MyCoRe-Importer

-- Dokumentation --

Inhaltsverzeichnis

1. Connect/Retrieve/Convert.....1
  1.1 Connect.....1
  1.2 Retrieve.....1
  1.3 Convert.....2
  1.4 Ein kurzes Beispiel.....2
2. Die Import-Datei.....3
  2.1 Konfiguration.....3
  2.2 Mapping.....4
    2.2.1 Get started.....4
    2.2.2 Mehrere Felder.....4
    2.2.3 Feld Bedingung.....5
    2.2.4 Escaping.....5
    2.2.5 URI-Resolver.....5
    2.2.6 Attribute.....6
    2.2.7 Kindelemente.....6
    2.2.8 Typen.....7
  2.3 Ein Beispiel (Weiterführung des obrigen).....8
  2.4 Erstellung der xml-Dateien.....9
3. Import.....9

In der Regel wird für jedes Projekt und für jeden Import ein Projektspezifischer Importer entwickelt. Das hat sich durch die unterschiedlichen Formate und Ausnahmebedingungen, die zweifelsohne Auftreten, so durchgesetzt. Die Programmstruktur und viele Programmabschnitte sind jedoch bei jedem Importer identisch oder zumindest sehr ähnlich. Den die Problematik ist immer die gleiche: Wie kann man die Daten des Ausgangsformats fehlerfrei und effizient in das Zielformat überführen?

Der MyCoRe-Importer bietet dem Importentwickler eine einfache, feste und logische Programmstruktur die ihm den strukturellen Teil der Arbeit abnimmt. Diese Programmstruktur ist in mehrere Teile gegliedert und für jeden Import identisch:

- Verbindung zur Quelle herstellen und Daten lesen
- gelesene Daten in ein Importer verständliches Format konvertieren
- konvertierte Daten auf MyCoRe-Objekte abbilden (mapping)
- Import der MyCoRe-Objekte

Dem Importentwickler soll dabei soviel Arbeit wie Möglich abgenommen werden, so das er sich ausschließlich auf die Konvertierung der Daten konzentrieren kann.

1. Connect/Retrieve/Convert

1.1 Connect

Der erste Schritt bei der Implementierung eines Importers ist die Verbindung zur Quelle. Das kann eine Datenbank, eine xml-Datei, eine Webseite usw. sein. Um die Verbindung herzustellen kann das Interface **MCRImportConnector<C>** verwendet werden. Dieses bietet die Methoden *connect* und *close* an. Wobei die *connect*-Methode ein Objekt des Typs **C** zurückliefert. Wie dieser Typ aussieht und verwendet wird ist dem Import-Entwickler überlassen.

1.2 Retrieve

Nachdem die Verbinung zur Quelle steht können Daten von dieser gelesen werden. Dieser Schritt ist sehr importspezifisch und eng mit dem folgenden Schritt, der Konvertierung, verbunden. Aus diesem Grund bietet der MyCoRe-Importer hierzu kein spezielles Interface. Prinzipiell gibt es zwei Möglichkeiten die Daten zu lesen und zu verarbeiten:

1. Es werden zuerst alle Daten gelesen und anschließend verarbeitet.
2. Es werden in einer Schleife Blöcke von Daten gelesen und direkt verarbeitet. Diese Variante bietet die Möglichkeit Threads zu verwenden. Auch das spätere Mapping könnte somit asynchron erfolgen.

1.3 Convert

Warum konvertieren?
Der MyCoRe-Importer stellt eine allgemeingültige API bereit die für alle Import-Aufgaben funktionieren soll. Um dies zu gewährleisten, müssen alle Daten von jeder Quelle in ein MyCoRe-Importer verständliches Format umgewandelt werden. Dieses Format wird später von der Mapping-Komponente verwendet.

In MyCoRe gibt es zwei Objekttypen, **MCRObject** und **MCRDerivate**. Für beide stellt der Importer ein Konvertier-Interface bereit. Das sind **MCRImportRecordConverter<T>** und **MCRImportDerivateConverter<T>**. Eine Klassen die eins der beiden Interfaces verwendet, muss die Methode convert implementieren. In dieser wird exakt EIN Objekt umgewandelt. Es wird also entweder eine **MCRImportRecord**- oder eine **MCRImportDerivate**-Instanz generiert.

MCRImportRecord
Ein Record ist die vereinfachte Version eines MyCoRe-Objekts. Es besitzt einen Namen, welches dem Objekttyp entspricht (person, institution etc.) und eine Liste von Feldern. In diesen Feldern (**MCRImportField**) werden die Informationen des Datensatzes (**T)** gespeichert.

MCRImportField
Eine Feld besteht aus einem Id/Wert-Paar. Jede Information die aus dem Quellformat gespeichert werden soll benötigt also eine Id. In der Regel sollte diese Id eindeutig sein, was aber nicht zwingend erforderlich ist. Sinnvoll ist dies z.B. bei alternativen Namen. Diese können durchaus mehr als einmal pro Person auftreten.

MCRImportDerivate
Ein Importderivat ist die Abstraktion eines **MCRDerivate**. Es besteht aus einer id, einem label und einer Liste von Dateien die zu diesem Derivat gehören.

1.4 Ein kurzes Beispiel

Angenommen die Aufgabe ist es eine Personendatenbank in MyCoRe zu importieren. Der erste Schritt ist die Verbindung zur Datenbank mit der Methode *connect* des **MCRImportConnector<T>**. Steht die Verbindung muss eine Entscheidung bezüglich des retrieving/converting getroffen werden. In diesem Beispiel ist es einfacher erst alle Daten zu lesen und diese anschließend zu konvertieren. Gelesen wird mithilfe eines select-Befehls. Hat die Tablle zum Beispiel die Spalten "Vorname, Nachname, Geschlecht, Geburtsdatum" dann können diese mit einem **MCRImportRecordConverter** in einen **MCRImportRecord** konvertiert werden:

```
/* die Ergebnis-Liste – wird später benötigt um die generierten Records
weiterzuverarbeiten */
List<MCRImportRecord> recordList = new ArrayList<MCRImportRecord>();
// starte die select-Abfrage und speichere die Ergebnisse in einer Liste
List<SelectResult> selectResultList = select.getResults();
// erstelle eine neue Instanz des Konvertieres
MyRecordConverter converter = new MyRecordConverter();
// durchlaufe die select Ergebnisse
for(SelectResult sr : selectResultList) {
  // konvertiere ein select result und erzeuge eine MCRImportRecord
  MCRImportRecord record = converter.convert(sr);
  // speichere den record in der Ergenis-Liste
  recordList.add(record);
}

public class MyRecordConverter<SelectResult> implements MCRImportRecordConverter {
  public MCRImportRecord convert(SelectResult sr) {
    // erstelle einen neuen MCRImportRecord vom Typ "person"
    MCRImportRecord record = new MCRImportRecord("person");
    // füge die verschiedenen Felder dem Record hinzu
    record.addField(new MCRImportField("lastname", sr.get(0)));
    record.addField(new MCRImportField("firstname", sr.get(1)));
    record.addField(new MCRImportField("gender", sr.get(2)));
    record.addField(new MCRImportField("birthdate", sr.get(3)));
    return record;
  }
}
```

Das Ergebnis stellt eine Liste von MCRImportRecords da die mithilfe des Mappings weitervarbeitet werden kann.

2. Die Import-Datei

Für die weitere Verarbeitung der Daten wird eine xml-Datei benötigt. Die sogenannte Import-Datei. Diese Datei ist für jeden Import individuell anzulegen. Als root-Tag muss "import" angegeben werden. Unterhalb des root-Tags ist sie in zwei Abschnitte gegliedert, der Konfiguration und das Mapping.

2.1 Konfiguration

Der Konfigurationsteil ist sowohl für das Mapping als auch den späteren Import in MyCoRe relevant. Das config-Element stellt dabei den Einstieg dar. Im folgenden ein vollständiges Beispiel einer Konfiguration:

```
<config>
  <projectName>DocPortal</projectName>
  <datamodelPath>/home/user/app/docportal/modules/app/config/</datamodelPath>
  <saveToPath>/home/user/import-dir/</saveToPath>
  <createClassificationMapping>false</createClassificationMapping>
  <derivates use="true">
    <createInImportDir>true</createInImportDir>
    <importToMycore>true</importToMycore>
    <importFilesToMycore>false</importFilesToMycore>
  </derivates>
</config>
```

projectName: Hier wird der Projektname angegeben. Dieser String wird bei allen Objekten die später importiert werden soll als Projektteil angenommen. Beispiel: {projectName}_author_00000001 -> DocPortal_author_00000001

datamodelPath: Der Pfad zum Datenmodel-Ordner. Er wird später erweitert mit dem datamodel-Attribut eines mcrobject. Diese Pfad ist zwingend anzugeben, es werden verschiedene Informationen aus den Datenmodellen benötigt. Beispiel: {datamodelPath}{datamodel}

saveToPath: Dieser Pfad gibt an wo die gemappten xml-Dateien gespeichert werden. Für jedes Objekt wird ein Unterordner mit dem Namen des Objekts erstellt. Ist das Klassifikationsmapping aktiv wird zusätzlich der Ordner "classification" angelegt.

createClassificationMapping: Das Klassifikationsmapping ist standardmässig aktiviert und muss in diesem Fall nicht explizit angegeben werden. Hat man einmal seine Klassifikationsmapping-Dateien ausgefüllt, kann dieses Flag auf *false* gesetzt werden um seine Dateien nicht zu überschreiben. Weitere Informationen zum Klassifikationsmapping stehen unter *2.2.8 - Typen*.

derivates: Dieses element definiert nur ob Derivate verwendet werden oder nicht. Standardmässig ist es auf false eingestellt und muss somit auch nicht angegeben werden wenn Derivate keine Rolle beim Import spielen.

createInImportDir: Definiert ob beim mapping die Derivat-xml-Objekte angelegt werden. Wenn aktiv, werden die xml-Dokumente im Unterordner "derivates" des *saveToPaths* gespeichert.

importToMycore/importFilesToMycore: Ist das **importToMycore** Flag aktiv, dann werden die xml-Derivate beim Import in das System mit übernommen. Die Dateien welche das Derivat enthält sind davon unbeeinflußt. Sie können mit dem Flag **importFilesToMycore** importiert werden. Diese Funktionalität ist zweigeteilt, weil man nicht zwingend die gesamten dazugehörigen Bilder und Dateien mit hochladen möchte.

2.2 Mapping

Das Mapping stellt die Verbindung zwischen dem Importer-Format (**MCRImportRecord**, **MCRImportDerivate**) und der MyCoRe xml-Objektrepräsentation dar. Das Mapping bietet dabei die Möglichkeit, mit relativ wenig (oder gar keinem!) Programmieraufwand, die Records in MyCoRe-Objekte umzuwandeln. Im umschließenden mapping-Tag können die einzelnen MyCoRe-Objektrepräsentation angegeben werden. Die generelle Struktur sieht so aus:

```
<mapping>
  <resolvers>
    <resolver prefix="myPrefix" class="org.mycore.dataimport.resolver.uri.MyResolver" />
    ...
  </resolvers>

  <mcrobjects>
    <mcrobject name="person" datamodel="datamodel/person.xml">
    ...
  </mcrobjects>
</mapping>
```

Das mapping-tag hat zwei Kinder. Im resolvers-tag kann eine Liste von URI-Resolvern angegeben werden. Wie dies genau funktioniert und welche Aufgaben URI-Resolver übernehmen wird in Unterpunkt 5. näher erläutert.

Im mcrobjects-tag geschieht das eigentliche mapping der MyCoRe-Objekte. Für jedes MyCoRe-Objekt welches gemappt werden soll muss ein Kindelement mit dem tag mcrobject definiert werden. Dieses enthält zwei Attribute. Den Namen des Objekts (identisch mit dem Namen eines Records) und einen Unterpfad zum Datenmodel. Weiterhin hat jedes mcrobject-Element eine Liste von "mappings". Diese mappings bilden die Felder eines Records auf die MyCoRe-xml-Struktur ab. Wie diese Syntax genau funktioniert wird an den folgenden Beispielen gezeigt.

2.2.1 Get started

Bei der einfachsten Form des mappings muss nur das Feld und der Zielmetadatenabschnitt angegeben werden. Der Wert aus dem Feld *var1* wird in diesem Fall in den Textbereich geschrieben.

```
<map fields="var1" to="metadata" />
```

```
<def.metadata class="MCRMetaLangText">
  <metadata xml:lang="de" form="plain">Wert aus var1</metadata>
</def.metadata>
```

In diesem Beispiel wurde davon ausgegangen das *metadata* vom Typ MCRMetaLangText ist. Für diesen Typ wird standardmässig (falls nicht anders angegeben) die Sprache auf „de“ und die *form* auf „plain“ gestellt. Diese Informationen werden alle dynamisch aus dem Datenmodel ausgelesen.

Das Feld *var1* entspricht einen **MCRImportField**. Besitzt ein **MCRImportRecord** kein **MCRImportField** mit dieser id (*var1*), dann wird dieser mapping Abschnitt vollständig igonoriert. Das heißt das die finale xml-Datei, kein def.metadata-Element enthalten wird.

2.2.2 Mehrere Felder

Möchte man mehrere Felder in einen Metadatenabschnitt speichern, müssen diese mit Komma getrennt im fields-Attribut angegeben werden.

```
<map fields="var1,var2" to="metadata" />
```

In diesem Fall werden die Werte der beiden Felder zusammengefügt und im Textbereich gespeichert. Jedoch ist ein einfaches zusammenfügen oft nicht erwünscht. In dieser Situation müssen die Felder getrennt dargestellt werden. Mithilfe des text-Elements und des value-Attributs kann man Felder und Text beliebig kombinieren.

```
<map fields="var1,var2" to="metadata">
  <text value="{var1}, {var2}" />
</map>
```

```
<metadata xml:lang="de" form="plain">Wert aus var1, Wert aus var2</metadata>
```

Wie man an dem Beispiel sieht, werden die beiden Felder durch ein Komma und ein Leerzeichen getrennt dargestellt. Felder werden im Importer immer innerhalb einer geschweifte Klammer angegeben.

2.2.3 Feld Bedingung

Leider sind die zu importierenden Daten selten vollständig. Oft trifft man z.B. auf Personen die nur einen Nachnamen und keinen Vornamen besitzen. Nimmt man jetzt folgende Import-Vorgaben an:

- Nach- und Vorname existiert → Trennung durch Komma + Leerzeichen
- nur der Nachname existiert → nimm nur diesen

Für dieses Problem wurden im Importer die eckigen Klammern eingeführt. Ein Feld welches in eine eckige Klammer eingebettet ist muss einen Wert besitzen, ansonsten wird die gesamte Klammer ignoriert.

```
<map fields="nachname,vorname" to="name">
  <text value="{nachname}[, {vorname}]" />
</map>
```

Ist der Wert des Feldes 'vorname' leer wird die gesamte Klammer '[, {vorname}]' ignoriert. Ein weiteres hübsches Beispiel für diese Funktionalität ist folgende Datumsangabe.

```
<map fields="year,month,day" to="date">
  <text value="{year}[-{month}[-{day}]]" />
</map>
```

2.2.4 Escaping

Der Importer unterstützt escaping, die vordefinierten Klammern können also trotzdem als Text verwendet werden.

```
<map fields="nachname,vorname" to="name">
  <text value="\{{nachname}\}[, {vorname}]" />
</map>
```

```
<metadata xml:lang="de" form="plain">{Mustermann}, Manfred</metadata>
```

2.2.5 URI-Resolver

Nicht alle Werte können mit diesem Schema aufgelöst werden. Aus diesem Grund gibt es die Möglichkeit eigene URIResolver im config-Abschnitt der xml-Datei zu definiert. Die angegebene Klasse muss das Interface **MCRImportURIResolver** implementieren.

```
<resolvers>
  <resolver prefix="resolverId"
    class="org.mycore.dataimport.pica.resolver.uri.MyResolver" />
  ...
</resolvers>
```

Im mapping-Abschnitt kann auf diese dann zugegriffen werden.

```
<map fields="var1,var2" to="metadata">
  <text value="{var1}, {var2}" resolver="resolverId"/>
</map>
```

Dabei werden zuerst die Felder aufgelößt und das Ergebnis anschließend mit dem Resolver weiterverarbeitet.

2.2.6 Attribute

Attribute können analog zu Text auf folgende Weise angegeben werden:

```
<map fields="var1,var2" to="metadata">
  <attributes>
    <attribute name="metainfo1" value="{var1}" />
    <attribute name="metainfo2" value="[Info2: {var2}]" />
  </attributes>
</map>
```

```
<metadata metainfo1="Wert aus var1" metainfo2="Info2: Wert aus var2" />
```

Einige Attribute verwenden Namespaces, diese können über das namespace-Attribut mit angegeben werden. Um Schreibarbeit zu sparen sind die meisten Namespaces über Kürzel ansprechbar. Dazu zählen xml, xlink, xsi, xsl und dv. Alle anderen müssen mit der Syntax „kürzel,url“ angegeben werden.

```
<attributes>
  <attribute name="metainfo1" value="{var1}" namespace="xlink" />
  <attribute name="metainfo2" value="text" namespace="myns,www.myns.net/myns/" />
</attributes>
```

Natürlich ist es auch für Attribute möglich eigene Resolver zu verwenden. Dies geschieht analog zu dem Textelement.

```
<attribute name="metainfo1" value="{var1}" resolver="resolverId" />
```

2.2.7 Kindelemente

Um Kindelemente in die MyCoRe-xml-Struktur zu integrieren wird das children-Element verwendent. Jedes children-Element kann eine Reihe von child-Elementen besitzen die über das tag-Attribut definiert werden.

```
<map fields="var1,var2" to="metadata">
  <children>
    <child tag="lastName">
      <text value="{var1}" />
    </child>
    <child tag="firstName">
      <text value="{var2}" />
    </child>
  </children>
</map>
```

Alle child-Elemente unterstützen dieselben Funktionen wie das map-Element. Es können Text, Attribute, Bedingungen, Resolver und weitere Kinder definiert werden.

```
<children>
  <child tag="child">
    <attributes>
      <attribute name="childAttr" value="{var0}" resolver="myRes" />
    </attributes>
    <children>
```

```
      <child tag="subChild">
        <text value="{var1}[ : {var2}]" />
       </child>
    </children>
  </child>
</children>
```

```
<def.metadata class="MCRMetaXML" >
  <metadata xml:lang="de">
    <child childAttr="Wert aus var0">
      <subChild>Wert aus var1 : Wert aus var2</subChild>
    </child>
  </metadata>
</def.metadata>
```

2.2.8 Typen

Bisher wurden ausschließlich Fälle betrachtet wo die zu importierenden Daten auf einen Metadatensatz abgebildet werden. Die MyCoRe-xml-Struktur ist aber etwas komplexer. Zusätzlich zu den Metadaten gibt es noch Eltern, Service-Informationen, die Objekt-ID, das Objekt-Label etc. Um diese Informationen abzubilden wird das type-Attribut verwendet. Standardmässig ist type immer auf "metadata", alle anderen müssen angegeben werden. Es ist wichtig Anzumerken, daß jeder Typ eine unterschiedliche xml-Definitions-Syntax verwendet, da jeder eine individuelle Aufgabe verfolgt.

Im folgenden eine Liste von Typen die zusätzlich zu den metadaten implementiert sind.

ID:
Für jedes MyCoRe-Import-Objekt wird eine Id benötigt. Sie muss nicht MyCoRe valide sein (soetwas wie DocPortal_author_00000004). Es reicht aus, wenn sie für jedes Objekt eindeutig ist. Während des finalen Schritts, dem Import zu MyCoRe, wird jede intere Import-Id mit einer gültigen MyCoRe-Id ersetzt. Das gleiche gilt für Link- und Parentids. Auch diese Ids müssen den Import-Ids entsprechen und werden über diese referenziert.

Der Importentwickler hat zwei Möglichkeiten. Per Standardeinstellung wird die Id für jedes Objekt nach dem Schema "{recordName}_{aufsteigende Nummer}" automatisch genieriert. Um selbst die Kontrolle über die Id zu erhalten kann man den Typ "id" verwenden:

```
<map fields="id" type="id" value="authorId_{id}" resolver="authorRes"/>
<mycoreobject ID="authorId_08033301">
```

Um Objekte innerhalb eines Importvorgangs zu referenzieren, ist es immer besser die Id selbst zu erzeugen.

Label:
```
<map fields="label" type="label" value="{label}" />
<mycoreobject ID="DocPortal_author_08033301" label="Mein Label!" >
```

Parent (identisch zu einem normalen Link):
```
<map fields="link_href,link_label" type="parent" >
  <attributes>
    <attribute name="href" namespace="xlink" value="{link_href}" />
    <attribute name="label" namespace="xlink" value="{link_label}" />
  </attributes>
</map>
```

Kinder werden bei der Importstruktur von MyCoRe nicht expliziet mit angegeben. Die MyCoRe-Import Funktionalität geht davon aus, das das Eltern-Objekt zuerst

angelegt wurde und damit immer existiert.

Klassifikationen
Klassifikationen werden mit einer Ausnahme wie normale Metadaten behandelt. Zusätzlich wird zu jeder Klassifikation im "classification"-Verzeichnis des Speicherpfads eine Klassifikationsmapping-Datei angelegt. Diese Dateien müssen nach dem Mapping manuell vervollständigt werden. Beim Import in das System werden die Klassifkationsmapping-Dateien mit den Importdateien verknüpft um die korrekten Werte einzubinden. Dieses Klassifikationsmapping kann mit der Konfigurationsanweisung
`<createClassificationMapping>false</createClassificationMapping>` ausgeschaltet werden.

Multi data
Wie in Punkt 1.3 (Convert - MCRImportField) schon angedeutet, kommt es vor, das ein **MCRImportRecord** mehrere Felder mit der gleichen Id besitzt. Zum Beispiel bei alternativen Namen:

- Wolfgang Amadeus Mozart
- Joannes Chrysostomus Wolfgangus Theophilus Mozart

Angenommen wir haben eine Person Mozart mit genau diesen beiden alternativen Namen. Beide Felden verwenden als Id "altName". Im MyCoRe Datenmodel von Personen werden alternative Namen mit dem tag "alternativeName" gespeichert. Um diese beiden Namen nun zu mappen wurde der Typ "multidata" eingeführt.

```
<map fields="altName" to="alternativeName" type="multidata" />
```

Diese Zeile wäre vollkomen ausreichend wenn es sich bei dem Ziel-Metadatensatz um einen MCRMetaLangText hält. Das Ergebnis würde folgendermaßen aussieht:

```
<def.alternativeName class="MCRMetaLangText" >
  <alternativeName xml:lang="de" form="plain">
    Wolfgang Amadeus Mozart
  </alternativeName>
  <alternativeName xml:lang="de" form="plain">
    Joannes Chrysostomus Wolfgangus Theophilus Mozart
  </alternativeName>
</def.metadata>
```

Theoretisch ist es auch möglich, Felder mit unterschiedlichen Ids auf einen Metadatensatz zu mappen:

```
<map fields="altName,altName2,testName" to="alternativeName" type="multidata">
  <text value="[{altName}][{altName2}][{testName}]" />
</map>
```

Hierbei wird jedes Feld einzeln betrachtet. Durch das Ausschlußverfahren wird nur das aktuelle Feld im Text angezeigt.

Eigene Typen
Um eigene Typen in das Importsystem zu integrieren muss die aufzulösende Klasse das Interface **MCRImportMapper** implementieren und an der Klasse **MCRImportMapperManager** mit der Methode *addMapper* registriert werden. Der dort angegebne Typ kann dann in der mapping Datei verwendet werden.

2.3 Ein Beispiel (Weiterführung des obrigen)

Die Personenrecords mit ihren vier Feldern müssen nun auf die MyCoRe-Objektstruktur "gemapped" werden. Die entsprechende Import-Datei könnte z.B. So aussehen:

```
<import>
  <config>
   <projectName>DocPortal</projectName>
   <datamodelPath>/home/usr/app/jportal/modules/app/config/</datamodelPath>
   <saveToPath>/home/usr/import-dir/</saveToPath>
  </config>

  <mapping>

   <mcrobjects>
     <mcrobject name="person" datamodel="datamodel/person.xml">
       <!-- LABEL -->
       <map fields="lastname,firstname" type="label" value="{lastname}[,{firstname}]" />

       <!-- NAMES -->
       <map fields="firstname,lastname" to="heading" >
         <children>
           <child tag="lastName">
             <text value="{lastname}" />
           </child>
           <child tag="firstName">
             <text value="{firstname}" />
           </child>
         </children>
       </map>

       <!-- GENDER -->
       <map fields="gender" to="heading" >
         <attributes>
           <attribute name="classid" value="urmal_class_000000001" />
           <attribute name="categid" value="{gender}" />
         </attributes>
       </map>

       <!-- DATE OF BIRTH -->
       <map fields="dateofbirth" to="dateOfBirth" />
      </mcrobject>
    </mcrobjects>
  </mapping>
</import>
```

Jedes MyCoRe-Objekt benötigt eine eindeutige Id. Da in diesem Beispiel in der Tabelle keine Id existiert, muss sie automatisch generiert werden. Dies gewährleistet der URI-Resolver *idGen*. In dieser Klasse wird an den Ursprungswert (app_person_) eine sich steigerne statische Zahl angehängt. So erhält jedes Objekt eine eindeutige Id. Dieser URIResolver ist schon implementiert und kann von jedem Importentwickler verwendet werden. Siehe *org.mycore.importer.mapping.resolver.uri.**MCRImportIdGenerationURIResolver.***

Anschließend wird für das MyCoRe-Objekt das Label im root-tag gesetzt. Dieses setzt sich zusammen aus den Vor- und Nachnamen die durch ein Komma und ein Leerzeichen getrennt sind. Existiert kein Vorname wird auschließlich der Nachname als Label gesetzt.

Danach werden die Metadaten Name, Geschlecht und das Geburtsdatum in die xml-Datei geschrieben. In diesem Beispiel ist Metadatenfeld *heading* als MCRMetaXML definiert. Erwartet werden die beiden Kindelemente *lastName* und *firstName*, in dennen jeweils der entsprechende Name als Text steht. Im mapping-File lässt sich

diese Struktur durch das children-Element darstellen. Das Geschlecht (gender) ist als Klassifikation definiert. Eine Klassifikation benötigt genau zwei Attribute. Zum einen die *classid*, welche die Id der Klassifikation bestimmt und zum anderen die *categid*, welche den Wert der Klassifkation definiert. Im letzten mapping Abschnitt wird das Geburtsdatum definiert. In diesem Beispiel wird davon ausgegangen das die Datumsangabe vom Quellformat in der Form YYYY-MM-DD vorliegt. Würde ein anderes Format vorliegen müsste dies mit einem eigenen URI-Resolver konvertiert werden.

2.4 Erstellung der xml-Dateien

Steht die Import-Datei kann mit der eigentlichen Erstellung der xml-Dateien begonnen werden (welche dann später importiert werden sollen). Es ist nicht notwendig das MyCoRe (DB, Lucene etc.) läuft, ein eigenens kleines Java-Projekt welches den Importer einbindet ist völlig ausreichend.

Als Vorraussetzung für diesen Schritt wird eine Liste von **MCRImportRecord**/**MCRImportDerivate** (*siehe 1.4*) und die Import-Datei benötigt. Diese Liste von Records/Derivaten wird nun mithilfe der Import-Datei gemapped. Als Ergebnis erhält man für jeden Record/Derivate ein entsprechende xml-Datei im *saveToPath*.

Die Singleton-Klasse **MCRImportMappingManager** stellt den Einstieg für das Mapping dar. Sie muss am Anfang mit der Import-Datei verknüpft werden. Dies geschieht über die *init*-Methode. In fast allen Fällen kann das Mapping dann schon gestartet werden. Hierfür ist die Methode *startMapping(Liste von MCRImportRecords)* zuständig. Verwendet man Derivate, muss vor dem Aufruf von *startMapping()*, die Methode *setDerivateList(Liste von MCRImportDerivate)* aufgerufen werden. Der Rest geschieht vollkommen automatisch.

Mapping Status
Um sich über den Status des Mappings zu informieren ist es möglich einen **MCRImportStatusListener**/**Adapter** zu registrieren.

Eigene Mapper
Eigene Mapper können mithilfe des MCRImportMapperManager hinzugefügt werden. Dabei reicht es aus den Typ und die Klasse anzugeben.

```
MCRImportMappingManager mm = MCRImportMappingManager.getInstance();
mm.getMapperManager().addMapper("type name", MyMapper.class);
```

3. Import

Sind die xml-Objekte einmal erstellt, dann ist auch der Import zu MyCoRe keine schwierige Angelegenheit mehr. Der folgende Codeabschnitt muss bei laufender MyCoRe-Umgebung ausgeführt werden, am besten in einem CLI-Kommando.

```
MCRImportImporter im = new MCRImportImporter(new File("Pfad zur Import-Datei"));
im.startImport();
```

Wie beim mapping kann auch beim Import ein **MCRImportStatusListener/Adapter** registriert werden um sich über den Fortschritt zu informieren.