Dokumentation UGH Bibliothek

Regelsätze und Referenzdokumentation

Version 3.0.0 – UGH 3.0.0

14. Januar 2016

Inhaltsverzeichnis

1 Einführung

Die Verwaltung von Dokumenten ist seit jeher die Hauptaufgabe von Bibliotheken. Mit dem Beginn der Digitalisierung von Informationen hat sich die Aufgabe stark gewandelt. Vermehrt werden Metadaten zum Auffinden, Sortieren und Beurteilen von Informationen genutzt. Das gebundene Buch als Informationseinheit wird zunehmend in eine Vielzahl kleinerer Einheiten (Strukturen/Dokumentstrukturen) aufgeteilt, um diese im Internet sinnvoll und gut nutzbar präsentieren zu können. Um diese Funktionalitäten umsetzen zu können, wurden in den vergangenen Jahren unterschiedliche Datenformate erarbeitet, die die erforderlichen Meta-Informationen zu einem Dokument abspeichern und auf den eigentlichen Inhalt verweisen (z.B. TIFF-, XML- oder HTML- Dateien). Ebenso ist es übliche Praxis, Metadaten zu einem Dokument aus unterschiedlichen Quellen zu aggregieren und gegebenenfalls zu ergänzen.

Für die Softwareentwicklung ergeben sich daraus einige Besonderheiten. Da sich Datenformate schnell ändern bzw. erweitert werden sowie unterschiedliche Projekte unterschiedliche Anforderungen an Datenformate haben (es müssen unterschiedliche Daten abgespeichert werden), ist es ratsam, ein möglichst flexibles und wohldefiniertes Datenformat zu nutzen. Um unterschiedliche Formate be- und verarbeiten zu können ist es ferner ratsam, dass alle Tools und Programme auf eine einheitliche Programmierschnittstelle (API) aufsetzen.

Die **UGH**[1] Bibliothek stellt eine solche Programmierschnittstelle dar, die durch unterschiedliche Tools und Programme genutzt werden kann und soll, um Daten zu laden und zu speichern. Diese Schnittstelle implementiert ein universelles Dokumentmodell, welches von unterliegenden Formaten zur Beschreibung von Metainformationen unabhängig ist. Vielmehr existieren unterschiedliche Klassen zur Serialisierung eines Dokumentes. Jede dieser Klassen implementiert genau ein Datenformat. Überliegende Programmschichten (bspw. die Business-Logik) ist somit vollkommen unabhängig vom Datenformat. Zu beachten ist jedoch, dass nicht immer alle Datenformate auch das komplette universelle Dokumentmodell umsetzen können. Ebenso können einige Serialisierungsklassen lediglich Daten lesen, da ein Schreiben für dieses Datenformat nicht benötigt wird oder wenig sinnvoll erscheint. Dies gilt beispielsweise für Klassen, die Datenformate aus Bibliothekskatalogen übernehmen. Da dort üblicherweise nur die bibliographische Ebene abgebildet wird, könnten sämtliche Strukturinformationen sowie Referenzen nicht geschrieben werden. Auf eine Schreibmethode wurde daher für diese Klassen verzichtet.

Nachfolgend wird zunächst das interne, universelle Dokumentmodell der **UGH** Bibliothek kurz skizziert, um anschließend auf die allgemeine Konfiguration und auf die einzelnen Serialisierungsformate einzugehen.

Der Quellkode der **UGH** Bibliothek, die verschiedenen JAR-Dateien – eine für den allgemeinen Gebrauch inklusive aller benötigten Java-Bibliotheken und eine für die Benutzung mit Goobi, sowie auch diese Dokumentation sind im Goobi-Wiki unter folgender URL verlinkt:

---

1 Der Name der UGH Bibliothek leitet sich von einem Laut des Bibliothekars (einem Orang-Utan) der Unsichtbaren Universität in Terry Pratchetts Scheibenwelt-Zyklus – in der deutschen Übersetzung – ab. Diesem ist ein eigener Artikel in Wikipedia gewidmet: http://de.wikipedia.org/wiki/Figuren_und_Schaupl%C3%A4tze_der_Scheibenwelt-Romane#Der_Bibliothekar

```
http://repository.digiverso.com/gitweb/?p=ugh.git;a=summary
```

2 Dokumentmodell

2.1 Strukturdaten

Strukturdaten beschreiben die Struktur eines Dokumentes. Hierzu wird allerdings ein recht weitgehender Strukturbegriff verwendet, der nicht nur den internen, hierarchischen Aufbau eines Dokumentes betrifft, sondern das Dokument an sich ebenfalls als Teil dieser Struktur versteht – als bibliographische Einheit „Dokument".

Neben dieser logischen Struktur kann das Modell die physische Struktur als zweite Struktur abbilden. Diese Struktur speichert üblicherweise die „gebundene Einheit" sowie ihre Untereinheiten (d.h. die einzelnen Seiten des Werkes). Sowohl in der logischen als auch in der physischen Struktur können Einheiten beliebig tief hierarchisch geschachtelt werden, wobei jede Einheit nur eine einzige Elterneinheit besitzen kann. Ferner können Einheiten aus der logischen und der physischen Struktur miteinander verknüpft werden, um so bspw. das Verhältnis zwischen einem Kapitel und seinen entsprechenden Seiten wiederzugeben. Beispiel: Kapitel fünf (logische Struktur) umfasst die Seiten 11 bis 27 (physische Struktur).

Jede Struktureinheit muss immer über einen Typ verfügen (Strukturtyp/DocStruct). Dieser Klassifiziert die Struktureinheit und definiert somit auch, welche weiteren Struktureinheiten als untergeordneten Einheiten vorhanden sein dürfen. Gleiches gilt für Metadaten. In der Praxis wird somit bspw. eine fehlerhafte Zuordnung von Metadaten zu einem Strukturtyp vermieden. Das Zuweisen einer ISBN zu einem Kapitel ist damit eventuell nicht möglich.

2.2 Inhaltsdateien

Jeder Struktureinheit können ferner Inhaltsdateien (Contentfiles) zugeordnet sein, wobei eine m:n Relation zwischen Struktureinheiten und Inhaltsdateien möglich ist. Diese Dateien enthalten den eigentlichen Inhalt des Dokuments entweder als Text, Bild, Ton oder Video. Das Datenformat der Inhaltsdatei ist nicht festgelegt. Das gesamte Dokument kann aus einer Inhaltsdatei oder aus einer Reihe von Inhaltsdateien bestehen. Der Zugriff auf eine Inhaltsdatei geschieht immer über das jeweilige Strukturelement. Die Benennung der jeweiligen Inhaltsdatei hat keinerlei Bedeutung für das Dokument. Vielmehr hängt zum Beispiel die Reihenfolge, in der bestimmte Inhaltsdateien präsentiert werden sollen, von der entsprechenden Reihenfolge der verknüpften Struktureinheiten ab.

2.3 Metadaten

Sowohl Inhaltsdateien als auch Struktureinheiten können Metadaten besitzen. Ein Metadatum zeichnet sich dadurch aus, dass es einen bestimmten Typ sowie einen Wert besitzt und entweder einer Struktureinheit oder einer Inhaltsdatei zugeordnet ist; es ist ein Typ-Wert Paar, welches zur Beschreibung des verknüpften Objekts dient. Ein Metadatum kann immer nur einem Objekt zugeordnet sein. Die Länge sowie der Typ des Metadatenwerts sind unbestimmt. Prinzipiell geht die API immer von beliebig langen Zeichenketten (Strings) aus. Abhängig von den jeweiligen

Serialisierungsklassen können jedoch bestimmte Wert-Typen vorausgesetzt werden, entweder weil bestimmte Felder Werte gemäß ISO-Konventionen voraussetzen (zum Beispiel Datum-, Sprach- oder Ländercodes) oder aber Datenbankspalten bestimmte Maximallängen haben. Derzeit ist diese Problematik von übergeordneten Applikationsschichten abzufangen.

Ein spezieller Typ von Metadatum sind Personen. Sie unterscheiden sich nicht grundsätzlich von herkömmlichen Metadaten, jedoch wird das einfache Typ-Wert-Schema um weitere Merkmale ergänzt. So wird beispielsweise zwischen Vor- und Nachname einer Person unterschieden, und es gibt einen Namen zur Anzeige (DisplayName) und eine Differenzierung, ob es sich um eine natürliche oder juristische Person (Firma oder Organisation) handelt. Der Metadaten-Typ steht bei Personen immer für eine bestimmte Rolle; ähnlich wie bei einem regulären Metadatum hat jede Person einen Typ.

2.4 Metadatengruppen

Seit der Version 1.10.0 gibt es die Möglichkeit, Metadaten zu gruppieren. Damit ist es beispielsweise möglich, Titel in einen nonSort- und einen Sort-Teil zu zerteilen und diese Zusammengehörigkeit auch über viele verschiedene Titel hinweg beizubehalten. Oder man könnte zu jeder Person einen Verweis auf die entsprechende Wikipedia-Seite mitspeichern.

Metadatengruppen können aus allen definierten Metadatentypen gebildet werden. Die einzige Einschränkung ist, das jeder Metadatentyp nur einmal vorkommen darf. Gruppierungen können immer nur einem Objekt zugeordnet sein.

2.5 Anker

Als Anker (Anchor) wird eine Dokumentstruktur verstanden, die eher virtueller Natur ist und andere Struktureinheiten zu einer Gruppe zusammenfügen kann. Solche Anker sind beispielsweise die Strukturen Mehrbändiges Werk oder Zeitschrift, welches alle Bände des Werkes bzw. alle Bände der Zeitschrift zusammenhält. Der Anker selber besitzt keine Inhaltsdateien – also kein Imageset mit Bild-Dateien, er besteht lediglich aus einer Struktureinheit, die über deskriptive Metadaten verfügt. Das wesentliche Metadatum ist ein Identifier, mit welchem der Anker eindeutig identifiziert werden kann.

Prinzipiell gibt es zwei Möglichkeiten, den Anker abzuspeichern:

- In derselben Datei wie die unterliegende Struktureinheit. In diesem Fall wird die Struktureinheit, die als Anker dient, redundant in jeder Metadatendatei gespeichert. Das RDF/XML Format beispielsweise speichert die Dokumentstruktur des Ankers als oberste Struktureinheit in der Datei.
- In einer separaten Datei, so dass aus unterliegenden Struktureinheiten auf den Anker verwiesen werden muss. Für den Import in Datenbanken und Repositories bedeutet dies, dass der Anker immer zuerst importiert werden muss. Um die Referenzierung zu gewährleisten, enthält der Anker einen Identifier, der im Verweis in der unterliegenden Struktureinheit genutzt wird.

3 Regelsatz

Das oben beschriebene Dokumentmodell ist recht allgemein gehalten, es legt keinerlei Metadaten- oder Strukturtypen fest. Um überhaupt Instanzen von Struktureinheiten und Metadaten anlegen zu können, ist es notwendig, in einem Regelsatz entsprechende Typen zu definieren. Weiterhin enthält dieser auch entsprechende Abschnitte für die Serialisierungsinformationen der jeweiligen Serialisierungsklassen.

Technisch wird der Regelsatz beim ersten API-Aufruf durch die Applikation geladen und nur der Teil interpretiert, der die Meta- und Strukturtypen definiert. Erst wenn entsprechende Klassen zum Serialisieren der Daten aufgerufen werden, wird der entsprechende Abschnitt zur Konfiguration der Serialisierung interpretiert (da dies erst in den entsprechenden Serialisierungsklassen implementiert ist). Das bedeutet, dass bei fehlerhafter Konfiguration auch während der Laufzeit Fehler auftreten können. Diese werden als Exceptions von der **UGH** Bibliothek geworfen und müssen entsprechend durch die Applikation abzufangen werden.

Der Regelsatz selber ist als XML-Datei implementiert. Der Wurzelknoten in jedem Regelsatz ist das `<Preferences>` Element. Innerhalb dieses Elementes befinden sich alle Definitionen der Metadaten- und Strukturtypen. Wichtig ist hierbei, dass Metadatentypen vor Gruppierungen und Strukturtypen definiert werden müssen.

3.1 Definition von Metadatentypen

Ein Metadatentyp wird innerhalb des `<MetadataType>` Elements definiert. Dieses Element muss als einziges Kind das `<Name>` Element besitzen, welches den internen Namen des Metadatentyps enthält. Der interne Name wird auch zur Referenzierung auf den Metadatentyp aus der Serialisierungskonfiguration heraus genutzt. Er darf keine Leerzeichen enthalten.

Weiterhin kann das `<MetadataType>` Element beliebig viele `<language>` Elemente als Kind enthalten. Diese `<language>` Elemente übersetzen den internen Namen des Metadatentyps in die jeweilige Sprache, die im Attribut name angegeben ist. Daher ist es sinnvoll, mindestens ein `<language>` Element pro Metadatentyp zu definieren, damit dessen Wert in der Benutzeroberfläche angezeigt und genutzt werden kann. Pro Sprache und `<MetadataType>` Element darf es nur ein einziges `<language>` Element geben.

Beispiel: Definition des Haupttitels „TitleDocMain“

```
<MetadataType>
      <Name>TitleDocMain</Name>
      <language name="de">HauptTitel</language>
      <language name="en">main title</language>
</MetadataType>
```

Handelt es sich bei dem definierten Feld nicht um ein Metadatum, sondern um eine Person, so ist dem `<MetadataType>` Element ein Attribut `type` mit dem Wert „person“ hinzuzufügen.

Beispiel: Definition einer Person

```
<MetadataType type="person">
      <Name>Author</Name>
```

```
        <language name="en">Author</language>
        <language name="de">Autor</language>
    </MetadataType>
```

Die Reihenfolge der `<language>` Elemente spielt keine Rolle, da sie intern über die API anhand der Werte im Attribut `name` aufgerufen werden. Ist keine Sprache definiert, so gibt die API einen NULL-Wert zurück und die Applikation sollte den internen Namen des Metadatentyps nutzen.

Soll der Anker in einer separaten Datei gespeichert werden, wird zwingend ein Metadatum benötigt, das eine Referenzierung von Ankerdatei und der Datei der unterliegenden Struktureinheit ermöglicht. Daher muss in diesem Fall der Metadatentyp, der hierfür genutzt werden soll, ein Attribut `type` mit dem Wert `identifier` besitzen.

Beispiel: Festlegung eines Identifiers für die oben genannte Referenzierung

```
    <MetadataType type="identifier">
        <Name>CatalogIDDigital</Name>
        <language name="de">PPN (digital)</language>
        <language name="en">PPN</language>
    </MetadataType>
```

Metadatentypen, die mit einem Unterstrich „_“ beginnen, sind sogenannte „interne“ Metadatentypen. Diese können zum Beispiel beim Anlegen von Vorgängen mit Werten versehen werden – etwa durch einen OPAC-Import, der im Regelsatz festgelegt ist oder durch eine Definition in der Projekt-Konfigurationsdatei `projects.xml` von Goobi. Diese internen Metadatentypen werden nicht mit den Rückgabelisten für Strukturtypen – zum Beispiel `DocStruct.getAllVisibleMetadata()` – zurückgegeben. So sind sie beispielsweise im Goobi-Metadateneditor für Nutzer nicht sichtbar und auch nicht als neue Metadaten anlegbar.

Beispiel: Definition eines Metadatums mit Normdaten

```
    <MetadataType normdata="true">
        <Name>Classification</Name>
        <language name="en">Classification</language>
        <language name="de">Klassifizierung </language>
    </MetadataType>
```

Metadaten, bei denen das Attribut `normdata="true"` gesetzt ist, besitzen weitere Felder, um einen Normdatensatz durch eine URL, einen Identifier und den Namen der Datenbank zu beschreiben.

Beispiel: Definition einer Person mit zusätzlichen Feldern

```
    <MetadataType type="person" namepart="true">
        <Name>Author</Name>
        <language name="en">Author</language>
        <language name="de">Autor</language>
    </MetadataType>
```

Bei Definitionen von Personen kann zusätzlich zum weiter oben beschriebenen Attribut `normdata` auch das Attribut `namepart` gesetzt werden. Dadurch können zusätzliche Felder

freigeschaltet werden, in denen weitere Informationen wie Lebensdaten oder Ansetzungsform erfasst werden können.

3.2 Definition von Metadatengruppen

Eine Metadatengruppe wird innerhalb des `<Group>` Elements definiert. Dieses Element muss als einziges Kind das `<Name>` Element besitzen, welches den internen Namen des Metadatentyps enthält. Der interne Name wird auch zur Referenzierung auf den Metadatentyp aus der Serialisierungskonfiguration heraus genutzt. Er darf keine Leerzeichen enthalten.

Weiterhin kann das `<Group>` Element beliebig viele `<language>` Elemente als Kind enthalten. Diese `<language>` Elemente übersetzen den internen Namen des Metadatentyps in die jeweilige Sprache, die im Attribut name angegeben ist. Daher ist es sinnvoll, mindestens ein `<language>` Element pro Metadatentyp zu definieren, damit dessen Wert in der Benutzeroberfläche angezeigt und genutzt werden kann. Pro Sprache und `<Group>` Element darf es nur ein einziges `<language>` Element geben.

Außerdem muss das `<Group>` Element mindestens ein `<metadata>` Element enthalten. Über diese Elemente können die Metadaten zur Gruppe hinzugefügt werden. Das Element muss dazu den internen Metadatennamen einer `<MetadataType>` Definition enthalten.

```
<Group>
      <Name>Title</Name>
      <language name="de">Titel</language>
      <language name="en">Title</language>
      <metadata>NonSort</metadata>
      <metadata>Sort</metadata>
</Group>
```

3.3 Definition von Strukturtypen

Strukturtypen werden ähnlich definiert wie Metadatentypen. Das Element zur Definition eines Strukturtyps heißt `<DocStrctType>` und genau wie bei den Metadatentypen existieren auch hier die Elemente `<Name>` und `<language>` mit denselben Eigenschaften.

Beispiel: Minimaldefinition eines Strukturtyps mit zusätzlichen <language> Elementen

```
<DocStrctType>
      <Name>Acknowledgment</Name>
      <language name="de">Danksagung</language>
      <language name="en">Acknowledgment</language>
</DocStrctType>
```

Sollen einem Strukturtyp weitere Strukturtypen als Kinder untergeordnet werden, muss im Regelsatz diese potentielle Zugehörigkeit definiert werden. Das Element `<allowedchildtype>` enthält dazu den internen Namen des entsprechende Strukturtyps. Hierbei ist die Reihenfolge der Strukturtypen egal.

Beispiel: Strukturtypdefinition mit potentiellen Kindern

```
<DocStrctType>
      <Name>Acknowledgment</Name>
      <language name="de">Danksagung</language>
      <language name="en">Acknowledgment</language>
      <allowedchildtype>OtherDocStrct</allowedchildtype>
</DocStrctType>
```

Ähnlich wie die Definition potentieller Kinder eines Strukturtyps definiert das Element `<metadata>` potentielle Metadatentypen, die zu einem Strukturtyp hinzugefügt werden dürfen. Das Element muss dazu den entsprechenden internen Metadatennamen enthalten. Darüberhinaus kann das `<metadata>` Element weitere Attribute enthalten, die beispielsweise die Häufigkeit der Verwendung sowie die Art der Nutzung eines Metadatentyps im Kontext des Strukturtyps angeben. Ein Strukturtyp „Artikel“ soll zum Beispiel zwingend einen Titel als Metadatum bekommen, jedoch keine oder beliebig viele Autoren.

Diese Attribute sind:

<table>
<tr><td>num</td><td>Das Attribut muss eines der folgende Werte enthalten:

<table><tr><td>„*“</td><td>kein mal oder beliebig oft (0...n)</td></tr><tr><td>„+“</td><td>ein mal oder beliebig oft (1...n)</td></tr><tr><td>„1o“</td><td>kein mal oder genau einmal (0...1)</td></tr><tr><td>„1m“</td><td>genau einmal (1)</td></tr></table>
Ist kein Attribut angegeben, wird standardmäßig der Wert „*“ angenommen.</td></tr>
<tr><td>DefaultDisplay</td><td>Das Attribut gibt an, ob der Metadatentyp für das entsprechende Strukturelement standardmäßig angezeigt werden soll (auch dann, wenn es noch keinen Inhalt hat). Metadaten mit Inhalt werden in jedem Fall angezeigt. In einer Metadaten-Erfassungsmaske lassen sich durch Abfragen dieses Attributs dann Leermasken generieren. Dazu muss das Attribut den Wert „true“ besitzen. Andere Werte oder ein Fehlen dieses Attributs ist mit dem Wert „false“ gleichzusetzen.</td></tr>
</table>

Beispiel: Komplexe Strukturtypinformation

```
<DocStrctType>
      <Name>PeriodicalVolume</Name>
      <language name="de">Zeitschriften-Band</language>
      <language name="en">Periodical Volume</language>
      <metadata num="*">Author</metadata>
      <metadata DefaultDisplay="true" num="1m">
            CatalogID
      </metadata>
      <group num="*">CauseEntry</group>
```

```
            <metadata num="*">SICI</metadata>
            <allowedchildtype>Advertising</allowedchildtype>
            <allowedchildtype>Appendix</allowedchildtype>
            <allowedchildtype>Article</allowedchildtype>
            <allowedchildtype>OtherDocStrct</allowedchildtype>
            <allowedchildtype>PeriodicalIssue</allowedchildtype>
            <allowedchildtype>PeriodicalPart</allowedchildtype>
            <allowedchildtype>TitlePage</allowedchildtype>
        </DocStrctType>
```

Ähnlich wie die Definition potentieller Metadaten eines Strukturtyps definiert das Element `<group>` potentielle Metadatengruppen, die zu einem Strukturtyp hinzugefügt werden dürfen. Das Element muss dazu den entsprechenden internen Gruppennamen enthalten.

Dabei stehen zur Konfiguration der Häufigkeit die selben Attribute wie bei den `<metadata>` Elementen zur Verfügung.

Strukturelemente, die als Anker dienen, werden mittels das Attributs `anchor` als solche gekennzeichnet. Diese Strukturelemente müssen immer die obersten Strukturelemente eines Dokuments sein. Das `anchor` Attribut des `<DocStrctType>` Elements enthält in diesen Fällen den Wert „true“.

4 Serialisierung

Daten, die mittels der API angelegt wurden, sollen natürlich auch geschrieben und gelesen werden können. Dazu verfügt die API über ein internes JAVA-Interface, welches durch unterschiedliche Klassen implementiert werden kann und Methoden zum Lesen und Schreiben bereitstellt. Ob diese Lese- und Schreibvorgänge dazu eine Datenbank oder das Dateisystem nutzten, oder welches Dateiformat genutzt wird, ist nicht relevant. Diese spezifischen Dinge werden durch das Interface gekapselt.

Durch die Implementierung des Interfaces können nachträglich neue Formate durch die **UGH** Bibliothek unterstützt werden, wobei eine Serialisierungsklasse genau ein Datenformat implementiert. Bei der Nutzung der Serialisierungsklassen ist darauf zu achten, dass diese das komplette Dokumentmodell unterstützten – also das Dokument ohne Verlust von Daten speichern und laden können. Eventuelle Einschränkungen müssen aus der Dokumentation der Serialisierungsklasse hervorgehen.

Eine Serialisierung kann immer nur für ein komplettes Dokument erfolgen. Änderungen an einzelnen Dokumentteilen führen also immer dazu, dass das komplette Dokument neu geschrieben wird. Die Applikation hat daher selbst darauf zu achten, dass Schreibzugriffe nicht konkurrierend stattfinden.

Da die Serialisierung genau genommen nicht Bestandteil der API ist – es existiert ein Java-Interface, das von beliebigen weiteren Serialisierungsklassen implementiert werden kann, wird diese in einem speziellen Bereich des Regelsatzes definiert. Dieser Bereich ist mit dem `<Formats>` Element umschlossen. Innerhalb dieses Elements gibt es für jede Serialisierungsklasse genau ein Element, welches die komplette Konfiguration für das jeweilige Dokumentformat enthält. Zumeist handelt es sich hierbei um Mapping-Informationen, wie zum Beispiel interne Struktur- und

Metadatentypen in das jeweilige Format gemappt werden können. Die Konfigurationsmöglichkeiten sind dabei dem Format angepasst.

Als Serialisierungsformat für die interne Speicherung wird METS/MODS empfohlen, das die Strukturdaten im METS-Format ablegt und die Metadaten in MODS-Containern (pro Strukturelement) in einem Goobi-XML-Namespace. Damit sind die Daten menschenlesbar und auch außerhalb der **UGH** Bibliothek editierbar – zum Beispiel in einem Text- oder XML-Editor. Für den Export/Import von DFG-Viewer-METS[2] mit MODS-Metadaten siehe das entsprechende Kapitel.

2 http://dfg-viewer.de/profil-der-metadaten/

4.1 MARC

Serialisierungsklassen:

```
de.intranda.ugh.extension.MarcFileformat
```

4.1.1 Einführung

Das Format MARC (**MA**chine-**R**eadable **C**ataloging) ist ein Datenformat, das zum Austausch von bibliographischen Daten entwickelt wurde. Das Format enthält Metadaten zu einer einzelnen Struktureinheit. Strukturdaten können über das Format nicht beschrieben werden. Daher kann das Format lediglich zum Import von Datensätzen genutzt werden, das Schreiben von Metadaten im MARC Format ist in UGH nicht vorgesehen.

Die vorliegende UGH Erweiterung erwartet als Eingabe einen Datensatz im Format MARCXML, wie es von der Library of Congress[3] beschrieben wird.

Die Erweiterung kann nicht selbst auf einen Katalog zugreifen, der Datensatz muss bereits lokal vorliegen. Er kann entweder in Form einer Datei oder als Zeichenkette an die UGH Erweiterung übergeben werden.

Das Format MARC XML besteht aus drei Bereichen. Jeder `<record>` enthält genau ein `<leader>` Element. Hier sind einige Angaben zum Dokumententyp codiert. Anschließend folgt eine Reihe von `<controlfield>` Elementen. In 001 ist der Identifier des Datensatzes enthalten, 007 und 008 enthalten weitere Angaben zum Dokumententyp. Anschließend folgen `<datafield>` Elemente, die wiederum eine Liste von `<subfield>` enthalten.

```
<record xmlns="http://www.loc.gov/MARC21/slim">
        <leader>xxxxxntm a22yyyyy c 4500</leader>
        <controlfield tag="001">722181507</controlfield>
        <controlfield tag="003">DE-601</controlfield>
        <controlfield tag="005">20150128152124.0</controlfield>
        <controlfield tag="008">120823m18611871gw          000 0 lat d</controlfield>
        <datafield tag="035" ind1=" " ind2=" ">
                <subfield code="a">(DE-599)GBV722181507</subfield>
        </datafield>
        <datafield tag="040" ind1=" " ind2=" ">
                <subfield code="a">GBVCP</subfield>
                <subfield code="b">ger</subfield>
                <subfield code="c">GBVCP</subfield>
                <subfield code="e">rakwb</subfield>
        </datafield>
        <datafield tag="041" ind1="0" ind2=" ">
                <subfield code="a">lat</subfield>
                <subfield code="a">ger</subfield>
        </datafield>
        <datafield tag="100" ind1="1" ind2=" ">
                <subfield code="a">Pichler, Pauline</subfield>
                <subfield code="e">Besitzer</subfield>
                <subfield code="0">(DE-601)722181795</subfield>
```

[3] Fußnote: http://www.loc.gov/marc/bibliographic/

```
            <subfield code="0">(DE-588)1025374479</subfield>
        </datafield>
        <datafield tag="245" ind1="1" ind2="0">
            <subfield code="a">[Stammbuch Pauline Pichler]</subfield>
            <subfield code="h">Manuskript</subfield>
        </datafield>
....
</record>
```

4.1.2 Konfiguration

Die Konfiguration erfolgt im Regelsatz innerhalb des `<Formats>` Elements. Dort kann ein `<Marc>` Element angelegt werden. Innerhalb dieses Elements sind die vier Elemente `<Person>`, `<Metadata>`, `<Group>` und `<DocStruct>` erlaubt, die jeweils mit einem internen Datentyp verknüpft werden müssen.

4.1.2.1 Metadata

Mit Hilfe des `<Metadata>` Elements wird der Import von einfachen Metadaten konfiguriert. Innerhalb des Elements sind folgende Unterelemente erlaubt:

`<Name>`	Der interne Metadatenname. Dieses Element muss genau einmal vorhanden sein. Der Inhalt des Feldes muss einem Namen eines `<MetadataType>` Elements entsprechen.
`<field>`	Mit diesem Feld wird ein `datafield` eines MARC Records beschrieben. Dieses Feld ist wiederholbar und muss mindestens einmal existieren. Es wird in der Sektion `field` (Kapitel 2.3) genauer beschrieben.
`<identifierField>`	Mit diesem Element kann das `subfield` definiert werden, das den Identifier des Metadatums enthält. Dieser Identifier kann auf Normadatenbanken wie die GND verweisen. Dieses Feld ist optional.
`<identifierConditionField>`	In diesem Element kann ein regulärer Ausdruck definiert werden, den der Identifier erfüllen muss. Dies kann notwendig werden, wenn mehrere Identifier zu verschiedenen Datenbanken angegeben sind. Dieses Feld ist optional.
`<identifierReplacement>`	In diesem Feld kann ein regulärer Ausdruck definiert werden, der auf den Identifier ausgeführt wird. Dies kann zum Beispiel genutzt werden, um einen unerwünschten Prefix zu entfernen. Dieses Feld ist optional.

<conditionField>	In diesem Element kann der Code eines <subfield> definiert werden, dessen Inhalt auf eine in <conditionValue> definierte Bedingung überprüft werden soll. Das Metadatum wird nur dann erzeugt, wenn das <subfield> existiert und der Bedingung entspricht. Dieses Feld ist optional. Wenn es existiert, muss auch <conditionValue> existieren.
<conditionValue>	Hier kann ein regulärer Ausdruck definiert werden. Der Inhalt des subfields, das in <conditionField> definiert wurde, muss diesen Ausdruck erfüllen, damit das Metadatum erzeugt wird. Hiermit kann zum Beispiel geprüft werden, ob eine Person oder Institution einer bestimmten Rolle entspricht. Dieses Feld ist optional. Wenn es existiert, muss auch <conditionField> konfiguriert sein.
<fieldReplacement>	In diesem Feld kann ein regulärer Ausdruck definiert werden, der auf den Feldinhalt angewendet wird. Damit können zum Beispiel unerwünschte Klammern von Datumsangaben entfernt werden.
<separateEntries>	Dieses Feld steuert, wie mit mehrfach vorkommenden Feldern umgegangen werden soll. Es kann die Werte true und false enthalten. Wurde true gesetzt, wird für jedes gefundene Feld ein eigenes Metadatum erstellt. Wenn der Wert false ist, werden alle Inhalte in ein einzelnes Metadatum geschrieben, die einzelnen Werte werden dann durch den Inhalt des <separator> Feldes getrennt. Wenn das Feld nicht konfiguriert wurde, wird jedes Feld als eigenes Metadatum angelegt.
<separator>	Dieses Feld definiert die Zeichenfolge, die als Trennzeichen zwischen den Inhalten bei mehrfach vorkommenden Feldern genutzt werden soll, wenn <separateEntries> auf false gesetzt wurde. Fehlt dieses Feld, wird Semikolon, gefolgt von einem Leerzeichen genutzt.

4.1.2.2 Person

Mit dem <Person> Element wird der Import von Personen konfiguriert. Innerhalb des Feldes sind alle Elemente erlaubt, die auch für <Metadata> möglich sind. Der einzige Unterschied der beiden Definitionen wird in field (Kapitel 4.1.2.3) beschrieben.

4.1.2.3 field

Das <field> Element erlaubt die genaue Beschreibung zu einem einzelnen Metadatum innerhalb des MARC records.

<fieldMainTag>	Die <datafield> tag Nummer. Das Feld muss genau einmal vorhanden sein.
<fieldInd1>	Der Wert des ind1 Attributes des <datafield> Elements. Hier kann eine Zahl, ein Leerzeichen oder any eingegeben werden. Fehlt das Feld, wird any genutzt.
<fieldInd2>	Der Wert des ind2 Attributes des <datafield> Elements. Hier kann eine Zahl, ein Leerzeichen oder any eingegeben werden. Fehlt das Feld, wird any genutzt.
<fieldSubTag>	Der Wert des code Attributes des <subfield> Elements, in dem sich der zu importierende Text befindet. Das Feld muss bei <Metadata> genau einmal vorhanden sein.
<firstname>	Der Wert des code Attributes des <subfield> Elements, in dem der Vorname einer Person enthalten ist. Bei <Person> muss entweder <firstname> und <lastname> oder <expansion> einmal vorhanden sein.
<lastname>	Der Wert des code Attributes des <subfield> Elements, in dem der Nachname einer Person enthalten ist. Bei <Person> muss entweder <firstname> und <lastname> oder <expansion> einmal vorhanden sein.
<expansion>	Der Wert des code Attributes des <subfield> Elements, in dem die Expansion des Namens enthalten ist. Beim Import wird der Name am Komma in Vorname und Nachname aufgeteilt. Bei <Person> muss entweder <firstname> und <lastname> oder <expansion> einmal vorhanden sein.

4.1.2.4 Group

In <Group> kann eine Metadatengruppe importiert werden. Das <Group> Element kann folgende Unterelemente enthalten:

<Name>	Der interne Name der Gruppe. Dieses Element muss genau einmal vorhanden sein. Der Inhalt des Feldes muss einem Namen einer <Group> Definition entsprechen.
<Metadata>	Konfiguration eines Metadatums innerhalb der Gruppe. Die Konfiguration entspricht der <Metadata> Konfiguration. Es können nur Metadaten definiert werden, die innerhalb der Gruppe erlaubt sind. Das Feld ist wiederholbar.
<Person>	Konfiguration einer Person innerhalb der Gruppe. Die Konfiguration entspricht der <Person> Konfiguration. Es können nur Personen definiert werden, die innerhalb der Gruppe erlaubt sind. Das Feld ist wiederholbar.

4.1.2.5 DocStruct

Mit Hilfe der <DocStruct> Definitionen können die einzelnen Dokumententypen definiert werden.

<Name>	Der interne Name des Strukturelements. Dieses Element muss genau einmal vorhanden sein. Der Inhalt des Feldes muss dem Namen eines <DocStrctType> Elements entsprechen.
<leader6>	In diesem Feld kann der Wert gesetzt werden, der an sechster Position im <leader> erwartet wird. Dieses Feld muss genau einmal existieren.
<leader7>	In diesem Feld kann der Wert gesetzt werden, der an siebter Position im <leader> erwartet wird. Dieses Feld muss genau einmal existieren.
<leader19>	In diesem Feld kann der Wert gesetzt werden, der an der 19. Position im <leader> erwartet wird. Dieses Feld ist optional.
<field007_0>	In diesem Feld kann der Wert gesetzt werden, der an nullter

	Position im `<controlfield tag="007">` erwartet wird. Dieses Feld ist optional. Wenn das Feld einen Wert enthält, muss das `controlfield` auch existieren.
`<field007_1>`	In diesem Feld kann der Wert gesetzt werden, der an erster Position im `<controlfield tag="007">` erwartet wird. Dieses Feld ist optional. Wenn das Feld einen Wert enthält, muss das `controlfield` auch existieren.
`<field008_21>`	In diesem Feld kann der Wert gesetzt werden, der an 21. Position im `<controlfield tag="008">` erwartet wird. Dieses Feld ist optional. Wenn das Feld einen Wert enthält, muss das controlfield auch existieren.

4.1.3 Beispielkonfigurationen

4.1.3.1 Beispiel: Mapping von Metadaten

Daten aus MARC XML:

```
<datafield tag="245" ind1="1" ind2="0">
<subfield code="a">[Stammbuch Pauline Pichler]</subfield>
<subfield code="h">Manuskript</subfield>
</datafield>
```

Mapping im Regelsatz zur Datenübernahme:

```
<Metadata>
<Name>TitleDocMain</Name>
<field>
        <fieldMainTag>245</fieldMainTag>
                <fieldSubTag>a</fieldSubTag>
                <fieldInd1>any</fieldInd1>
                <fieldInd2>0</fieldInd2>
        </field>
</Metadata>
```

4.1.3.2 Beispiel: Entfernen von Klammern

Daten aus MARC XML:

```
<datafield tag="245" ind1="1" ind2="0">
        <subfield code="a">[Stammbuch Pauline Pichler]</subfield>
        <subfield code="h">Manuskript</subfield>
</datafield>
```

Mapping im Regelsatz zur Datenübernahme:

```
<Metadata>
        <Name>TitleDocMain</Name>
        <field>
                <fieldMainTag>245</fieldMainTag>
                <fieldSubTag>a</fieldSubTag>
                <fieldInd1>any</fieldInd1>
                <fieldInd2>0</fieldInd2>
        </field>
        <fieldReplacement>s/\[(.+)\]/$1/g</fieldReplacement>
</Metadata>
```

4.1.3.3 Beispiel: Übernahme von Normdaten

Daten aus MARC XML:

```
<datafield tag="650" ind1=" " ind2="7">
        <subfield code="0">(DE-601)106075438</subfield>
        <subfield code="0">(DE-588)4079163-4</subfield>
        <subfield code="a">Weltkrieg</subfield>
        <subfield code="9">g:1914-1918</subfield>
        <subfield code="2">gnd</subfield>
</datafield>
```

Mapping im Regelsatz zur Datenübernahme:

```
<Metadata>
        <Name>Classification</Name>
        <field>
                <fieldMainTag>650</fieldMainTag>
                <fieldSubTag>a</fieldSubTag>
                <fieldInd1>any</fieldInd1>
                <fieldInd2>any</fieldInd2>
        </field>
        <identifierField>0</identifierField>
        <identifierConditionField>/DE-588/</identifierConditionField>
        <identifierReplacement>s/\(.+\)//g</identifierReplacement>
</Metadata>
```

4.1.3.4 Beispiel: Übernahme des Autors aus MARC XML

Daten aus MARC XML:

```
<datafield tag="100" ind1="1" ind2=" ">
        <subfield code="a">Klein, Felix</subfield>
        <subfield code="0">(DE-601)133790177</subfield>
        <subfield code="0">(DE-588)11856286X</subfield>
        <subfield code="4">aut</subfield>
</datafield>
```

Mapping im Regelsatz zur Datenübernahme:

```
<Person>
	<Name>Author</Name>
	<field>
		<fieldMainTag>100</fieldMainTag>
		<expansion>a</expansion>
		<fieldInd1>1</fieldInd1>
		<fieldInd1> </fieldInd1>
	</field>
	<identifierField>0</identifierField>
	<identifierConditionField>/DE-588/</identifierConditionField>
	<identifierReplacement>s/\(.+\)//g</identifierReplacement>
	<conditionField>4</conditionField>
	<conditionValue>/aut/</conditionValue>
</Person>
```

4.1.3.5 Beispiel: Mapping einer Handschrift

Daten aus MARC XML:

```
<leader>xxxxxntm a22yyyyy c 4500</leader>
```

Mapping im Regelsatz zur Datenübernahme:

```
<DocStruct>
	<Name>Manuscript</Name>
	<leader6>t</leader6>
	<leader7>m</leader7>
</DocStruct>
```

4.1.3.6 Beispiel: Mapping einer Karte

Daten aus MARC XML:

```
<leader>xxxxxnem a22yyyyy c 4500</leader>
<controlfield tag="001">840562748</controlfield>
<controlfield tag="003">DE-601</controlfield>
<controlfield tag="005">20151123080919.0</controlfield>
<controlfield tag="007">au auuun</controlfield>
```

Mapping im Regelsatz zur Datenübernahme:

```
<DocStruct>
	<Name>Map</Name>
	<leader6>e</leader6>
	<leader7>m</leader7>
	<field007_0>a</field007_0>
	<field007_1>u</field007_1>
	<field008_21> </field008_21>
</DocStruct>
```

4.1.3.7 Beispiel: Angaben zum Verlag, Ort und Erscheinungsjahr

Daten aus MARC XML:

```
<datafield tag="260" ind1="3" ind2=" ">
        <subfield code="6">880-01</subfield>
        <subfield code="a">Tōkyō</subfield>
        <subfield code="b">Shōbunsha</subfield>
        <subfield code="c">2016</subfield>
</datafield>
```

Mapping im Regelsatz zur Datenübernahme:

```
<Group>
        <Name>Publishing</Name>
        <Metadata>
                <Name>PlaceOfPublication</Name>
                <field>
                        <fieldMainTag>260</fieldMainTag>
                        <fieldSubTag>a</fieldSubTag>
                        <fieldInd1>3</fieldInd1>
                </field>
        </Metadata>

        <Metadata>
                <Name>PublisherName</Name>
                <field>
                        <fieldMainTag>260</fieldMainTag>
                        <fieldSubTag>b</fieldSubTag>
                        <fieldInd1>3</fieldInd1>
                </field>
        </Metadata>

        <Metadata>
                <Name>PublicationYear</Name>
                <field>
                        <fieldMainTag>260</fieldMainTag>
                        <fieldSubTag>c</fieldSubTag>
                        <fieldInd1>3</fieldInd1>
                </field>
        </Metadata>
</Group>
```

4.2 METS/MODS

Serialisierungsklassen:

```
ugh.fileformats.mets.MetsMods (v1.9)
ugh.fileformats.mets.MetsModsImportExport (v1.9)
```

4.2.1 Einführung

Das METS/MODS XML-Format kann das interne Dokumentmodell vollständig serialisieren, es sind sowohl Lese- und Schreibmethoden implementiert. Da es sich lediglich um eine Untermenge von METS handelt – es werden nicht alle METS Elemente genutzt – existiert eine Beschreibung des dieser Implementierung zugrunde liegenden Formats auf den Seiten des DFG-Viewer[4]: Das DFG-Viewer METS-Profil[5] in Version 2.0.

Die Strukturtypen werden grundsätzlich in METS serialisiert, die Metadaten je nach genutzter Klasse im MODS-Format des DFG-Viewer-Profils[6] (`ugh.fileformat.mets.MetsModsImportExport`) oder in einem Goobi-Namespace innerhalb des MODS Extension-Tags (`ugh.fileformat.mets.MetsMods`). Diese Trennung ist nötig, weil für eine Speicherung im MODS-Anwendungsprofil ein Mapping der Metadatentypen des Dokumentmodells zu den MODS-Typen UND UMGEKEHRT zwingend nötig ist. Da die Metadatentypen jedoch frei im Regelsatz definiert werden können, ist ein Mapping nach MODS immer möglich, bei verschiedenen Autoren beispielsweise – für die es nicht notwendigerweise eine voneinander unterschiedliche Entsprechung in MODS gibt – aber dann nicht in der anderen Richtung.

Für die vollständige Serialisierung des Dokumentmodells kann die Klasse `MetsMods` genutzt werden, während die Klasse `MetsModsImportExport` zunächst den Export unterstützt, so dass METS-Dateien im DFG-Viewer METS-Profil exportiert werden können. Lesender Zugriff auf Dateien mit Metadaten im MODS-Anwendungsprofil ist bereits implementiert, unterliegt jedoch gewissen Einschränkungen, die im Kapitel „Mapping für den Import“ erläutert werden.

4.2.2 Allgemeine Konfiguration

Die Serialisierung in METS/MODS mit der Klasse `MetsMods` bedarf keiner weiteren Konfiguration, nahezu alle Tags innerhalb des `<METS>` Tags in der Format-Sektion des Regelsatzes beziehen sich ausschließlich auf die Klasse `MetsModsImportExport`. Lediglich der Tag `<AnchorIdentifierMetadataType>` muss hier vorhanden sein.

Beispiel: Die notwendige Konfiguration für die Speicherung per `MetsMods`

```
<METS>
      <AnchorIdentifierMetadataType>
            CatalogIDDigital
      </AnchorIdentifierMetadataType>
</METS>
```

4 http://dfg-viewer.de/

5 Als PDF: http://dfg-viewer.de/fileadmin/groups/dfgviewer/METS_Anwendungsprofil_2.0.pdf und als XML-Profil: http://dfg-viewer.de/fileadmin/groups/dfgviewer/METS_Anwendungsprofil_2.0.xml

6 http://dfg-viewer.de/fileadmin/groups/dfgviewer/MODS_Anwendungsprofil_1.0.pdf

Da das Metadatenformat innerhalb von METS (`MetsModsImportExport`) recht flexibel ist, müssen zunächst einige grundlegende Dinge konfiguriert werden. Diese befinden sich direkt unterhalb des umschließenden `<METS>` Elements.

4.2.2.1 NamespaceDefinition

Das Element `<NamespaceDefinition>` enthält eine Namespace-Definition, die zusätzlich zu den von **UGH** sowieso genutzten Namespaces im MODS-Mapping benötigt wird. Die Namespaces und auch die SchemaLocations (falls vorhanden) von METS, MODS, XLINK, GOOBI, XSI und DV sind bereits in der Klasse MetsMods definiert und werden – falls im Regelsatz vorhanden – mit den hier definierten Werten überschrieben. Vorsicht ist hier geboten, sollte eine andere Version eines bereits in **UGH** definierten Schemas im Regelsatz geändert werden, denn im Allgemeinen ist **UGH** hier auf eine bestimmte Version festgelegt. Als Elemente enthält `<prefix>` die Definition des Prefixes, das Element `<URI>` die URI des Namespaces an und schließlich gibt das Element `<schemaLocation>` die URI des XML-Schema-Datei an, gegen die validiert werden soll.

Beispiel: Definition eines zusätzlichen im MODS-Mapping genutzten Namespaces

```
<NamespaceDefinition>
      <URI>http://zvdd.gdz-cms.de/</URI>
      <prefix>zvdd</prefix>
      <schemaLocation>
            http://pfad/zu/schema/xsd/zvdd.xsd
      </schemaLocation>
</NamespaceDefinition>
```

4.2.2.2 Der Link zur Anker-Datei

Eine Besonderheit stellt die Konfiguration der Anker dar: Wenn Anker in einer separaten Datei gespeichert werden, muss ein Metadatum zur Referenzierung auf diesen Anker vorhanden sein. Dieses Metadatum enthält den Wert des entsprechenden Identifiers, welches den Anker eindeutig referenziert. Dieses Metadatum wird in einem entsprechenden XML-Feld innerhalb des Metadatensatzes gespeichert.

Das Element `<XPathAnchorQuery>` definiert das Element innerhalb der Metadaten (MODS), dessen Inhalt der Identifier der Anker-Datei ist. Die Anker-Datei ist in einer zusätzlichen METS-Datei gespeichert, die separat gespeichert und dann auch wieder geladen wird. Die Angabe ist als XPath-Query eingetragen.

Beispiel: Ein XPath zum Anker-Datei-Identifier

```
<XPathAnchorQuery>
     ./mods:mods/mods:relatedItem[@type='host']/mods:recordInfo/mods:reco
rdIdentifier
     [@source='gbv-ppn']
</XPathAnchorQuery>
```

Das folgende Beispiel beschreibt den XPath zu dem Element `<recordIdentifier>` mit dem Wert `PPN123456789`. Hier ein Auszug aus einer METS-Datei einer untergeordneten Struktur (Der Inhalt des Attributs `source` kommt aus dem Metadatenmapping des Metadatums `CatalogIDDigital`, dazu jedoch später):

```
     <mods:mods>
          <mods:relatedItem type="host">
               <mods:recordInfo>
               <mods:recordIdentifier source="gbv-ppn">
                    PPN123456789
               </mods:recordIdentifier>
               </mods:recordInfo>
          </mods:relatedItem>
     </mods:mods>
```

Hier die entsprechende Stelle in der Anker METS-Datei:

```
     <mods:recordInfo>
          <mods:recordIdentifier source="gbv-ppn">
               PPN123456789
          </mods:recordIdentifier>
     </mods:recordInfo>
```

4.2.2.3 Der Anker Identifier Metadatentyp

Das Element `<AnchorIdentifierMetadataType>` beschreibt den internen Metadatentyp desjenigen Elements, das als Anker-Identifier dienen soll. In unserem Beispiel das Element `CatalogIDDigital` mit dem Wert `PPN123456789`.

Beispiel: Ein Anker Identifier Metadatentyp

```
     <AnchorIdentifierMetadataType>
          CatalogIDDigital
     </AnchorIdentifierMetadataType>
```

Diese Referenzierung von Ankerdatei (übergeordnete Dokumentstruktur) und nachfolgender Dokumentstruktur ist im oben genannten METS-Profil definiert (siehe „dmdSec Anforderung 4: Hierarchische Verknüpfung von Dokumenten mittels MODS“).

Soll der Inhalt des Anker-Metadatums per regulärem Ausdruck verändert werden, ist dies mit dem Tag `<ValueRegExp>` an dieser Stelle möglich (zu regulären Ausdrücken siehe weiter unten).

4.2.3 Konfiguration des Mappings von Struktur- und Metadaten

4.2.3.1 Dokumentstrukturen und Metadaten

Das Mapping von Dokumentstrukturtypen des Dokumentmodells – also die im Regelsatz definierten Strukturtypen – zu METS-Strukturtypen ist im unteren Teil der METS-Sektion definiert. Die Elemente `<DocStruct>` definieren dieses Mapping. Werden hier Elemente nicht erwähnt, wird der Name des internen Strukturtyps auch im METS-Dokument verwendet. Ein Mapping ist vor allem für die Darstellung im DFG-Viewer nötig (physische Strukturtypen), sowie für ein Mapping zum Beispiel nach zvdd – hier müssen interne Strukturtypen auf die in zvdd vorhandenen gemappt werden. Innerhalb des Elements `<DocStruct>` muss es folgende zwei Unterelemente geben:

`<InternalName>`	Interner Name des Dokumentstrukturtyps.
`<MetsType>`	Der Name, der im Attribut type des <mets:div> Elements erscheinen soll.

Beispiel: Mapping von Dokumentstrukturtypen (physische Struktur)

```
<DocStruct>
      <InternalName>BoundBook</InternalName>
      <MetsType>physSequence</MetsType>
</DocStruct>
```

Auszug aus der METS-Datei (Ausschnitt):

```
<mets:structMap TYPE="PHYSICAL">
      <mets:div type="physSequence"/>
</mets:structMap>
```

Das Metadatenmapping gestaltet sich etwas komplizierter, da MODS als zu unterstützendes Metadatenformat in sich stark strukturiert ist. Um diese Struktur abbilden zu können, wird sich gängiger XML-Standards bedient. Das Mapping eines XML-Elements auf einen internen Metadatentyp und umgekehrt wird mittels eines XPath-Ausdrucks definiert. Dieser XPath-Ausdruck enthält den relativen Pfad ausgehend vom Element `<mets:xmlData>`. Hierbei sollte dieser XPath-Ausdruck das Element auswählen, welches den Wert des Metadatums zum Inhalt hat.

Für den Import wird das Mapping eines MODS-Elements per XPath auf einen internen Metadatentyp gemappt, wobei das Element `<XPath>` genutzt wird. Für den Export wird der Inhalt eines internen Metadatums in das MODS-Element geschrieben, das per `<WriteXPath>` definiert wird.

4.2.3.2 Mapping für den MODS-Export – Metadaten

Für das Schreiben von XML-Dateien kann lediglich ein Subset von XPath verwendet werden, das im Folgenden erläutert wird. Hierdurch kann es vorkommen, dass der XPath-Ausdruck zum Schreiben von Metadaten (Export) von dem abweicht, der zum Lesen (Import) von Metadaten genutzt wird. Daher werden die beiden Elemente `<XPath>` und `<WriteXPath>` unterschieden, die jeweils als Unterelement eines `<Metadata>` Elements genutzt werden. Pro internem Metadatum können

Mappings auch mehrmals erfolgen, siehe hierzu auch die weiter unten angeführten Beispiele zu bedingten Mappings sowie der Manipulation von Metadatenwerten per regulären Ausdrücken.

Für das `<WriteXPath>` Element gelten folgende Bedingungen:

- Der Schreibausdruck muss zwingend mit „`./`" beginnen und anschließend den kompletten Pfad vom `<xmlData>` Element ausgehen enthalten. Für MODS bedeutet dies, dass selbst das umschließende `<mods>` Element definiert wird. Das hat damit zu tun, dass beim internen Aufbau der XML-Struktur entsprechende Elemente angelegt werden, sollten sie noch nicht vorhanden sein.
- Die Hierarchie von Elementen im XPath können nur durch „`/`" getrennt werden.
- Elemente können auf jeder Hierarchiestufe gefiltert werden. Der Filterausdruck muss zwischen eckigen Klammern stehen „`[]`". Beim Anlegen werden die entsprechenden Elemente und Attribute, die in diesem Filterausdruck definiert werden, ebenfalls angelegt.
- Attribute werden durch ein vorangestellten Klammeraffen „`@`" als solche gekennzeichnet.
- Innerhalb von Filterausdrücken kann einem Element oder einem Attribut ein Wert zugewiesen werden. Diese Zuweisungen sind direkt hinter den Attribut- oder Elementnamen zu schreiben, mit einfachen Anführungszeichen „`'`" zu umschließen und von dem Namen mit einem Gleichheitszeichen „`=`" zu trennen. Eine gültige Zuweisung wäre bspw. `@attribut='wert'` oder aber auch `element='wert'`.
- Außerhalb von einfachen Anführungszeichen wird z.B. das Zeichen „`/`" als Trennzeichen der einzelnen Tags verwendet, innerhalb einfacher Anführungszeichen sind auch Sonderzeichen erlaubt.
- Zuweisungen außerhalb von Filterausdrücken sind nicht zulässig, da der Wert ja durch den Wert des zugewiesenen Metadatums bestimmt wird.
- Existieren mehrere Filterausdrücke pro Element, so sind diese in zwei separaten Klammern unterzubringen. Es wird eine UND-Verknüpfung angenommen. Ein gültiger Ausdruck wäre also zum Beispiel `./element[@attribut1='1'][@attribut2='2']`.
- Eine gleichzeitige Zuweisung von Werten zu Elementen und Attributen ist möglich durch die Angabe `='wert'` direkt hinter den Attribut-Zuweisungen. Beispiel: `./element[@attribut1='1'][@attribut2='2']='wert'`.
- Weitere Funktionen wie beispielsweise `not` werden für das Schreiben ignoriert.

Zu beachten ist im allgemeinen, dass die im XPath-Ausdruck verwendeten Elementnamen des Datenformats entsprechend der Einstellungen in den `<NamespaceDefinition>` Elementen des Regelsatzes über einen Prefix verfügen können, falls sich die angestrebten Elemente nicht im Default-Namespace befinden.

Beispiel: XPath-Ausdrücke für das Mapping

```
<Metadata>
      <InternalName>TitleDocSub</InternalName>
      <WriteXPath>
            ./mods:mods/mods:titleInfo/mods:subTitle
      </WriteXPath>
</Metadata>
```

Grundsätzlich berücksichtigt das System beim Anlegen von Elementen entsprechend des XPath-Ausdruck die bereits vorhandene XML-Struktur. Diese enthält natürlich auch alle bereits angelegten Elemente anderer und derselben Metadaten. Dabei geht das System so vor, dass es sich Element für Element den Pfad hinunter hangelt, um fehlende Elemente am Ende des Pfades anzulegen. Für den obigen Pfad im obigen Beispiel prüft **UGH**, ob ein Element `<mods>` bereits vorhanden ist; falls nicht, wird ein solches angelegt. Falls doch, wird das nächste Element als Kind-Element des `<mods>` Elements überprüft. In diesem Fall also `<titleInfo>`. Ist ein solches Element nicht vorhanden, wird es immer als Kind des bereits vorhandenen Elements angelegt, und so weiter und so fort. Dieses Vorgehen funktioniert prächtig, solange es nur ein Metadatum vom selben Typ pro Dokumentstruktur gibt. Gibt es mehrere davon, würde dieses Vorgehen dazu führen, dass entsprechende Elemente gerade nicht mehr angelegt werden, da der entsprechende Pfad im DOM-Baum ja bereits für das erste Metadatum angelegt wurde.

Wenn es also mehrere gleiche Metadatentypen pro Dokumentstruktur gibt, wird das Hash-Zeichen „#" genutzt, um zu kennzeichnen, dass ein jedes davon angelegt wird und somit einen eigenen Teilbaum in der XML-Struktur bekommt. Dieses Zeichen darf sich ausschließlich im `<WriteXPath>` Element befinden, es gilt also nur zum Schreiben. Es kennzeichnet die Stelle im XPath-Ausdruck, an der mit der Überprüfung des Pfads gestoppt wird und ein neuer Teilpfad angelegt werden soll. Wären jetzt zum Beispiel zwei Untertitel für eine Dokumentstruktur vorhanden, die nach obigen XPath-Ausdruck MODS-konform geschrieben werden sollen, so muss der XPath-Ausdruck zum Schreiben wie folgt aussehen:

```
./mods:mods/mods:titleInfo/#mods:subTitle
```

Dies führt dazu, dass das Element `<subTitle>` innerhalb von `<titleInfo>` für jedes entsprechende Metadatum wiederholt angelegt wird. Wichtig ist, dass das Hash-Zeichen nur zu Beginn eines Elementnamens stehen darf, das heißt, es muss sich auch vor dem Prefix des entsprechenden Namespaces befinden. Stünde der „#" vor dem Element `mods:titleInfo`, würde für jedes Metadatum `TitleDocSub` ein solcher Teilbaum im XML-Dokument angelegt.

Beispiel: Verwendung von Attributen in Filtern

```
<Metadata>
      <InternalName>singleDigCollection</InternalName>
      <WriteXPath>
            ./mods:mods/#mods:classification[@authority='ZVDD']
      </WriteXPath>
</Metadata>
```

... erzeugt die folgende XML-Struktur...

```
<mods:mods>
```

```
            <mods:classification authority="ZVDD">
                  VD17-nova
            </mods:classification>
      </mods:mods>
```

Soll ein Attribut mit dem Inhalt des Metadatenfeldes erzeugt werden, wird an das Ende des Xpath-Ausdrucks ein „/@" mit folgendem Metadatennamen angehängt. Für alle nicht-MODS Namespaces sind hier die Namespace-Präfixe mit anzugeben – zum Beispiel `/@slub:displayLabel`.

Beispiel: Verwendung von Attributen zum Speichern des Wertes

```
      <Metadata>
            <InternalName>CurrentNoSorting</InternalName>
            <WriteXPath>
                  ./mods:mods/#mods:part[@type='host']/@order
            </WriteXPath>
      </Metadata>
```

...und...

```
      <Metadata>
            <InternalName>CurrentNo</InternalName>
            <WriteXPath>
                  ./mods:mods/mods:part/mods:detail/mods:number
            </WriteXPath>
      </Metadata>
```

...erzeugen die XML-Struktur...

```
      <mods:mods>
            <mods:part order="100" type="host">
                  <mods:detail>
                        <mods:number>1</mods:number>
                  </mods:detail>
            </mods:part>
      </mods:mods>
```

MODS Elemente können durch Anhängen von „[]" an den Elementnamen und einer darin enthaltenen Gruppierungsnummer gruppiert werden. Alle Elemente mit derselben Gruppierungsnummer werden gemeinsam in ein Element geschrieben. So ist es zum Beispiel möglich, im Regelsatz zwei verschiedene `<originInfo>` Elemente zu definieren – eines für das Digitalisat und eines für das originale Werk.

Beispiel: Gruppierung von MODS-Elementen

```
      <Metadata>
            <InternalName>PublisherName</InternalName>
            <WriteXPath>
                  ./mods:mods/mods:originInfo[1]/#mods:publisher
            </WriteXPath>
      </Metadata>
      <Metadata>
            <InternalName>PlaceOfPublication</InternalName>
```

```
            <WriteXPath>
./mods:mods/mods:originInfo[1]/#mods:place/mods:placeTerm[@type='text']
            </WriteXPath>
      </Metadata>
```

...und...

```
      <Metadata>
            <InternalName>_placeOfElectronicOrigin</InternalName>
            <WriteXPath>
./mods:mods/mods:originInfo[2]/#mods:place/mods:placeTerm[@type='text']
            </WriteXPath>
      </Metadata>
      <Metadata>
            <InternalName>_dateDigitization</InternalName>
            <WriteXPath>
./mods:mods/mods:originInfo[2]/#mods:dateCaptured[@encoding='w3cdtf']
            </WriteXPath>
      </Metadata>
```

...erzeugen die XML-Struktur...

```
      <mods:originInfo>
            <mods:publisher>Tanzer</mods:publisher>
            <mods:place>
                  <mods:placeTerm type="text">Grätz</mods:placeTerm>
            </mods:place>
      </mods:originInfo>
      <mods:originInfo>
            <mods:place>
                  <mods:placeTerm type="text">Göttingen</mods:placeTerm>
            </mods:place>
            <mods:dateCaptured encoding="w3cdtf">2009</mods:dateCaptured>
      </mods:originInfo>
```

Beispiel: Gruppierung durch eine Gruppe von Metadaten

```
<Group>
    <InternalName>Title</InternalName>
    <WriteXPath>./mods:mods/#mods:titleInfo</WriteXPath>
    <Metadata>
        <InternalName>NonSort</InternalName>
        <WriteXPath>./mods:nonSort</WriteXPath>
    </Metadata>
    <Metadata>
        <InternalName>TitleDocMain</InternalName>
        <WriteXPath>./mods:title</WriteXPath>
    </Metadata>
    <Metadata>
        <InternalName>TitleDocSub</InternalName>
        <WriteXPath>./mods:subTitle</WriteXPath>
    </Metadata>
</Group>
```

...erzeugt die folgende XML-Struktur...

```
<mods:titleInfo>
     <mods:nonSort>Die</mods:nonSort>
     <mods:title>Bau- und Kunstdenkmäler im Regierungsbezirk
                Cassel</mods:title>
     <mods:subTitle>Kreis Gelnhausen</mods:subTitle>
</mods:titleInfo>
```

Die Gruppe `<WriteXPath>` Definition innerhalb des <Group> Elementes erzeugt den Grundpfad. Von da an wird der relative Pfad durch jedes `<WriteXPath>` Element für die einzelnen Metadaten gebildet. Das mappen von Personendaten ist hier ebenfalls möglich und wird im Abschnitt **Mapping für den MODS-Export – Personen** genauer erläutert.

Die Elemente `<ValueCondition>` und `<ValueRegExp>` können – genau wie beim PICA+ Import – zur bedingten Zuweisung und Manipulation von Metadatenwerten genutzt werden, beispielsweise zum Entfernen der der PPN vorangestellten Zeichenkette „PPN“ oder generell zur Bildung von komplexeren Ausdrücken. Auch hier geschieht dies durch die Nutzung von regulären Ausdrücken in Perl5-Syntax.

Beispiel: Bildung einer PURL aus der CatalogIDDigital

```
     <Metadata>
          <InternalName>CatalogIDDigital</InternalName>
          <ValueRegExp>
s/(.*)/http:\/\/resolver\.sub\.uni\-goettingen\.de\/purl\?$1/
          </ValueRegExp>
          <WriteXPath>
               ./mods:mods/mods:identifier[@type='purl']
          </WriteXPath>
     </Metadata>
```

Beispiel: Entfernen der vorangestellten Zeichenfolge „PPN“

```
     <Metadata>
          <InternalName>CatalogIDDigital</InternalName>
          <ValueRegExp>s/^PPN(.*)/$1/</ValueRegExp>
          <WriteXPath>
./mods:mods/mods:recordInfo/#mods:recordIdentifier[@source='gbv-ppn']
          </WriteXPath>
     </Metadata>
```

Beispiel: Durch Präfix bedingtes Mapping eines Identifier-Metadatums

```
     <Metadata>
          <ValueCondition>/^VD17/</ValueCondition>
          <InternalName>CatalogFieldVDid</InternalName>
          <WriteXPath>
               ./mods:mods/#mods:identifier[@type='vd17']
          </WriteXPath>
     </Metadata>
     <Metadata>
          <ValueCondition>/^VD18/</ValueCondition>
          <InternalName>CatalogFieldVDid</InternalName>
          <WriteXPath>
               ./mods:mods/#mods:identifier[@type='vd18']
```

```
            </WriteXPath>
      </Metadata>
```

Das letzte Beispiel zeigt ein bedingtes Mapping, das je nach Präfix des Wertes des internen Metadatums `CatalogFieldVDid` entweder den Typ „vd17“ oder „vd18“ erzeugt.

4.2.3.3 Mapping für den MODS-Export – Personen

Da Personen weitere Merkmale haben, müssen diese Merkmale ebenfalls per XPath geschrieben werden können. Dazu können dem `<WriteXpath>` Element der Regelsatzdatei weitere Elemente folgen. Diese selektieren entsprechende Elemente ausgehend von dem Element, welches durch den in dem `<WriteXpath>` Element angegebenen Ausdruck ausgewählt wurde. Derzeit können die folgenden Elemente für den Export genutzt werden:

`<FirstnameXPath>`	Selektiert das Feld, in das der Vorname der Person geschrieben werden soll.
`<LastnameXPath>`	Selektiert das Feld, in das der Nachname der Person geschrieben werden soll.
`<DisplayNameXPath>`	Selektiert das Feld, in das der Name der Person geschrieben werden soll, der zur Anzeigen dient (hier wird, wenn kein Metadatum dafür existiert, der Name aus Nachname und Vorname aggregiert, in der Form "Nachname, Vorname"). Für den Export wird in dieses Feld der Name der Person nach dem Muster „Nachname, Vorname“ eingetragen, falls kein anderer Wert existiert.

`<IdentifierXPath>`	Selektiert das Feld, in das der Identifier der Person (im Beispiel eine ID aus der Personen-Norm-Datei PND) geschrieben werden soll. Diese Funktion ist noch direkt in die **UGH** Bibliothek integriert. Eine Nutzung ist momentan nur wie im folgenden Beispiel möglich:
	`<IdentifierXPath>`
	`  ../mods:name[@authority='pnd'][@ID='']`
	`</IdentifierXPath>`

Eine allgemein exaktere Abbildung der folgenden noch vorhandenen Attribute von Personen aus dem Dokumentmodell nach MODS ist geplant: `affiliation`, `institution`, `identifierType`, `role`, `personType` und `isCorporation`.

Beispiel: Generierung von Personen

```
      <Metadata>
            <InternalName>Author</InternalName>
            <WriteXPath>
                  ./mods:mods/#mods:name[@type='personal']
                  [mods:role/mods:roleTerm="aut"
                  [@authority='marcrelator'][@type='code']]
            </WriteXPath>
```

```
            <FirstnameXPath>
                  ./mods:namePart[@type='given']
            </FirstnameXPath>
            <LastnameXPath>
                  ./mods:namePart[@type='family']
            </LastnameXPath>
            <DisplayNameXPath>./mods:displayForm</DisplayNameXPath>
            <IdentifierXPath>
                  ../mods:name[@authority='pnd'][@ID='']
            </IdentifierXPath>
      </Metadata>
```

...erzeugt folgende XML-Struktur...

```
      <mods:name ID="pnd07658111X" authority="pnd" type="personal">
            <mods:role>
                  <mods:roleTerm authority="marcrelator" type="code">
                        aut
                  </mods:roleTerm>
            </mods:role>
            <mods:namePart type="family">Castelli</mods:namePart>
            <mods:namePart type="given">Pietro</mods:namePart>
            <mods:displayForm>Castelli, Pietro</mods:displayForm>
      </mods:name>
```

Besitzt eine Dokumentstruktur ein Metadatum mit Namen „TitleDocMain“, so wird dieser Titel als LABEL dieser Struktur in der StructMap eingetragen. Dies soll in zukünftigen Versionen der **UGH** Bibliothek konfigurierbar sein.

Beispiel: Verwendung des Titels als Label für die StructMap (unvollständiger Ausschnitt)

```
      <mets:structMap TYPE="LOGICAL">
            <mets:div LABEL="Allgemeine deutsche Bibliothek"
                  TYPE="Periodical">
                  <mets:div LABEL="Allgemeine deutsche Bibliothek"
                        TYPE="PeriodicalVolume">
                        <mets:div LABEL="Des ersten Bandes erstes Stück."
                        TYPE="PeriodicalIssue">
                              <mets:div LABEL="Inhalt"
                              TYPE="TableOfContents" />
                        </mets:div>
                  </mets:div>
            </mets:div>
      </mets:structMap>
```

4.2.3.4 Metadatenmapping für den Import

Wie bereits in der Einleitung erläutert, ist ein Import von METS-Dateien, dessen Metadaten im Goobi-Namespace gespeichert wurden, ohne Probleme möglich. Die Klasse `ugh.fileformats.mets.MetsMods` implementiert lesenden und schreibenden Zugriff des Interfaces `Fileformat`.

Für den Import von METS-Dateien mit Metadaten im DFG-Viewer MODS-Format muss ein eindeutiges Mapping von MODS-Metadaten zu den internen Metadatentypen des

Dokumentmodells existieren. Da ein 1:1-Mapping von beliebigen internen Metadatentypen nach MODS – vor allem bei bereits vorhandenen Daten und Regelsätzen – oft nicht möglich ist, da MODS im Gegensatz zu den internen Metadatentypen beschränkt ist in seinen beschreibenden Möglichkeiten, wäre ein Import von solchen exportierten Dateien mit einem Mapping der MODS-Metadaten zu den bestehenden internen Metadatentypen nicht mehr eindeutig möglich. Daher ist der lesender Zugriff auf METS-Dateien mit MODS-Metadaten im DFG-Viewer MODS-Format bereits implementiert, jedoch noch nicht aktiviert in der Klasse `ugh.fileformats.mets.MetsModsImportExport`.

4.2.4 Erweiterte Möglicheiten des METS-Exports

Für diverse Möglichkeiten des zvdd/DFG-Viewer METS-Formats sind keine äquivalenten Daten in unserem Dokumentmodell vorgesehen. Für einen ordentlichen METS-Export sind diese Daten jedoch unerlässlich. Für die Festlegung von Werten der folgenden Felder sind in der Klasse `ugh.fileformats.mets.MetsModsImportExport` Setter- und Getter-Methoden implementiert – nähere Erläuterungen finden sich im bereits erwähnten zvdd/DFG-Viewer METS-Profil:

- Die Attribute rightsOwner, rightsOwnerLogo, rightsOwnerSiteURL, digiprovReference, digiprovPresentation, digiprovReferenceAnchor und digiprovPresentationAnchor setzen Werte für die administrative Metadaten-Sektion der METS-Datei, das sind die Metadaten zu Rechteinhaber, Urheber, Herkunft und Online-Präsentation – vergleiche Kapitel „Administrative Metadaten-Sektion“ im zvdd/DFG-Viewer METS-Profil.
- Das Attribut `purlUrl` setzt den Wert eines persistenten Identifiers, der in der METS-Datei als PURL für das gesamte Werk an der entsprechenden Stelle in der logischen StructMap gesetzt wird (siehe „structMap Anforderung 3: Komplexes Dokumentenmodell“).
- Das Attribut `contentIds` hingegen dient der Referenzierung der einzelnen Seiten der physischen Struktur. Momentan wird es als Pfad zur einzelnen Datei genutzt, an das noch der Dateiname angehängt wird (Dieses Feature ist noch experimentell).
- Wird am Ende eines solchen Metadadatums ein regulärer Ausdruck (in Perl5-Syntax) übergeben – in der Form `$REGEXP(s///)` – dann wird dieser reguläre Ausdruck auf den gesamten Wert angewendet, bevor er in das METS übernommen wird.

Beispiel:

Wird für das Metadatum `digiprovReference` beispielsweise der Wert „`http://opac.sub.uni-goettingen.de/DB=1/PPN?PPN=PPN123456789`“ übergeben, und kommt das „`PPN123456789`“ aus einem bestimmten Metadatenfeld und ist in einem bestimmten Kontext nicht beeinflussbar, kann durch „`http://opac.sub.uni-goettingen.de/DB=1/PPN?PPN=PPN123456789$REGEXP(s/PPN=PPN/PPN/)`“ das `PPN` vor der eigentlichen Nummer entfernt werden.

Einer weiteren Erläuterung bedürfen die METS FileGroups, die per `VirtualFileGroup` übergeben werden. Eine jede `VirtualFileGroup` im Objekt `FileSet` des `DigitalDocument` wird als eine METS FileGroup exportiert wie in Kapitel „fileSec Anforderungen 2: File-Groups“ des METS-Profils erläutert. Diese können mit der folgenden Methode

```
DigitalDocument.getFileSet().addVirtualFileGroup()
```

dem Dokumentmodell hinzugefügt werden. Alle benötigten Werte können innerhalb des `VirtualFileGroup` Objekts gesetzt werden, siehe hierzu die Implementierung im Quellkode der Klasse `UghConvert`.

4.2.4.1 Normdaten

Normdaten können in UGH zu jedem Metadatum und jeder Person erfasst werden. Normdaten bestehen immer aus den drei Informationen Kürzel der Datenbank, URL der Datenbank und Wert innerhalb der Datenbank. Sie können mit der Methode `ugh.dl.Metadata.setAutorityFile(String authorityID, String authorityURI, String authorityValue)` gesetzt werden.

Der METS-Export erzeugt dann aus den Werten die Attribute `authority`, `authorityURI` und `valueURI`.

```
<mods:name type="personal" authority="gnd"    authorityURI="http://d-
nb.info/gnd/"    valueURI="http://d-nb.info/gnd/116733721">
     <mods:namePart>Mann, Monika</mods:namePart>
     <mods:role>
                 <mods:roleTerm type="code"
                 authority="marcrelator">aut</mods:roleTerm>
     </mods:role>
</mods:name>
```

```
<mods:subject>
     <mods:topic authority="gnd"   authorityURI="http://d-nb.info/gnd/"
     valueURI="http://d-nb.info/gnd/4077445-4">Silicium</mods:topic>
</mods:subject>
```

4.2.4.2 Persistente Identifier

Das Attribut `contentIDs` des zvdd/DFG-Viewer METS-Formats wird beim METS-Export automatisch erzeugt, sofern das Metadatum `_urn` für die Struktureinheit existiert.

4.2.5 UghConvert

Um die oben beschriebenen Serialisierungsformate des genutzten Dokumentmodells unabhängig von einer externen Applikation zwischeneinander zu konvertieren, kann `UghConvert` genutzt werden. Es handelt sich dabei um eine Java-Applikation, die auf Kommandozeilen-Ebene mit mitgelieferten Skripten für Windows- und Linux-Systeme die jeweiligen Formate lesen und schreiben kann. Zur Ausführung wird lediglich eine vollständige Version der **UGH** Bibliothek (`ughCLI`) sowie ein Java-Runtime-Environment in Version 1.5 benötigt. Die Bibliotheken und der **UGH** Quellkode sowie diese Dokumentation sind im Goobi-Wiki verlinkt[7].

7 http://wiki.goobi.org/index.php/UGH_-_Java_Metadaten_Bibliothek

`UghConvert` liest ein `DigitalDocument` in einem gegebenen Serialisierungsformat ein – wenn nötig unter Angabe des genutzten Regelsatzes – und serialisiert es anschließend wieder in einem ebenfalls gegebenen Format. Es können Daten auch nur gelesen werden sowie in einem Format gelesen und im selben Format wieder gespeichert werden.

4.2.5.1 Die Kommandozeile

Das Skript wird wie folgt aufgerufen, wobei der Pfad zum Java Runtime Environment gesetzt sein muss:

```
java -jar ughCLI-1.6.jar
```

Die grundlegenden Parameter sind die folgenden:

-c, --config <file>	Der Pfad zur genutzten Regelsatzdatei.
-r, --read <format>	Das zu lesende Format: Hier kommen die folgenden in Frage: `mets`, `rdf`, `xstream` oder `picaplus`. Als METS Variante kann hier – wie schon oben beschrieben – zunächst nur das interne METS-Format gelesen werden.
-w, --write <format>	Das zu schreibende Format: Die folgenden Serialisierungsformate kommen hierfür in Frage: `mets`, `dvmets`, `rdf` oder `xstream`, wobei hier zwischen dem internen METS-Format (`mets`) und dem zvdd/DFG-Viewer-METS unterschieden wird (`dvmets`).
-i, --input <file>	Der Pfad zur Quelldatei, im Format wie mit `-r` angegeben.
-o, --output <file>	Der Pfad, unter der die Zieldatei gespeichert werden soll; diese wird im Format wie unter `-o` angegeben serialisiert.
-v, --verbose	Mit diesem Parameter werden die Struktur- und Metadaten des Dokuments beim Lesen ausgegeben.
-h, --help	Gibt eine Hilfe zur Syntax aus.
-q, --quiet	Vermeidet jegliche Art der Ausgabe, Fehler jedoch ausgenommen.
-V, --version	Gibt die Versionen der einzelnen von **UGH** genutzten `Fileformat` Implementierungen aus.

Beispiel: Lesen einer XStream-Datei mit Ausgabe der Struktur- und Metadaten

```
java -jar ughCLI-1.6.jar -c digizeit.xml -i meta.xstream.xml
-r xstream -v
```

Beispiel: Konvertieren einer XStream-Datei in eine (interne) METS-Datei

```
java -jar ughCLI-1.6.jar -c vd17_nova.xml -i meta.xstream.xml
-r xstream -o meta.mets.xml -w mets
```

Beispiel: Konvertieren einer RDF-Datei in eine (interne) METS-Datei, incl. Ausgabe der Struktur- und Metadaten

```
java -jar ughCLI-1.6.jar -c gdz.xml -i meta.rdf.xml
-r rdf -o meta.mets.xml -w mets -v
```

4.2.5.2 Administrative Metadatensektion

Für die Konvertierung in das zvdd/DFG-Viewer METS-Format sind noch einige weitere Werte konfigurierbar, die sich in drei Gruppen einteilen lassen (siehe auch "Eigenheiten des METS-Exports"). Eine davon enthält die Daten für die administrative Metadaten-Sektion `dv:rights` und `dv:links`.

-mro, --metsrightsowner <owner>	Der Urheber des Digitalisats.
-mrl, --metsrightslogo <url>	Eine URL des Logos des Urhebers; dieses Logo wird durch den DFG-Viewer sowie das zvdd-Portal entsprechend angezeigt.
-mru, --metsrightsurl <url>	Eine URL der Homepage des Urhebers.
-mrc, --metsrightscontact <url>	Eine URL zu einem Kontaktformular der Homepage des Urhebers oder alternativ eine E-Mail-Adresse. Hiermit soll vom DFG-Viewer ein direkter Kontakt zum Urheber ermöglicht werden.
-mdr, --metsdigiprovreference <url>	Eine URL auf den Katalogeintrag des Digitalisats – oder auf die untergeordnete Struktur, zum Beispiel einen Band – falls eine solche existiert.
-mdra, --metsdigiprovreferenceanchor <url>	Eine URL auf den Katalogeintrag des Digitalisats – oder auf die Übergeordnete Struktur, zum Beispiel eine Zeitschrift – wenn eine solche existiert.
-mdp, --metsdigiprovpresentation <url>	Eine URL auf die Online-Präsentation des Digitalisats – oder auf die untergeordnete Struktur, zum Beispiel einen Band – falls eine solche existiert.
-mdpa, --metsdigiprovpresentationanchor <url>	Eine URL auf die Online-Präsentation des Digitalisats – oder auf die übergeordnete Struktur, zum Beispiel eine Zeitschrift – falls eine solche existiert.

4.2.5.3 Dateigruppen (FileGroups)

Daten für die für den DFG-Viewer benötigten Dateigruppen (FileGroups), mindestens erforderlich sind FileGroups mit den Namen `MIN` und `DEFAULT`.

-fmin, --minfilesuffix <filesuffix>	Die Dateiendung der Dateinamen in der FileGroup `MIN`.
-mmin, --minmimetype <mimetype>	Der Mimetype der Dateien in der FileGroup `MIN`.
-pmin, --minpath <path>	Der Pfad der Dateien in der FileGroup `MIN`, der Dateiname wird aus den Eigenschaften des Objekts `ContentFile` genommen.
-smin, --minidsuffix <idSuffix>	Die Endung der XML ID, die in der METS-Datei für die FileGroup `MIN` verwendet wird.

Alle weiteren Angaben sind für die Dateigruppen `DEFAULT`, `MAX`, `DOWNLOAD`, `LOCAL`, `PRESENTATION` und `THUMBS` äquivalent. Die Dateigruppe `LOCAL` wird automatisch erzeugt, kann jedoch bei Bedarf mit eigenen Werten überschrieben werden. Die Parameter für die weiteren Dateigruppen sind äquivalent zu `minfilesuffix`, `minmimetype`, `minpath` und `minidsuffix` festgelegt, zum Beispiel `defaultpath`, `maxpath`, `downloadpath`, `localpath`, `presentationpath` und `thumbspath`. Alle Parameter finden sich in der Hilfe von `UghConvert`:

```
java -jar ughCLI-1.6.jar -h
```

Für die Konfiguration der erweiterten Parameter (administrative Metadaten und Dateigruppen) sollten die langen Parameter (`--mimidsufffix`, `--minmimetype`, etc.) verwendet werden.

Beispiel: Ausschnitt aus einer METS-Datei (FileGroups)

Aus dem Angaben

```
java -jar ughCLI-1.6.jar
-c vd17_nova.xml
-i meta.rdf.xml
-r rdf
-o meta.mets.xml
-w dvmets
--default filesuffix .jpg
--defaultmimetype image/jpeg
--defaultpath /pfad/zu/den/bildern/
--defaultidsuffix _DEF
--metsrightsowner "SUB Göttingen"
--metsrightslogo http://gdz.sub.uni-goettingen.de/logo_gdz_dfgv.png
--metsrightsowner mailto:dfg-viewer@gdz.sub.uni-goettingen.de
--metsrightsurl http://gdz.sub.uni-goettingen.de
--metsdigiprovreference
      http://opac.sub.uni-goettingen.de/DB=1/PPN?PPN=590628720
--metsdigiprovpresentation
      http://resolver.sub.uni-goettingen.de/purl?PPN590628720
```

ergibt sich die folgende METS FileGroup `DEFAULT`...

```
      <mets:fileGrp USE="DEFAULT">
            <mets:file ID="FILE_0000_DEF" MIMETYPE="image/jpeg">
                  <mets:FLocat LOCTYPE="URL"
                  xlink:href="/pfad/zu/den/bildern/00000001.jpg"
                  xmlns:xlink="http://www.w3.org/1999/xlink" />
            </mets:file>
            <mets:file ID="FILE_0001_DEF" MIMETYPE="image/jpeg">
                  <mets:FLocat LOCTYPE="URL"
                  xlink:href="/pfad/zu/den/bildern/00000002.jpg"
                  xmlns:xlink="http://www.w3.org/1999/xlink" />
            </mets:file>
            ...
      </mets:fileGrp>
```

...und folgende administrative METS Metadatensektion:

```
      <mets:amdSec ID="AMD">
            <mets:rightsMD ID="RIGHTS">
                  <mets:mdWrap MDTYPE="OTHER" MIMETYPE="text/xml"
                  OTHERMDTYPE="DVRIGHTS">
                        <mets:xmlData>
                              <dv:rights xmlns:dv="http://dfg-viewer.de/">
                                    <dv:owner>SUB Göttingen</dv:owner>
                                    <dv:ownerLogo>
                               http://gdz.sub.uni-
goettingen.de/logo_gdz_dfgv.png
                                    </dv:ownerLogo>
                                    <dv:ownerSiteURL>
                                          http://gdz.sub.uni-goettingen.de
                                    </dv:ownerSiteURL>
                                    <dv:ownerContact>
                              mailto:dfg-viewer@gdz.sub.uni-goettingen.de
                                    </dv:ownerContact>
                              </dv:rights>
                        </mets:xmlData>
                  </mets:mdWrap>
            </mets:rightsMD>
            <mets:digiprovMD ID="DIGIPROV">
                  <mets:mdWrap MDTYPE="OTHER" MIMETYPE="text/xml"
                  OTHERMDTYPE="DVLINKS">
                        <mets:xmlData>
                              <dv:links xmlns:dv="http://dfg-viewer.de/">
                                    <dv:reference>
                  http://opac.sub.uni-goettingen.de/DB=1/PPN?PPN=590628720
                                    </dv:reference>
                                    <dv:presentation>
                  http://resolver.sub.uni-goettingen.de/purl?PPN590628720
                                    </dv:presentation>
                              </dv:links>
                        </mets:xmlData>
                  </mets:mdWrap>
            </mets:digiprovMD>
```

```
    </mets:amdSec>
```

4.3 PICA+

Serialisierungsklasse:

```
ugh.fileformats.opac.PicaPlus (v1.3)
```

4.3.1 Einführung

Das PICA+ Format wird überwiegend in Bibliothekskatalogen genutzt und beschreibt lediglich die bibliographische Einheit; es sind nur Metadaten für eine Struktureinheit in PICA+ zu finden und keinerlei Strukturdaten. Gegebenenfalls können einige Metadatenfelder Verweise auf andere Struktureinheiten enthalten, die jedoch durch die Applikation nochmals aus dem Bibliothekskatalog geholt werden und über die API entsprechend verknüpft werden muss.

Die Grundlage der PICA+ Serialisierung der **UGH** Bibliothek bildet das Format PICA+ XML, wie es durch den GBV in seinem Verbund-WIKI beschrieben[8] wird und durch die getOpac-Klassen von Jens Ludwig implementiert wurde. Die Serialisierungsklasse kann nicht die Daten selbständig aus einem Katalog herunterladen, sondern muss mit einer entsprechenden XML-Datei „gefüttert" werden.

Die PICA+ Serialisierungsklasse kann aus einem PICA-Datensatz lediglich eine Struktureinheit mit ihren Metadaten erstellen. Diese Struktureinheit ist immer die oberste logische Struktureinheit des Dokumentes. Die Klasse verfügt lediglich über eine Lese-Methode. Ein Schreiben in das PICA+ Format erscheint nicht sinnvoll, da nicht zuletzt auch entsprechende Schnittstellen zum Upload der Daten in Bibliothekssystem kaum vorhanden sind.

4.3.2 Konfiguration

Die PICA+ Konfiguration erfolgt innerhalb des umschließenden `<PicaPlus>` Elements, welches direkt dem `<Formats>` Element untergeordnet ist. Innerhalb des `<PicaPlus>` Elements kommen die typ-spezifischen Elemente `<Person>`, `<Metadata>` und `<DocStruct>` zu Einsatz, die jeweils einen internen Datentyp mappen. Ebenso wie beim RDF/XML-Format findet hier immer ein 1:1 Mapping statt.

Für das Mapping wird auf die Merkmale der PICA+ Struktur zurückgegriffen, innerhalb dieser typ-spezifischen Elemente sind folgende Unterelemente erlaubt:

`<picaMainTag>`	Die PICA+ Feldnummer. Dieses Element muss genau einmal vorhanden sein.
`<picaSubTag>`	Der PICA+ Feldtrenner innerhalb des mit `<picaMainTag>` angegebenen Feldes. Der Inhalt dieses Unterfeldes entspricht dem Wert des im `<Name>` Element angegebenen Metadatums. Innerhalb des `<Metadata>` und `<DocStruct>` Elements muss dieses Feld genau einmal vorhanden sein. Für das `<Person>` Element gelten andere Regeln (siehe unten).
`<valueCondition>`	Bedingtes Mapping: Mit der Angabe eines regulären Ausdrucks

8 http://www.gbv.de/wikis/cls/PICAplus_in_XML

	(Perl5-Syntax) kann hier eine Bedingung angegeben werden. Nur, wenn diese Bedingung auf den Inhalt des oben definierten PICA+ Feldes zutrifft, wird der Wert dem internen Metadatum zugewiesen. Dieses Feld ist optional, Beispiel siehe unten.
`<name>`	Der interne Metadatenname. Dieses Element muss ebenfalls genau einmal vorhanden sein.
`<valueRegExp>`	Nachträgliche Bearbeitung von PICA+ Feldwerten: Mit der Angabe eines regulären Ausdrucks (Perl5-Syntax) kann der Wert des Metadatums manipuliert werden. Dieses Feld ist optional, Beispiel siehe unten.

Beispiel: Mapping von PICA+ Felder

```
<Metadata>
      <picaMainTag>021A</picaMainTag>
      <picaSubTag>a</picaSubTag>
      <Name>TitleDocMain</Name>
</Metadata>
```

Beispiel: Nutzung von bedingten Zuweisungen

```
<Metadata>
      <picaMainTag>007S</picaMainTag>
      <picaSubTag>0</picaSubTag>
      <ValueCondition>/^VD17/</ValueCondition>
      <Name>CatalogFieldVDseventeen</Name>
      <ValueRegExp>s/^VD17\s(.*)/$1/</ValueRegExp>
</Metadata>
<Metadata>
      <picaMainTag>007S</picaMainTag>
      <picaSubTag>0</picaSubTag>
      <ValueCondition>/^VD18/</ValueCondition>
      <Name>CatalogFieldVDeighteen</Name>
      <ValueRegExp>s/^VD18\s(.*)/$1/</ValueRegExp>
</Metadata>
```

Der Wert von `picaMainTag 007s` und `picaSubTag 0` wird nur dann dem internen Metadatum `CatalogFieldVDseventeen` zugewiesen, wenn dessen Inhalt mit „VD17“ beginnt. Sollte der Inhalt des PICA+ Feldes mit „VD18“ beginnen, wird dieser dem internen Metadatum `CatalogFieldVDeighteen` zugewiesen.

Wenn eine der Bedingungen erfüllt ist, wird zusätzlich noch das „VD17 “ oder das „VD18 “ vor der eigentlichen VD17- bzw. VD18-Nummer (incl. dem Leerzeichen) entfernt.

Beispiel: Manipulation von Metadaten beim Import

```
<Metadata>
      <picaMainTag>003@</picaMainTag>
      <picaSubTag>0</picaSubTag>
      <Name>CatalogIDDigital</Name>
      <ValueRegExp>s/(.*)/PPN$1/</ValueRegExp>
</Metadata>
```

Mit diesem regulären Ausdruck wird der Wert des PICA+ (PPN) um die Zeichenkette „PPN“ ergänzt.

Da Personen weitere Merkmale aufweisen und nicht nur ein einfaches Typ-Wert-Paar sind, kann für jedes dieser Merkmale ein `<picaSubTag>` innerhalb des `<Person>` Elements existieren. Das entsprechende Attribut `type` gibt das entsprechende Merkmal an. Folgende Werte für dieses Attribut sind gültig:

firstname	Vorname der Person
lastname	Nachname der Person
identifier	Identifier der Person (bspw. aus der Personennormdatenbank)
expansion	Es können weiterhin Vor- und Nachname aus dem Pica-Feld „Expansion der Ansetzungsform“ extrahiert werden.

Beispiel: Mapping von Personen

```
<Person>
      <picaMainTag>028A</picaMainTag>
      <Name>Author</Name>
      <picaSubTag type="firstname">d</picaSubTag>
      <picaSubTag type="lastname">a</picaSubTag>
      <picaSubTag type="identifier">9</picaSubTag>
      <picaSubTag type="expansion">8</picaSubTag>
</Person>
```

Da der Typ der jeweiligen Dokumentstruktur abhängig vom Wert eines Feldes ist, existiert innerhalb des Elements `<DocStrct>` noch ein weiteres Pflichtelement `<picaContent>`. Nur wenn das mit `<picaMainTag>` und `<picaSubTag>` spezifizierte Element den in `<picaContent>` definierten Content aufweist, wird eine entsprechende Dokumentstruktur anlegt und es werden ihr die Metadaten zugeordnet. Der Typ der Dokumentstruktur wird im Element `<Name>` spezifiziert.

Hierbei kann es auch zu einem n:1 Mapping kommen, das heißt, es können mehrere Typen aus dem Pica-System einem internen Typ zugewiesen werden. Dies ist notwendig, da die Buchstabenkombination des PICA+ Formats nicht nur den bibliographischen Typ, sondern auch die Erscheinungsform (gedruckt, digital, Microform) enthält. Entsprechend muss für jede Buchstabenkombination ein Mapping vorgenommen werden.

Beispiel: Mapping eines MultiVolume-Werkes

```
      <DocStruct>
            <picaMainTag>002@</picaMainTag>
            <picaSubTag>0</picaSubTag>
            <picaContent>Oc</picaContent>
            <Name>MultivolumeWork</Name>
      </DocStruct>
      <DocStruct>
            <picaMainTag>002@</picaMainTag>
            <picaSubTag>0</picaSubTag>
            <picaContent>Ac</picaContent>
            <Name>MultivolumeWork</Name>
      </DocStruct>
```

Für das Mapping von Dokumentstrukturen ist zu beachten, dass lediglich die ersten im Regelsatz definierten Buchstaben verglichen werden. Dies ist quasi mit einer trunkierten Suche gleichzusetzen. Da in dem Beispiel oben lediglich zwei Buchstaben im Feld <picaContent> angegeben sind, werden auch nur die ersten beiden Buchstaben in der PICA+ Datei berücksichtigt. Ferner erfolgt der Vergleich unter Beachtung der Groß/Kleinschreibung (case-sensitive).

4.4 XStream

Serialisierungsklasse:

```
ugh.fileformats.mets.XStream (v1.2)
```

Die Serialisierung via XStream hat zunächst das RDF/XML-Schema abgelöst, um unabhängig zu sein von Mappings und zusätzlichen Einträgen im Regelsatz. XStream kann das Dokumentmodell komplett serialisieren, hat jedoch einige Nachteile (siehe unten), so dass dieses Format in Zukunft aus der UGH Bibliothek entfernt wird, lesender Zugriff wird jedoch weiterhin aus Kompatibilitätsgründen möglich sein!

XStream[9] serialisiert das komplette Java-Objekt `DigitalDocument` im XStream XML-Format, so dass alle Strukturtypen, Metadaten, Referenzen, etc. des Dokumentmodells gesichert werden. Einträge im Regelsatz oder ein Mapping sind hier nicht nötig. Da es jedoch enge Beziehungen des `DigitalDocument` zum Regelsatz gibt (Namen der Metadaten, Quantifier, etc.), werden zunächst einige Teile des Regelsatzes mit serialisiert. Nach dem Laden des serialisierten Objekts wird es deswegen mit dem aktuellen Regelsatz abgeglichen, so dass Änderungen im Regelsatz auf das Objekt abgebildet werden.

Obwohl XStream als Serialisierungsformat den Vorteil hat, dass keine weitere Konfiguration vorgenommen werden muss, um es zu nutzen, hat es den doch den Nachteil, dass sehr große und komplexe XML-Dateien in einem komplexen Format entstehen, die schlecht bis gar nicht außerhalb der **UGH** Bibliothek – evtl. per Hand – bearbeitet werden können.

9 http://xstream.codehaus.org/

4.5 RDF/XML

Serialisierungsklasse:

```
ugh.fileformats.exel.RDFFile (v1.2)
```

Das Serialisierungsformat RDF/XML ist aus Kompatibilitätsgründen Teil der UGH Bibliothek (beispielsweise zum Lesen alter RDF/XML-Bestände, die in ein neues Format überführt werden sollen), denn einige neuere Teile der Dokumentmodells können damit nicht mehr serialisiert werden und gingen somit bei einer Speicherung verloren. Dies betrifft momentan zum Beispiel die erweiterten Paginierungsarten „Spaltenzählung“ und „Blattzählung“. Daher sollte das Format nicht mehr zum internen Speichern genutzt werden!

4.5.1 Einführung

Das RDF/XML-Format ist das ursprünglich für das AGORA Dokument-Management-System (AGORA DMS) entwickelte Datenformat, welches von diesem für den Import unterstützt wird. Ferner generiert das dazugehörige Meta- und Strukturdatenerfassungsprogramm AGORA Editor dieses Format. Es handelt sich also um kein generisches RDF-basiertes XML-Format, sondern vielmehr um eine ganz spezielle Ausprägung, um digitalisierte Dokumente zu beschreiben, wobei davon ausgegangen wird, dass jede Seite als einzelnes Image gespeichert wird. Um Daten in diesem Format übernehmen und in das DMS importieren zu können, wird das entsprechende Datenformat von **UGH** unterstützt.

Das RDF/XML-Format bildet nicht das komplette Dokumentmodell ab. Während die logische Struktur mit ihren Metadaten frei konfigurierbar ist, ist die physische Struktur nur stark eingeschränkt erhalten. Seiten werden zu so genannten Seitenbereichen mit identischer Paginierung zusammengefasst (Paginierungssequenzen), hierdurch kann ein logisches Strukturelement lediglich eine eine Startseite und eine Endseite besitzen. Unterbrechungen im Seitenverlauf sind nicht möglich.

Damit das Mapping entsprechend funktioniert, sind die Strukturtypen für die physische Struktur vorgegeben. Das oberste physische Strukturelement muss vom Typ `BoundBook` sein. Die darunterliegenden Seiten sind vom Typ `page`. Bei abweichenden Typen besteht Gefahr, dass die Paginierungssequenzen nicht erzeugt werden können.

Ferner kennt das RDF/XML-Format keine Inhaltsdateien. Um entsprechende Objekte intern zu erzeugen, müssen die Imagedateien einer konkreten Namenskonvention erfolgen: 8-stellig aufsteigend beginnend mit 1 (mit vorangestellten Nullen) und 3 Stellen für die Endung `tif`, beispielsweise `00000001.tif`.

4.5.2 Konfiguration

Die Konfiguration des Formats findet ausschließlich im umschließenden `<RDF>` Element statt.

Strukturtypen werden einfach 1:1 gemappt. Daher umfasst das `<DocStrct>` Element auch lediglich die beiden Unterelemente `<Name>` und `<RDFName>`. `<Name>` enthält den internen Namen des Strukturtyps wie im Regelsatz oben definiert. `<RDFName>` ist der Name des Strukturtyps, so wie er im Attribut `TYPE` des Elements `<DocStruct>` in der RDF/XML-Datei gespeichert wird.

Das Metadaten-Mapping ist prinzipiell sehr ähnlich, jedoch aufgrund der Struktur von RDF/XML etwas erweitert: Das `<Metadata>` Element umfasst die Konfiguration für einen Metadatentyp. Der interne Name wird im Element `<Name>` festgelegt, der entsprechende Name des XML-Elements in der XML-Datei in dem Element `<RDFName>`.

Beispiel: Konfiguration des Metadatenmappings für RDF/XML

```
<Metadata>
      <Name>TitleDocMain</Name>
      <RDFName>AGORA:TitleDocMain</RDFName>
</Metadata>
```

Beispiel: Umsetzung in der RDF/XML Datei

```
<AGORA:DocStrct AGORA:Type="AGORA:Monograph">
      <AGORA:TitleDocMain>Juristenzeitung</AGORA:TitleDocMain>
</AGORA:DocStrct>
```

Als Ergänzung zu dem 1:1 Mapping können Metadaten auch in `<RD:Bag>` oder `<RDF:Seq>` Elementen gespeichert werden. Dies kann mittels der Attribute `rdfList` und `rdfListType` angepasst werden. `rdfList` enthält dabei den Namen des umschliessenden XML-Elementes, in denen ein `<RDF:Bag>` oder `<RDF:Seq>` Element enthalten ist. Das eigentliche Metadatum wir dann als separates Element innerhalb des `<RDF:Li>` Elements gespeichert.

Beispiel: RDF-List Konfiguration

```
<Metadata rdfList="AGORA:ListOfCreators" rdfListType="seq">
      <Name>IllustratorArtist</Name>
      <RDFName>AGORA:Illustrator</RDFName>
</Metadata>
```

Beispiel: Umsetzung in der RDF/XML-Datei

```
<AGORA:ListOfCreators>
      <RDF:Seq>
            <RDF:Li>
                  <AGORA:Author>
                  <AGORA:CreatorLastName>Meier</
                        AGORA:CreatorLastName>
                  <AGORA:CreatorFirstName>T</
                        AGORA:CreatorFirstName>
                  </AGORA:Author>
            </RDF:Li>
      </RDF:Seq>
</AGORA:ListOfCreators>
```

Wie an dem Beispiel zu sehen ist, sind die RDF-Listen vor allem für Personen sinnvoll, da nur in Ihnen zwischen Vor- und Nachnamen der Person unterschieden wird.

Für das Einlesen der Daten ist theoretisch auch ein n:1 Mapping möglich, das heißt, unterschiedliche RDF/XML-Typen werden auf denselben internen Metadatentyp gemappt. Für das Schreiben wird jedoch immer die erste Mappingdefinition verwendet, hier ist also nur ein 1:1 Mapping möglich.

4.6 AGORA-Database

Die AGORA-Datenbank Serialisierung ist nicht mehr in der UGH Bibliothek enthalten, das folgende Kapitel ist jedoch aus Dokumentationsgründen erhalten geblieben!

4.6.1 Einführung

Das AGORA DMS-System setzt auf einer klar definierten Datenbankstruktur auf. Diese Datenbankstruktur ist so flexibel gehalten, dass in ihr ebenfalls unterschiedliche Meta- und Strukturtypen definiert werden können. Allerdings bildet die Struktur lediglich eine Untermenge des oben erläuterten Dokumentmodells ab. Diese Untermenge ist identisch mit der des RDF/XML Formats. Weitere Erläuterungen finden sich dort. Derzeit kann diese Klasse lediglich Daten in die AGORA-Datenbank schreiben.

4.6.2 Datenbankkonfiguration

Die Konfiguration befindet sich in dem umschließenden `<AGORADATABASE>` Element, welches sich dem `<Formats>` Element unterordnet. Zunächst müssen einige Basisparameter der Datenbank angegeben werden. Dies sind im Einzelnen:

<databaseURL>	URL der Datenbank, auf die zugegriffen werden soll.
<databaseUser>	Benutzer, der entsprechenden Lese-/Schreibzugriff auf die Datenbank hat.
<databasePassword>	Passwort des Benutzers.
<databaseDriver>	Name der Klasse des JDBC-Treibers, die für die Kommunikation mit der Datenbank zuständig ist.

4.6.3 Definition des Persistenten Identifiers in der Datenbank

Weiterhin sind einige funktional relevante Angaben zu machen. So legt das Element `<persistentIdentifier>` zum Beispiel fest, welcher Metadatentyp zur persistenten Identifizierung einer Struktureinheit gilt. Dieses Element muss genau einmal vorhanden sein, da anhand dessen überprüft wird, ob ein Dokument bereits im Repository importiert wurde und so gegebenenfalls die Serialisierung abgelehnt werden muss.

Beispiel: Konfiguration der PPN als persistenter Identifier:

```
<persistentIdentifier>CatalogIDDigital</persistentIdentifier>
```

4.6.4 Vererbbare Metadatentypen

Das Element `<inheirableMetadataTypes>` enthält all die Metadatentypen die von einer Struktureinheit auf unterliegende Struktureinheiten vererbt werden soll. Entsprechende Einträge werden in der Datenbank dupliziert und erlauben so einen schnellen Zugriff bzw. eine schnelle Filterung. Die Metadatentypen werden jedoch nur dann dupliziert, wenn die Struktureinheit kein eigenes Metadatum dieses Typs besitzt. Innerhalb das `<inheirableMetadataTypes>` Elements befindet sich für jeden Metadatentyp ein `<internalName>` Element.

Beispiel: Vererbung der Digitalen Kollektion beim Datenbankimport

```
    <inheirableMetadataTypes>
                <internalName>singleDigCollection</internalName>
    </inheirableMetadataTypes>
```

4.6.5 Wertelisten

Eine Werteliste kann beim Schreiben in die Datenbank bestimmte Werte ersetzen. Dies kann dann sinnvoll sein, um Sprachcodes umzuschreiben. In einer entsprechenden Mappingdatei werden die alten Werten (aus der API) den neuen Werten (in der Datenbank) gegenüber gestellt. Die Mappingdatei ist nichts weiter als eine Textdatei, die pro Zeile ein Wertepaar enthält. Der erste Wert ist der zu ersetzende Wert (also der in der **UGH** Bibliothek verwendete Wert), der zweite Wert ist der entsprechende Wert aus der Datenbank. Beide Werte sind durch ein Leerzeichen getrennt.

Beispiel: Aufbau der Mappingdatei

```
de 57
en 58
fr 626
```

Eine Werteliste hat immer einen Namen und einen Pfad, der in das lokale Dateisystem zeigt. Der Name der Werteliste wird im `<ListName>` Element gespeichert, der Pfad im `<FileName>` Element.

Beispiel: Definition einer Werteliste

```
    <ValueList>
          <ListName>languagelist</ListName>
          <FileName>C:/olms/language.txt</FileName>
    </ValueList>
```

Der Name der Werteliste dient dazu, sie aus dem Metadatentyp-Mapping heraus ansprechen zu können. Daher muss deren Name eindeutig sein.

4.6.6 Metadatentyp-Mapping

Hauptbestandteil der Konfiguration für diese Serialisierungsklasse ist jedoch das Mapping der Metadatentypen in bestimmte Tabellen und Tabellenspalten. Für das Verständnis der unten beschriebenen Erläuterungen ist die Kenntnis des AGORA-Datenbankschemas daher sinnvoll.

Für jeden Metadatentyp existiert ein `<Metadata>` Element. Innerhalb dieses Elements wird die Tabelle, die Spalte oder sogar ein Wert ausgewählt. Jedes `<Metadata>` Element muss ein Unterelement `<InternalName>` und `<TableName>` besitzen. `<InternalName>` enthält den internen Namen des Metadatentyps und `<TableName>` enthält den Namen der Tabelle, in welcher der Wert des Metadatums gespeichert werden soll. Je nach Datenbankmanagementsystem ist hier evtl. auf Groß/Kleinschreibung zu achten. Entsprechend der Relation zwischen Struktureinheit und Metadatum gibt es mehrere Möglichkeiten, das Metadatum zu speichern:

1:1	Metadaten des entsprechenden Typs dürfen max. einmal pro Struktureinheit auftreten, das Metadatum wird daher in der gleichen Tabelle gespeichert wie die Struktureinheit. Dies ist in dem AGORA-Datenbankschema die DOC-Tabelle für Struktureinheiten der logischen Struktur und die ISET-Tabelle für Struktureinheiten der physischen Struktur. Hierzu muss lediglich die entsprechende Tabellenspalte im Element `<TableColumn>` angegeben werden. Zu beachten ist jedoch, dass die entsprechende Relation zwischen Strukturtyp und Metadatum im Regelsatz (num-Attribut) auf `1m`, `1o` oder `+` steht.
	Beispiel:
	`<Metadata>`
	`  <InternalName>CatalogIDDigital</InternalName>`
	`  <TableName>Doc</TableName>`
	`  <TableColumn>CatalogIDDigital</TableColumn>`
	`</Metadata>`

1:n	Das Metadatum kann mehrmals für eine Struktureinheit vorkommen. In diesem Fall gibt es zwei Tabellen. Die DocAttribute-Tabelle enthält den Typ des Metadatums, die DocAttribValue-Tabelle verknüpft zzgl. des Metadatumwerts mit der Tabelle für die Struktureinheit (DOC-Tabelle). In diesem Fall ist in `<TableName>` der Wert `DocAttribute` anzugeben. Das Element `<FieldValue>` muss dann den Wert des internen Attributnamen besitzen, wie er in der DocAttribute Tabelle gespeichert ist. Das Element `<TableColumn>` existiert nicht.
	Beispiel:
	`<Metadata>`
	`  <InternalName>PublisherName</InternalName>`
	`  <TableName>DocAttribute</TableName>`
	`  <FieldValue>Publisher</FieldValue>`
	`</Metadata>`

m:n	Ein Metadatum kann mehrmals für eine Struktureinheit vorkommen. Im Gegensatz zu der 1:n Verknüpfung wird allerdings der Metadatenwert mehrmals für unterschiedliche Struktureinheiten verwendet. Dieses Modell findet für die DigitalCollection- und die PlacePublication-Tabelle Anwendung. Sollen Daten in einer dieser beiden Tabellen gespeichert werden, muss das `<Metadata>` Element lediglich ein `<TableName>` Element enthalten. Dieses muss den Wert `PlacePublication` oder `DigitalCollection` enthalten.
	Beispiel:
	`<Metadata>`
	`  <InternalName>PlaceOfPublication</InternalName>`
	`  <TableName>PlacePublication</TableName>`

	`</Metadata>`

Personen	Personen werden in einer eigenen Tabelle gespeichert. Dies ist notwendig, da die zusätzlichen Merkmale eines Person-Objekts keinen Platz in den herkömmlichen Metadatentabellen haben. Das `<FieldValue>` Element gibt in diesem Fall den Rollennamen an, den dieser Personentyp in der Datenbank besitzt. Dieser wird in der Datenbank in der CreatorType-Tabelle gespeichert. Das Element `<TableName>` muss dazu zwingend den Wert `Creator` besitzen. Beispiel:
	`<Metadata>`
	`<InternalName>Author</InternalName>`
	`<TableName>Creator</TableName>`
	`<FieldValue>AUTHOR</FieldValue>`
	`</Metadata>`

Zusätzlich können nicht nur die Typen konvertiert werden, sondern auch die Werte eines Metadatums. Dazu dienen Wertelisten, die innerhalb des `<AGORADATABASE>` Elements definiert sein müssen. Eine solche Werteliste kann für die Wertkonvertierung durch einfügen des Elements `<ValueList>` genutzt werden. Dieses Element muss den Namen der Werteliste enthalten. Es ist sicherzustellen, dass eine entsprechende Werteliste mit diesem Namen auch existiert.

Beispiel: Nutzung einer Werteliste für die Wertkonvertierung

```
<Metadata>
      <InternalName>DocLanguage</InternalName>
      <TableName>Doc</TableName>
      <TableColumn>IDLanguage</TableColumn>
      <ValueList>languagelist</ValueList>
</Metadata>
```

4.6.7 Strukturtyp-Mapping

Der Dokumentstrukturtyp wird relativ einfach gemappt. Da der Name des Strukturtyps ebenfalls in der Datenbank gespeichert wird, jedoch mittels einer separaten Tabelle über eine ID mit der eigentlichen Struktureinheit verknüpft wird, ist der entsprechende Typname lediglich ein einziges Mal in der Datenbank gespeichert.

Neue Strukturtypen werden durch die Serialisierungsklasse automatisch in der Tabelle hinzugefügt, sobald die Klasse ihre Präferenzen geladen hat, das heißt, ein Update der Strukturtypen erfolgt automatisch vor den eigentlichen Schreiboperationen für das jeweilige Dokument.

Mittels des `<DocStruct>` Elements kann der entsprechende interne Name mit dem in der Datenbank gespeicherten Namen gemappt werden. Der interne Name wird dabei im `<InternalName>` Element definiert. Das `<DBName>` Element enthält den Namen aus der BibCategory-Tabelle der Datenbank.

Beispiel: Mapping der Strukturtypen für die AGORA-Datenbank

```
    <DocStruct>
        <InternalName>PeriodicalIssue</InternalName>
        <DBName>ZSCHR_HEFT</DBName>
    </DocStruct>
```

5 Ausblick

Dieses Dokument hat kurz einen Überblick über das Dokumentmodell und dessen Anpassbarkeit mittels eines Regelsatzes gegeben. Ferner sind kurz einige Klassen und deren Konfigurationsmöglichkeiten zur Serialisierung der Daten beschrieben worden.

Zukünftig mag es sinnvoll sein, die Konfigurationsmöglichkeiten des Regelsatzes weiter auszubauen. Es ist bspw. eine Typisierung von Metadatenwerten denkbar. Derzeit wird ein Metadatenwert lediglich als Zeichenkette betrachtet. Zukünftig mag es für einige Anwendungsfälle sinnvoll sein, einem bestimmten Schema – wie einem bestimmten Zeit- oder Datumsformat – zu folgen oder aber lediglich nur ganzzahlige Werte für bestimmte Metadatentypen zu erlauben. Der darüber liegenden Applikation würde viel zusätzliche Businesslogik erspart bleiben, die derzeit entsprechende Eingaben vor der Weiterverarbeitung prüfen muss. Ein beschreibendes XML-Schema, das solche Dinge abbildet, wäre sehr wünschenswert und hilfreich. Weiterhin wäre ein Mechanismus sehr sinnvoll, der auch Dinge abbilden kann, die zunächst nicht mit einem XML-Schema abgedeckt werden können, wie zum Beispiel eine Prüfung, ob ein am Anfang des Regelsatzes definiertes Metadatum an allen benötigten Bereichen im Regelsatz wieder auftaucht. Zum Beispiel in einer bestimmten `<formats>` Sektion oder bei den Strukturdaten; so erweitert, könnte eine Methode des **UGH** Bibliothek die Korrektheit eines Regelsatzes problemlos umfassend prüfen und auf Fehler hinweisen.

Weiterhin kann der Anwendungszweck der **UGH** Bibliothek ausgedehnt werden, wenn diese mittels einer Persistenzschicht die einzelnen Objekte direkt ansprechen könnte, ohne über Serialisierungsklassen zu gehen. Dieser Verzicht der Serialisierungsschicht würde dazu führen, dass zum Ändern/Löschen/Aktualisieren von einzelnen Objekten nicht immer das komplette Dokument gelesen und geschrieben werden müsste. Serialisierungsklassen würden dann lediglich zum Import und Export genutzt werden. Auf eine solche Klassenbibliothek ließe sich auch direkt ein Repository aufbauen, welches aufgrund der identischen API all jene Tools nutzen kann, die derzeit auf beispielsweise METS-Dateien anwendbar sind.

Als Beispiel für weitere Vereinfachungen bzw. Standardisierungen der **UGH** Bibliothek könnte man für die Konfiguration der Anzahl der erlaubten Metadatenfelder pro Strukturelement (`num` Attribut) auf bewährte Analogien zurückgreifen und nicht eigene Konventionen nutzen, hier bietet sich zum Beispiel die Quantifizierungssyntax regulärer Ausdrücke an. Auch wenn solche Änderungen sicherlich teilweise weitreichende Anpassungen der **UGH** Klassen zur Folge haben: Die Kompatibilität zu früheren Versionen sollte im besten Fall bestehen bleiben.

6 Zusätzliche Details

Diese Dokumentation wurde von Markus Enders, Stefan E. Funk und Robert Sehr verfasst. Für weitere Details stehen die Entwickler von UGH, Goobi und der Plugins jederzeit zur Verfügung. Bitte wenden Sie sich an folgende Ansprechpartner:

Ansprechpartner:	**Kontakt:**
Florian Alpers Steffen Hankiewicz Robert Sehr	intranda GmbH Bertha-von-Suttner Str. 9 D – 37085 Göttingen http://www.intranda.com info@intranda.com