Persistenz Layer Tutorial

1 Persistenz-Layer

Dieser Persistenz Layer ist eine generische Komponente, mit der definierbare Daten aus einer relationalen Datenbank entnommen werden können und verändert wieder zurückgeschrieben. Zusätzlich kann das generische Transfermodell `JDataSet` an die Oberfläche gebunden werden.

Konzeptionell lehnt sich dieses Framework teilweise an ADO.NET an; geht aber insoweit darüber hinaus, als hier die Zugriffe auf die Datenbank nicht mehr programmiert, sondern definiert werden.

Dieses ist kein Objekt-Relationales Mapping; dieses ist auch kein Code-Generator.

1.1 Highlights

Der hier vorgestellte Persistenz-Layer für Java-Anwendungen zeichnet sich durch folgende Features/Alleinstellungsmerkmale aus:

- Generischer Persistenz-Layer für Dialog-Anwendungen.
- Die Datenbankzugriffschicht einer Anwendung wird hier in XML definiert, und nicht programmiert.
- Läuft mit allen Datenbanken, für die ein JDBC-Treiber verfügbar ist.
- Nutzung des Persistenz-Layers über SOAP für verteilte Anwendungen.
- Anbindung der Oberfläche (GuiBuilder) an das generische Transfermodell.
- Durch bescheidene Ansprüche an Ressourcen (Bandbreite, Latenzzeiten) auch im Internet einsetzbar.
- Bescheidene Anforderungen an die Server-Hardware (Windows 2000 oder Linux) für bis zu 1000 Nutzer.
- Hohe Performanz durch Wiederverwendung der Datenbankzugriffe.
- Kurze Entwicklungszeiten: Wenn das Datendesign fertig ist, ist auch die Anwendung fertig.
- Paralleler Zugriff auf mehrere Datenbanken, auch auf verschiedenen Servern.
- Hier wird alles weggelassen, was umständlich, teuer, langsam oder Ressourcen-fressend ist: Kein Einsatz von J2EE, kein Code-Generator, kein Objekt-Relationales Mapping, kein CORBA, kein RMI.

Der Anwendungsrealisierer wird so massiv von Routine-Tätigkeiten entlastet: Die Oberfläche und der Zugriff auf die Daten muss nicht mehr programmiert werden; er kann sich so auf die eigentliche Arbeit konzentrieren: Die Implementierung der Geschäftslogik.

Die Anwendung ist leicht änderbar: Auch massive Änderungen/Ergänzungen an der Datenstruktur und die damit verbundene Änderungen an der Oberfläche werden ohne jede Programmierung rein deklarativ vorgenommen. Diese Änderungen kann ein geschulter Anwender ohne Programmierkenntnisse selbst vornehmen.

Da alle Informationen in XML-Dokumenten abgelegt werden, können im Prinzip auch voll generische Anwendungen realisiert werden, also die Datenzugriffe und die Oberfläche zur Laufzeit generieren – und das ohne Performanzverlust.

1.2 Was macht der Persistenz-Layer?

Der Persistenz-Layer ist eine generische Komponente zur Speicherung von Daten in einer (relationalen) Datenbank.

Üblicherweise umfasst ein Datenmodell ein Vielzahl von Tabellen, die auf komplexe Art untereinander in Beziehung stehen. Für einen konkreten Anwendungsfall werden zumeist mehrere Datensätze aus verschiedenen Tabellen der Datenbank angefordert.

Wird diese Zusammenstellung hart kodiert in der Software vorgenommen, so ist das wenig flexibel. Bei der „klassischen" Programmierung führen selbst kleine Änderungen (Beispiel: Der Datentyp eines Feldes soll von ganzzahlig auf zwei Nachkommastellen geändert werden) bei mittleren und größeren Projekten zu einem beeindruckenden Aufwand bei zweifelhafter Stabilität. Hierfür haben Endanwenden und Laien wenig Verständnis.

Das nächste Problem besteht darin, wie die Daten zwischen den verschiedenen Schichten der Anwendung transportiert werden. Hierfür gibt es verschiedene Ansätze:

1. RPC
   Die Geschäftsobjekte werden als „Remote Object" implementiert. Diese bedeutet, dass sie eine Vielzahl von Methoden (getName, setName) im Netz bereitstellen, auf die die Clients Zugriff haben.
   Diese Vorgehen (CORBA „classic") hat sich als nicht praktikabel erwiesen, da sehr hohe Netzlast erzeugt wird.
2. Copy by value
   Die Geschäftsobjekte werden aus den verschiedenen Tabellen „zusammengebaut" und als Objekt übertragen („By Value").
   Dieses Vorgehen ist dann problematisch, wenn Objekte sehr groß werden (alle Buchungen eines Kunden) oder wenn es nicht erwünscht ist, die Geschäftsobjekte bis zum Client durchzureichen.
   Bei einer verteilten Anwendung wird ein entsprechendes Protokoll benötigt, welches Objekte über Netzwerke „verschicken" kann (RMI, SOAP, CORBA). Stichwort „Marshalling, Unmarshalling".
3. Transfermodell
   Es werden nicht die Geschäftsobjekte direkt, sondern je nach Geschäftsvorfall nur bestimmte Daten übermittelt; dieses „Transfermodell" wird gesondert implementiert.
   Der Vorteil liegt hier darin, dass je nach Geschäftsvorfall nur die benötigten Daten übermittelt werden; der Nachteil ist, dass eine sehr große Zahl solcher Transfermodelle gibt, und die „Lieferkette" – Datenbank, Geschäftsobjekt, Transfermodell, Client-Oberfläche, Transfermodell, Geschäftsobjekt, Datenbank – sehr lang wird und damit die Implementierung sehr aufwendig wird und wenig flexibel ist.
4. Generisches Transfermodell („DataSet")
   Es wird ein generischer Ansatz für das Transfermodell verfolgt:
   Die Transfermodelle werden nicht einzeln realisiert, sondern es wird ein abstrakter Container implementiert, der beliebige Datenstrukturen beinhalten kann.
   Er stellte außer dem Hinzufügen von Daten und deren Auslesen keine fachliche Funktionalität zur Verfügung.
   Im .NET Framework ist das die Klasse „DataSet". Dieser Container beinhaltet – ähnlich wie eine Datenbank – Tabellen, Spalten, Zeilen und Referenzen.

Der hier implementierte Ansatz verfolgt ein generisches Transfermodell, wobei bei der Definition dieses Modells noch einen Schritt weiter gegangen wird:

Die zentrale Idee besteht darin, dass mit einer Beschreibungssprache definiert wird, welche Daten aus der Datenbank entnommen werden sollen.

Zusätzlich besteht die Möglichkeit, die Daten an die Oberfläche zu binden.

1.3 Installation

jdataset.jar in ein beliebiges Verzeichnis kopieren. Im gleichen Verzeichnis muss eine Datei „PLConfig.xml" vorhanden sein, welche den Zugriff auf die Datenbank definiert.

Selbstredend muss jdataset.jar sowie der JDBC-Treiber zu Datenbank im CLASSPATH Ihrer Anwendung enthalten sein.

Anschließend sind die gewünschten Datenbank mit ihren Datenbankzugriffe zu definieren.

2 Definition der Datenbank-Zugriffs-Schicht

Vorgehen:

- Definition der Datenbank; ER-Modellierung
- Die definierte Datenbank dem Persistenz Layer bekannt machen.
- Datenbankzugriffe definieren
- Bindung an die Oberfläche
- Programmierung des Datenaustauschs zwischen Datenbank und Oberfläche

2.1 Definition der Datenbank (PLConfig.xml)

Diese Einstellungen dienen dem Persistenz-Layer, um Zugriff auf eine bestimmte Datenbank zu erhalten.

Hierzu gehören der zu verwendende JDBC-Treiber, die URL zur Datenbank, sowie Username und Password.

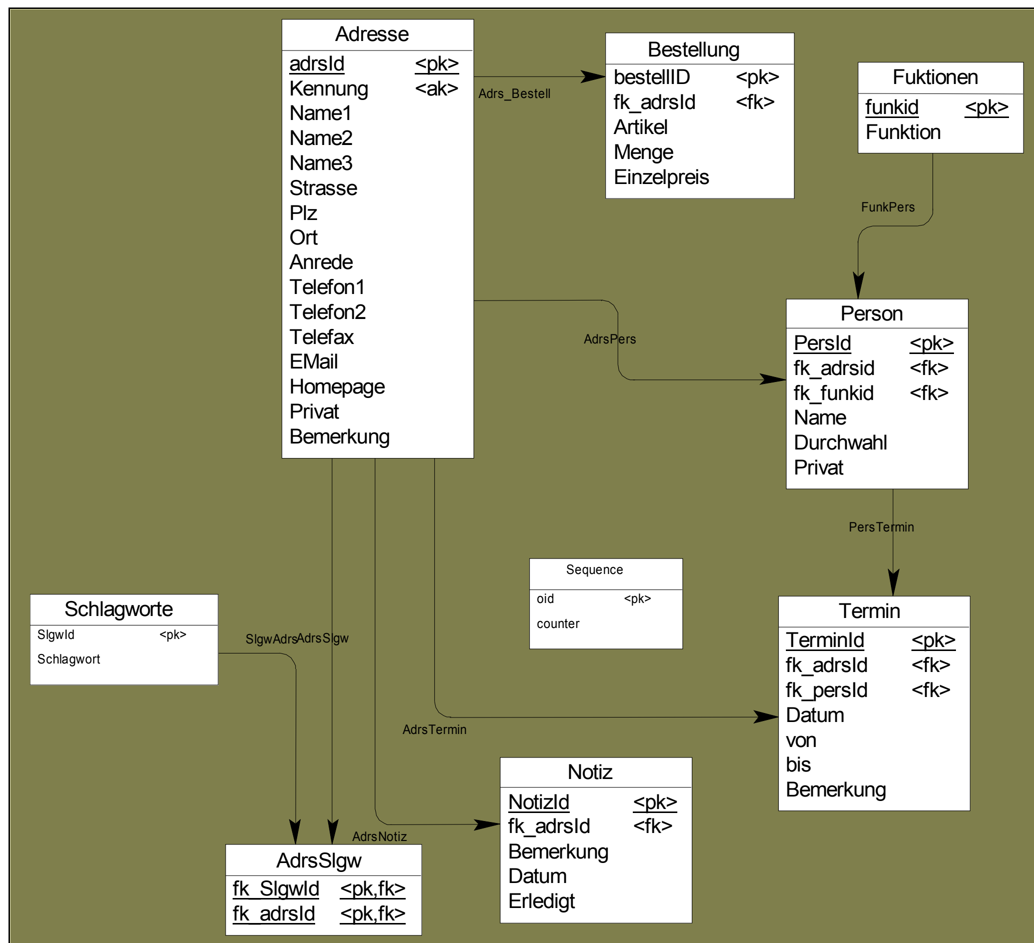
Grundsätzlich ist jede Datenbank die über einen JDBC-Driver verfügt ansprechbar; die Erfahrung lehrt jedoch, dass einige dieser Treiber sehr schlecht implementiert sind. So z.B. der JDBC-ODBC-Driver von Sun, von dessen Einsatz dringend abgeraten wird.

Einige Datenbanken genügen nicht den Minimalanforderungen an eine relationale Datenbank. So ist MySql nur dann sinnvoll einsetzbar, wenn als Tabellentyp **InnoDB** verwendet wird.

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<!DOCTYPE Server SYSTEM 'http://jdataset.de/PLConfig.dtd'>
<Server>
  <!— In dieser Datei sind zu Zugriffe auf die Datenbank definiert->
  <DatasetDefinitionFile>DatabaseConfig.xml</DatasetDefinitionFile>
  <Database>
    <!-- Postgres -->
    <JDBC-Driver>org.postgresql.Driver</JDBC-Driver>
    <URL>jdbc:postgresql://localhost/adrsdemo</URL>
    <!-- Datenbankschema angeben.
     Schema kann weggelassen werden:
           Dann werden die Metadaten aller Schemata eingelesen.
     Auch mehrere Schemata angeben: Dann mit Komma getrennt. -->
    <Schema>public</Schema>
    <Username>myUsername</Username>
    <Password>myPassword</Password>
    <MaxActiveConnections value="30"/>
    <MaxIdleConnections value="15"/>
    <MinIdleConnections value="1"/>
    <!-- 300 000 = 5 Minuten -->
    <ConnectionTimeOut value="300000"/>
    <!-- Ein einfaches Statement zur Überprüfung der Datenbankverbindung
(see DBCP)-->
    <ValidationQuery>"SELECT 1"</ValidationQuery>
```

```
    <!-- Default-Sequence für die Vergabe von Primary Keys -->
    <Sequence>
      <get>SELECT NEXTVAL('counter')</get>
    </Sequence>
    <TransactionIsolationLevel value="TRANSACTION_SERIALIZABLE" />
    <!-- Feldname für optimistisches Locking -->
    <OptimisticLockingField value='version'/>
    <!-- Feldname für User und Datum INSERT -->
    <CreateUserField value="Created"/>
    <!-- Feldname für User und Datum letztes UPDATE -->
    <UpdateUserField value="Updated"/>
  </Database>
  <Dataset>
    <Encoding value='ISO-8859-1'/>
    <ElementName value='DataSet'/>
  </Dataset>
  <Format>
    <DateFormat value='dd.MM.yyyy'/>
    <TimeFormat value='HH:mm'/>
    <TimestampFormat value='dd.MM.yyyy HH:mm:ss'/>
    <DecimalFormat value='#0.00'/>
  </Format>
  <Options>
    <Debug value='false'/>
  </Options>
```

2.2 *Aufbau einer Beispieldatenbank*



Bei der Definition von Datenstrukturen müssen die folgende Einschränkung berücksichtigt werden:

- Jede Tabelle sollte ein einziges Feld als Primärschlüssel haben; üblicherweise ein long.[1]
- Ausnahme ist hier die „klassische“ m-n-Beziehung, die keine doppelten Einträge zulässt, wie in diesem Beispiel die Tabelle „AdrsSlgw“. Hier besteht der Primärschlüssel aus den beiden Fremdschlüsseln.
- In der Datenbank müssen die Foreign Key Constraints vollständig definiert werden.

2.3 Definition der Datenbank-Zugriffe

Die Anforderung besteht darin, die Daten für eine Oberfläche bereitzustellen.

Hierzu dient die Datei „DatabaseConfig.xml“.

In ihr werden alle fachlichen Zugriffe („View“) auf die Datenbank definiert. Diese Zugriffe sollten sich an den Use Cases der Fachanwendung orientieren. Solche Views erhalten einen Namen, unter dem auf sie zugegriffen wird.

Diese Definition leistet dabei folgendes:

Es wird eine Wurzeltabelle („RootTable“) angegeben, die als Ausgangspunkt für die Navigation zu weiteren Tabellen dient. In unseren Beispiel also „Adresse“.[2]

Von diesem Ausgangspunkt wird entlang der im Datenmodell festgelegten Referenzen zu weiteren Tabellen navigiert, die entweder abhängige Daten („Child“) oder übergeordnete sind („Parent“). Von einem „Child“ kann zu weiteren Childs navigiert werden („Enkel“, „Urenkel“); zu einem „Child“ können auch „Parent“ eingelesen werden und deren „Parent“ („Vater“, „Großvater“).

Es ist aber in dieser Implementierung nicht erlaubt, von einem „Parent“ zu dessen „Child“ zu navigieren, da das in aller Regel fachlich keinen Sinn macht und die Gefahr von zyklischen Datenstrukturen besteht. Ein weitere Einschränkung ist, dass „Parent“ immer als „readonly“ gekennzeichnet werden; sie können also eingelesen, aber nicht geändert werden (dies macht in aller Regel fachlich keinen Sinn).

Ausgehend von „Adresse“ kommen als „Child“ die Tabellen „Person“, „Termin“, „Bestellung“, „Notiz“, „AdrsSlgw“ infrage. Von „AdrsSlgw“ können wir zu dessen „Parent“ „Schlagwort“ navigieren; von „Person“ zu ihrem „Parent“ „Funktion“.

Um den Zusammenhang zwischen den Tabellen herzustellen, müssen wir jeweils angeben, über welche Primär- und Fremdschlüssel auf die Daten zugegriffen werden soll (siehe die Attribute „pk“ und „fk“).

Das macht dann zusammen immerhin ein „JOIN“ aus acht Tabellen. Bei genauerer Betrachtung werden wird aber deutlich, dass „Funktion“ nicht mit eingelesen werden muss, sondern wir die Anzeige einer ComboBox überlassen.

Zu jeder Tabelle – egal ob „Root“, „Child“ oder „Parent“ können wir ihre Spalten auswählen (siehe Element „Column“).

Im einfachsten Fall wird hier „*“ angegeben, also alle Columns. Das hat den Vorteil, dass – wenn sich die Datenstruktur ändert – diese Definition nicht angepasst werden muss.

Es ist aber auch möglich, die gewünschten Columns einzeln aufzuführen. Zum einen, um die Menge der zu übertragenden Daten zu begrenzen, zum anderen aber auch, um ihnen einen anderen Namen zu geben („alias“) oder Daten zu berechnen.

Unser „View“ sieht dann etwa so aus:

```
<View name="AdresseEinzeln">
<RootTable tablename="adresse" pk="adrsid">
```

[1] Diese Forderung kann inzwischen soweit eingeschränkt werden, als dieses ein sinnvolles Vorgehen ist; der Persistenz Layer akzeptiert auch Tabellen mit mehreren Felder als Primärschlüssel.

[2] Für spätere Versionen des Persistenz-Layers ist geplant, auch mehrere solcher Wurzel-Tabellen zuzulassen.

```
    <Column name="*"/>
    <Child tablename="person" pk="persid" fk="fk_adrsid"
                 element="personen" orderby="name">
      <Column name="*" />
    </Child>
    <Child tablename="termin" pk="terminid" fk="fk_adrsid"
                 orderby="datum,von">
      <Column name="*" />
    </Child>
    <Child tablename="notiz" pk="notizid" fk="fk_adrsid">
      <Column name="*" />
    </Child>
    <Child tablename="bestellung" pk="bestellid" fk="fk_adrsid">
      <Column name="*"/>
      <Column name="Menge * Einzelpreis" alias="Wert"
        readonly="true"/>
    </Child>
    <Child tablename="adrsslgw" pk="fk_adrsid,fk_slgwid"
                 fk="fk_adrsid" alias="AdressSchlagworte">
      <Column name="*"/>
    </Child>
</RootTable>
</View>
```

Zusätzliche Features:

- Mit dem Attribut „orderby“ kann selbstredend die Sortierreihenfolge der Childs festgelegt werden. Es können mit Komma getrennt beliebig viele Spalten aufgeführt werden.
- Die Felder eines „Parent“ können in den jeweiligen Datensatz eingebettet werden. Hierzu dient das Attribut „join=’true’“. Dieses bewirkt, dass die Datenstruktur „flachgeklopft“ wird. Natürlich muss hier auf doppelte Spaltenname geachtet werden (was mit dem Attribut „alias“ vermieden werden kann).
- Bei „Child“ kann über das Attribut „element“ ein Zwischenknoten definiert werden, der alle abhängigen Daten aufnimmt (in unseren Beispiel „Personen“ → „Person“). Dieses ist nur für die XML-Darstellung wichtig.

Named Parameter:

Bei einem Request können Parameter benannt werden, die erste zur Laufzeit gesetzt und ausgewertet werden:

```
<View name="AdresseParameter">
<RootTable ... where="Name1 LIKE $name1 AND Vorname = $vorname”>
    ...
```

Auf dieser Art wird eine Prepared Statement vorbereitet:

```
… WHERE Name1 LIKE ? AND Vorname = ? …
```

Welches erst zu Laufzeit mit den Werten gefüttert wird:

```
ParameterList list = new ParameterListe();
list.add(new NVPair(„Name1“, „Müller%“));
list.add(new NVPair(„Vorname“, „Karl“));
JDataset ds = pl.getDataset(„AdresseParameter“, lst);
```

2.4 Beispiel für ein Ergebnisdokument

Der Persistenz-Layer stellt die eingelesen Daten als JDataSet (siehe unten) zur Verfügung.

Die toString()-Methode des DataSet verwandelt seinen Inhalt in ein XML-Dokument:

Der Node „Schema“ hält die Definition des DataSet; der Node „Data“ selbstredend die Daten.

```
<JDataSet name='AdresseEinzeln'>
  <Schema>
    <adresse tablename='adresse' tabletype='root'>
      <Column name='adrsid' type='4' pk='true' notnull='true'/>
      <Column name='kennung' type='12' notnull='true'/>
      <Column name='name1' type='12' notnull='true'/>
      <Column name='name2' type='12'/>
      <Column name='name3' type='12'/>
      <Column name='strasse' type='12'/>
      <Column name='plz' type='12'/>
      <Column name='ort' type='12'/>
      <Column name='anrede' type='12'/>
      <Column name='telefon1' type='12'/>
      <Column name='telefon2' type='12'/>
      <Column name='telefax' type='12'/>
      <Column name='email' type='12'/>
      <Column name='homepage' type='12'/>
      <Column name='privat' type='-6' notnull='true' default='0'/>
      <Column name='bemerkung' type='-1'/>
      <Column name='version' type='4'/>
      <person tablename='person' tabletype='child' element='personen' fk='fk_adrsid'>
        <Column name='persid' type='4' pk='true' notnull='true'/>
        <Column name='fk_adrsid' type='4' notnull='true' fk='true'/>
        <Column name='fk_funkid' type='4'/>
        <Column name='name' type='12' notnull='true'/>
        <Column name='durchwahl' type='12'/>
        <Column name='privat' type='-6' notnull='true' default='0'/>
      </person>
      <termin tablename='termin' tabletype='child' fk='fk_adrsid'>
        <Column name='terminid' type='4' pk='true' notnull='true'/>
        <Column name='fk_adrsid' type='4' fk='true'/>
        <Column name='fk_persid' type='4'/>
        <Column name='datum' type='91' notnull='true'/>
        <Column name='von' type='92'/>
        <Column name='bis' type='92'/>
        <Column name='bemerkung' type='-1'/>
      </termin>
      <notiz tablename='notiz' tabletype='child' fk='fk_adrsid'>
        <Column name='notizid' type='4' pk='true' notnull='true'/>
        <Column name='fk_adrsid' type='4' fk='true'/>
        <Column name='bemerkung' type='-1' notnull='true'/>
        <Column name='datum' type='91'/>
        <Column name='erledigt' type='-6' notnull='true' default='0'/>
      </notiz>
      <bestellung tablename='bestellung' tabletype='child' fk='fk_adrsid'>
        <Column name='bestellid' type='4' pk='true' notnull='true'/>
        <Column name='fk_adrsid' type='4' fk='true'/>
        <Column name='artikel' type='12'/>
        <Column name='menge' type='4'/>
        <Column name='einzelpreis' type='3'/>
        <Column name='wert' type='8' readonly='true'/>
      </bestellung>
      <AdressSchlagworte tablename='adrsslgw' tabletype='child' supress='true'
fk='fk_adrsid'>
        <Column name='fk_slgwid' type='4' pk='true' notnull='true' fk='true'/>
        <Column name='fk_adrsid' type='4' pk='true' notnull='true' fk='true'/>
        <schlagworte tablename='schlagworte' tabletype='parent' inline='true'
fk='fk_slgwid'>
          <Column name='schlagwort' type='12' notnull='true'/>
```

```
        </schlagworte>
      </AdressSchlagworte>
    </adresse>
  </Schema>
  <Data>
    <adresse rowtype='root'>
      <adrsid>1</adrsid>
      <kennung>RUDI</kennung>
      <name1>Rudi Müller</name1>
      <name2/>
      <name3></name3>
      <strasse>Milchstraße 13</strasse>
      <plz>10969</plz>
      <ort>Berlin-Mitte</ort>
      <anrede/>
      <telefon1/>
      <telefon2>0172 / 47 11</telefon2>
      <telefax/>
      <email/>
      <homepage/>
      <privat>0</privat>
      <bemerkung>Hallo Rudi!</bemerkung>
      <version>34</version>
      <personen rowtype='node'>
        <person rowtype='child'>
          <persid>2</persid>
          <fk_adrsid>1</fk_adrsid>
          <fk_funkid>4</fk_funkid>
          <name>Johanna</name>
          <durchwahl>444</durchwahl>
          <privat>0</privat>
        </person>
      </personen>
      <personen rowtype='node'>
        <person rowtype='child'>
             […]
        </person>
      </personen>
      <termin rowtype='child'>
        <terminid>12</terminid>
        <fk_adrsid>1</fk_adrsid>
        <fk_persid/>
        <datum>20.03.2003</datum>
        <von>01.01.1970</von>
        <bis>01.01.1970</bis>
        <bemerkung>Wichtig</bemerkung>
      </termin>
      <termin rowtype='child'>
             […]
      </termin>
      <bestellung rowtype='child'>
        <bestellid>10</bestellid>
        <fk_adrsid>1</fk_adrsid>
        <artikel>Kaffekanne</artikel>
        <menge>1</menge>
        <einzelpreis>5.10</einzelpreis>
        <wert>5.1</wert>
      </bestellung>
      <bestellung rowtype='child'>
             [...]
      </bestellung>
      <schlagwort>Kunde</schlagwort>
      <schlagwort>Lieferant</schlagwort>
    </adresse>
  </Data>
</JDataSet>
```

2.5 Methoden des Persistenz-Layers

Hier eine Übersicht zu einigen Methoden des Persistenz-Layer, die im Interface `IPL` definiert sind. Die wichtigsten Methoden sind naturgemäß `getDataset` und `setDataset`, mit den Daten aus der Datenbank ausgelesen und wieder zurückgeschrieben werden.

Wichtig für das Verständnis ist, dass beim Zurückschreiben der Daten dem Persistenz-Layer Hinweise gegeben werden müssen, welche Daten sich geändert haben (UPDATE), welche neu (INSERT) und welche zu löschen sind (DELETE). Dieses wird je Knoten mit den Attributen `modified='true'`, `inserted='true'` oder `deleted='true'` erreicht. Der GuiBuilder vergibt diese Attribute automatisch.

```
/**
 * Liefert einen Dataset mit dem angegebenen Namen und dem
 * angegebenen Primärschlüssel der Wurzeltabelle.
 * @param datasetname
 * @param oid
 * @return
 */
public JDataSet getDataset(String datasetname, long oid);

/**
 * Der vom Client geänderter Dataset wird in die Datenbank
 * zurückgeschrieben.
 * @param dataset Ein Dataset
 */
public int setDataset(JDataSet dataset);

/**
 * Liefert einen Dataset ohne Angabe eines Schlüssels;
 * also vor allem eine Menge von Datensätzen - bis zum Inhalt
 * einer kompletten Tabelle.
 * @param datasetname
 * @return String
 */
public JDataSet getAll(String datasetname);

/**
 * Liefert einen Dataset ohne Werte aber mit den Metadaten.
 * Wird bei der Erstellung eines neuen Datensatzes benötigt.
 * @param datasetname Der gewünschte Dataset
 * @return JDataSet
 */
public JDataSet getEmptyDataset(String datasetname);

/**
 * Liefert einen neuen eindeutigen Schlüssel für den Client.
 * @return long
 */
public long getOID();

/**
 * Liefert die Meta-Daten der Datenbank als XML-Document.
 * Tables, Columns, Parent-Tables, Child-Tables.
 * Je nach Datenbank-Typ sind diese Informationen mehr oder weniger
 * vollständig; dieses hängt von der Implementierung des
 * JDBC-Treibers ab.
```

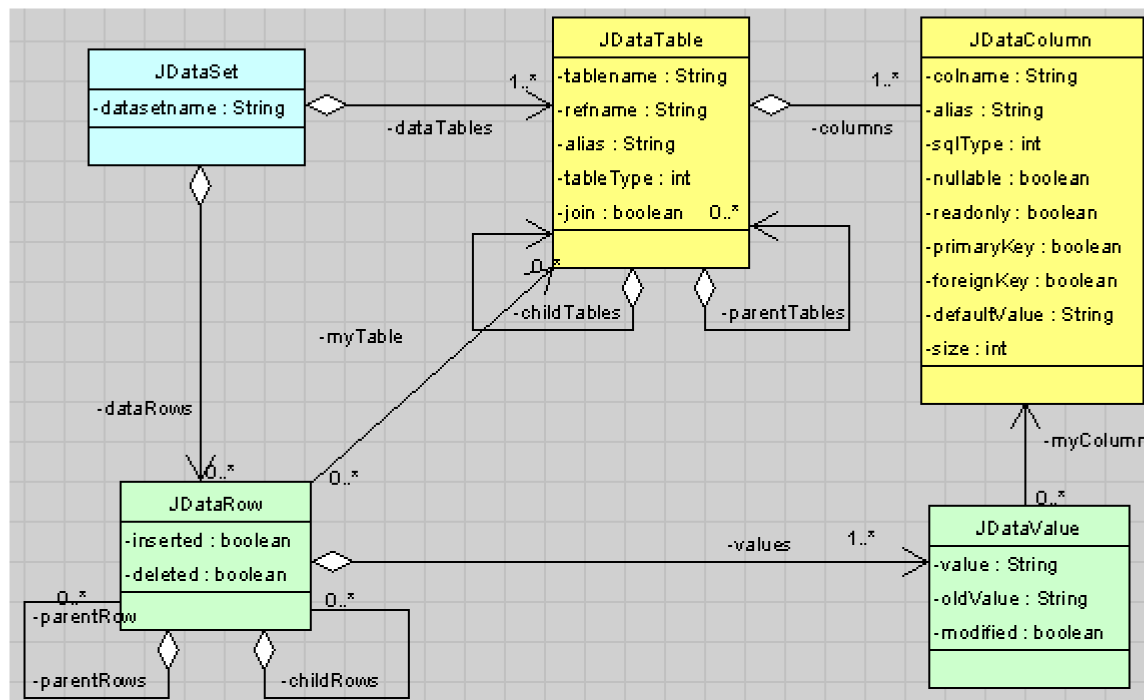
```
 * @return String
 */
public String getDatabaseMetaData(String databaseName);

/**
 * Setzt den Debug-Modus des Persistenzlayers.
 * @param b
 */
public void setDebug(boolean b);

/**
 * Liefert den Debug-Modus
 * @return
 */
public boolean isDebug();
```

3 JDataSet

Ein DataSet ist – wie oben beschrieben – ein generisches Transfermodell. Je nach Anwendungsfall kann seine innere Struktur verschieden sein, ohne das programmiert oder Code generiert wird.

Er hält Informationen über die Struktur der Daten sowie die Daten selbst.

Ein DataSet wird üblicherweise vom Persistenz-Layer aufgebaut; hier soll nur erläutert werden, welche Methoden der DataSet für die Client-Anbindung zur Verfügung stellt.



Die Klasse JDataSet selbst ist nur eine Container von Tabellen für die Struktur der Daten (gelb) sowie eine Menge von Zeilen für die Daten selbst (grün).

Bei der Definition des Persistenz-Layers erhalten alle DataSets einen eindeutigen Namen.

Z.Z. kann ein DataSet immer nur genau eine WurzelTabelle halten; dieses sollte eigentlich für die meisten Anwendungen ausreichen; Workaround wenn das nicht reicht: Einen weiteren DataSet definieren.

Solange dieses der Fall ist, kann mit der Methoden getDataTable() auf einfache Art auf diese Wurzeltabelle zugegriffen und mit addRow() dem DataSet einfach eine neue Zeile hinzugefügt werden.

Darüber hinaus gibt es fachliche Gründen, einen DataSet mit genau einer Zeile oder mit eine Menge von Zeilen zu haben:

Einzelobjekt

Der „einzeilige“ DataSet wird zumeist für komplexe Geschäftsobjekte wie „Kunde“ oder „Auftrag“ eingesetzt. Hier steht das einzelne Objekt um Vordergrund; allerdings hat dieses Objekt zumeist eine komplexe Unterstruktur aus abhängigen oder übergeordneten Daten wie Ansprechpartnern, Bestellungen, Terminen usw.

Aus dem Persistenz-Layer werden diese Objekte mit der Methode
```
JDataSet getDataset(String datasetName, long oid)
```
angefordert. Bei oid handelt es sich naturgemäß im den Primär-Schlüssel und wir täten uns wundern, wenn die Datenbank hier mehr als ein Objekt liefern würde.

Bei solch einem DataSet funktioniert also auch die Methode
`JDataRow getRow()`

Listenobjekt

Hier hält der DataSet eine ***Menge*** von Zeilen; dieser Typ von DataSet wird meist benutzt, um dem Anwender eine Auswahl von Objekten anzubieten, also das Ergebnis einer Datenbank-Recherche oder die Daten für eine ComboBox. Die einzelnen Zeilen selbst sind hier meist relativ einfach aufgebaut und werden oft in Tabellen angezeigt.

3.1 Änderungen an den Daten verarbeiten

Indem dem DataSet Zeilen hinzugefügt, gelöscht oder Werte verändert werden, ändert sich sein Inhalt.

Um dieses zu prüfen gibt es die Methode `hasChanges()` sie liefert nur dann `true`, wenn sich irgendwas am Inhalt verändert hat.

Die veränderten Daten müssen zurück in die Datenbank; hierzu dient die Methode `getChanges()` sie liefert wiederum einen DataSet, der aber nur die veränderten Inhalte umfasst, also die Zeilen, bei geändert, hinzugefügt oder gelöscht[3] wurden.

Diesen DataSet übergeben wir dem Persistenz-Layer:
`int setDataset(JDataSet changes)`

Der Persistenz-Layer wird nun in einer Transaktion alle getätigten Änderungen in die Datenbank schreiben, also die als gelöscht markierten Zeilen jetzt wirklich löschen, die neuen Zeilen einfügen und die geänderten updaten. Als Rückgabewert erhalten wir die Zahl der geänderten Datensätze.

Naturgemäß kann hier einiges schief gehen, wie z.B. dass die Daten inzwischen von einem anderen Anwender gelöscht oder manipuliert wurden.

Nur wenn keine Exception geworfen wird teilen wir dem DataSet mit, dass auch er nun die Äderungen akzeptieren kann: `commitChanges()`

Beispiel-Code:

```
JDataSet myDataset = myPersistenzLayer.getDataset(myName, myOid);
```

[Daten ändern/hinzufügen/löschen]

```
if (myDataset.hasChanges()) {
    try {
      JDataSet dsChanges = myDataset.getChanges();
      int numRows = myPersistenzLayer.setDataset(dsChanges);
      myDataset.commitChanges();
    } catch (Exception ex) {
      // Fehlerbehandlung
    }
}
```

Hier die beiden DataSets bitte nicht verwechseln!

`myDataset` ist der, den ich ursprünglich aus dem Persistenz-Layer gelesen habe, an ihm nehme ich alle Änderungen vor.

`dsChanges` wird nur temporär zum Update der Datenbank eingesetzt; das commit muss natürlich bei `myDataset` erfolgen.

Natürlich ist es auch möglich, den ursprünglichen DataSet `myDataset` dem Persistenz-Layer zu übergeben; ich verschwende hier höchsten Ressourcen, wenn ich unnötig Daten über das Netz schicke, an denen sich gar nichts geändert hat.

[3] Die Methode JDataRow#setDeleted(true) löscht die Zeilen nicht wirklich, sondern markiert sie nur als gelöscht.

3.2 JDataRow / JDataValue

Spalteninhalte lesen / ändern:

```
String getValue(String columnName);
setValue(String columnName, String value);
boolean isModified();
```

Eine DataRow speichert intern alle Felder als Strings; dieses tut sie auch dann, wenn ein long, int oder boolean zugewiesen wird; dann wird intern dieser Wert in einen String verwandelt. Welchen Wert dieser String intern hat, kann aus dem Rückgabewert der Methoden String setValue(...) ermittelt werden.

Liefert setValue ein anderes Ergebnis als der gespeicherte Wert, wird die Eigenschaft modified dieser Spalte gesetzt; eine Row gilt als geändert, wenn irgendeine Spalte sich geändert hat (isModified).

Auch wenn alle Werte als String gehalten werden, ist der SQL-Datentyp jeder Spalte bekannt. Genau genommen wird dieser aus den Metadaten der Datenbank ermittelt.

Mit `JDataValue#getDataType()` kann darauf zugegriffen werden; die hier gelieferten Werte entsprechen denen in java.sql.Types definierten.

4 Anbindung an die Oberfläche

Im der Datei GuiBuilderConfig.xml muss eine Adapter-Klasse für den Zugriff auf den Persistenz-Layer eingetragen werden; z.Z. gibt es hierzu nur eine einzige:

```
<ApplicationAdapter>de.guibuilder.adapter.PersistenceAdapter</Applic
ationAdapter>
```

Der Persistenz-Layer selbst stellt generische Datasets zur Verfügung, bzw. nimmt die (in der Oberfläche) geänderten Datasets entgegen.

Das Problem besteht also darin, wie die Oberflächen-Komponenten mit den Zeilen und Spalten des Dataset verbunden werden soll.

4.1 GUI-Mapping

Das Mapping zwischen dem Dataset und den Oberflächen-Widgets erfolgt grundsätzlich über eine Notation, die sich an XPath anlehnt. Ähnlich wie bei XPath mit Elementen und Attributen habe ich beim Dataset des Problem auf Zeilen und Spalten zuzugreifen.

4.2 Mapping - Notation

Die Notation lautet wie folgt:
`root.child1.child2#parent1#parent2@column1`

Mit einem „.“ wird also zu einer abhängigen Child Table navigiert, währen mit „#“ eine übergeordnete Parent Table angesprochen wird. „@“ trennt am Ende den Namen der Spalte ab.

Außerdem ist zu beachten, dass es einen Unterschied macht, ob über die Struktur des Dataset zugegriffen wird (Table, Column) oder seinen Inhalt (Row, Value).

Von Child Rows kann es beliebig viele geben; also benötigen wir eine Notation, die es ermöglicht auf *bestimmte* Zeilen zuzugreifen. Diese geschieht wir bei XPath mit der Angabe des Index der Zeile:
`Root[x].child1[y].child2[z]#parent1#parent2@column1`

Im Unterschied zu XPath sind hier die Index-Angaben allerdings 0-relativ!

Lassen wir die Index-Angabe weg, wird die erste (äh, also die nullte) Zeile geliefert.

Zu beachten ist, dass genau genommen nicht die Namen der Tabellen hier aufgeführt werden, sondern die Namen der *Referenz* auf diese Tabelle. Es ist ja durchaus denkbar, dass dieselbe Tabelle mehrfach als Child oder Parent definiert werden kann.

4.3 Mapping - Definition

Beim Mapping werden dem Oberflächen-Widget bei der Spezifikation zusätzliche Attribute beigegeben, über die die Abbildung auf den Dataset ermöglicht wird.

Die wichtigsten Attribute sind `element="[path]"` und `root-element="[path"]`

Ein Root-Element kann bei Containern definiert werden, wenn diese eine Unterstruktur aus dem Dataset enthalten sollen (Form, Dialog, Table).

Wichtig für das Verständnis ist, dass sich der Pfad zu einer Zeile und Spalte im Dataset hierarchisch aus der Container-Schachtelung der Oberfläche ergibt, also eine einzelnes Widget sich die Pfadangaben aus seinen Parent-Container zusammensucht.

Wenn unser Dataset etwas so aussieht:
```
<JDataSet>
   <Adresse>
     <Name1>Rudi Müller</Name1>
```

Dann kann ein Beispiel für das Formular so aussehen:
```
<Form label="Adresse eingeben" root-element="Adresse" >
```

Auf Groß- und Kleinschreibung ist zu achten![4]

Soll also jetzt in einem Textfeld „Name1“ aus dem Model editiert werden, dann ist das Element wie folgt anzugeben:
```
<Text label="Name1:" element="@Name1" />
```

Der Path-Ausdruck zu diesem Feld würde also folgendes ergeben:
```
Adresse@Name1
```

4.4 Mapping von Tabellen

Eine Tabelle kann mehrere gleichartige Objekte in Zeilen anordnen. In unserem Dataset müssen wir damit rechnen, dass bestimmte Einträge mehrfach vorgenommen:
```
<JDataset>
   <Adresse>
     <Name1>Rudi Müller</Name1>
     <Termin>
       <Datum>30.5.2003</Artikel>
       <Text>Weltuntergang nicht verpassen!</Text>
     </Termin>
     <Termin>
       <Datum>1.6.2003</Artikel>
       <Text>Mal sehn wies weitergeht</Text>
     </Termin>
```

So wie unser Dataset aufgebaut ist, muss die Tabelle entsprechend ausgezeichnet werden.

```
<Table element=".Termin">
   <Date label="Datum" element="@Datum" />
   <Text label="Bezeichnung" element="@Text" />
</Table>
```

Wichtig ist der „.“ vor dem Termin, da hiermit gesteuert wird, dass der Termin eine Child Table von Adresse ist.

4.5 Mapping von Combo- und Listboxen

Hier besteht die Anforderung darin, dass eine Combobox mit einer Menge von Werten aus der Datenbank gefüllt wird, und sich der Anwender daraus einen Eintrag auswählen kann. Es soll zumeist nicht die Bezeichnung des Wertes verwendet werden, sondern seine (unsichtbare) ID-Nummer.

Das ergibt dann die folgenden zusätzlichen Attribute für List- und Comboboxen:

```
dataset="[DatasetName]"
displayMember="[Element-Darstellung]"
valueMember="[Element-ID]"
```

Beispiel:

[4] Es scheint sinnvoll zu sein, hier nicht Case-sensitiv vorzugehen

In einer Tabelle “Funktionen” sind alle Funktionen einer Person aufgeführt; wir haben einen View „Funktionen“ definiert, der uns die Daten für die Combobox liefern soll:

```
<View name="Funktionen">
   <RootTable tablename="funktion" pk="funkid"
               orderby="bezeichnung">
     <Column name="*" />
   </RootTable>
</View>
```

Die Daten des eingelesenen Dataset sähe so aus:

```
<JDataset name="Funktionen">
   <Funktion>
     <funkid>4711</funkid>
     <Bezeichnung>Siebenundvierzig...</Bezeichnung>
   </Funktion>
   <Funktion>
     <funkid>4712</funkid>
     <Bezeichnung>...
```

In der Tabelle „Person“ haben wir zudem einen Fremdschlüssel „fk_funkid“.

Die Definition unserer ComboBox sieht dann so aus:

```
<Combo label="Funktion" element="@fk_funkid" dataset="Funktionen"
displayMember="Bezeichnung" valueMember="funkid"/>
```

Der GuiBuilder selbst sorgt automatisch dafür, dass bei der Initialisierung eines Dialoges die ComboBox mit den Werten aus dem Model gefüllt wird.

4.6 Mapping von Baumstrukturen (Tree)

TODO!

5 ...und jetzt bitte alle zusammen !

Um die Daten nun wirklich aus der Datenbank zu lesen, in der Oberfläche darzustellen, dort zu ändern und die geänderten Daten zurück in die Datenbank zu transportieren benötigen wir noch etwas Funktionalität, die wird der Einfachheit halber mit der Scriptsprache „BeanShell“ implementieren.

Der folgende Code-Schnipsel initialisiert einen Dialog für die Anbindung an den Persistenz-Layer.

```
<Script language="BeanShell">
<!--
import de.guibuilder.framework.*;
import de.guibuilder.framework.event.*;
import de.pkjs.pl.*;
import de.jdataset.*;
import electric.xml.*;
// Hier haben wir eine Referenz auf den Persistenz-Layer
pl = GuiSession.getInstance().getAdapter().getDatabase();
pl.setDebug(false);
datasetName = "AdresseEinzeln";
```

Jetzt brauchen wird noch zwei Methoden für Lesen und Speichern (dafür nehmen wir im einfach zwei Buttons).

Daten aus der Datenbank lesen und anzeigen:

```
read(event) {
   // Die Adresse mit dem Primärschlüssel „1“ wird gelesen...
   JSataSet ds = pl.getDataset(modelName, 1);
   // ...und wir weisen dieses der Oberfläche zu.
   event.window.setDatasetValues(ds);
}
```

Die Daten aus der Datenbank werden nun in der Oberfläche angezeigt, und der Benutzer kann sie nach Herzenslust ändern.

Geänderte Daten zurückspeichern:

```
save(event) {
   // Daten aus der Oberfläche abholen
   JDatSet ds = event.window.getDatasetValues();
   // Nur speichern wenn die Daten auch geändert wurden.
   if(ds.hasChanges() ) {
      // Änderungen aus dem Dataset abholen...
      JDataSet dsChanges = ds.getChanges();
      // ...und die Änderungen an den Persistenzlayer übergeben
      pl.setDataset(dsChanges);
      // Alles was als geändert, ergänzt oder gelöscht
      // gekennzeichnet wurde übernehmen.
      ds.commitChanges();
   }
}
```

6 Dataset in Files speichern

Die Methode `JDataSet#getXml()` liefert einen Dataset als XML-Dokument; man kann aus diesem Dokument den Dataset auch wieder erzeugen, indem es dem Constructor übergeben wird:

```
JDataSet clone = new JDataSet(myDataset.getXml));
```
[5]

Das führt zu der Idee, dass man Datasets nicht nur in der Datenbank speichern muss, sondern diese u.U. auch in Files (zwischen-)speichern kann. Alles andere, also auch das Binding mit der Oberfläche, funktioniert wird oben beschrieben.

Beispiel:

Dataset aus File einlesen:

```
JDataSet ds  = new JDataSet(new Document(new File("MyFile.xml")));
ds.commitChanges(); // Alte Änderungen vorsichtshalber committen.
event.window.setDatasetValues(ds);
```

Änderungen speichern:

```
JDataSet ds  = event.window.getDatasetValues();
if (ds.hasChanges()) {
    Document doc = ds.getXml();
    doc.write(new File(filename));
    ds.commitChanges();
}
```

Wenn wir Datasets in einem File speichern, müssen wir naturgemäß den gesamten Dataset dort ablegen, und nicht nur die getätigten Änderungen!

Außerdem sollten wir nach dem Einlesen des Dataset aus der Datei vorsichtshalber ein commitChanges absetzen. Da eine File-System ist keine Datenbank ist, bleiben die vom GuiBuilder gesetzten Eigenschaften „modified“, „inserted“ und „deleted“ im XML-Document erhalten. Alternativ kann ***vor*** dem Speichern ein commitChanges ausgeführt werden, damit die Änderungen nicht in der Datei ersichtlich sind.

Beispiel-Anwendung:

Die Eigenschaften des GuiBuilder werden in der Datei „GuiBuilderConfig.xml“ gespeichert; dieses ist nichts anderes als ein Dataset!

Wird im GuiBuilder der Menüpunkt „Bearbeiten / Eigenschaften“ aufgerufen, wird das GuiBuilder-Script „GuiBuilderConfigEditor.xml“ gestartet und darin die Einstellungen für den GuiBuilder vorgenommen.

7 API Tutorial

7.1 Request erzeugen.

Üblicherweise wird die Zugriffschicht wie oben beschrieben über XML-Dokumente definiert. Es besteht jedoch auch die Möglichkeit, diese Definitionen „zu Fuß“ über die API vorzunehmen.

Ausgangspunkt ist der Persistenz Layer selbst, da er alle Zugriffe auf die Datenbank verwaltet. Der Name muss eindeutig sein.

```
Request req = myPL.createRequest("test");
// Default-Datenbank zuweisen.
req.setDefaultDatabase(pl.getDatabase("MyDatabase"));
```

[5] Wie hier ersichtlich kann man Datasets auf diese Art auch bequem clonen!

```
// Table-Request erzeugen;
// „*“ bedeutet, dass alle Spalten ausgegeben werden sollen;
// ansonsten die gewünschten Spalten mit Komma getrennt aufführen.
TableRequest tr = req.createRootTableRequest("MyTableName", "*");
// Primary Key festlegen.
tr.setPK("oid");
// Daten lesen unter Angabe des Dataset und des Primärschlüssels
JDataSet ds = pl.getDataset("test", 1);
// Daten ausgeben
System.out.println(ds.getXml());
```

7.2 JDataSet „zu Fuß“ erzeugen

Datasets – und ihr Binding an die Oberfläche – können auch ganz und gar ohne jede Datenbank „zu Fuß“ erstellt werden.

Dazu müssen wir (mindestens) folgende Dinge tun:

1. Einen JDataSet erstellen
2. Eine DataTable erstellen
3. Die Tabelle mit Spalten versehen
4. Row(s) zu der Tabelle erzeugen.
5. Die Row(s) mit Werten füllen.

```
// 1. Create DataSet
JDataSet ds = new JDataSet(„Test“);
// 2. Create Table
JDataTable tblA = new JDataTable(“Adresse”);
ds.addRootTable(tblA);
// 3. Create Columns
JDataColumn colID = tblA.addColumn(“oid”, java.sql.Types.INTEGER);
colID.setNullable(false);
JDataColumn colName = tblA.addColumn(“Name”, Types.VARCHAR);
// 4. Add empty Row
JDataRow row1 = tblA.createNewRow();
ds.addRow(row1);
// 5. Set Values
row1.setValue(“oid”, 42);
row1.setValue(“Name”, “Rudi”);
// DataSet als XML-Document ausgeben.
System.out.println(ds.getXml());
```

Das Ergebnis sieht dann wie folgt aus:

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<JDataSet name='Test'>
  <Schema>
    <Adresse tablename='Adresse' tabletype='root'>
      <Column name='oid' type='4' notnull='true'/>
      <Column name='Name' type='12'/>
    </Adresse>
  </Schema>
  <Data>
    <Adresse inserted='true' rowtype='root'>
      <oid modified='true'>42</oid>
      <Name modified='true'>Rudi</Name>
    </Adresse>
  </Data>
</JDataSet>
```

8 Known Bugs / TODO

Der Anspruch bestand darin, ein „View“ auch aus verschiedenen Datenbanken zusammenstellen zu können; das hat noch nicht wirklich funktioniert.

Ob als Primärschlüssel auch andere Datentypen als int/long funktionieren, und ob ein Primärschlüssel auch aus mehreren Feldern bestehen kann, ist nicht vollständig getestet.

8.1 TODO

- HAVING, DEFAULT, IN
- Namespaces für Persistenzlayer und GuiBuilder
- View-Definition anhand der Metadaten überprüfen.
- Ein Konzept für die Plausibilisierung eines Dataset fehlt.
- BLOB?
- Anbindung von Tree an Model fehlt.
- Mehrere Root Tables je Dataset
- VIEWS unterstützen (Mckoi, SQL-Server, ...)

8.1.1 JOIN für Parent Tables

Man spart sich SQL-Statements, wenn die Parents über einen LEFT JOIN mit eingelesen werden; Voraussetzung ist naturgemäß, dass die ColumnNamen auch eindeutig sind oder ein entsprechendes Prefix angegeben wird:

```
<RootTable tablename="Bestellung">
   <Parent tablename="Artikel" join="LEFT" prefix="A_">
```

8.1.2 Alle Objekte XML-serialisierbar machen

Alle Klassen erhalten eine Methode `getElement()` um Objekte zu Human readable zu serialisieren und einen Constructor `MyClass(Element ele)` um sie wieder zu erzeugen.

8.1.3 DataSet aus verschiedenen Datenbank zusammenstellen

Das Problem ist nicht die Definition; man könnte ja bei jeder Tabelle angeben, aus welcher Datenbank sie stammt.

Das Problem ist das Two Phase Commit über die beteiligten Datenbanken.

8.2 Wunschliste

Oberfläche zum grafischen Entwurf von Datasets / Datenbanken erstellen.

DDL-Scripts aus der Definition erstellen.