Transformations – die API des Oracle Data Modeler

Dr. Gudrun Pabst
Trivadis GmbH
München

Schlüsselworte:

Oracle Data Modeler, Transformations, API

Einleitung

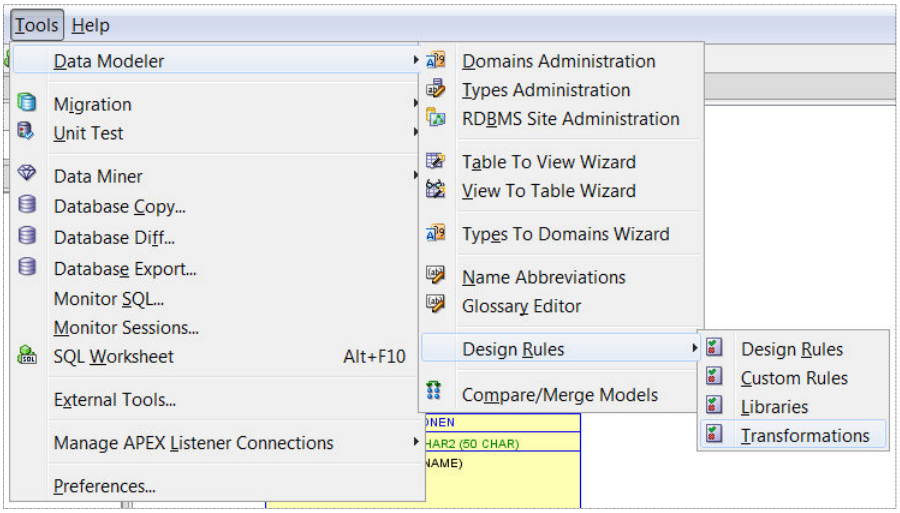
Bei der Datenmodellierung fallen oft Routinearbeiten an wie das Hinzufügen von Audit-Spalten an alle Tabellen oder das Erstellen von Triggern mit Standardcode. Die Automatisierung dieser Tätigkeiten erleichtert nicht nur die Arbeit des Datenmodellierers, sondern erhöht auch die Qualität des Modells, da sichergestellt wird, dass alle Objekte tatsächlich den vorgegebenen Konventionen entsprechen.
Mit den Transformations bietet der Oracle Data Modeler eine Schnittstelle, über die das Datenmodell programmgesteuert angepasst werden kann. Im folgenden wird die Schnittstelle vorgestellt und an Beispielen gezeigt, wie diese Funktionalität eingesetzt werden kann. Hierbei wird der Oracle Data Modeler in der Variante als Extension des SQL Developer verwendet.

Grundlagen

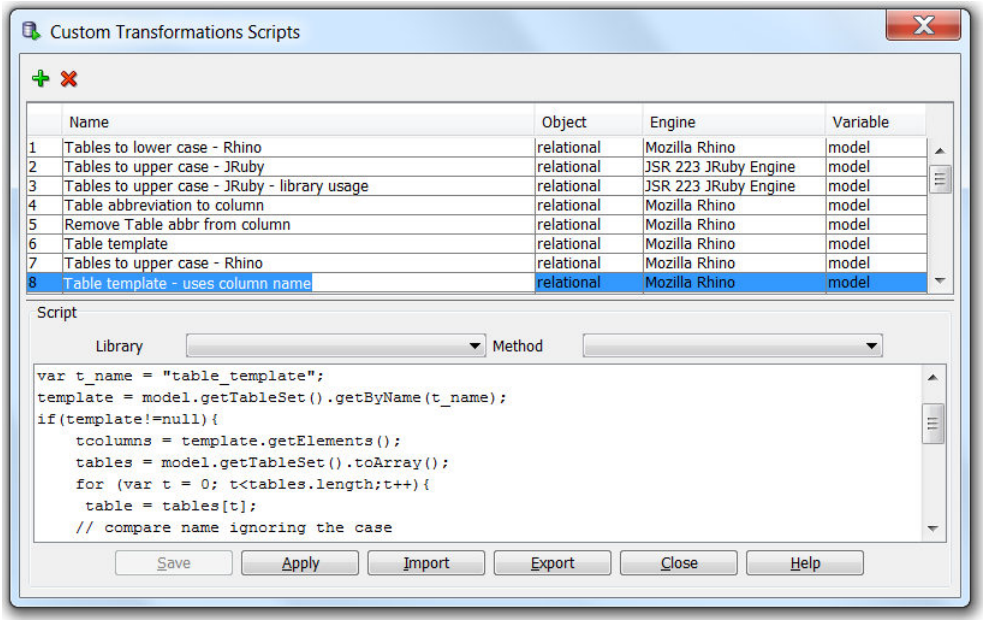
Java stellt über „Scripting for Java“ (JSR 233 - http://jcp.org/en/jsr/detail?id=223) die Möglichkeit zur Verfügung, Skript-Sprachen in Java-Programme einzubinden und den Endanwendern so eine Erweiterung des Programms durch eigene Funktionalitäten zu ermöglichen. Im Artikel „How Java Plays With Scripting Languages“ (http://www.oracle.com/technetwork/articles/dsl/languages-137862.html) wird diese Integration beschrieben.
Es gibt verschiedene Implementierungen der Anbindung von Skript-Sprachen. Eine davon ist Mozilla Rhino, das die Anbindung von JavaScript umsetzt. Mozilla Rhino ist im JDK 6 und höher bereits enthalten.

Der Oracle Data Modeler bietet dem Datenmodellierer diese Erweiterung durch Skript-Sprachen über die Transformations.
Da Mozilla Rhino im JDK enthalten ist, steht JavaScript bereits für die Transformations zur Verfügung.
Für JRuby werden zwar Beispielskripte mitgeliefert, aber die Scripting Engine muss erst eingebunden werden. Ebenso können für weitere Skript-Sprachen Engines eingebunden werden.

Das Fenster für die Transformations wird im SQL Developer geöffnet über den Menüpunkt „Tools → Data Modeler → Design Rules → Transformations“:



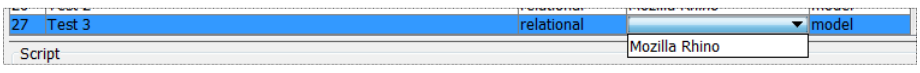
Das Fenster für die Transformations zeigt zunächst nur die mitgelieferten Skripte, später dann auch die eigenen Skripte:



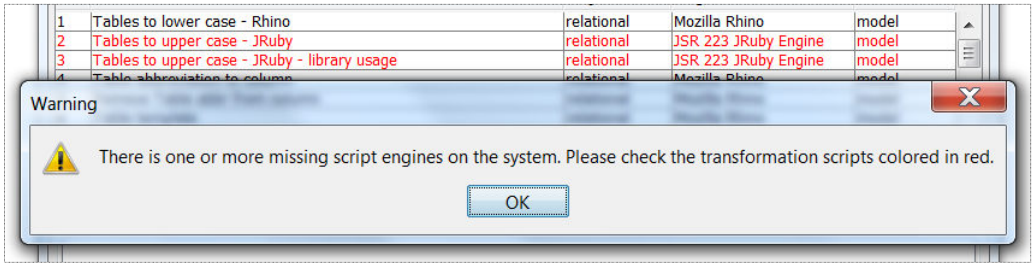
Bei jedem Skript ist angegeben, ob es sich auf die logische oder auf die relationale Ebene bezieht. Unter „Engine“ wird festgelegt, mit welcher Scripting Engine das Skript ausgeführt wird.

Einbinden einer Scripting Engine

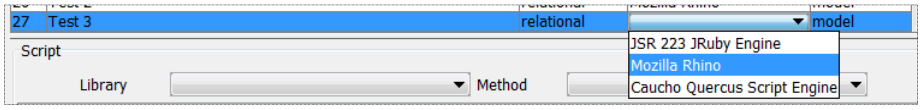
Zunächst wird nur die Engine „Mozilla Rhino“ angeboten:



Die Engine „JSR 223 JRuby Engine“ wird bei den Beispielskripten zu dieser Engine in der Liste aufgeführt, obwohl sie nicht eingebunden ist. Beim Öffnen der Transformationen erscheint daher zunächst eine entsprechende Fehlermeldung:



Um eine Scripting Engine einzubinden, wird die Jar-Datei der Scripting Engine benötigt. Diese Datei wird im Verzeichnis `jre\lib\ext` der JVM abgelegt, die vom Oracle SQL Developer verwendet wird. Anschließend muss der Oracle SQL Developer neu gestartet werden. Dann zeigt die Auswahlliste die neue Scripting Engine. Nach dem Einbinden von „JSR 223 JRuby Engine“ für JRuby und „Caucho Quercus Script Engine“ für PHP stehen diese Einträge zur Verfügung:



Mitgelieferte Dokumentation

Die eingebundenen Scripting Engines können sowohl für Custom Design Rules, d.h. selbstdefinierte Konsistenzprüfungen, als auch für Transformations verwendet werden. Die Transformations dienen dazu, umfangreiche Änderungen am Datenmodell mit Hilfe von Skripten durchzuführen.
Zum Vorgehen und zur Verwendung der Transformations bringt der SQL Developer im Verzeichnis `…\sqldeveloper\sqldeveloper\extensions\oracle.datamodeler\xmlmetadata\doc` folgende Dateien zur Dokumentation mit:

README.rtf	Kurzbeschreibung zum Umgang mit der API
index.html	Master für die Navigations- und die Inhaltsseite; Beschreibung des XML-Formats des Datenmodells

Zusätzlich werden im SQL Developer einige Beispielskripte mitgeliefert, über die die Benutzung der Transformations demonstriert wird.

Ein mitgeliefertes Beispiel

Das Skript „Table template - uses column name“ zeigt das Anlegen von Audit-Spalten über das Auslesen von Spalten einer Template-Tabelle und Anfügen an alle Tabellen, die noch nicht über diese Spalten verfügen:

```
var t_name = "table_template";
template = model.getTableSet().getByName(t_name);
if(template!=null){
```

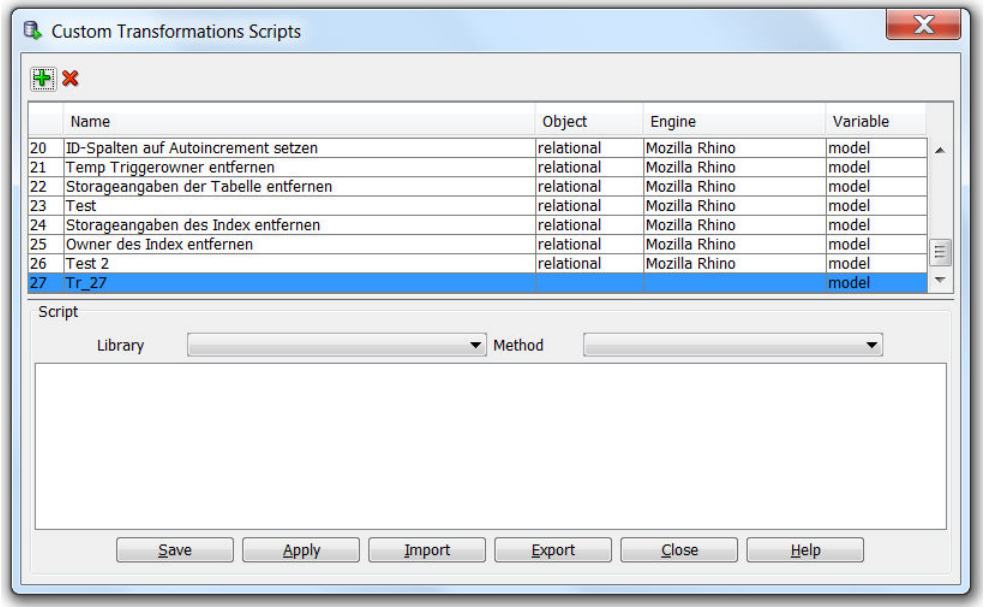
```
    tcolumns = template.getElements();
    tables = model.getTableSet().toArray();
    for (var t = 0; t<tables.length;t++){
     table = tables[t];
    // compare name ignoring the case
     if(!table.getName().equalsIgnoreCase(t_name)){
         for (var i = 0; i < tcolumns.length; i++) {
            column = tcolumns[i];
            col = table.getElementByName(column.getName());
            if(col==null){
             col = table.createColumn();
            }
            column.copy(col);
            table.setDirty(true);
         }
     }
    }
}
```

Mit `model` wird das aktuelle Datenmodell auf oberster Ebene angesprochen; `getTableSet()` liest daraus alle Tabellendefinitionen als Collection aus. Über `getByName(<Name>)` kann aus einer Collection ein Element anhand seines Namens ermittelt werden, hier die Template-Tabelle. Mit `toArray()` wird eine Collection in ein Array konvertiert, das in einer Schleife durchlaufen und auf dessen Elemente dann mittels `…[t]` zugegriffen werden kann.
Die Methode `getElements()` des Tabellenobjekts ruft alle Spalten der Template-Tabelle ab. Diese Methode ist eine Abkürzung für `getElementsCollection().toArray()` und liefert die Spalten gleich in einem Array.
Im weiteren Verlauf wird für alle Tabellen des Modells außer der Template-Tabelle geprüft, ob die Spalten mit demselben Namen wie die Spalten der Template-Tabelle bereits existieren. Dazu wird mit `getElementByName(<Name>)` zur aktuellen Tabelle die Spalte mit dem jeweiligen Namen ermittelt. Gibt es diese Spalte nicht, wird `null` zurückgeliefert. In diesem Fall wird mit der Methode `createColumn()` des Tabellenobjekts eine neue Spalte nur mit Default-Eigenschaften angelegt.
Sowohl für bestehende als auch für neuangelegte Spalten werden mit `<Quellspalte>.copy(<Zielspalte>)` die Eigenschaften der Template-Spalte auf die Tabellenspalte übertragen.
Abschließend wird die Tabelle mit `setDirty(true)` zum Speichern markiert.

Eigene Skripte

Die Transformations bieten die Möglichkeit, eigene Skripte einzubinden. Das Fenster für die Transformations hat dafür den Button . Nach Betätigen des Buttons steht eine neuer Eintrag zur Verfügung:

trivadis
makes IT easier.



Das neue Skript muss einen sprechenden Namen erhalten. Unter „Object“ muss angegeben werden, ob es sich auf die logische oder die relationale Ebene bezieht. Die korrekte Engine für das Skript muss ausgewählt werden. Dann kann im unteren Teil des Fensters der Skriptcode eingegeben werden.
Bevor ein Skript ausgeführt werden kann, muss es mit „Save“ gespeichert werden. Die Ausführung des Skripts wird dann mit „Apply“ ausgelöst.

Ausgabe von Text

Zur Kontrolle des Ablaufs des Skripts ist es häufig nötig, Informationen an den Bearbeiter zurückzugeben. Bei dem von Mozilla Rhino zur Verfügung gestellten JavaScript handelt es sich um serverseitiges JavaScript, daher steht die Ausgabe von Text mit Hilfe von `alert` nicht zur Verfügung.

Zur Ausgabe kurzer Meldungen im Skriptablauf können wir benutzen, dass das Scripting in einer Java-Umgebung stattfindet. Mit `importPackage` werden Java-Pakete in das JavaScript eingebunden. Dann stehen uns die Javaklassen des Pakets zur Verfügung. Wird nur eine Klasse gebraucht, kann alternativ mit `importClass` auch nur die benötigte Klasse eingebunden werden.
Mit dem Skript

```
importClass(javax.swing.JOptionPane);
var vText = "Demo der Ausgabe von Text";
JOptionPane.showMessageDialog(null, vText);
```

wird der in der Variable `vText` hinterlegte Text ausgegeben.

Für längere Ausgaben ist diese Methode eher nicht geeignet, da die Ausgabe dann unübersichtlich ist und sich der Text auch nicht kopieren und weiterverarbeiten lässt. In diesem Fall können wir im Modell eine „Note“ anlegen. Leider haben wir im SQL Developer keinen Zugriff auf die Eigenschaften der Notes außer den Text. Um den Namen der neu angelegten Note zu ermitteln, hilft folgendes Skript:

```
importClass(javax.swing.JOptionPane);
var vText = "";
var vNote;

var vNotes = model.getNoteSet().toArray();
for ( var n=0; n<vNotes.length; n++ ) {
  vNote = vNotes[n];
  vText = vText + vNotes[n].getName()    + " - " +
                  vNotes[n].getComment() + "\n";
}

JOptionPane.showMessageDialog(null, vText);
```

Das Skript ermittelt alle Notes im Modell und gibt ihren Namen und den in der Note hinterlegten Text aus. Mit `setName("Ausgabe")` können wir der Note für die Ausgabe noch einen sprechenden Namen geben. Längere Text werden dann zurückgegeben mit

```
var vText = "";
var vNote = model.getNoteSet().getByName("Ausgabe");
...
vNote.setComment(vText);
```

Interaktion mit dem Aufrufer

Das Einbinden von `JOptionPane` ermöglicht es auch, die Skripte interaktiv zu gestalten. Mit der Methode `showConfirmDialog` können wir im Ablauf des Skripts nach Bestätigungen fragen:

```
importClass(javax.swing.JOptionPane);
...
var vAntwort = JOptionPane.showConfirmDialog (
                 null , "Weitermachen?" , "Weitermachen?" ,
                 JOptionPane.YES_NO_OPTION );

if ( vAntwort == 0 ) {
  ...
}
```

Die Methode `showInputDialog` zeigt ein Dialogfenster mit einem Eingabefeld und gibt den eingegebenen Wert zurück:

```
importClass(javax.swing.JOptionPane);

var vTabelle = JOptionPane.showInputDialog (
                 null , "Welche Tabelle soll bearbeitet werden?" ,
                 "Tabellenauswahl" ,
                 JOptionPane.PLAIN_MESSAGE );

JOptionPane.showMessageDialog ( null , vTabelle );
```

Eine ausführliche Dokumentation von `JOptionPane` und seiner Methoden findet sich in der Java-Dokumentation.

Erstellen eines eigenen Skripts

Für das Vorgehen zum Erstellen eines eigenen Skripts verwenden wir folgende Aufgabenstellung:
Im Datenmodell gibt es mehrere Tabellen, die Gültigkeitsspalten haben. Auf allen derartigen Tabellen soll ein Check Constraint angelegt werden, der prüft, ob der Start des Gültigkeitszeitraums vor dem Ende des Gültigkeitszeitraums liegt.
Alle betroffenen Spalten folgen dabei einer Namenskonvention, sie heißen `<Tabellenkürzel>_GUELTIG_VON` bzw. `<Tabellenkürzel>_GUELTIG_BIS`. Auch der Check Constraint soll einer Namenskonvention folgen, er erhält den Namen `<Tabellenkürzel>_GUELTIG_VON_BIS_CK`.

Der Ablauf des Skripts ist wie folgt:
1. Zunächst werden alle Tabellen des Modells ermittelt und in einer Schleife durchlaufen.
2. Zur gerade aktuellen Tabelle werden alle Spalten ermittelt und ebenfalls in einer Schleife durchlaufen.
3. Entspricht der Name der aktuellen Spalte einem der beiden oben genannten Schemas, wird die Spalte in der passenden Variable gespeichert.
4. Sind nach dem Durchlaufen aller Spalten der aktuellen Tabelle beide Variablen gefüllt, benötigt die Tabelle den Check Constraint. Daher werden alle Constraints der Tabelle durchlaufen.
5. Wird der Check Constraint nicht gefunden, wird er angelegt.

Zum Schritt 1:
Unter der Klasse „RelationalDesign“ finden wir in der HTML-Dokumentation die Collections:

Collections		
Name	Type	Getter
importConnectionStamps	List	`getImportConnectionStamps()`
importDDLFileStamps	List	`getImportDDLFileStamps()`
Schemas	SchemaObjectSet	`getSchemaObjectSet()`
Tables	TableSet	`getTableSet()`
TableViews	TableViewSet	`getTableViewSet()`
ForeignKeys	FKIndexAssociationSet	`getFKIndexAssociationSet()`
FKArcs	FKArcSet	`getArcSet()`
NoteObjects	NoteSet	`getNoteSet()`
Diagrams	List	`getListOfSubviews()`
StorageDesigns	StorageDesignSet	`getSorageDesigns()`

Mit `getTableSet()` können wir alle Tabellen in unserem Modell auslesen. Die Methode `toArray()` konvertiert dabei die Collection in ein Array, das in einer Schleife durchlaufen werden kann und über den Schleifenindex bei jedem Durchlauf die aktuelle Tabelle zur Verfügung stellt.

Zu den Schritten 2, 4 und 5:
Unter der Klasse „Table“ finden wir in der HTML-Dokumentation die Collections:

Collections				
Name	Type	Getter	Create Item	Add Item
columns	Collec-tion	`getElementsCollection()`	`createColumn`	
indexes	List	`getAllInds_FKeyInds()`	`createIndex`	
tableCheckConstraints	List	`getCheckConstraints()`	`createCheck Constraint`	`addCheck Constraint`
columnGroups	List	`getColumnGroupsSet()`	`createColumn Group`	
spatialDefinitions	List	`getSpatialDefinitions()`	`createSpatial Definition`	

Wie aus dem mitgelieferten Beispiel für die Audit-Spalten zu ersehen ist, können wir die Liste der Spalten auch erhalten über `getElements()` .
Die Check Constraints werden über `getCheckConstraints()` ermittelt. Die Tabelle gibt an, dass ein Check Constraint mit der Methode `createCheckConstraint` erstellt wird. Um einen Check Constraint tatsächlich zur Tabelle hinzuzufügen, muss die Methode `addCheckConstraint` verwendet werden.

Zum Schritt 3:
Jedes Objekt im Datenmodell hat einen Namen, der über `getName()` abgefragt werden kann. Das Tabellenkürzel finden wir laut der HTML-Dokumentation über `getAbbreviation()`.

Daraus ergibt sich folgendes Skript:

```
importClass(javax.swing.JOptionPane);
var vText = "Check Constraint anlegen für\n";

var vGueltigVonName = "GUELTIG_VON";
var vGueltigBisName = "GUELTIG_BIS";
var vConstraintName = "GUELTIG_VON_BIS_CK";

var vTables;
var vTable;
var vTableProxy;
var vColumns;
var vColumn;
var vColumnVon = null;
var vColumnBis = null;
var vCheckConstraints;
var vCheckConstraint;
var vCheckConstraintVonBis = null;
var vKuerzel;

vTables = model.getTableSet().toArray();
for ( var t=0; t < vTables.length; t++ ) {
  vTable = vTables[t];
  vColumnVon = null;
```

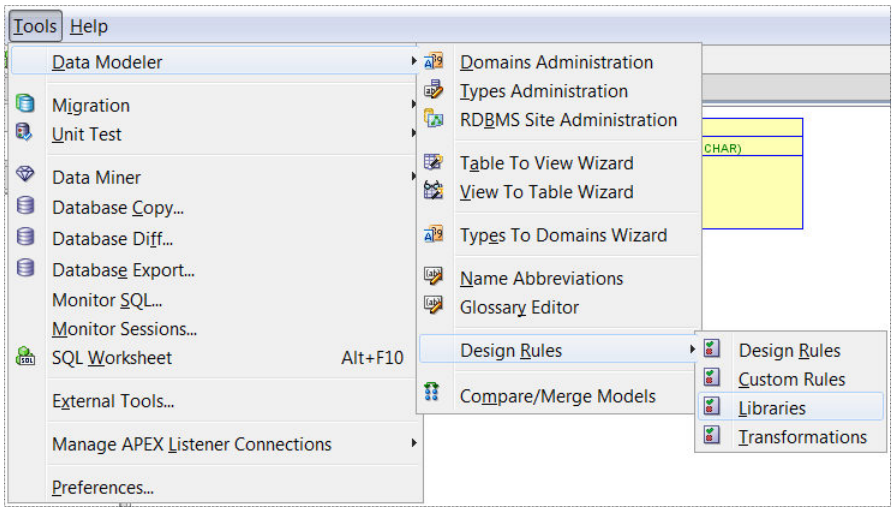

trivadis makes IT easier.

```
  vColumnBis = null;
  vCheckConstraintVonBis = null;
  vKuerzel = vTable.getAbbreviation();
  // Ermitteln der Gültigkeitsspalten
  vColumns = vTable.getElements();
  for ( var c = 0; c < vColumns.length; c++ ){
    vColumn = vColumns[c];
    // Namensvergleiche
    if ( vColumn.getName().equalsIgnoreCase(
                            vKuerzel + "_" + vGueltigVonName ) ) {
      vColumnVon = vColumn;
    } else if ( vColumn.getName().equalsIgnoreCase(
                            vKuerzel + "_" + vGueltigBisName ) ) {
      vColumnBis = vColumn;
    }
  }
  // Beide Spalten vorhanden?
  if ( vColumnVon != null && vColumnBis != null ) {
    vText = vText + vTable.getName() + "\n";
    vCheckConstraints = vTable.getCheckConstraints().toArray();
    // Ermitteln des Gültigkeitsconstraint
    for ( var c = 0; c < vCheckConstraints.length; c++ ){
      vCheckConstraint = vCheckConstraints[c];
      // Namensvergleich
      if ( vCheckConstraint.getName().equalsIgnoreCase(
                            vKuerzel + "_" + vConstraintName ) ) {
        vCheckConstraintVonBis = vCheckConstraint;
      }
    }
    // Ggf. Anlegen des Check Constraint
    if ( vCheckConstraintVonBis == null ) {
      vCheckConstraintVonBis = vTable.createCheckConstraint ();
      vTable.addCheckConstraint ( vCheckConstraintVonBis );
      vCheckConstraintVonBis.setName (
        vKuerzel + "_" + vConstraintName );
      vCheckConstraintVonBis.setRule (
        vKuerzel + "_" + vGueltigVonName + " <= " +
        vKuerzel + "_" + vGueltigBisName );

      vTable.setDirty(true);
    }
  }
}

JOptionPane.showMessageDialog(null, vText);
```

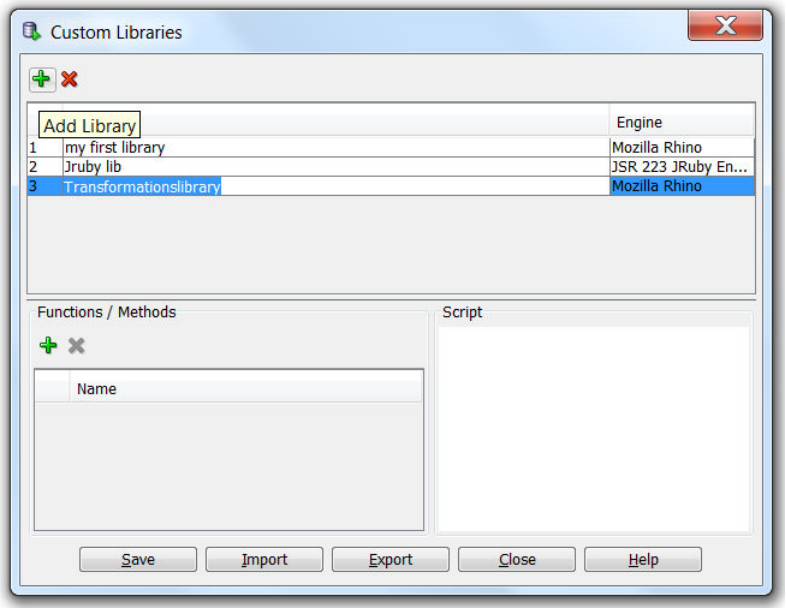
Libraries

Die Skripte können in Libraries zusammengefasst und verwaltet werden. Da hierfür eine komplette JavaScript-Datei als Library im SQL Developer hinterlegt wird, können dann auch übergreifende Variablen und Methoden deklariert und verwendet werden.

Um die Library-Funktionalität zu nutzen, sind mehrere Schritte nötig:
Zunächst wird die Library als JavaScript-Datei erstellt.
Dann wird im SQL Developer über das Menü „Tools → Data Modeler → Design Rules → Libraries“ das Fenster für die Libraries geöffnet:

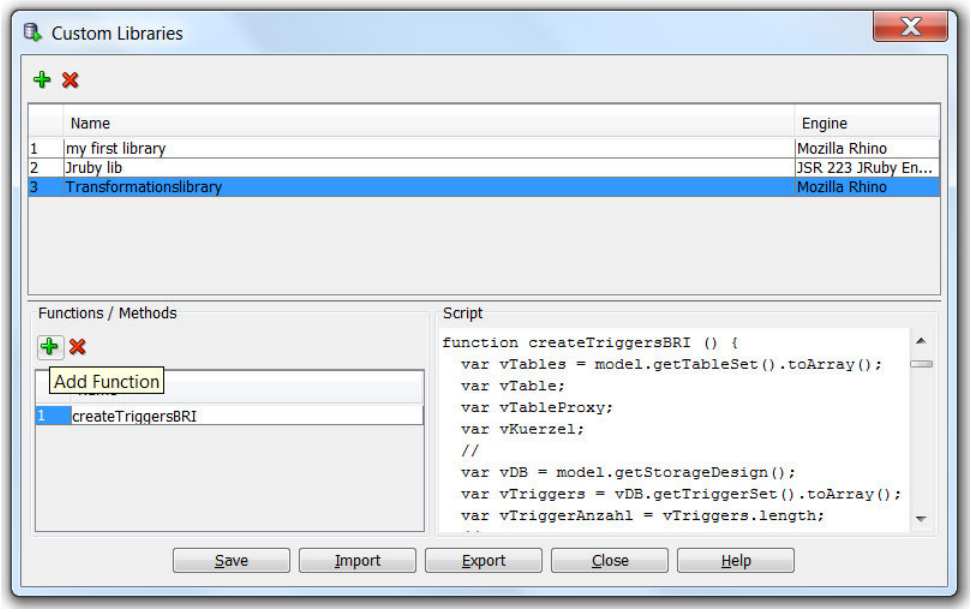


In diesem Fenster wird über den oberen Button  eine neue Library angelegt:
Wie bei den Skripten muss für die Library ein sprechender Name angegeben und die gewünschte Engine ausgewählt werden.

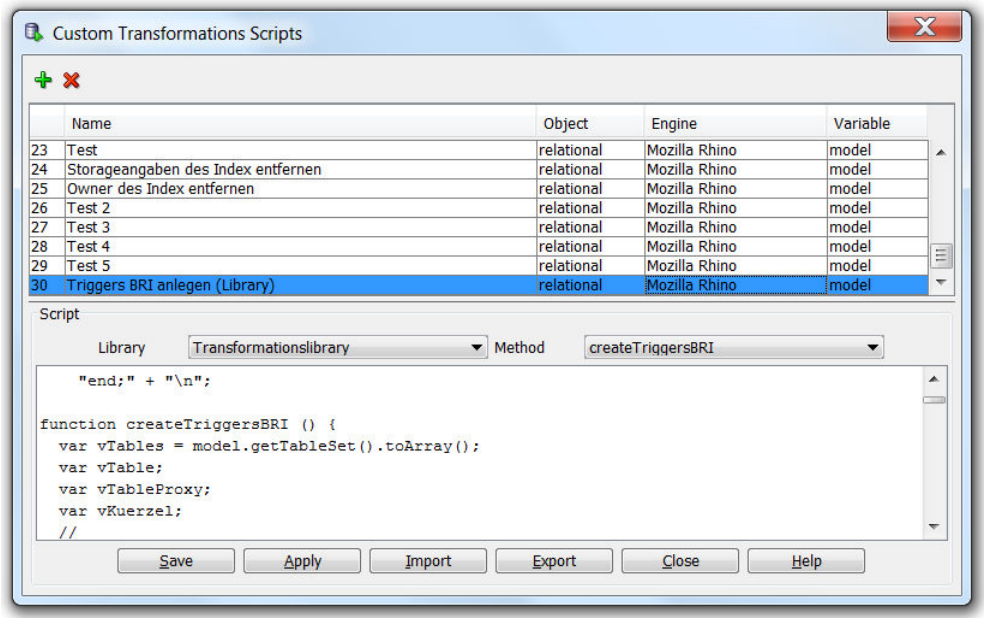


In das Feld „Script“ wird der komplette Code der Library kopiert.

Damit eine Methode in der Library als Skript verwendet werden kann, muss sie im Bereich links neben „Script“ bekanntgemacht werden. Dazu wird mit dem unteren Button  eine neue Zeile angelegt und dort der Name der Function wie in der Library eingetragen:



Um die jetzt eingetragene Funktion als Skript zu verwenden, wird im Fenster der Transformations ein neues Skript angelegt und mit einen Namen, dem Object und der Engine versehen. Für das Skript wird kein Code eingetragen, sondern unter „Library“ wird die gewünschte Library ausgewählt. Unter „Methods“ kann dann aus allen bekanntgemachten Methoden der Library die gewünschte Funktion gewählt werden:



Der Code-Bereich zeigt nach der Auswahl der Library den kompletten Code der Library an, Änderungen sind hier aber natürlich nicht möglich.

Fazit

Mit den Transformations kann der Data Modeler einfach um Skripte erweitert werden, die automatisiert Standardaufgaben durchführen. Durch die Möglichkeit, weitere Scripting Engines zusätzlich zu Mozilla Rhino einzubinden, können die Skripte in der persönlich bevorzugten Skript-Sprache erstellt werden.
Über Libraries lassen sich die Skripte organisieren, wiederverwendbare Teile können dort einmal hinterlegt und mehrfach aufgerufen werden.

Kontaktadresse:

Dr. Gudrun Pabst
Trivadis GmbH
Lehrer-Wirth-Straße 4
D-81829 München

Telefon: +49 (0) 89-99 27 59 30
Fax: +49 (0) 89-99 27 59 59
E-Mail gudrun.pabst@trivadis.com
Internet: www.trivadis.com