode auch mit "is" (z.B. isEngineRunning) beginnen. Set Methoden sollten keinen Rückgabetyp besitzen (also vom Typ void sein), und sollten nur einen Parameter vom Typ der Eigenschaft (z.B. String) erwarten. Get Methoden sollten entsprechend den Wert als Typ der Eigenschaft zurück liefern und keinen Parameter erwarten. Es ist zwar nicht zwingend erforderlich, aber die Typen bei Get- und Set-Methoden sollten übereinstimmen. JavaBeans sollten auch das Serializable Interface implementieren. JavaBeans unterstützen noch weitere Funktionen (events etc.), und müssen über einen argumentlosen Konstruktor verfügen; sind aber im Zusammenhang mit dem Data Mapper oder einer Web-Applikation unwichtig. Daher ist folgende Klasse ein Beispiel für eine JavaBean:

```
public class Product implements Serializable {

  private String id;
  private String description;

  public String getId() {
     return id;
  }
  public void setId(String id) {
     this.id = id;
  }

  public String getDescription() {
     return description;
  }
  public void setDescription(String description) {
     this.description = description;
  }
}
```

Achtung! Bringen sie nicht die Typen bei den get und set Methoden für eine bestimmte Eigenschaft durcheinander. Für eine ganzzahlige Eigenschaft "account" beispielsweise sollten die Methoden für die beiden getter und setter Methoden heißen:

```
public void setAccount (int acct) {….}
public int getAccount () {….}
```

Beide verwenden den Typ "int". Würde die get-Methode stattdessen "long" liefern, gäbe dies Probleme.

Achtung! Beachten sie auch nur eine Methode mit Namen getXxxx() oder setXxxx() zu haben. Seien sie also sparsam mit derartigen polymorphen Methoden. Derartige Methoden sollten sie einen aussagekräftigeren Namen spendieren.

Achtung! Bei booleschen Eigenschaften gibt es eine alternative Methoden zum Laden der Eigenschaft.. Die get Methode kann auch in der Form isXxxxx() angegeben werden. Beachten sie, entweder eine "is" Methode oder eine "get" Methode zu haben, aber nicht beide!

Glüchwunsch! Sie haben den Kursus bestanden!

Na gut, zwei Seiten

Zusatz: Objekt Navigation (JavaBeans Properties, Maps, Lists)

In diesem Dokument haben sie bisher kennen gelernt auf Objekte zuzugreifen, die ihnen aus anderen JavaBean kompatiblen Frameworks (struts,..) vertraut vorkommen mag. Das Data Mapper Framework erlaubt es in Objekt Graphen mit Hilfe von JavaBean Eigenschaften, Maps (key/value) oder Listen zu navigieren,. Beachten sie folgende Navigation (mit einer List, einer Map und einer JavaBean):

```
Employee emp = getSomeEmployeeFromSomewhere();
((Address) ( (Map)emp.getDepartmentList().get(3) ).get ("address")).getCity();
```

In den Eigenschaften des employee Objektes kann wie folgt navigiert werden:

```
"departmentList[3].address.city"
```

Wichtig: Diese Art der Syntax wird von iBATIS' nur mit dynamischen SQL Elementen unterstützt. Dieses geht nicht bei Eigenschaften von <result> oder <parameter> Abbildungen.

Ressourcen (com.ibatis.common.resources.*)

Die Resources Klasse stellt Methoden bereit, mit denen Ressourcen einfach vom Klassenpfad geladen werden können. Das Hantieren mit Classloader ist herausfordernd, besonders im Umfeld eines Applikation Server/Container. Die Resources Klasse versucht diese oftmals lästige Arbeit zu vereinfachen.

Häufige Anwendungsgebiete für Ressourcen-Dateien sind:

- Laden der SQL Map Konfiguration (z.B. sqlMap-config.xml) vom Klassenpfad.
- Laden diverser *.properties Dateien vom Klassenpfad.
- Etc.

Es gibt verschiedene Arten wie eine Ressource geladen werden kann, zum Beispiel:

- Durch einen Reader: Für einfache Nur-Lese Textdaten.
- Durch einen InputStream: Für einfache Binärdaten.
- Durch ein File: Zum Lesen/Schreiben von binären oder Textdateien.
- Durch eine Properties Datei : Zum Nur-Lesen von Konfigurationsdateien (properties-Format).

Die entsprechenden Methoden der Resources Klasse:

```
Reader getResourceAsReader(String resource);
InputStream getResourceAsStream(String resource);
File getResourceAsFile(String resource);
Properties getResourceAsProperties(String resource);
```

In jedem Fall wird zum Laden der Ressource der gleiche Klassenlader benutzt, der die Resources Klasse geladen hat. Schlägt das fehl wird der System-Classloader aufgerufen. In einer Umgebung in der es Schwierigkeiten mit dem ClassLoader gibt (z.B. mit einigen Applikationsserver), können sie einen spezifischen ClassLoader angeben (z.B. nutzen sie einen besonderen ClassLoader um Klassen ihrer eigenen Applikation Klassen zu laden). Jede der oben genannten Methoden hat eine alternative Methode die den ClassLoader als ersten Parameter,angegeben bekommt. Es sind:

```
Reader getResourceAsReader (ClassLoader classLoader, String resource);
InputStream getResourceAsStream (ClassLoader classLoader, String resource);
File getResourceAsFile (ClassLoader classLoader, String resource);
Properties getResourceAsProperties (ClassLoader classLoader, String resource);
```

Der Name des *resource P*arameters sollte voll qualifiziert sein (package Name mit vollständiger Name der Datei/Ressource). Wenn sie also zum Beispiel die Ressource 'com.domain.mypackage.MyPropertiesFile.properties'in ihrem Klassenpfad haben, können sie diese als Properties Datei mit der Resources Klasse wie folgt laden (beachten sie, das die Ressource nicht mit einem Schrägstrich "/" startet):

```
String resource = "com/domain/mypackage/MyPropertiesFile.properties";
Properties props = Resources.getResourceAsProperties (resource);
```

Ebenso können sie ihre SqlMap Konfigurationsdatei vom Klassenpfad laden. Nehmen wir an, diese sei in der Datei sqlMap-config.xml im properties Package enthalten:

```
String resource = "properties/sqlMap-config.xml";
Reader reader = Resources.getResourceAsReader(resource);
SqlMapClient sqlMap = SqlMapClientBuilder.buildSqlMap(reader);
```

Internationalisieren von Ressourcen

Hinweise: Diese Information gilt ab iBATIS Version 2.3.

Für iBATIS ist der Hauptbelang, soweit es um Internationalisierung geht, die XML Konfigurationsdateien. Haben diese eine ungewöhnliche Kodierung und entsprechen nicht der systemweiten Kodierung, kann es zu Fehlern kommen. iBATIS stellt zwei verschiedene Lösungen bereit.

Internationalisierung mit Character Reader

Verwenden sie einen Reader, nutzt iBATIS die Java Klasse InputStreamReader um Dateien zu encodieren. Voreingestellt verwendet die Klasse das systemweite Encoding und ignorierte das aktuelle Encoding der Datei. In einigen Umgebungen harmoniert das systemweite Encoding nicht gut mit dem Unicode-Encoding in XML Dateien. Wenn sie Probleme beim Parsen von XML Dateien, in denen ein Reader die Eingabe ist, beobachten, können sie das voreingestellte Encoding ändern, damit es zum Encoding der XML Datei passt,. Zum Beispiel:

```
String resource = “properties/sqlMap-config.xml”;
Resources.setCharset(Charset.forName('UTF-8”));   // ändere das Encoding
Reader reader = Resources.getResourceAsReader(resource);
SqlMapClient sqlMap = SqlMapClientBuilder.buildSqlMap(reader);
```

Die “setCharset” Methode ändert das Encoding für alle weiteren Aufrufe von “getResourceAsReader”. Wenn sie wieder zur systemweiten Voreinstellung zurückkehren möchten, rufen sie einfach “setCharset(null)” auf.

Internationalisierung mit Byte Input Streams

Wenn sie über einen Byte-InputStream eine XML Konfigurationsdatei laden ,wird der Parser das Datei-Encoding automatisch erkennen. Dieses ist oft der beste Weg::

```
String resource = “properties/sqlMap-config.xml”;
InputStream inputStream = Resources.getResourceAsStream(resource);
SqlMapClient sqlMap = SqlMapClientBuilder.buildSqlMap(inputStream);
```

Diese Methode verlässt sich auf die natürliche Unterstützung der Zeichen-Encoding des Parsers. Bei Schwierigkeiten lesen sie in der Dokumentation des Parsers wie sie das voreingestellte Schema überschreiben können.

SimpleDataSource (com.ibatis.common.jdbc.*)

Die SimpleDataSource Klasse ist eine *einfache* Implementierung für eine JDBC 2.0 konforme DataSource. Sie enthält einen Connection-Pool und ist synchronisiert (erzeugt keine weiteren Threads). Dieses macht sie zu einer leicht gewichtigen und portablen Lösung für einen Connection Pool. SimpleDataSource wird genau so verwendet wie jede andere JDBC DataSource Implementierung und wird als Teil der JDBC Standard Extensions API dokumentiert, die sie hier finden können:
http://java.sun.com/products/jdbc/jdbc20.stdext.javadoc/

Achtung!: Die JDBC 2.0 API ist Bestandteil von J2SE 1.4.x

Achtung!: SimpleDataSource ist recht praktisch, effizient und effektiv. Für unternehmenskritische Anwendungen empfehlen wir aber eine unternehmensweite, strategische DataSource Implementierung (wie sie bei Applikations-Server oder kommerziellen O/R Mapping Werkzeugen vorkommen) zu nehmen.

Der Konstruktor von SimpleDataSource erwartet einen Properties Parameter, der eine Menge von Konfigurationswerten enthält. Folgende Tabelle beschreibt die Eigenschaften. Nur Eigenschaften mit "JDBC." sind unbedingt erforderlich.

Eigenschaftsname (Property Name)	Erforderlich	Voreinstellung	Beschreibung
JDBC.Driver	Ja	entfällt	Der Klassenname für den JDBC Treiber.
JDBC.ConnectionURL	Ja	entfällt	Die Verbindung-URL für die JDBC Verbindung.
JDBC.Username	Ja	entfällt	Der Benutzername, um sich an der Datenbank anzumelden.
JDBC.Password	Ja	entfällt	Das Passwort, um sich an der Datenbank anzumelden.
JDBC.DefaultAutoCommit	Nein	Treiber abhängig	Die Einstellung, ob autocommit, für alle Verbindungen die der Pool liefert, voreingestellt ist,.
Pool.MaximumActiveConnections	Nein	10	Maximale Anzahl von gleichzeitig offenen Verbindungen.
Pool.MaximumIdleConnections	Nein	5	Anzahl von möglichen untätigen, inaktiven Verbindungen im Pool.
Pool.MaximumCheckoutTime	Nein	20000	Die maximale Zeit (in Millisekunden) die eine Verbindung ausgecheckt bleiben darf, bevor sie wieder eingesammelt wird.

Pool.TimeToWait	Nein	20000	Die Zeit, die ein Client auf eine Connection warten darf (weil alle in Benutzung sind). Die Zeitangabe ist die maximale Zeit (in Millisekunden), die der Thread wartet bevor er erneut versucht eine Verbindung zu erhalten. Es ist gut möglich, das eine Verbindung zwischenzeitlich an den Pool zurückgegeben wird und den wartenden Thread benachrichtigt. Daher muss der Thread nicht die ganze Zeit warten. Der Wert gibt also nur die maximal zulässige Wartezeit an.
Pool.PingQuery	Nein		Die Ping Abfrage wird gegen die Datenbank gesendet um die Verbindung zu testen. In Umgebungen, in denen Verbindungen nicht stabil sind, kann mit der Ping Abfrage garantiert werden, dass Verbindungen aus dem Pool immer eine noch gültige Verbindung zur Datenbank darstellen. Dieses kann allerdings großen Einfluss auf die Performance haben. Achten sie daher auf die Einrichtung der Ping Abfrage und testen sie dieses ausgiebig.

SimpleDataSource (fortgesetzt…)

Pool.PingEnabled	Nein	false	Ein-oder Abschalten einer Ping Abfrage, um zu erkennen ob die Datenbank erreichbar ist. Für die meisten Applikationen ist eine Ping Abfrage nicht notwendig.
Pool.PingConnectionsOlderThan	Nein	0	Verbindungen, die älter als der angegebene Wert (in Millisekunden, sind, werten mit der Ping Abfrage getestet. Dies ist hilfreich in Umgebungen, in der die Datenbank Verbindungen selbständig schließt (z.B. nach 12 Stunden).
Pool.PingConnectionsNotUsedFor	Nein	0	Verbindungen, die für die angegebene Zeit (in Millisekunden), inaktiv waren ,werden mit der Ping Abfrage getestet. Dies ist hilfreich in Umgebungen, in der die Datenbank inaktive Verbindungen selbständig schließt (z.B. nach 12 Stunden).
*Driver.**	Nein		Viele JDBC Treiber unterstützen Konfigurationsparameter. Diese können angegeben werden, indem solchen Parametern "Driver." vorangestellt wird. Hat beispielsweise der Treiber die Eigenschaft "compressionEnabled" können sie SimpleDataSource konfigurieren, indem sie die Eigenschaft „Driver.compressionEnabled=true" angeben. Hinweis: Diese Eigenschaft können auch in der sqlMap-config.xml Datei angegeben werden.

Beispiel: Verwendung von SimpleDataSource

```
// properties normalerweise aus einer Datei geladen
DataSource dataSource = new SimpleDataSource(props);
Connection conn = dataSource.getConnection();
// ... Datenbank Abfrage und Änderungen
conn.commit();
// Verbindungen von der SimpleDataSource werden wieder an den Pool zurückgegeben wenn sie geschlossen werden
conn.close();
```