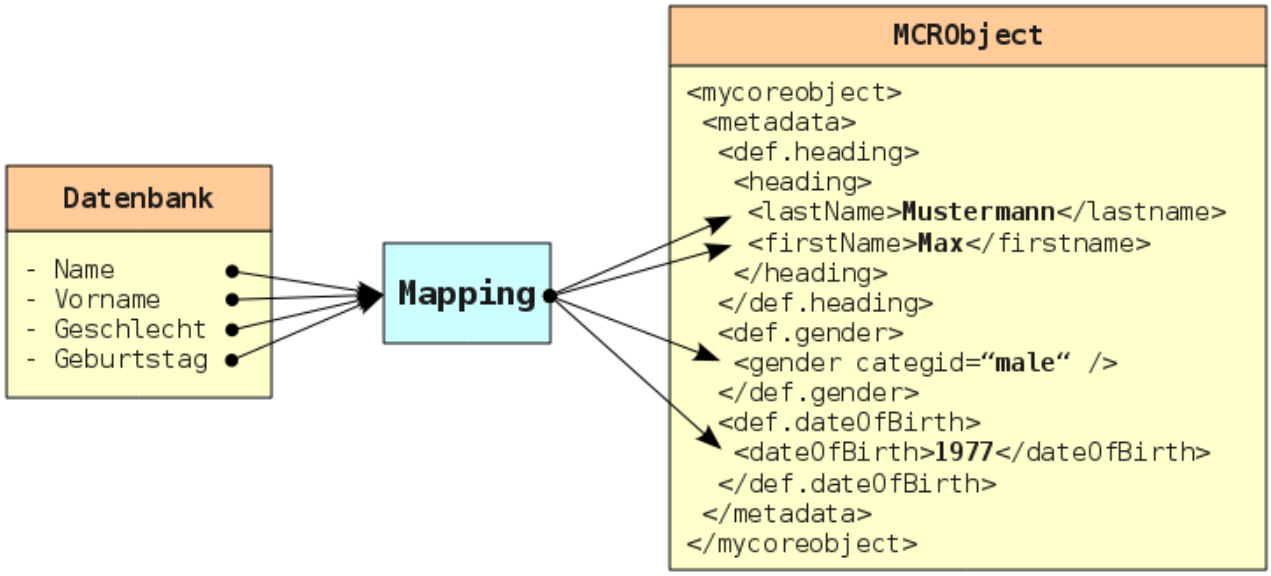
MyCoRe-Importer

-- Dokumentation --

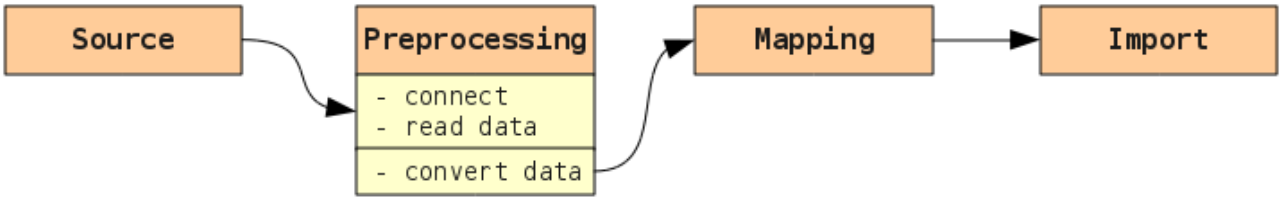
Inhaltsverzeichnis

1. Einleitung

Der MyCoRe-Importer bietet dem Importentwickler ein Migrationstool mit einer einfachen, festen und logischen Programmstruktur. Die Idee hinter dem Importer ist ein Abbildungsverfahren (Mapping). Jede Information einer Datenquelle, z. B. der Name einer Person in einer Datenbank, gehört an eine ganz bestimmte Stelle im MyCoRe-Objekt. Im Importer wird dieses Mapping über eine xml-Datei geregelt.



Der MyCoRe-Importer unterstützt natürlich nicht nur Datenbanken, auch andere Quellen, wie z.B. xml-Dateien sind denkbar. Dazu müssen die Quelldaten vorverarbeitet (preprocessing) und in ein propiotäres, für das Mapping verständliches, Format konvertiert werden.



Der Importentwickler hat also zwei Hauptaufgaben. Zuerst muss er eine Verbindung zur Quelle herstellen, Daten von dieser lesen und sie in ein Importer verständliches Format konvertieren. Anschließend können diese Daten mittels des Abbildungsverfahren in MyCoRe-Objekte umgewandelt werden. Ein einfacher CLI-Befehl reicht dann aus, um die Daten in das MyCoRe-System zu importieren.

2. Vorverarbeitung

2.1. Verbinden

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. Im folgeden ein Beispiel für die Verbindung zu einer MySQL Datenbank.

```
public class MyConnector implements MCRImportConnector<Connection> {
  protected Connection conn;
  public Connection connect() {
    final String hostname   = "localhost";
    final String port       = "3306";
    final String dbname     = "myDB";
    final String user       = "root";
    final String password   = "abc123";
    try {
      // Treiber laden
      Class.forName("org.gjt.mm.mysql.Driver").newInstance();
    } catch (Exception e) {
      e.printStackTrace();
    }
    try {
      // Verbindung aufbauen
      String url = "jdbc:mysql://"+hostname+":"+port+"/"+dbname;
      conn = DriverManager.getConnection(url, user, password);
    } catch (SQLException sqlExc) {
      sqlExc.printStackTrace();
    }
    return conn;
  }
  public void close() {
    try {
      conn.close();
    } catch(SQLException sqlExc) {
      sqlExc.printStackTrace();
    }
  }
}
```

2.2. Daten lesen

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.

```
// Build connection
MyConnector connector = new MyConnector();
Connection con = connector.connect();
// do query
String qry = "SELECT lastname,firstname,gender,bithdate FROM persons";
```

```
Statement stmt = con.createStatement(ResultSet.TYPE_FORWARD, ResultSet.READ_ONLY);
ResultSet result = stmt.executeQuery(personen);
```

2.3. Daten konvertieren

In diesem Abschnitt geht es darum die Quelldaten in ein für das Abbildungsverfahren verständliches Format zu konvertieren. Das Format ist sehr einfach gehalten und besteht aus lediglich drei Klassen. Das sind **MCRImportRecord**, **MCRImportDerivate** und **MCRImportField**. Bevor diese näher erläutert werden kurz etwas zu den Konvertierschnittstellen. Der Importer bietet zwei Interfaces **MCRImportRecordConverter<T>** und **MCRImportDerivateConverter<T>** um die oben genanten Objekte zu erzeugen. Diese Interfaces können verwendet werden um den Konvertiervorgang zu vereinheitlichen. Beide Interfaces bietet die Methode *convert(T)* an. In dieser wird genau EIN Objekt umgewandelt. Wie das genau funktioniert wird in den unten stehenden Beispielen demonstriert.

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 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 (siehe Typ "multidata" 3.2.8).

```
ArrayList<MCRImportRecord> recordList = new ArrayList<MCRImportRecord>();
// erstelle eine neue Instanz des Konvertieres
MyPersonConverter converter = new MyPersonConverter();
// durchlaufe die Datensätze der SELECT-Anweisung
while(result.next()) {
  // konvertiere ein select result und erzeuge eine MCRImportRecord
  MCRImportRecord record = converter.convert(result);
  // speichere den record in der Ergenis-Liste
  recordList.add(record);
}

public class MyPersonConverter<ResultSet> implements MCRImportRecordConverter {
  public MCRImportRecord convert(ResultSet rs) {
    // erstelle einen neuen MCRImportRecord vom Typ "person"
    MCRImportRecord record = new MCRImportRecord("person");
    // füge die verschiedenen Felder dem Record hinzu
    record.addField(new MCRImportField("lastname", rs.getString(1)));
    record.addField(new MCRImportField("firstname", rs.get(2)));
    record.addField(new MCRImportField("gender", rs.get(3)));
    record.addField(new MCRImportField("birthdate", rs.get(4)));
    return record;
  }
}
```

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.

```
public class MyDerivateConverter implements MCRImportDerivateConverter<String> {
    public MCRImportDerivate convert(String id) {
        MCRImportDerivate derivate = new MCRImportDerivate(id);
        derivate.addFile("/home/user/importfiles/" + id + ".tif");
        return derivate;
    }
}
```

2.4. Mapping

Sind die Quelldaten in **MCRImportRecords** und **MCRImportDerivates** abgelegt, kann mit dem eigentlichen Mapping begonnen werden. Es ist nicht notwendig das MyCoRe (DB, Lucene etc.) läuft, ein eigenens kleines Java-Projekt welches den Importer einbindet ist völlig ausreichend.

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.

```
MCRImportMappingMananger mm = MCRImportMappingManager.getInstance();
// Mapping manager mit der Mapping Datei initialisieren
mm.init(new File("meine-mapping-datei.xml"));
// wenn derivate vorhanden sind hinzufügen
mm.setDerivateList(derivateList);
// das mapping starten
mm.startMapping(recordList);
```

Hinweis:
In der aktuellen Version von MyCoRe wird in der Initialisierungsmethode der Klasse **MCRConfiguration** versucht eine mycore.properties Datei zu laden. Dies führt zu einer Exception wenn der Konvertier- und Mapping Prozess nicht zur Laufzeit von MyCoRe ausgeführt wird. Um dies zu umgehen kann folgender Code verwendet werden: System.*setProperty*("MCR.Configuration.File", "");

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

3. Die Mapping-Datei

Für die weitere Verarbeitung der Daten wird eine xml-Datei benötigt. 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.

Um Fehler während des Bearbeitungsprozesses in der Mapping-Datei zu vermeiden wurde eine XML Schema Datei erstellt. Diese liegt unter "mycore-importer/src/main/resources/schema/import.xsd". Um die Schema Datei einzubinden muss das import-Element erweitert werden:

```
<import xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="src/main/resources/schema/import.xsd">
```

3.1. Konfiguration

Der Konfigurationsabschnitt 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 (Siehe 3.2). 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. Um einmal erstellte und vollständig ausgefüllte Klassifikationsmapping-Dateien nicht zu überschreiben sollte dieses Flag auf false gestellt werden. Weitere Informationen zum Klassifikationsmapping stehen unter 3.2.8 - Typen.

derivates: Dieses Element definiert, ob Derivate verwendet werden oder nicht. Standardmässig ist es auf false eingestellt und muss somit 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: Das **importToMycore** Element definiert, ob Derivate importiert werden oder nicht. Unabhängig davon, kann expliziet angegeben werden, ob die dazugehörigen Dateien (Bilder, Filme, Pdfs etc.) mit import werden. Dafür ist das Element **importFilesToMycore** zuständig. Um das Mapping zu testen ist es ratsam den Dateiupload zu deaktivieren.

3.2. 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 Punkt 3.2.5 näher erläutert.

Im mcrobjects-Element 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.

3.2.1 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 zur Laufzeit aus dem angegebenen Datenmodel ausgelesen.

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

3.2.2 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.

3.2.3 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>
```

3.2.4 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>
```

3.2.5 URI-Resolver

Nicht alle Werte können mit Hilfe dieses Schemas 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.

3.2.6 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" />
```

3.2.7 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>
```

3.2.8 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.

Weiterhin ist es möglich das Klassifikationsmapping für einen ganz bestimmten Metadatensatz abzuschalten. Dazu muss im map-Element das Attribut *useClassificationMapping* auf *false* gestellt werden.

Multi data

Wie in Punkt 2.3 schon angedeutet kommt es vor, daß ein **MCRImportRecord** mehrere Felder mit der gleichen Id besitzt. Zum Beispiel bei alternativen Namen:

- Wolfgang Amadeus Mozart
- Joannes Chrysostomus Wolfgangus Theophilus Mozart

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

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

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

```
<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>
```

Weiterhin ist es 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 berücksichtigt.

Derivate
Die Verknüpfung von Derivaten mit Importobjekten geschieht über den Typ *derivate*. Im map-Element werden zwei Attribute benötigt, *fields* und *derivateId*. Die Auflösung geschieht analog zum Typ *multi data*. Es können also meherere Derivate verlinkt werden.

```
<map fields="abbildungsId" derivateId="did_{abbildungsId}" type="derivate" />
```

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 angegebene Typ kann dann in der Mapping-Datei verwendet werden.

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

3.2.9 Umschließende Attribute

Umschließende Attribute entsprechen den Attributen des Elternelements eines Metadatensatzes. In der Regel sind das Attribute wie *heritable*, *notinherit* oder *class*. Der Importer bietet Zugriff auf diese Attribute über das *enclosingAttributes* Element.

```
<map fields="myAttrValue,name" to="name">
  <enclosingAttributes>
    <attribute name="heritable" value="true"/>
    <attribute name="myAttr" value="{myAttrValue}" />
  </enclosingAttributes>
  <text value="{name}"/>
</map>
```

```
<def.name class="MCRMetaLangText" heritable="true" myAttr="my value">
```

```
  <name xml:lang="de" form="plain">
    My Name
  </name>
</def.name>
```

3.3. Vor-/ und Nachverarbeitung

Die Vor-/ und Nachverarbeitung bietet eine Möglichkeit Zugriff auf die Importobjekte zur Laufzeit des Importvorgangs zu erhalten. Diese können dann mit Hilfe der API manipuliert werden. Der erste Eingriffspunkt ist direkt vor dem Mapping, der zweite bei erfolgreichem Abschluß.

Um diese Verarbeitungsmöglichkeit zu verwenden muss man eine Klasse erstellen welche das Interface **MCRImportMappingProcessor** einbindet. Diese Klasse muss dann bei dem entsprechenden Mapping-Element angegeben werden.

```
<mcrobject name="person" ... processor="org.mycore.MyMappingProcessor">
```

Es wird pro Import nur **eine** Instanz des jeweiligen Processors erstellt. Die Methoden *pre-* und *postProcessing* werden von dieser Instanz immer wieder aufgerufen. Weiterhin ist es wichtig das die Klasse keinen oder mindestens einen Konstruktor ohne Parameter besitzt, da sie per *newInstance* angelegt wird.

3.4. Ein Beispiel (Weiterführung des obrigen)

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

```
<import xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="src/main/resources/schema/import.xsd">
  <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 (siehe 3.2.8 - Typen).

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.

4. Import

Sind die xml-Objekte einmal erstellt, dann ist auch der Import zu MyCoRe keine schwierige Angelegenheit mehr. Folgendes CLI-Kommando startet den Import:

```
import from mapping file {0}
```

Wobei {0} mit dem Pfad der Mapping-Datei zu ersetzen ist.