kopal Library for Retrieval and Ingest

– DOKUMENTATION –

Stefan Funk, Kadir Karaca Koçer, Sabine Liess, Jens Ludwig, Matthias Neubauer

© Projekt kopal,

Deutsche Nationalbibliothek /

Niedersächsische Staats- und Universitätsbibliothek Göttingen

koLibRI v1.0 – Juli 2007

Inhaltsverzeichnis

1 Überblick über die kopal Library of Retrieval and Ingest **4**
- 1.1 Funktion . . . 4
- 1.2 Projektrahmen . . . 6
- 1.3 Distribution . . . 6

2 Installation und Konfiguration **7**
- 2.1 Voraussetzungen . . . 7
- 2.2 Installation . . . 7
- 2.3 Konfiguration von koLibRI . . . 8

3 Nutzung für den Ingest-Workflow **12**
- 3.1 Übersicht über ProcessStarter und ActionModule . . . 14
- 3.2 Los geht's! – Informationen zu den Beispiel-Konfigurationen . . . 20

4 Nutzung für Retrieval **25**

5 Der Migration Manager **26**
- 5.1 Beschreibung . . . 26
- 5.2 Aufbau . . . 27
- 5.3 Nutzung . . . 28
- 5.4 Klassenbeschreibungen und Erweiterungen . . . 30

6 Die koLibRI-Datenbank **33**
- 6.1 Installation . . . 33
- 6.2 Aufbau des Datenbankschemas . . . 35

7 Eigene koLibRI-Erweiterungen **38**
- 7.1 Die Struktur koLibRIs . . . 38
- 7.2 Konfiguration von Klassen . . . 42
- 7.3 Metadatenformate . . . 43

8 Implementation der DIAS-Administrations- und Suchschnittstelle **44**

9 koLibRI als Web-Service **47**
- 9.1 Einführung . . . 47
- 9.2 Installation . . . 47
- 9.3 Konfiguration . . . 48
- 9.4 Starten des Web-Services . . . 48

9.5 Funktionsweise des Ingests 48
9.6 Funktionsweise des Retrievals 49
9.7 Organisation des Zwischenspeichers 49
9.8 Mögliche Operationen 52
9.9 Weitere Information für Programmierer 53

10 Anhang **55**
10.1 Nutzung von JHOVE in koLibRI 55
10.2 Die WSDL-Datei des koLibRI Web-Services 58
10.3 Direkter Zugriff auf die Schnittstellen des DIAS 73
10.4 Der TIFF Image Metadata Processor 74
10.5 Fehlercodes bei Programmende 75
10.6 Fehlerbehandlung und Loglevel 76

1 Überblick über die kopal Library of Retrieval and Ingest

koLibRI ist ein Framework zur Integration eines Langzeitarchivs wie dem IBM Digital Information Archiving System (DIAS) [1] in die Infrastruktur einer Institution. Insbesondere organisiert koLibRI das Erstellen und Einspielen von Archivpaketen in DIAS und stellt Funktionen zur Verfügung, um diese abzurufen und zu verwalten.

Dieses Dokument beschreibt die Installation und Einstellung eines funktionsfähigen koLibRI-Systems sowie den grundlegenden internen Aufbau, um Eigenentwicklungen zu ermöglichen. Ein gewisses Grundwissen über Langzeitarchivierung [7] sowie über die Standards OAIS [2], URN [3], METS [4], LMER [5] und das DIAS ist für das Verständnis des Systems unvermeidlich. Das nötige Wissen ist jedoch über das Internet frei zugänglich.

koLibRI wurde zum erfolgreichen Ende des Projektes kopal [6] vorgelegt und bietet einen voll funktionsfähigen und stabilen Zustand. Es gelten die auf der kopal Website veröffentlichten Nutzungs- und Haftungsbedingungen[1].

1.1 Funktion

Im einfachen Fall generiert koLibRI aus den mit dem zu archivierenden Objekt gelieferten Metadaten sowie den von JHOVE [8] maschinell extrahierten Metadaten eine XML-Datei nach METS Schema, verpackt diese zusammen mit dem Objekt in einer Archivdatei (.zip oder .tar) und liefert diese Datei als Submission Information Package (SIP) an DIAS.

So gesehen ist koLibRI für eine vollständige Langzeitarchivierungslösung mit DIAS entwickelt worden. Jedoch kann koLibRI auch als eigenständige Software zur Generierung der METS Dateien oder kompletten SIPs nach dem Universellen Objektformat [22] vollkommen ohne DIAS eingesetzt werden. Die auf diese Weise generierten XML-Metadatendateien oder die kompletten SIPs können für den Datenaustausch zwischen verschiedenen Institutionen verwendet werden; ein Aspekt, der bei der Entwicklung des UOF besonders im Vordergrund stand. Alternativ kann durch den modularen Aufbau der koLibRI auch mit vertretbarem Aufwand an ein anderes Archivsystem oder ein anderes Metadatenformat angepasst werden, da die Schnittstellen ausreichend spezifiziert sind.

[1] http://kopal.langzeitarchivierung.de/index_koLibRI.php.de

1.1.1 Ingest

Um ein Archivpaket zu erstellen und einzuspielen, muss definiert werden, welche Arbeitsschritte dafür erforderlich sind. Diese Arbeitsschritte können für verschiedene Arten digitaler Objekte sehr unterschiedlich sein.

Beispielsweise können die Quellen der Objekte (CD-ROM im lokalen Computer, Dokumentenserver im Intranet, etc.) und ihrer deskriptiven Metadaten (dem Objekt beigelegte XML-Datei, OAI-Schnittstelle eines Katalogsystems, etc.) für elektronische Dissertationen, digitalisierte Bücher und auch innerhalb solcher Objektarten stark variieren. Um die verschiedensten Arbeitsabläufe zu integrieren, ist koLibRI modular konzeptioniert worden. Es erlaubt die Notation der Arbeitsabläufe in einer XML-Datei namens `policies.xml`. Die einzelnen Arbeitsschritte werden als *steps* bezeichnet, die durch *action modules* implementiert sind.

Typische Module übernehmen Aufgaben wie das Herunterladen der Dateien für das Archivpaket, das Validieren der Dateien, Extrahieren technischer Metadaten, Generieren der zum UOF gehörenden Metadatendatei und Einspielen des fertigen Archivpakets in DIAS. Auch die Aufteilung der Arbeitsabläufe auf verschiedene Rechner ist möglich, um z.B. in verschiedenen Abteilungen Archivpakete erstellen zu lassen, aber eine zentrale Instanz zur Qualitätskontrolle, zur Pflege von Nachweissystemen und zum Einspeisen zu benutzen.

1.1.2 koLibRI-Datenbank

Für viele Nutzungsszenarien empfiehlt es sich, im Ingest-Arbeitsablauf Daten über die behandelten Objekte zu erheben und zu speichern. Zu diesem Zweck verfügt koLibRI über eine Datenbankunterstützung, so dass z.B. eigene Identifier, Dublin Core Metadaten oder einige technische Metadaten auf Dateiebene abgespeichert werden können. Auf diese Weise stehen Daten für Funktionalitäten außerhalb koLibRIs zur Verfügung, wie z.B. Statistiken, Identifier-Resolving oder eine OAI-Schnittstelle.

1.1.3 Retrieval

Die Funktionalität zum Abrufen von Archivpaketen ist zum einen über einen Web-Service (siehe Kapitel 9) nutzbar. Zum anderen ist die Nutzung des Kommandozeilentools `DiasAccess` bzw. die Nutzung der Java-Methoden der gleichnamigen Klasse für eigene Programme möglich.

1.2 Projektrahmen

koLibRI ist im Rahmen des Projekts kopal – Kooperativer Aufbau eines Langzeitarchivs digitaler Informationen [6] von der Deutschen Nationalbibliothek und der Niedersächsischen Staats- und Universitätsbibliothek Göttingen für die Benutzung des Systems *kopal-Solution* entwickelt worden. Inhalt des dreijährigen Projekts kopal (2004 - 2007) war die praktische Erprobung und Implementierung eines kooperativ erstellten und betriebenen Langzeitarchivierungssystems für digitale Publikationen.

Als Verbundpartner haben die Deutsche Nationalbibliothek [18], die Niedersächsische Staats- und Universitätsbibliothek Göttingen [19] und die IBM Deutschland GmbH [20] in diesem Vorhaben eine nachnutzbare Lösung für die Langzeiterhaltung digitaler Ressourcen erstellt. Der technische Systembetrieb erfolgt durch die Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen [21].

1.3 Distribution

koLibRI kann unter

http://kopal.langzeitarchivierung.de/kolibri

sowohl vorkompiliert als JAR-Datei und als JAVA Quellcode kostenfrei und ohne Registrierung heruntergeladen werden. Bei Problemen, Fragen und gerne auch für Rückmeldungen jeglicher Art kann eine E-Mail an die Adresse

kolibri@kopal.langzeitarchivierung.de

geschickt werden.

Es steht jedem Interessenten frei, die Software beliebig zu modifizieren, weiter zu entwickeln und weiter zu geben. Es müssen jedoch die mitgelieferten Lizenzbedingungen beachtet werden. Die Software-Bibliotheken und Dienstprogramme, welche außerhalb des kopal-Projekts entwickelt worden sind und von koLibRI benutzt werden, unterliegen teilweise anderen Lizenzbedingungen, die jeweils in eigenen Lizenzdateien zu finden sind. Fragen zur Verwendung dieser Software sind ausschliesslich an die Autoren der entsprechenden Softwarepakete zu richten.

2 Installation und Konfiguration

2.1 Voraussetzungen

Da die kopal Library for Retrieval and Ingest komplett in Java implementiert wurde, sollte die Software auf jeder Plattform laufen, die eine Java Virtual Machine in der Version 1.5 zur Verfügung stellt. Allerdings sind auch verschiedene Java-Umgebungen nicht immer kompatibel, vereinzelt wurden Probleme mit unterschiedlichen XML-Parser-Implementationen beobachtet. Das Entwicklerteam hat koLibRI auf verschiedenen Prozessorarchitekturen und Betriebsystemen entwickelt und getestet[2].

koLibRI ist eine freie Software im Sinne der Free Software Foundation [9]. Es kann ohne weiteres eine Portierung auf eine eigene bevorzugte Plattform realisiert werden. Das kopal-Team würde sich in einem solchen Fall über eine Benachrichtigung über die Anpassung sehr freuen.

2.2 Installation

Für die vorkompilierte Distribution besteht die Installation aus den folgenden Komponenten:

1. Dem vorkompilierten Programmpaket `kolibri.jar`,
2. dem Verzeichnis mit eingebundenen Softwarepaketen `/lib` und
3. den Beispiel- und Konfigurationsdateien im Verzeichnis `/config`.

Dazu kommen mehrere Batchdateien für MS Windows und Unix zur einfachen Ausführung der Programmteile. Diese brauchen lediglich aus dem gepackten Archiv in ein bevorzugtes Verzeichnis entpackt zu werden. Falls koLibRI bei Bedarf selbst aus den Quelldateien kompiliert werden soll, müssen die gepackten `.java` Dateien entpackt und – am besten mit Hilfe des mitgelieferten Ant-Skripts – kompiliert werden.

Nachdem `kolibri.jar` auf der Festplatte zur Verfügung steht, muss die Batchdatei `workflowtool.bat` bzw. dessen Pendant für Unix-Systeme `workflowtool`, siehe Abbildung 1, mit Hilfe eines Texteditors angepasst werden. Die Dateien `diasingest.bat` und `dias-access.bat` müssen ebenfalls für einen Zugriff auf das Archivsystem DIAS an lokale Pfade angepasst werden. Bedenken Sie bitte, dass Sie evtl. keinen Zugriff auf das DIAS System

[2] Genutzt und getestet wurde koLibRI mit verschiedenen Linux-Distributionen, unter Mac OS X 10.4 und unter MS Windows XP

des Projektes kopal haben werden. Wenn alle Pfade und Parameter korrekt sind, können die verschiedenen Funktionalitäten des Programmes anhand dieser Batchdateien bequem gestartet werden.

Da hierfür Pfade und Parameter angegeben werden müssen, sollte wenn nötig der Systemadministrator für der richtigen Einträge hinzugezogen werden. Es wird empfohlen, über die Parameter – falls möglich – 512 Megabyte oder mehr für den Heapspeicher zu reservieren, da koLibRI viel mit komplexen XML-Strukturen arbeitet und diese unter Umständen viel Speicher benötigen.

In Abbildung 1 sind exemplarisch die wichtigsten Abschnitte der `workflowtool` Datei gelistet. Angepasst werden sollten folgende Zeilen:

- `KOLIBRI_HOME` ist das Verzeichnis, in dem sich die `kolibri.jar` Datei befindet.
- `JAVA_HOME` ist das Verzeichnis der lokalen Java Installation.
- `JAVA` ist das Verzeichnis innerhalb von `JAVA_HOME`, das die ausführbare Java Virtual Machine beinhaltet.
- `EXTRA_JARS` sind vollständige Pfade und Dateinamen zu gegebenenfalls einzubindenden zusätzlichen Programmbibliotheken, die durch ein Semikolon (MS Windows) oder einen Doppelpunkt (Unix/Linux) getrennt werden müssen. Dies können beispielsweise institutionsspezifische Erweiterungen von koLibRI sein. Hier sind alle von koLibRI genutzten und mitgelieferten JAR-Dateien bereits eingetragen.
- Wenn Sie den der Java Virtual Machine zugewiesenen Speicher erhöhen wollen, ersetzen Sie bitte das `-Xmx512` durch beispielsweise `-Xmx768`, `-Xmx1024` oder auch mehr, je nach verfügbarem Hauptspeicher.

2.3 Konfiguration von koLibRI

Die für die Konfiguration von koLibRI anzupassenden Dateien sind die allgemeine Konfigurationsdatei `config.xml` (beim Start von koLibRI kann über den Kommandozeilenparameter `-c` ein alternativer Dateiname bzw. Pfad angegeben werden) und die Policy-Datei `policies.xml`.

In der Konfigurationsdatei befinden sich globale und modul-/klassenspezifische Konfigurationsvariablen, in der Policy-Datei können für einzelne Arbeitsabläufe spezifische Konfigurationsvariablen gesetzt werden. Zur Auswahl der jeweils passenden Variablen wird zuerst

```
#!/bin/sh

##############################################################################
# Copyright 2005-2007 by Project kopal
#
# http://kopal.langzeitarchivierung.de/
#

 ...
 ...

# Configuration constants:

KOLIBRI_HOME=/users/user/projects/kopal/kolibri  # The koLibRI location
JAVA_HOME=/usr/java                              # Java JRE directory
JAVA=$JAVA_HOME/bin/java                         # Java interpreter
EXTRA_JARS=lib/clibwrapper_jiio.jar: ... ...

# NOTE: Nothing below this line should be edited
##############################################################################

CP=${KOLIBRI_HOME}/kolibri.jar:${EXTRA_JARS}

# Retrieve a copy of all command line arguments to pass to the application.

ARGS=""
for ARG do
    ARGS="$ARGS $ARG"
done

# Set the CLASSPATH and invoke the Java loader.
${JAVA} -classpath $CP -Xmx512m de.langzeitarchivierung.kopal.WorkflowTool $ARGS

##############################################################################
```

Abbildung 1: Die Datei `workflowtool`

versucht, einen für den gerade laufenden Arbeitsschritt definierten Konfigurationswert zu finden, dann einen für das Modul bzw. die Klasse und schließlich wird auf globale Konfigurationsvariablen zurückgegriffen.

Die Konfigurationsdatei hat folgende Struktur (für formale Spezifikation siehe die Datei `config.xsd`):

```
<config>
  <common>
    <property>
      <field>logLevel</field>
      <value>INFO</value>
      <description>...</description>
    </property>
  </common>

  <modules>
    <class name="actionmodule.MetadataGenerator">
      <property>
        <field>acceptedUnknownFileFormats</field>
        <value>.xms</value>
        <value>.doc</value>
        <description>...</description>
      </property>
    </class>
  </modules>
</config>
```

Der `common` Block definiert die globalen Variablen, der `modules` Block die modul- und klassenspezifischen Variablen. Jede einzelne Konfigurationsvariable ist in `property` Tags eingebettet, der `field` Tag definiert den Namen (Groß-/Kleinschreibung ist hier irrelevant) und in einem oder mehreren `value` Tags wird der Wert festgesetzt. Die Wiederholbarkeit eines `value` Tags hängt von der jeweiligen Variable ab, deren Bedeutung in den `description` Tags dokumentiert ist. Der `class` Tag bündelt alle Konfigurationsvariablen für ein bestimmtes Modul, das `name` Attribut gibt dabei den Klassennamen des entsprechenden Javamoduls an (entweder mit vollem Paketnamen oder ohne das übliche Präfix `de.langzeitarchivierung.kopal.`).

Die Angabe von Konfigurationswerten für einzelne Arbeitsschritte in der Policy-Datei geschieht folgendermaßen (für formale Spezifikation siehe die Datei `policies.xsd`, für die Erklärung der Policy-Datei siehe Kapitel 3, Seite 12):

```
<step class="MetadataGenerator">
  <config>
    <property>
      <field>showPdfPages</field>
      <value>false</value>
    </property>
  </config>
  ...
</step>
```

Wenn sich eine Hierarchieebene unterhalb eines `step` Tags ein `config` Tag befindet, werden alle darin sich befindenen Konfigurationsvariablen auf die Klasse des entsprechenden `step` Tags bezogen. Näheres zu den Konfigurationsvariablen siehe Kapitel 3.2, Seite 20. Für eine individuelle Konfiguration ist es sinnvoll, von den mitgelieferten Beispielen auszugehen.

3 Nutzung für den Ingest-Workflow

Um koLibRI für das Einspeisen von Archivpaketen zu nutzen, ist die Definition der Arbeitsabläufe[3] in der Konfigurationsdatei `policies.xml` wesentlich. Die formale Syntax ist in der Beispieldatei `policies.xsd` im `config` Verzeichnis definiert, die Funktionsweise der Beispiel-Arbeitsabläufe ist zusammen mit den Beispielkonfigurationsdateien in Kapitel 3.2 auf Seite 20 erläutert.

Arbeitsabläufe werden als Bäume im Sinne der Graphentheorie betrachtet, die von der Wurzel aus zu den Blättern abgearbeitet werden. Jeder Knoten ist ein Arbeitsschritt, dessen Kinderknoten nur ausgeführt werden, wenn der Elternknoten erfolgreich durchgeführt wurde. Die Kinderknoten werden ggf. parallel ausgeführt, was für zeitaufwändige Nebenaufgaben (z.B. zusätzliches Übertragen der Daten in ein weiteres Archiv oder Brennen auf CD) sinnvoll sein kann[4].

Ein beispielhafter Arbeitsablauf würde folgende sechs Schritte nacheinander durchführen:

1. Ausgewählte Dateien in ein Bearbeitungsverzeichnis kopieren (ActionModule `FileCopyBase`),
2. deskriptive Metadaten zu den Dateien beschaffen (ActionModule `MetadataExtractorDmd`),
3. technische Metadaten aus den vorhandenen Dateien extrahieren (ActionModule `MetadataGenerator`, siehe Kapitel 10.1, Seite 55),
4. die Metadatendatei `mets.xml` generieren (ActionModule `MetsBuilder`),
5. alle Dateien zu einem Archivpaket komprimieren (ActionModule `Zip`) und schließlich
6. das Archivpaket in DIAS (oder ein anderes Archivsystem) einspielen (ActionModule `SubmitDummySipToArchive`).

Die zugehörige Policies-Datei sieht aus wie folgt:

[3] Welche Arbeitsschritte notwendig sind, hängt auch von den eigenen Ansprüchen an die Archivierung ab. Diese Ansprüche müssen Institutionen für sich selbst klären; der Prozess, sich eine Langzeitarchivierungs-Policy zu geben, kann durch keine technische Lösung ersetzt werden. Für entsprechende Informationen verweisen wir auf das Partnerprojekt nestor [7].

[4] Diese Funktionalität ist in koLibRI 1.0 leider noch nicht realisiert.

```
<policies>
  <policy name="example">
    <step class="FileCopyBase">
      <step class="MetadataExtractorDmd">
        <step class="MetadataGenerator">
          <config>
            <property>
              <field>showHtmlImages</field>
              <value>true</value>
              <description>...</description>
            </property>
          </config>
          <step class="MetsBuilder">
            <step class="Zip">
              <step class="SubmitDummySipToArchive">
              </step>
            </step>
          </step>
        </step>
      </step>
    </step>
  </policy>
</policies>
```

Neben den Arbeitsabläufen selbst muss auch festgelegt werden, wie neue Arbeitsabläufe angestoßen werden und welche Startwerte ihnen mitgegeben werden. Dies ist die Aufgabe der sogenannten *process starter*, die beim Start durch den Kommandozeilenparameter `-p` oder in der Konfigurationsdatei mit der Variablen `defaultProcessStarter` festgelegt werden. Einfache Fälle wären die ProcessStarter `MonitorHotfolderExample` und `MonitorHttpLocationExample`, die entweder aus allen Dateien und Unterverzeichnissen eines angegebenen Verzeichnisses oder einer URL ein Archivpaket erstellen, ein anspruchsvollerer wäre ein Server (`Server` und `ClientLoaderServerThread`), durch den ein Rechner darauf wartet, von anderen Rechnern Archivpakete anzunehmen, um sie weiterzubehandeln. Es ist auch möglich, mehrere ProcessStarter gleichzeitig laufen zu lassen, indem für die Konfigurationsvariable `defaultProcessStarter` mehrere Werte angegeben werden.

Start einer WorkflowTool-Instanz

Um eine WorkflowTool-Instanz nach der korrekten Installation zu starten, wird eines der dem Release beiliegenden Batchfiles verwendet. Hierzu müssen die Batchfiles lediglich mit

den jeweiligen lokalen Konfigurationen ergänzt werden. Zusätzlich können über diese Batchfiles optionale Modulpakete anderer Institutionen eingebunden und verwendet werden (siehe Kapitel 2, Seite 7).

`workflowtool -h` erklärt die Kommandozeilenoptionen:

```
usage: WorkflowTool
    -c, --config-file        The config file to use.
    -s, --show-properties    Prints the system properties and continues.
    -p, --process-starter    The process starter module which chooses items for
                               processing.
    -h, --help               Prints this dialog.
```

Optional zum Aufruf der Batchfiles kann eine WorkflowTool-Instanz auch über `java -jar kolibri.jar [OPTIONS]` gestartet werden.

Da der Zugang zu DIAS üblicherweise über eine verschlüsselte Verbindung erfolgt, wird nach der Installation des Zertifikats der DIAS-Hosting-Partners mit dem Java keytool der Parameter `-Djavax.net.ssl.trustStore=KEYSTORE_LOCATION` notwendig. Der Pfad zur Keystore-Datei und zur von SSH benötigten `known_hosts` Datei wird in der Konfigurationsdatei konfiguriert.

3.1 Übersicht über ProcessStarter und ActionModule

3.1.1 ProcessStarter

`Server` Die koLibRI-Instanz wartet auf Netzwerkverbindungen und startet für jede Anfrage einen Thread, der die relevanten Aktionen durchführt. In der Konfigurationsdatei muss dieser `ServerThread` als `serverClassName` angegeben werden, wie zum Beispiel `ClientLoaderServerThread` (nimmt Archivpakete von anderen koLibRI-Instanzen an). Für Beispielwerte siehe `example_config_clientloader.xml`.

Wichtige Konfigurationswerte:

- serverClassName
- defaultConnectionPort

`MonitorHotfolderExample` Beispiel-ProcessStarter, der für jedes Unterverzeichnis eines angegebenen Verzeichnisses ein Archivpaket – ein SIP – baut. Er beobachtet das unter `hotfolderDir` angegebene Verzeichnis und bearbeitet zunächst alle vorhandenen Verzeichnisse und später alle hinzukommenden. Die Namen der Unterverzeichnisse werden in dieser Beispielklasse als persistente Identifier der SIPs interpretiert.

`MonitorHotfolderExample` erbt von `MonitorHotfolderBase`, weswegen die Werte der zuletzt genannten Klasse konfiguriert werden.

Wichtige Konfigurationswerte:

- hotfolderDir
- readDirectoriesOnly
- startingOffset
- checkingOffset
- addingOffset

`MonitorHttpLocationExample` Wie `MonitorHotfolderExample`, nur dass die Dateien und Verzeichnisse von einer URL geladen werden können. Auch diese wird beobachtet und neu hinzugekommende Verzeichnisse werden bearbeitet. Als persistente Identifier werden wie oben die Verzeichnisnamen genutzt. Hier werden die Werte von `MonitorHttpLocationBaese` konfiguriert.

Wichtige Konfigurationswerte:

- urlToMonitor
- readDirectoriesOnly
- startingOffset
- checkingOffset
- addingOffset

3.1.2 ActionModule

`AdaptHtmlPage` Tauscht Links innerhalb einer HTML-Datei gegen andere Links aus. Dieses ActionModule wird vom `MigrationManager` benutzt, um Dateinamen gegen ihre migrierten und evtl. veränderten Dateinamen auszutauschen, damit eine HTML-Seite nach der Migration noch korrekt angezeigt wird.

Wichtige Konfigurationswerte:

- htmlPageFilename
- replaceFulltextOnly

`AddDataToDb` Fügt der Datenbank konfigurierbar verschiedene Informationen hinzu, wenn `useDatabase` in der Konfigurationsdatei auf `TRUE` gesetzt ist.

Wichtige Konfigurationswerte:

- storeFileData

- storeFileDataTechMd
- storeDc
- storeIds
- storeCustomData
- customDataEelements

`AlreadyIngestedChecker` Testet anhand der XOR-Checksumme in der koLibRI-Datenbank, ob ein SIP bereits in das DIAS eingespielt wurde.

Wichtige Konfigurationswerte:

- waitUntilNextTry
- errorIfChecksumExists

`AskDIASifIngested` Testet per Zugriff auf das DIAS anhand einer externen ID, ob ein SIP mit eben dieser externen ID bereits in das DIAS eingespielt wurde.

`CleanPathToContentFiles` Die Verzeichnisse und Dateien in dem zur Bearbeitung genutzten temporären Verzeichnis werden gelöscht.

`CleanUpFiles` Die erzeugten ZIP-Files und evtl. entpackte METS-Dateien werden aus dem Destination-Verzeichnis entfernt.

`DiasIngestFeedback` Nutzt das Ticketing-System des SIP-Interfaces [10], um den Status eines DIAS Ingest-Vorgangs abzufragen. Je nach Konfiguration können alle bis dahin in die koLibRI-Datenbank eingetragenen Daten bei einem nicht erfolgreichen Ingestversuch wieder entfernt werden.

Wichtige Konfigurationswerte:

- removeFileOnError
- removeTechmdOnError
- removeDublincoreOnError
- removeIdentifierOnError
- removeVariousOnError
- removeDiasOnError

`Executor` Mit Hilfe des Executors können systemeigene Kommandozeilenbefehle ausgeführt werden. Die statischen Argumente werden in der Konfigurationsdatei angegeben.

Wichtiger Konfigurationswert:

- command

`FileCopyBase` Es werden Verzeichnisse und beinhaltete Dateien in das temporäre Verzeichnis kopiert, um hier die Dateien zu bearbeiten. Dieses ActionModul kann mit eigenen Klassen erweitert werden, um z.B. während des Kopierens die Schreibweise der Dateinamen zu korrigieren oder um bestimmte Dateien vor der Archivpaketerstellung zu entfernen. Außerdem bleiben die Originaldateien auf diese Weise unverändert.

`FileCopyHttpBase` Kopiert Verzeichnisse und Dateien per HTTP. So können auch Dateien von einem Web-Server archiviert werden, ansonsten siehe `FileCopyBase`.

`MetadataExtractorDmd` Bindet deskriptive Metadaten in das Archivpaket ein. Dieses Modul muss den eigenen Bedürfnissen angepasst werden. In unserem Beispiel wird lediglich aufgezeigt, wie die Daten in das SIP eingebunden werden. Die Herkunft der deskriptiven Metadaten ist abhängig von den Strukturen einzelner Institutionen. Erbt von `MetadataExtractorBase`.

`MetadataExtractorDigiprovmd` Bindet Provenance-Metadaten (Herkunfts-Metadaten) in das Archivpaket ein. Auch dieses Modul muss den Bedürfnissen der eigenen Institution angepasst werden. Erbt ebenfalls von `MetadataExtractorBase`.

`MetadataGenerator` Generiert technische Metadaten zu allen vorhandenen Content-Files des Archivpakets unter Nutzung des JHOVE-Toolkits der Harvard-Universität (siehe Kapitel 10.1, Seite 55).

Wichtige Konfigurationswerte:

- acceptedUnknownFileFormat
- showGifGlobalColorTable
- showPdfPages
- showPdfImages
- showPdfOutlines
- showPdfFonts
- showMixColorMap
- showTiffPhotoshopProperties
- showWaveAESAudioMetadata
- showHtmlLinks
- showHtmlImages
- showIso9660FileStructure
- jhoveIsVerbose

- errorIfSigmatchDiffers

`MetsBuilder` Erstellt aus allen gesammelten Informationen die METS-Datei des Archivpakets. Verantwortlich hierfür ist die in der Konfigurationsdatei unter `mdClassName` angegebene Klasse. Für kopal erzeugt die Implementation `Uof.java` des Interfaces `MetadataFormat` eine METS-Datei im Universellen Objektformat (UOF) des kopal-Projekts.

`MigrateFiles` Wird vom MigrationMagager genutzt, um einzelne Dateien in ein anderes – normalerweise aktuelleres – Dateiformat zu migrieren.

Wichtige Konfigurationswerte:

- migrationToolExecutionCommand
- sourceFileName
- destFileName

`PrepareMigration` Dieses Modul erweitert das UOF um die benötigten Herkunfts-Metadaten für eine Migration.

Wichtige Konfigurationswerte:

- migrateMetadataRecordCreator
- provmdComments
- provmdPermission
- provmdCreator
- provmdPurpose
- provmdResult
- provmdSteps

`SubmitDummySipToArchive` Beispiel-Modul zur Demonstration.

`SubmitSipToClientLoader` Übermittelt ein SIP an einen ClientLoader. Dieses Modul kann von mehreren AssetBuildern gleichzeitig genutzt werden, die ihre Archivpakete an eine zentrale WorkflowTool-Instanz senden. Näheres hierzu siehe unter Informationen zur Beispiel-Konfiguration.

Wichtige Konfigurationswerte:

- clientLoaderServer
- clientLoaderPort
- submitPolicyName

- fileSubmitNotificationTime

`SubmitSipToDias` Übermittelt ein SIP an das von kopal genutzte DIAS und ist die Implementation der von IBM bereitgestellten SIP-Spezifikation [10] zum Einspielen in kopal-DIAS. Einige der Konfigurationswerte werden im globalen Teil der Konfigurationsdatei eingestellt.

Wichtige Konfigurationswerte:

- cmUser
- cmPwd
- ingestPort
- ingestServer
- knownHostsFile
- preloadArea

`TiffImageMetadataProcessor` Validiert zunächst alle im Archivpaket vorhandenen TIFF Bilddateien mit Hilfe von JHOVE. Einige Fehler in den TIFF-Header-Metadaten können automatisch korrigiert werden. Siehe Kapitel 10.4, Seite 74.

Wichtige Konfigurationswerte:

- correctInvalidTiffHeaders
- verbose
- separateLogging
- separateLogfileDir
- jhoveConfigFile
- nonVerboseReportingTime

`Unzip` Entpackt eine ZIP-Datei in das angegebene Verzeichnis.

Wichtige Konfigurationswerte:

- tempDir
- sourceFile
- deleteZipOnExit

`Utf8Controller` Mit Hilfe dieses Moduls kann die erstellte METS-Datei, die immer im UTF-8-Format vorliegt, auf ungültige UTF-8 Zeichen überprüft werden und kann diese auch korrigieren. Die Quelle des genutzten Utf8FilterInputStreams ist der utf8-conditioner von Simeon Warner (`simeon@cs.cornell.edu`), der für die Nutzung in koLibRI nach JAVA portiert wurde.

Wichtige Konfigurationswerte:

- filename
- storeOriginalFile
- originalPostfix

`XorFileChecksums` Erstellt eine XOR-Checksumme aus allen vorhandenen Datei-Checksummen und fügt sie in die Datenbank ein. Diese XOR-Checksumme wird später genutzt, um festzustellen, ob ein SIP bereits in ein Archiv eingespielt wurde.

Wichtiger Konfigurationswert:

- errorIfAlreadyIngested

`Zip` Komprimiert die Content-Files inclusive der METS-Datei (mets.xml) zu einem handlichen Paket. Wichtige Konfigurationswerte:

Wichtiger Konfigurationswert:

- compressionLevel

Weitere, teilweise sehr spezielle, ActionModule und ProcessStarter sind von den kopal-Projektpartnern Deutsche Nationalbibliothek (DNB) und Niedersächsische Staats- und Universitätsbibliothek Göttingen (SUB) entwickelt worden und evtl. auf Anfrage verfügbar.

3.2 Los geht's! – Informationen zu den Beispiel-Konfigurationen

Es gibt drei Möglichkeiten, die Konfigurations-Beispieldateien zu nutzen.

Es kann zum einen für den Standalone Betrieb ein einzelner Rechner so konfiguriert werden, dass dieser die Archivpakete sowohl erstellt als auch an ein Archivsystem übermittelt – eine Kombination aus Archivpaket-Erstellung und Archiv-Transfer mit einer einzelnen Instanz der Klasse `WorkflowTool`.

Zum anderen besteht die Möglichkeit, mehrere Rechner so zu konfigurieren, dass diese als sogenannte `AssetBuilder` fungieren und jeweils aus verschiedenen Workflows resultierende Archivpakete erstellen. Diese Pakete werden an einen zentralen Rechner weitergeleitet. Dieser zentrale Rechner, der sogenannte `ClientLoader`, nimmt alle Archivpakete an und leitet diese Pakete an ein Archivsystem weiter – eine Trennung von Archivpaket-Erstellung und Archiv-Transfer mit einer Instanz des WorkflowTools als ClientLoader und mehreren Instanzen als AssetBuilder.

Als dritte Möglichkeit kann eine Datenbank zur Buchführung über die eingespielten und erstellten Archivpakete genutzt werden. Dies wurde bisher nur für eine einzelne Instanz des WorkflowTools im Standalone-Betrieb ausgiebig getestet, siehe oben.

VORSICHT: Die Nutzung der Datenbank ist bis jetzt nicht für die gleichzeitige Nutzung mehrerer WorkflowTools als Standalone-AssetBuilder getestet. Eine Nutzung mit mehreren unabhängigen Instanzen des WorkflowTools im Standalone-Betrieb ist zur Zeit theoretisch möglich, aber geschieht auf eigene Gefahr. Eine Nutzung der Datenbank durch einen Client-Loader ist derzeit nicht möglich.

3.2.1 Standalone-Nutzung – Archivpaket-Erstellung und Archiv-Transfer mit einer einzigen Instanz des WorkflowTools

Die Datei `example_config_standalone.xml` wird mit nur einer Instanz der Klasse WorkflowTool genutzt. Zum Ausführen des Beispiels müssen lediglich alle von den drei Sternchen (***) umschlossenen Tag-Inhalte sinvoll ausgefüllt werden. Dies sind vier Pfadangaben zu Arbeits- und Temporärverzeichnissen, je nach ProcessStarter eine URL oder eine weitere Pfadangabe und evtl. einen Pfad für separate Logfiles. Die `description` Tags in der Konfigurationsdatei geben Hinweise zur Bedeutung der Werte.

Es kann zur Zeit aus zwei ProcessStartern ausgewählt werden. `MonitorHotfolderExample` nimmt alle Unterverzeichnisse aus dem Verzeichnis, das unter `hotfolderDir` angegeben ist und behandelt im weiteren jedes Unterverzeichnis als ein einzuspielendes Archivpaket, indem diese Verzeichnisse der `ProcessQueue` hinzugefügt und bearbeitet werden. `MonitorHttpLocationExample` bearbeitet alle Unterverzeichnisse, die unter der URL `urlToMonitor` vorhanden sind und geht ebenso vor. Selbstverständlich müssen diese Unterverzeichnisse existieren und Dateien hinterlegt sein, aus denen JHOVE technische Metadaten extrahieren kann, um aussagekräftige und nutzbare Ergebnisse zu erhalten. Es können beliebige Dateistrukturen mit beliebigen Dateien hinterlegt werden, um dieses Modul zu testen. Näheres hierzu im Anhang in Kapitel 10.1, Seite 55.

Die hier genutzte Policy aus der Policy-Konfigurationsdatei `policies.xml` heisst `example_standalone` und führt folgende Actionmodule für jedes Archivpaket nacheinander aus:

- XorFileChecksums
- TiffImageMetadataProcessor
- MetadataExtractorDmd

- MetadataGenerator
- MetadataExtractorDigiprovmd
- MetsBuilder
- Utf8Controller
- Zip
- AlreadyIngestedChecker
- AddDataToDb
- SubmitDummySipToArchive
- CleanPathToContentFiles
- CleanUpFiles

Als ProcessStarter werden je nach Konfigurationsdatei entweder `FileCopyBase` oder `FileCopyHttpBase` gestartet. Einzelheiten zu den einzelnen Modulen können der Dokumentation der Java-Klassen sowie der `description` Tags der Konfigurationsdatei entnommen werden, siehe auch Kapitel 3.1, Seite 14.

Der Kommandozeilenbefehl zum Starten des Standalone-Beispiels lautet:

```
java -jar kolibri.jar -c config/example_config_standalone.xml
```

oder nach der Konfiguration der Skripte `workflowtool` (oder `workflowtool.bat`) mit dort zusätzlich gesetzten Werten:

```
workflowtool -c config/example_config_standalone.xml
```

3.2.2 Nutzung der ClientLoader/AssetBuilder-Kombination – Trennung von Archivpaket-Erstellung und Archiv-Transfer mit zwei oder mehr Instanzen des WorkflowTools

Die Datei `example_config_clientloader.xml` wird mit einer Instanz des WorkflowTools genutzt, die Datei `example_config_assetBuilder.xml` kann für jeweils eine Instanz des WorkflowTools als AssetBuilder genutzt werden. Es werden in den Konfigurationsdateien einige weitere Informationen zur Kommunikation zwischen den Instanzen benötigt. Die Arbeitsteilung wird im Folgenden beschrieben.

Die sogenannten AssetBuilder können auf verschiedenen Rechnern gestartet werden. So kann jeder Rechner einen anderen Workflow zur Archivpaket-Erstellung abarbeiten. Für unterschiedliche Workflows müssen dann sicherlich für jeden Workflow eigene Policies und eigene Konfigurationsdateien erstellt werden. Die AssetBuilder führen im vorliegenden Beispiel

zunächst einmal genau die Arbeitsschritte des ersten Beispiels aus, als ProcessStarter können ebenfalls beide oben genannten genutzt werden. Der Unterschied ist der, dass die fertig erstellten Archivpakete nicht an ein Archiv, sondern zunächst an den ClientLoader übermittelt werden. Hierfür nutzen die AssetBuilder das ActionModule `SubmitSipToClientLoader`. Die Policy lautet `example_assetBuilder`.

Als ClientLoader wird eine WorkflowTool-Instanz mit dem ProcessStarter `Server` gestartet. Dieser wartet auf eine Anfrage auf einem definierten Port, dem `defaultConnectionPort` in der Konfigurationsdatei. Bekommt dieser Server eine Anfrage von einem AssetBuilder, startet der Server einen ClientLoaderServerThread, konfiguriert mit `serverClassName`. Diese Threads warten auf die Übermittlung eines SIPs vom entsprechenden AssetBuilder. Es können mehrere Archivpakete von verschiedenen AssetBuildern gleichzeitig angenommen werden.

Die Policy lautet hier `example_clientloader` und enthält zwei ActionModule:

`SubmitDummySipToArchive` Dies ist dasselbe ActionModule wie im ersten Beispiel. Die Trennung in AssetBuilder und ClientLoader ist dann sinnvoll, wenn mehrere AssetBuilder eingesetzt werden, um beispielsweise unterschiedliche Workflows zeitgleich abzubilden und Archivpakete von verschiedensten Quellen zu erzeugen. Der ClientLoader sammelt dann zentral alle Archivpakete von allen AssetBuildern, kann evtl. noch verschiedene Aktionen für alle diese Pakete ausführen (zum Beispiel CDs von den Archivpaketen brennen), die METS-Dateien nochmals validieren oder auch andere Institutionen über neu eingespielte SIPs unterrichten.

`CleanUpFiles` Es werden nach erfolgreichem Ingest in das Archivsystem die von den AssetBuildern gelieferten gepackten Archivpakete gelöscht. Im Beispiel auskommentiert zur besseren Nachvollziehbarkeit.

Der Kommandozeilenbefehl zum Starten des AssetBuilder/ClientLoader-Beispiels lautet wie folgt:

1. Starten des ClientLoaders:
   ```
   java -jar kolibri.jar -c config/example_config_clientloader.xml
   ```
2. Starten eines AssetBuilders:
   ```
   java -jar kolibri.jar -c config/example_config_assetbuilder.xml
   ```

3. Starten beliebiger weiterer AssetBuilder (evtl. mit unterschiedlichen Konfigurationsdateien):
   `java -jar kolibri.jar -c config/example_config_assetbuilder.xml`

Oder wie gehabt nach der Konfiguration der Skripte `workflowtool` (oder `workflowtool.bat`) mit dort zusätzlich gesetzten Werten:

1. Starten des ClientLoaders:
   `workflowtool -c config/example_config_clientloader.xml`
2. Starten eines AssetBuilders:
   `workflowtool -c config/example_config_assetbuilder.xml`
3. Starten beliebiger weiterer AssetBuilder (evtl. mit unterschiedlichen Konfigurationsdateien):
   `workflowtool -c config/example_config_assetbuilder.xml`

3.2.3 Standalone-Nutzung mit Datenbank – Archivpaket-Erstellung und Archiv-Transfer mit einer Instanz des WorkflowTools und Nutzung der koLibRI-Datenbank

Zunächst muss eine Datenbank laut Datenbank-Dokumentation erstellt werden (Kapitel 6, Seite 33). Ist dies erfolgreich geschehen, sind noch die folgenden Schritte auszuführen:

1. Das WorkflowTool ist wie in Punkt 3.2.1 beschrieben zu konfigurieren.
2. Den Tag `useDatabase` der Konfigurationsdatei muss auf `TRUE` gesetzt werden.
3. Alle die Datenbank betreffenden Werte müssen mit sinnvollen Inhalten konfiguriert werden. Diese Werte sind in der Konfigurationsdatei mit ### gekennzeichnet.
4. Das WorkflowTool kann nun wie unter Punkt 3.2.1 genannt gestartet werden. Die Datenbank wird während des Einspiel-Vorgangs (Ingest) mit Informationen gefüllt. Die Datenbank-Dokumentation gibt hier weiterführende Informationen zur Nutzung der Datenbank.

Soll die Datenbank nicht genutzt werden, reicht es aus, den Wert von `useDatabase` in der globalen Sektion der Konfigurationsdatei auf `FALSE` zu setzen. Die Module, die die Datenbank nutzen, können auch weiterhin in der Policy-Datei eingetragen bleiben, da alle Datenbank-Module obigen Wert abfragen.

4 Nutzung für Retrieval

Die Funktionalität zum Abrufen von Archivpaketen ist zum einen über Web Service (siehe Kapitel über Web Service 9) nutzbar, zum anderen ist die Nutzung des Kommandozeilentools `DiasAccess` bzw. die Nutzung der Java-Methoden der gleichnamigen Klasse für eigene Programme möglich. Für ein Verständnis der DiasAccess Kommandozeilenoptionen ist die Kenntnis der DIP-Interface-Spezifikation [11] wichtig. Ein Überblick über die Optionen befindet sich in Kapitel 10 auf Seite 55.

5 Der Migration Manager

5.1 Beschreibung

Bei dem MigrationManager handelt es sich um eine koLibRI-Komponente, mit der Migrationen verwaltet und durchgeführt werden können. Die Objekte, die migriert werden sollen, und die, die Endprodukt der Migration sein sollen, können beschrieben werden. In Abhängigkeit von diesen Objekten und den eigenen Anforderungen können verschiedene Migrationsabläufe konfiguriert werden und unterschiedliche Werkzeuge zur Migration genutzt werden.

Der MigrationManager ist ein Prototyp und die vorhandenen Funktionalitäten liegen zwischen dem unten aufgeführten einfachen und komplexen Szenario. Er ist nicht ausreichend getestet, um für den produktiven Massenbetrieb automatisiert eingesetzt zu werden, und nicht alle Möglichkeiten, die durch DIAS, koLibRI und das UOF geboten werden sind implementiert. Für den exemplarischen Testbetrieb und zur Konzeption von Migrationsprozessen kann der MigrationManager aber durchaus als nutzbar betrachtet werden.

Migration selbst ist ein anspruchsvoller Prozess in einem Archivbetrieb, der nicht losgelöst von den anderen Prozessen und Planungen des Archivbetriebs und der zur Verfügung stehenden Infrastruktur betrachtet werden kann. Ein sehr einfaches Szenario, das ohne Weiteres mit der derzeitigen Version realisiert werden kann:

- Über Kommandozeile wird das Quellformat TIFF und das Zielformat JPEG2000 von Dateien in zu migrierenden Objekte angegeben,
- auf einer Menge von lokal vorhandenen METS-Dateien wird nach den entsprechende Objekten gesucht und
- ein einfacher Migrationsablauf bestehend aus dem Herunterladen der betroffenen Archivpakete, Migrieren der Dateien mit einem Kommandozeilenprogramm und dem erneuten Erstellen und Einspielen der Archivpakete wird durchgeführt.

Ein komplexes Szenario, das auch aufgrund fehlender Insitutionen-übergreifender Infrastruktur derzeit nicht vollständig realisiert werden kann, für das aber die grundlegende Architektur und Schnittstellen für weitere Entwicklungen vorhanden sind:

- Durch einen Technologiebeobachtungsdienst oder eine Format Registry (siehe Beispielsweise GDFR [26] oder PRONOM [27]) empfängt die koLibRI-Instanz die Nachricht, dass ein mit einem bestimmten Programm erstelltes Bildformat mit bestimmten technischen Eigenschaften in ein anderes Format überführt werden sollte,

- im DIAS-Datamanagement wird nach Objekten und Dateien mit der entsprechenden Verknüpfung technischer Eigenschaften gesucht,
- da die betroffenen Dateien und Objekte häufig als Vorlagen (z.B. Stylesheets, etc.) von anderen Objekten genutzt werden, müssen mehrere Objekte zusammen migriert und angepasst werden,
- in manchen Dateien sind die betroffenen Bildformate eingebettet,
- ein externes Diensteverzeichnis wird nach einem passenden Migrationsdienst (z.B. ein Grid-Dienst für die Konvertierung großer Datenmengen) für die Migration abgefragt und
- die abhängigen Dateien und Objekte werden angepasst.

5.2 Aufbau

Der MigrationManager selbst ist ein `ProcessStarter`, der aber aus einer Reihe von funktionalen Einheiten besteht:

`MigrationEventListener` Bekommt/Holt Nachrichten über potentiellen Migrationsbedarf. Dies geschieht zur Zeit über die Kommandozeile beim Start von koLibRI, wo eine Charakterisierung der Quell- und Zielobjekte übergeben wird.

`ObjectCharacter` Eine Charakterisierung von Objekten, wie z.B. der Quell- und Zieldateiformate, aber auch komplexerer Eigenschaften wie Erstellungssoftware oder Kodierungen. Da es derzeit keine allgemeinen Standards zur technischen Beschreibung von Objekten gibt, wird als Basis primär das kopal UOF benutzt und die einzelne Eigenschaften werden durch XPath-Ausdrücke definiert. Auch eine Objektcharakterisierung durch CQL [25] für das Dias-Datenmanagement ist möglich.

`ObjectCharacterTranslator` Übersetzt Objektcharakterisierungen in Anfragen an ein Datenmanagementsystem. Für XPath-Objektcharakterisierungen ist dies durch die Abfrage von METS-Dateien im lokalen Dateisystem, die Abfrage der koLibRI-Datenbank oder des Dias-Datenmanagements möglich, für CQL-Objektcharakterisierungen derzeit nur im Dias-Datenmanagement. Es ist zu beachten, dass nicht jedes Datenmanagement alle Abfragen unterstützt. Vollständig im Sinne des kopal UOF ist die Abfrage von lokalen METS-Dateien.

`ObjectDependencyAnalyser` Prüft Abhängigkeiten zwischen Objekten und fasst sie zusammen. Derzeit werden die `mptr` (METS-Pointer) der METS-Dateien analysiert und

alle sich dadurch in einem zusammenhängenden Graphen befindlichen Objekte zusammengefasst. Die Weiterverarbeitung von Objekten mit Abhängigkeiten zu anderen Objekten wird allerdings noch nicht genutzt.

`MigrationPolicyBuilder` Konfiguriert Migrationsabläufe anhand von Vorlagen. Je nach Vorlage werden noch passende Migrationsdienste ausgewählt.

Neben den Konfigurationsmöglichkeiten in der allgemeinen Konfigurationsdatei sind zwei weitere Konfigurationsdateien wichtig:

`migrationPolicyTemplates.xml` In dieser Datei (oder einer entsprechend der allgemeinen Konfiguration benannten Datei) werden die Policy-Vorlagen definiert, nach denen die einzelnen Migration durchgeführt werden.

`migrationServiceRegistry.xml` Hier werden die verfügbaren Migrationsdienste aufgeführt und definiert, wie sie aufgerufen werden können (derzeit sind nur Kommandozeilenprogramme getestet).

Als ActionModule sind für den MigrationManager besonders relevant:

`PrepareMigration` Passt das UOF für eine Migration an. Im Migrationsablauf muss dieses Modul sehr früh bzw vor der eigentlichen Dateikonvertierung aufgerufen werden.

`MigrateFiles` Führt externe Programme für Migrationen aus und passt die Metadaten im UOF entsprechend an. Migrationen sind derzeit nur durch das Aufrufen von Kommandozeilenprogrammen möglich.

`Executor` Führt externe Programme aus.

`AdaptHtmlPage` Passt die Links bzw. Dateireferenzierungen einer HTML-Seite an, wenn sich durch die Migration der Name einer Datei geändert hat.

5.3 Nutzung

Um Migrationen durchzuführen, müssen zuerst eine Reihe von Einstellungen vorgenommen werden und der eigentliche Migrationsablauf durch ein übergebenes Migrationsereignis an koLibRI angestoßen werden.

5.3.1 Konfiguration

1. Welches Datenmanagement soll oder kann genutzt werden? Die entsprechende `ObjCharacterTranslator`-Klasse muss in der Konfigurationsdatei als `ObjChrTranslatorClassName` für die Klasse `processstarter.migrationmanager.MigrationEventScheduler` angegeben werden. Wenn direkter Zugriff auf das DIAS besteht, kann `DiasCQLObjCharacterTranslator` für eine CQL-Charakterisierung oder `DiasObjCharacterTranslator` für eine XPath-Charakterisierung genutzt werden. Alternativ sind `FileObjCharacterTranslator` für im lokalen Dateisystem vorhandene METS-Dateien oder `DbObjCharacterTranslator` für die koLibRI-Datenbank möglich.

2. Für `FileObjCharacterTranslator` ist als `MetsFilesDir` anzugeben, unterhalb welchen Verzeichnisses die METS-Dateien bzw. Unterverzeichnisse mit METS-Dateien liegen.

3. Für die Klasse `processstarter.migrationmanager.MigrationPolicyBuilder` sind als `migrationServicesFileName` und `migrationPolicyTemplatesFileName` die Pfade zu den Konfigurationsdateien anzugeben, die die nutzbaren Migrationsdienste und Migrationsabläufe definieren.

4. Die verwendbaren Migrationsdienste müssen in der als `migrationServicesFileName` angegebenen Konfigurationsdatei definiert werden. Beschreibungen und Beispiele der Konfiguration sind in der Datei `config/example_migrationServiceRegistry.xml` zu finden. Zwei Aspekte sind wesentlich: Zum einen muss beschrieben werden, durch welches Actionmodule und mit welchen Parametern der Dienst aufgerufen werden kann. Zum anderen muss definiert werden, welche Eigenschaften der Dienst hat, damit er für eine bestimmte Migration ausgewählt wird. Dies kann durch den Namen, Klassifikationen oder Angabe von Quell- und Zielobjektcharakterisierungen mit XPath-Ausdrücken (z.B. Quell- und Zielformat) geschehen.

5. Die Vorlagen für Migrationsabläufe müssen in der als `migrationPolicyTemplatesFileName` angegebenen Konfigurationsdatei definiert werden. Beschreibungen und Beispiele der Konfiguration sind in der Datei `config/example_migrationPolicyTemplates.xml` zu finden. Hier ist ebenfalls die Formulierung der Anwendungskriterien einer Migrationsvorlage wesentlich. Die Kriterien werden derzeit ebenfalls als XPath-Objektcharakterisierungen ausgedrückt. Darauf folgt die Definition der bei Erfüllung der Kriterien zu verwendenen Policy-Bestandteile (`policyFragment`). In der Reihenfolge der Ausführung werden feste Policies aus der `policies.xml` (z.B. für konstante

Teile des Ingestprozesses) oder Kriterien für die Auswahl eines im Diensteverzeichnis definierten Migrationsdienstes aufgeführt.

5.3.2 Durchführung der Migration

Wenn die erforderlichen Konfigurationen aufeinander abgestimmt sind, wird der MigrationManager gestartet, indem koLibRI mit der Klasse `MigrationEventScheduler` gestartet wird. Es werden zwei Parameter erwartet, wenn nur ein Quell- und ein Zielformat angegeben werden soll. Dies geschieht durch Angabe der entsprechenden DIAS Format-IDs. Wenn vier Parameter angegeben werden, wird das erste Paar als XPath-Ausdruck und dessen erwartete Ergebniswert für die Quell-Objektcharakterisierung interpretiert, das zweite Paar entsprechend für die Ziel-Objektcharakterisierung. Eine Objektcharakterisierung durch XPath kann in der Langform als

```
/mets/amdSec/techMD/mdWrap/xmldata/format/text()
```

(Namensräume sind hier für bessere Lesbarkeit ausgelassen, müssen aber hinzugefügt werden) oder in der praktikabeleren Form als

```
//*[name()='lmerFile:format']
```

geschehen. Als Ergebnis kann z.B. GIF als

```
urn:diasid:fty:kopal:020050705000000000005
```

angegeben werden. Alle Objekte mit GIF-Dateien werden dann ausgewählt.

5.4 Klassenbeschreibungen und Erweiterungen

- `MigrationEventScheduler`
  Implementiert das `ProcessStarter`-Interface, initialisiert die weiteren Komponenten anhand der Konfiguration und steuert den Gesamtablauf.

- `MigrationEventListener`
  Interface für Klassen, die Ereignisse empfangen, die eine Prüfung des Migrationsbedarfs erfoderlich machen.

- `CommandLineMigrationEventListener`
  Implementiert das `MigrationEventListener`-Interface. Interpretiert Kommandozeilenargumente als entsprechende Ereignisse.

– `ObjCharacter`
  Objektcharakterisierung durch XPath-Ausdrücke. Erlaubt die Nutzung des `FileObjCharacterTranslator`, des `DbObjCharacterTranslator` und des `DiasObjCharacterTranslator`.

– `CQLObjCharacter`
  Objektcharakterisierung durch CQL-Ausdrücke für das DIAS-Datenmanagement. Erfordert die Nutzung des `DiasCQLObjCharacterTranslator`.

– `ObjCharacterTranslator`
  Interface für die Übersetzung von Objektcharakterisierungen in die Anfragen des jeweiligen Datenmanagements.

– `FileObjCharacterTranslator`
  Fragt als Quasi-Datenmanagement einzelne oder mehrere METS-Dateien ab. Dadurch eignet es sich auch intern für die Abbildung von XPath-Ausdrücken in die Anfragen anderer Datenmanagementsystemen. Dies kann realisiert werden, indem METS-Dateien durch den `FileObjCharacterTranslator` abgefragt werden, in denen statt der eigentlichen Werte der Archivobjekte die abzufragenden Felder des Datenmanagements gespeichert sind.

– `DbObjCharacterTranslator`
  Fragt als Datenmanagement die koLibRI-Datenbank ab, ob die durch XPath charakterisierten Objekte vorhanden sind. Nutzt für die Abbildung die Klasse `FileObjCharacterTranslator` und die Datei `MapUofToDB.xml`.

– `DiasObjCharacterTranslator`
  Fragt das DIAS-Datenmanagement ab, ob die durch XPath charakterisierten Objekte vorhanden sind. Nutzt für die Abbildung die Klasse `FileObjCharacterTranslator` und die Datei `MapUofToDIAS.xml`.

– `DiasCQLObjCharacterTranslator`
  Fragt durch CQLObjCharakter beschriebene Objekte im DIAS-Datenmanagement ab.

– `ObjectDependencyAnalysator`
  analysiert Abhängigkeiten zwischen Objekten und fasst sie zusammen. Zur Zeit werden nur durch METS-pointer verbundene Objekte analysiert.

– `MigrationPolicyBuilder`
  Baut und konfiguriert Migrationsabläufe mit Hilfe der Klassen `MigrationPolicyTem-`

`plateStore` und `MigrationServiceRegistry` für übergebene Objekte und Objektcharakterisierungen.

– `MigrationPolicyTemplateStore`
Klasse, die auf Basis der dazugehörigen Konfigurationsdatei die passenden Migrationsvorlagen für Abfragen liefert.

– `MigrationServiceRegistry`
Klasse, die auf Basis der dazugehörigen Konfigurationsdatei Beschreibungen und Aufrufparameter für gewünschte Migrationsdienste liefert.

6 Die koLibRI-Datenbank

koLibRI nutzt die Funktionalität von Hibernate [12] für sämtliche Datenbankzugriffe, um möglichst unabhängig von einem bestimmten Datenbanksystem zu sein. So kann durch die Nutzung eines anderen Datenbanksystems und durch Anpassung der Werte `hibernate-ConnectionDriver_class-` und `hibernateDialect` in der Konfigurationsdatei ein anderes Datenbanksystem aus der Fülle der von Hibernate unterstützten Systeme gewählt werden (Näheres hierzu ist in der Hibernate Dokumentation nachzulesen).

Derzeitige Einschränkungen

- Gleichzeitige Datenbankzugriffe von verschiedenen Rechnern aus sind noch nicht hinreichend getestet.
- Die koLibRI-Datenbank nutzt Fremdschlüssel, Transaktionen und die Autoinkrement-Funktionalität. Bei der Wahl einer MySQL-Datenbank schränkt dies die Wahl der Table-Engine auf `InnoDB` ein.

6.1 Installation

Benötigte Software/Jars

Im Klassenpfad müssen sich folgende Pakete befinden. Alle benötigten JARs befinden sich unter `kopal/lib`.

- `db-beans.jar`
- `hibernate3.jar`, inclusive aller von Hibernate benötigten Pakete
- ein JAR mit dem Datenbanktreiber für MySQL, z.B. mysql-connector-java-5.0.6-bin.jar

Installation einer Datenbank

Getestet wurde die koLibRI-Datenbank mit einer MySQL-Datenbank der Version 5.0 und der darin enthaltenen InnoDB-Table Engine. Für MySQL sind mit den Programmen *mySQL Administrator* und *mySQL Query Browser* einfache und frei erhältliche Verwaltungs-Tools für Windows verfügbar, aus denen heraus viele Konfigurations- und Abfrageaufgaben erledigt werden können. Ähnliche Tools sind auch für Unix und Mac OS X erhältlich. Für alle Datenbanken sind die entsprechenden Installationsanweisungen der Datenbank zu konsultieren.

Erstellen eines Datenbankschemas

Mit dem SQL-Skript `createDB.sql` im `config`-Verzeichnis kann das Datenbankschema für MySQL erstellt werden. Als Default-Datenbankname wird `kolibriDbTest` verwendet, das kann bei Bedarf durch Suchen-und-Ersetzen im Skript geändert werden. Zur Ausführung des Skripts kann entweder der Query-Browser benutzt werden oder man verbindet sich über die Kommandozeile mit

```
mysql -u [USER] -h [HOST] -p
```

und führt

```
source [PFAD]/createDB.sql
```

aus. Die Tabelen *owner* und *server* sind bereits mit einigen Default-Werten gefüllt, für eine sinnvolle Nutzung sind hier weitere Nutzer, Server und Passwörter einzurichten und in den Konfigurationsdateien entsprechend zu benutzen (`defaultOwner` und `defaultServer`).

Konfiguration der Datenbank

Es muss ein Datenbank-User mit Passwort für koLibRI angelegt werden, der von den Rechnern, auf denen koLibRI läuft, lesend und schreibend auf die Datenbank zugreifen darf. Im MySQL-Administrator kann dies im Menüpunkt *User Administration* erledigt werden, oder auch wie oben über die Kommandozeile. Firewall-Einstellungen sind bei Remote-Zugriffen zu beachten.

Konfigurieren von koLibRI

koLibRI muss zur Nutzung der Datenbank an mehreren Stellen konfiguriert werden. Neben der Einstellung, dass überhaupt eine Datenbank genutzt werden soll, müssen klassenspezifische Werte für `util.db.HibernateUtil` in der Konfigurationsdatei gesetzt werden (siehe Konfigurationsdatei). Desweiteren muss man in den zu nutzenden Policies den Zeitpunkt und die Art der zu speichernden Daten bestimmen. Dies geschieht durch Einfügen des Actionmodules `actionmodule.sub.AddDataToDb`. Als Konfigurationswerte stehen die Folgenden zur Verfügung:

- `storeFileData`
  füllt die Tabellen *file* und *fileformat* für jede Datei in der Metadatenklasse.
- `storeFileDataTechMd`
  füllt die Tabelle *techmd* für jede Datei in der Metadatenklasse.

- `storeDc`
  speichert alle Dublin Core Metadaten der Metadatenklasse in der Tabelle *dublincore*.
- `storeIds`
  speichert alle Einträge von `ProcessData.customIds` in der Tabelle *identifier*.
- `storeCustomData`
  speichert alle Einträge von `ProcessData.customData` in der Tabelle *various*.
- `customDataEelements`
  bezeichnet die Elemente aus `ProcessData.customData`, die in die Datenbank geschrieben werden sollen.

Damit ein Eintrag in die *dias*-Tabelle nur dann erfolgt, wenn das SIP auch erfolgreich eingespielt wurde, können die Actionmodule `actionmodule.AlreadyIngestedChecker` vor dem Einspielen und `actionmodule.DiasIngestFeedback` danach genutzt werden. Diese führen einige Checks durch und tragen alle Daten nur dann in die Tabelle ein, wenn diese Tests erfolgreich waren. Wenn auch das Eintragen erfolgreich war, werden die Pakete in das Archivsystem eingespielt. Gibt es beim Einspielen der Pakete Probleme, können die in die Tabelle eingetragenen Daten konfigurierbar wieder gelöscht werden. `DiasIngest` wird in den Beispieldateien nicht verwendet, diese Klasse kann lediglich zur Dokumentation der Funktionalität dienen, da der Zugriff auf das DIAS nötig ist.

Die für die Konfiguration wichtigen Parameter können in der Konfigurationsdatei `config.xml` unter den entsprechenden Klassen nachgeschlagen werden.

6.2 Aufbau des Datenbankschemas

In der Tabelle *input* werden Daten gespeichert, die koLibRI beim ersten „Kontakt“ mit dem Rohmaterial bekommt: Wem gehört das Objekt (Verweis auf *owner*), woher kommt der Auftrag/die Dateien (Verweis auf `server`) und wann war das (Spalte `startdate`, angegeben in Millisekunden seit 1970, siehe auch `java.lang.System.currentTimeMillis()`). Wenn schon vor dem Ausführen der Policy (d.h. nach dem ProcessStarter) Dateien im ProcessData-Objekt aufgeführt sind, wird eine ge-XOR-te SHA1-Checksumme aller Dateien vermerkt. Zusätzlich kann das Objekt optional mehrere Dateien, Dublin Core Metadaten, Identifier oder extra Informationen aufweisen (Verweise von *file*, *dublincore*, *identifier* oder *various* auf den *input-Eintrag*).

Während der Abarbeitung der Policy kann es zu Warnungen oder Fehlern kommen. Solche Ereignisse werden in der Tabelle *event* aufgeführt und verweisen auf den *input*-Eintrag,

bei dem es zu Problemen kam. Es wird auf den *modulestatus* und die Beschreibung des Fehlers *description* verwiesen. Zusätzlich werden der Zeitpunkt *eventdate* und der Modulname *module* protokolliert. Auch der Verweis auf eine Datei, die das Ereignis auslöste, ist möglich.

Schließlich wird das Archivpaket erfolgreich in DIAS eingespielt und ein Eintrag in *dias* vorgenommen. Neben der externen und internen DIAS-ID und dem Zeitpunkt des Einspielens wird auch ggf. bei einer Migration die interne ID des Vorgängerobjekts protokolliert. Falls sich die Dateien seit dem Eintrag in die *input*-Tabelle verändert haben, wird hier die veränderte ge-XOR-te SHA1-Checksumme aller Dateien gespeichert. Mit dem *deleted* Flag werden im Archiv gelöschte Pakete protokolliert.

Abbildung 2: Das Datenbankschema der koLibRI-Datenbank

7 Eigene koLibRI-Erweiterungen

Für Erweiterungen sind insbesondere die Interfaces `ActionModule` und `ProcessStarter` interessant. Eine gewisse Vertrautheit mit der Struktur koLibRIs und den Klassen des Workflow Frameworks `de.langzeitarchivierung.kopal` ist notwendig. Hier soll ein funktionaler Überblick gegeben werden, für die Detailanforderungen sollte die Javadoc-API-Dokumentation und der Quellcode konsultiert werden.

7.1 Die Struktur koLibRIs

Die Paketstruktur koLibRIs liegt unterhalb von `de.langzeitarchivierung.kopal`. In diesem Paket liegen die zentralen Klassen des Workflow Frameworks. Darunter liegen die folgenden Pakete, unterhalb denen jeweils für Institutionen spezifische Unterpakete möglich sind:

- `actionmodule`
  Enthält das Interface `ActionModule` und dessen implementierende Klassen.
- `administration`
  Enthält die Klassen, die das Such- und Administrations-Interface von DIAS implementieren, näheres siehe Kapitel 8, Seite 44.
- `formats`
  Enthält das Interface `MetadataFormat` und die implementierende Klasse `Uof` für das *Universelle Objektformat* des kopal-Projekts.
- `ingest`
  Hier sind die Klassen für das Einspeisen in das DIAS enthalten.
- `jhove`
  Hier sind die zusätzlichen Module für das *JSTOR/Harvard Object Validation Environment* enthalten, die von DNB und SUB programmiert wurden.
- `processstarter`
  Enthält das Interface für die `ProcessStarter` und dessen implementierende Klassen.
- `retrieval`
  Enthält die Klassen für den Zugriff auf die Archivpakete, die sich bereits im DIAS befinden.

- `ui`
  Hier sind Klassen für Nutzerinterfaces und Eventbehandlung enthalten.

- `util`
  Hier befinden sich allgemeine Hilfsklassen, wie zum Beispiel Methoden zur Dateiverwaltung, HTML- und HTTP-Zugriff, der TiffIMP, uswusf.

- `ws`
  Die koLibRI-Webservices, näheres siehe Kapitel 9.

Das Workflow Framework `de.langzeitarchivierung.kopal` besteht zur Zeit aus acht Klassen:

- `Policy`
  Verwaltet einen Workflow und baut aus seiner XML-Repräsentation in der Policy-Konfigurationsdatei einen Baum aus Steps auf.

- `PolicyIterator`
  Iteriert über die Arbeitsschritte eines Workflow-Baums.

- `ProcessData`
  Hält die zentralen Daten für ein zu bearbeitendes Archivpaket: Den Namen des Prozesses (z.B. URN des Archivpakets), dessen Policy, die Metadaten (das Uof Objekt) sowie die Dateiliste. Die `go`-Methode enthält die Logik zur Abarbeitung der Policy.

- `ProcessQueue`
  Die Queue aller zu bearbeitenden `ProcessData`-Objekte.

- `ProcessThreadPool`
  Ein ThreadPool, der ProcessStarter und ProcessData Objekte verarbeitet.

- `Status`
  Hält Statuswert und -beschreibung für den aktuellen Stand eines ActionModuls. Mit dem Statuswert wird die Abarbeitung eines Workflows gesteuert.

- `Step`
  Ein einzelner Arbeitsschritt im Workflow eines Archivpakets. Enthält u.a. Funktionen zum Laden und Starten der ActionModule und Setzen ihrer Statuswerte.

- `WorkflowTool` Stellt Kommandozeilen-Interface, startet die ProcessStarter und arbeitet die Prozesse ab. Es werden nur Arbeitsschritte eines Prozesses durchgeführt, die den Statuswert `Status.TODO` und deren vorherigen Schritte den Wert `Status.DONE` haben.

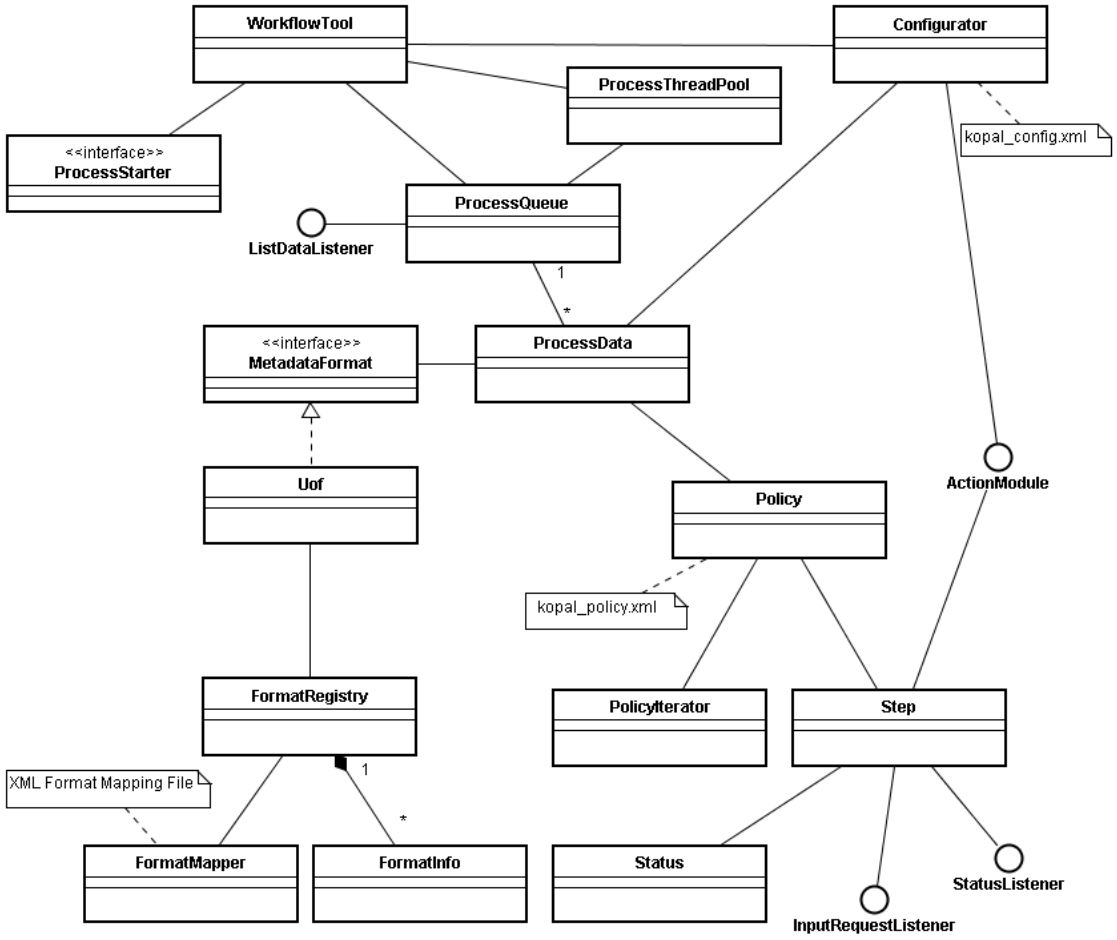

Abbildung 3: Das Klassendiagramm koLibRI

7.1.1 ProcessStarter

Für jedes neue Archivpaket muss der ProcessStarter im wesentlichen folgende Aktionen durchführen:

1. Ein neues ProcessData-Objekt anlegen,
2. ggf. Startwerte der Metadaten initialisieren, wenn ActionModule diese für ihre weiteren Aktionen benötigen und
3. das ProcessData-Objekt in die Liste abzuarbeitender Prozesse hinzufügen (`ProcessQueue.addElement()`).

Solange noch mindestens ein ProcessStarter läuft oder Elemente in der ProcessQueue enthalten sind, beendet sich koLibRI nicht, sondern wartet auf neue Prozesse oder die Abarbeitung der bestehenden.

7.1.2 ActionModule

Ein ActionModul muss

1. seinen Statuswert, wenn es aufgerufen wird, auf `Status.RUNNING` setzen und
2. seinen Statuswert, wenn es am Ende erfolgreich abgearbeitet ist, auf `Status.DONE` setzen.

Ein ActionModul sollte

- seinen Statuswert auf `Status.ERROR` setzen, wenn ein ernsthafter Fehler auftritt. Die Fehlermeldung wird separat protokolliert und Module mit diesem Statuswert werden nicht wieder aufgerufen. Es ist *nicht* gleichbedeutend mit dem Auslösen einer Java-Exception, d.h. das Modul wird in seiner Ausführung nicht unterbrochen und kann z.B. die Aktion nochmal ausführen und seinen Statuswert wieder ändern. Ein übliches Verfahren wäre z.B. bei einem Timeout den Status erst auf `Status.ERROR`, dann gleich auf `Status.TODO` zu ändern und vorläufig wieder mit `return` ins Workflow Framework zurückzuspringen. Auf diese Weise wird der Fehlschlag protokolliert und zu einem späteren Zeitpunkt ein neuer Versuch unternommen.
- den Statuswert `Status.WARNING` nutzen, wenn ein Ereignis auftritt, das als Warnung protokolliert werden soll (auch in der Datenbank), das aber nicht zum Abbruch des Module führt.

Ein ActionModul kann

- insbesondere die Metadaten und Dateien im ProcessData-Objekt bearbeiten.
- in der HashMap `customData` im ProcessData-Objekt Daten für andere Arbeitsschritte hinterlegen, für die kein Platz in der hinter dem UOF befindlichen Struktur vorhanden ist.
- auf die Konfigurationsdaten zugreifen.

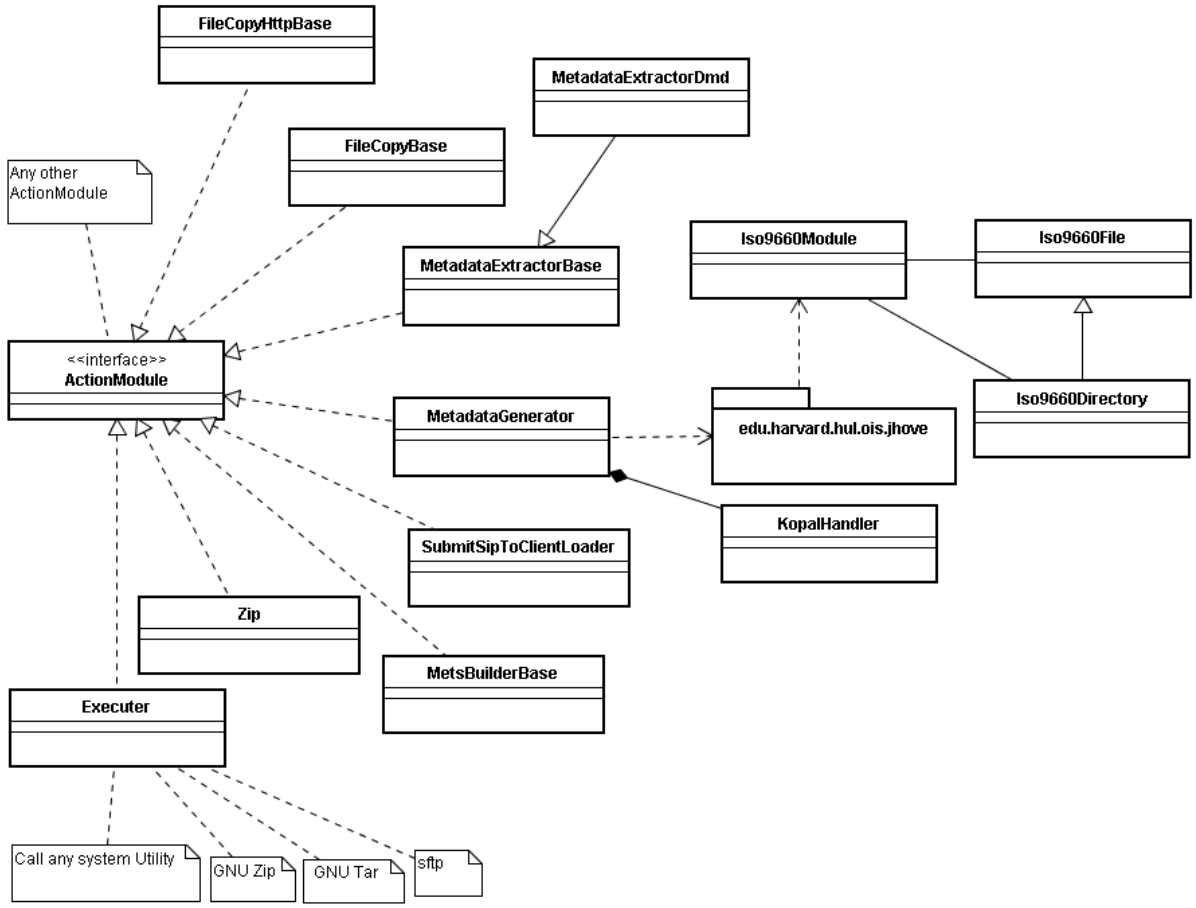

Abbildung 4: Klassendiagramm ActionModules

7.2 Konfiguration von Klassen

Die Konfiguration von Klassen geschieht durch die Klasse `Configurator`. Sie wird durch die Methode `setConfigDocument()` mit einer XML-Datei initialisiert und setzt dann durch verschiedene, überladene `configure`-Methoden die benötigten Konfigurationswerte von Klassen und Objekten, insbesondere von ActionModules und ProcessStarter. Dies geschieht dadurch, dass zu konfigurierende Klassen und Objekte entsprechend der JavaBeans-Spezifikation `set`-Methoden für die benötigten Werte zur Verfügung stellen.

Wenn z.B. ein ActionModule den Wert `useDatabase` aus der Konfiguration benötigt, stellt es eine `setUseDatabase`-Methode zur Verfügung, die im Programmablauf von der Configurator-Klasse mit dem für das ActionModule spezifischen Parameter aufgerufen wird, siehe auch Kapitel 2, Seite 7.

Auch die Übertragung von Konfigurationswerten zwischen Objekten und Klassen durch Setter- und Setter-Methodenpaare ist möglich, für Details ist die API-Dokumentation zu

konsultieren oder der Quellcode einzusehen.

7.3 Metadatenformate

Die metadatenformatspezifischen Funktionalitäten und Objektstrukturen sind weitgehend im Paket `formats` konzentriert. Das Interface MetadataFormat gibt die grundlegenden Funktionalitäten vor, die für die Verarbeitung benötigt werden. Nicht immer sind diese spezifisch genug, um eine vollständige Unabhängigkeit von dem konkreten Metadatenformat wie dem UOF zu erreichen.

Bei der vorliegenden koLibRI-Version ist das insbesondere bei der Abbildung der Metadaten auf die Datenbank durch die Klasse `MappToHib` der Fall, wie auch bei der Bearbeitung und Erstellung von Migrations-Metadaten (`PrepareMigration` und `MigrateFiles`).

Zur Implementierung des Interfaces `MetadataFormat` wurde auf ein Java-XML-Bindung durch XML-Beans [23] zurückgegriffen. Das dem UOF zugrundeliegende XML-Schema wurde mit dem XML-Beans scomp-Compiler in Java-Klassen übertragen. Auf diese Weise ist eine strukturierte und konsistente Nutzung des Schemas mit Java möglich. Insbesondere werden Funktionalitäten wie das Schreiben, Einlesen und Validieren entsprechender XML-Dateien bereitgestellt. Um eigene Schemata bzw. ein eigenes Metadatenformat zu integrieren, können ebenfalls mittels des XML-Beans scomp-Compiler entsprechende Java-Klassen generiert werden. Die zu erwartende Anpassungsarbeit liegt dann primär in der Implementierung des Interfaces MetadataFormat und zusätzlicher Hilfsmethoden, in der Anpassung vereinzelter ActionModule und bei der Nutzung der Datenbank in der Klasse `MappToHib`.

8 Implementation der DIAS-Administrations- und Suchschnittstelle

In koLibRI 1.0 wurden Klassen und Schnittstellen hinzugefügt, welche die Nutzung der von IBM implementierten Administrations- und Suchschnittstellen von DIAS-Core ermöglichen und unterstützen sollen. Dazu existieren sowohl Methoden zur komfortablen Generierung von entsprechenden HTTP-Anfragen an DIAS, als auch Methoden und Datenklassen, die die Interpretation und Weiterverarbeitung der XML Antworten von DIAS ermöglichen.

Alle dafür benötigten Klassen und Schnittstellen befinden sich im Unterpaket `administration` mit der zentralen Klasse `DiasAdmin.java`. In dieser Klasse sind alle Methoden vereint, um die Abfragen an DIAS zu formulieren und die XML Antworten zu parsen.

Grundsätzlich unterscheidet DIAS-Core zwischen zwei Schnittstellen. Zum einen ist das die Administrationsschnittstelle, die beispielsweise eine Löschfunktion für Objekte, Administrationsfunktionen für XML Schemata und die Liste bekannter Dateitypen, eine Konsistenzüberprüfung für eingespielte Objekte und eine Funktion zur Auflistung kürzlich veränderter Objekte bietet. Als zweites gibt es eine Suchschnittstelle, die Anfragen und Antworten nach dem SRU-Standard unterstützt.

Die `DiasAdmin`-Klasse vereint die Funktionen von beiden Schnittstellen, konkret sind dies:

- Einfügen eines neuen Dateityps in die Liste der DIAS bekannten Dateitypen
- Modifizierung eines Eintrags in der Liste der DIAS bekannten Dateitypen
- Löschung eines AIP (Datensatz eines Objektes zu einer internen DIAS ID, Datenbankeintrag in DIAS bleibt jedoch erhalten)
- Löschung eines Assets (Komplettes Objekt zu einer internen DIAS ID, Löschung jedoch nur möglich, wenn keine Kindobjekte existieren)
- Konsistenzüberprüfung eines eingespielten Objektes
- Bekanntmachung eines neuen XML-Schemas zur Verwendung innerhalb der METS-Metadaten
- Abfrage eines bestimmten XML Schemas
- Abfrage der Liste aller unterstützten XML Schemas
- Anfrage der Liste aller in einem definierten Zeitraum veränderten Objekte
- Übermittlung einer vorformulierten CQL-Abfrage an den SRU-Server von DIAS
- Generierung einer individuellen SRU-SearchRetrieve-Anfrage an den DIAS-Server

- Generierung einer individuellen SRU-Explain-Anfrage an den DIAS-Server
- Parsen einer Antwort der DIAS Administrations-Schnittstelle und ggf. Überführung der Daten in eine definierte Datenklasse
- Parsen einer Antwort der DIAS SRU-Such-Schnittstelle und ggf. Überführung der Daten in eine definierte Datenklasse

Um die Funktionalitäten der Klasse zu verwenden, muss der globalen Konfigurationsdatei lediglich ein Teil für die DiasAdmin-Klasse hinzugefügt werden. Im folgenden werden die einzelnen Konfigurationsparameter kurz erläutert:

```
<class name="administration.DiasAdmin">
    <field>useHttps</field>
          Steuert, ob die Verbindung über eine gesicherte HTTPS Verbindung oder über
          gewöhnliches HTTP erfolgen soll.
    <field>adminServer</field>
          Die Adresse des DIAS Administrations-Servers
    <field>adminPort</field>
          Der Port des DIAS Administrations-Servers
    <field>adminUser</field>
          Der Username für die Durchführung von administrativen Aufgaben
    <field>adminPwd</field>
          Das Administrations-Kennwort
    <field>keyStoreFile</field>
          Schlüsseldatei für die sichere Verbindung zum DIAS Administrations-Server
    <field>sruMaximumRecords</field>
          (Optional) Regelt die maximale Anzahl von Datensätzen, die in einer einzelnen
          SRU Antwort enthalten sein dürfen.
    <field>sruResultSetTTL</field>
          (Optional) Die Zeitspanne (in Sekunden), in der die ausgeführte Suche für
          weitere Zugriffe auf dem Server gespeichert wird.
    <field>sruSortKeys</field>
          (Optional) Sortierschlüssel, die vom SRU Server bei der Lieferung der Antwort
          berücksichtigt werden müssen.
    <field>sruStartRecord</field>
          (Optional) Die Nummer des ersten Datensatzes der Antwort
          (beispielsweise "100" für die Datensätze 100-200)
    <field>sruVersion</field>
          (Optional) Die Versionsnummer des SRU Servers. Der Standardwert ist "1.1"
    <field>sruX_additionalDataElements</field>
          (Optional) Zusätzliche DIAS Datenfelder, die mit dem Antwort-Datensätzen
          zurückgeliefert werden sollen.
    <field>sruX_listAffectedContentFiles</field>
          (Optional) Regelt, ob eine Liste von Datei IDs aller von der Suche betroffenen
```

```
        Dateien in der Antwort enthalten sein muss.
</class>
```

Die Klasse kann danach über den `Configurator` konfiguriert werden und je nach Einsatzziel in einem Actionmodul oder einer beliebigen anderen Klasse verwendet werden. Denkbar wäre auch die Entwicklung eines komfortablen Administrations-Tools mit grafischer Oberfläche. Konkreten Einsatz findet die Klasse derzeit im Prototypen des Migration Managers in den Klassen `DiasCQLObjCharacterTranslator.java` und `DiasCQLObjCharacterTranslator.java`, die dazu dienen, eine über Suchkriterien definierte Menge von archivierten Objekten für einen Migrationsvorgang auszuwählen. Siehe dazu den entsprechenden Teil der Dokumentation des *Migration Manager*.

Die Object Character Translator Klassen `DiasCQLObjCharacterTranslator` und `DiasObjCharacterTranslator` sind für die Abfrage von zu migrierenden Objekten über die SRU-Suchschnittstelle von DIAS verantwortlich.

`DiasCQLObjCharacterTranslator` verwendet dabei eine vorformulierte und komplette CQL-Abfrage, die in eine SRU-Anfrage verpackt und dann an das DIAS-System übergeben wird. Dies ermöglicht sehr komplexe Abfrageformulierungen und Einschränkungen. Für eine Definition von möglichen Anfragen ist die offizielle Spezifikation der *Administration and Search interfaces to DIAS-Core* von IBM zu konsultieren.

`DiasObjCharacterTranslator` verwendet ein Suchdefinitionsobjekt vom Typ `ObjCharacter` zur Formulierung der gesuchten Kriterien. Dies erlaubt beispielsweise die parallele Suche in mehreren Datenquellen, wie beispielsweise der kopal Datenbank und DIAS selbst. Für die Übersetzung zwischen den Datenfeldern innerhalb einer UOF-METS-Datei und den Datenfelder der DIAS-Datenbank kommt die Datei `MapUofToDiasData.xml` zum Einsatz, die sich im `/config`-Verzeichnis des Releases befindet und jederzeit ohne Anpassung am Sourcecode erweitert werden kann.

9 koLibRI als Web-Service

9.1 Einführung

Die Möglichkeit, koLibRI als Web-Service zu nutzen, ist eine der wichtigen Neuerungen der Version 1.0 gegenüber der früheren Version 0.7. Als Web-Service muss koLibRI nicht von der Befehlseingabe aus als Stapelverarbeitung der zu archivierenden Objekte gestartet werden, sondern wird von dem Administrator innerhalb eines Servlet-Containers gestartet und horcht ununterbrochen an einem vordefinierten Port, um Benutzeranfragen entgegen zu nehmen.

Eine jede solche Anfrage ist eine SOAP [13] Nachricht, die neben Benutzernamen, Kennwort und Institutionsnummer alle für die Verarbeitung dieser Anfrage nötigen Information beinhaltet. Die Java Klasse `WebServiceServer` bekommt eine solche Nachricht und ruft die entsprechende Methode in einer der drei Zwischenschichtklassen `IngestLayer`, `AccessLayer oder WSUtils` auf.

Alle Anfragen liefern ein `ResponseElemDocument` zurück, dieses enthält einen Statuscode, eine menschenlesbare Erklärung für diesen Code, die interne DIAS ID des digitalen Objektes (nur bei entsprechenden Anfragen), die Wartezeit (falls DIAS beschäftigt ist), einen Link zur angeforderten Datei (falls eine zum Herunterladen existiert), extra Informationstext (falls nötig), Fehlermeldungen (falls Fehler vorhanden) und ein benutzerdefiniertes XML Feld für Anworten, bei denen komplexere Daten – wie zum Beispiel Statistikwerte – zurückgeliefert werden müssen.

9.2 Installation

Die Installation von koLibRI als Web-Service ist sehr einfach, da nur das Entpacken der Inhalte aus dem heruntergeladenen ZIP-Archiv in das Axis2 Verzeichnis `/WEB-INF/services` und das Starten des Tomcat Application Servers nötig ist.

Benötigt werden nur drei Dateien: `kolibri_web_services.aar` beinhalten den nötigen Programmcode und alle Bibliotheken. Die weiteren zwei sind die Konfigurations- und Policy-Dateien im XML Format, die den Konfigurationsdateien in den vorherigen gleichen. Im reellen Produktionssystem wird eine zusätzliche Datei für Verschlüsselungszertifikate benötigt.

Die Installation ist nur so einfach, falls eine lauffähige Axis2-Umgebung [14] vorhanden ist. Falls nicht, muss erst Axis2 installiert werden. koLibRI wurde unter Axis2 Version 1.2 getestet. Zwar kommt Axis2 mit einem eingebauten Web-Server und ist daher auch ohne externen Web-Server lauffähig, aber es wird empfohlen, in produktiven Systemen einen *echten* Web-Server zu benutzen. Der koLibRI Web-Service wurde unter Tomcat Version 5.5.23 [15] getestet.

9.3 Konfiguration

Das Betriebssystem, Java SDK, Tomcat und Axis2 müssen laut deren jeweils eigenen Dokumentationen konfiguriert werden. Jede koLibRI-relevante Konfiguration wird über die koLibRI-eigene Konfigurationsdatei, die in den vorherigen Kapiteln ausführlich erklärt wurde, konfiguriert.

9.4 Starten des Web-Services

Starten Sie Tomcat mit Hilfe des `startup`-Scripts im `$CATALINA_HOME/bin` Verzeichnis. Vergewissern Sie sich anhand der jeweiligen Testseiten, dass Tomcat und Axis2 ordnungsgemäß laufen und falls nötig, rufen sie die `setParameters` Methode auf, um aktuelle Parameter zu ändern. Dafür kann sowohl die Klasse `TestWebServices` – als einfaches Beispiel – oder ein selbstgeschriebes Programm, welches als Vereinfachung `WebServiceClient` benutzt, eingesetzt werden.

Der koLibRI Web-Service ist dann bereit zum Start. Rufen Sie `initCache` auf, um den Zwischenspeicher zu initialisieren – ohne Zwischenspeicher wird der Web-Service nicht funktionieren. Nach erfolgreicher Initialisierung des Caches geben Sie den Ingest und/oder das Retrieval frei mit Hilfe von `allowIngestRequests` und/oder `allowRetrievalRequests`. Wenn keine Fehlermeldung erfolgt, ist der Web-Service einsatzbereit.

Bitte haben Sie Verständnis, dass aus Sicherheitsgründen DIAS alle Verbindungsversuche nach deren Berechtigung prüft. Daher versuchen Sie *nicht*, eine Verbindung zu DIAS aufzubauen oder gar Ihre generierte SIPs zum DIAS-Loader hochzuladen.

9.5 Funktionsweise des Ingests

Wie in der WSDL-Datei zu sehen ist, braucht Ingest eine Policy, die beschreibt, nach welchen Kriterien das jeweilige digitale Dokument zu verarbeiten ist, sowie ein bis sechs Metadatenblöcke von Typ `customXMLType`. Der erste ist Pflicht und beschreibt – unter anderem – die *unverzichtbare* Information des Persistent Identifiers (in unserem Falle eine URN) und außerdem, wo sich die einzuspielenden Dateien befinden. Weitere Informationen wie Dokumentname, Dateichecksummen, u.ä. sind erwünscht, momentan aber kein Pflicht.

Theoretisch kann `customXMLType` beliebig valides XML übertragen. Diese erste Version des Web-Services nutzt LMER als Metadatenformat, da es auch das vom UOF genutzte Format ist.

Die anderen fünf optionalen Metadatenblöcke sind für deskriptive Metadaten vorgesehen. Sie werden ohne weitere Verarbeitung direkt in den `DescMd` Teil der METS Datei

übernommen. Laut DIAS-Dokumentation muss, falls Descriptive Metadaten vorhanden sind, der erste Block im Dublin Core Simple [16] Format vorliegen.

In dieser ersten Version ist der Web Service Ingest nicht vollständig implementiert. Um komplette SIPs zu generieren, benutzen Sie bitte die standalone koLibRI-Version.

9.6 Funktionsweise des Retrievals

Das Sequenzdiagramm in Abbildung 5 zeigt den zeitlichen Ablauf des Retrievals. Typischerweise suchen BenutzerInnen einer Bibliothek benötigte Informationen im Bibliothekskatalog – der *kein* Teil von koLibRI ist – und bekommen eine Liste von für sie mehr oder weniger interessanten Publikationen als Suchergebnis. Falls sie der Meinung sind, dass eine bestimmte Publikation die richtige ist, fordern sie diese Publikation an.

Für den Fall, dass diese Publikation in elektronischer Form vorliegt, schickt die Benutzungsschnittstelle die URN dieser Publikation an koLibRI und fordert die dazugehörigen Metadaten. Der Web-Service wiederum fordert von DIAS die kompletten Metadaten für diese URN. DIAS antwortet mit einem Link zur gezippten METS-Datei dieser Publikation und ihrer internen DIAS ID. Der Web-Service generiert eine eindeutige Pfadangabe anhand dieser (ebenfalls eindeutigen) internen ID und prüft, ob das gepackte Objekt (`mets.zip`) bereits vorhanden ist. Ist das der Fall, spart er sich ein nochmaliges Herunterladen und liefert einen Link zur heruntergeladenen Datei.

Die Benutzungsschnittstelle entpackt die `mets.xml`, verarbeitet sie und generiert für die BenutzerInnen ein Inhaltsverzeichnis. Nachdem eine bestimmte Datei aus dieser Liste ausgewählt wurde, fordert die Benutzungskomponente diese Datei mit ihrer internen DIAS ID und den Dateinamen vom Web-Service. Dieser prüft seinen Cache auf das Vorhandensein dieser Datei und entscheidet, ob er sie aus dem DIAS holen muss.

9.7 Organisation des Zwischenspeichers

Der koLibRI Web-Service benutzt ein Verzeichnis, um die heruntergeladenen Metadaten und DIPs zu speichern. Dieses Verzeichnis wird als Cache bezeichnet aufgrund des positiven Nebeneffekts, dass der Web-Service bestimmen kann, ob eine bestimmte Datei vor kurzer Zeit bereits heruntergeladen wurde. Diese Datei muss dann nicht wiederholt von DIAS angefordert werden. Besonders für Dateien, die DIAS auf den Bandspeicher abgelegt hat, oder auch für extrem große Dateien, beschleunigt der Cache das Retrieval enorm und entlastet gleichzeitig das DIAS-System.

In Wirklichkeit ist der Cache nicht ein einziges Verzeichnis, sondern ein ziemlich großer Baum von Verzeichnissen (siehe Abbildung 6). Er ist in drei Stufen organisiert, jeweils mit

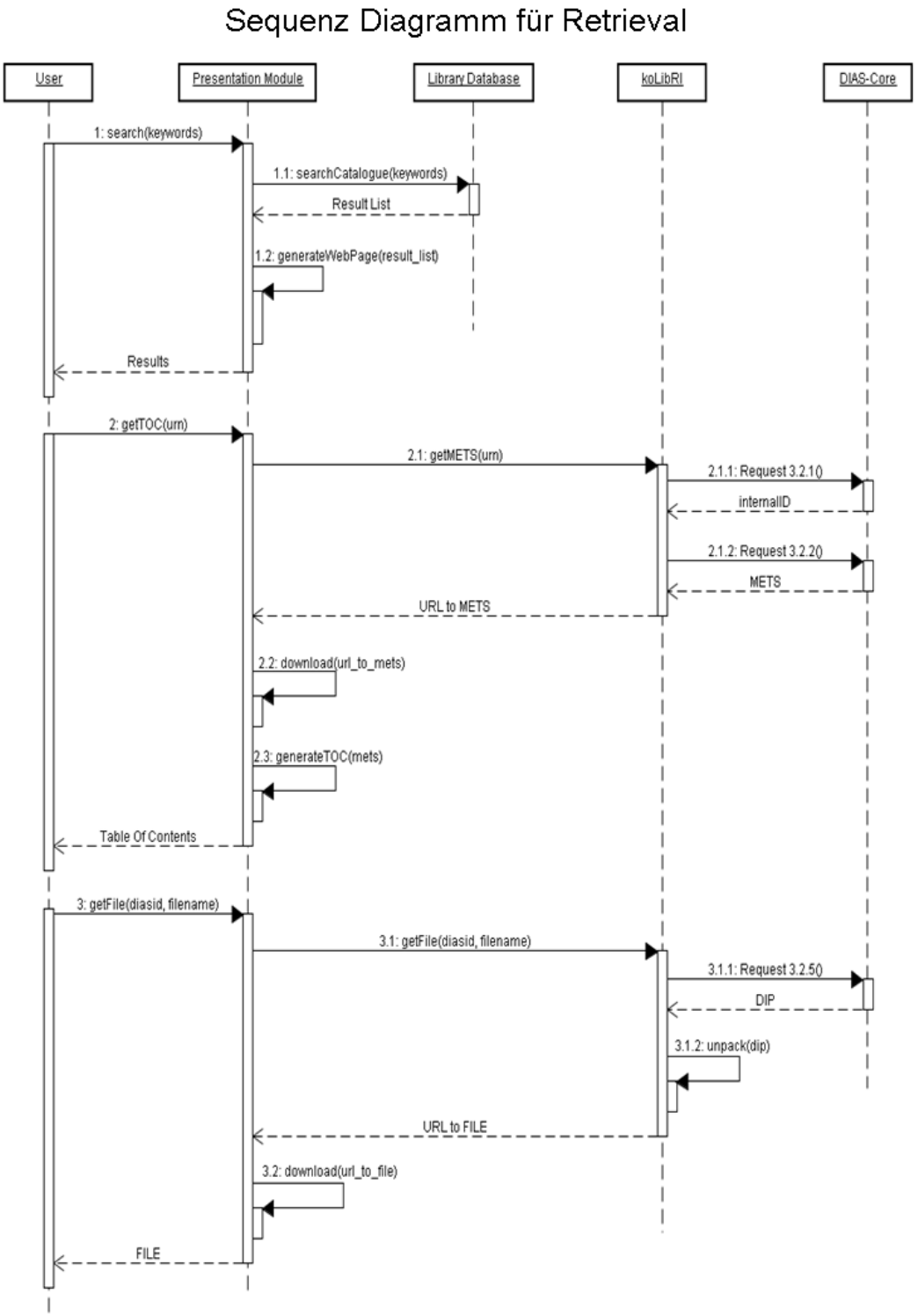

Abbildung 5: Sequezdiagramm des Retrievals inklusive externer Benutzungskomponente

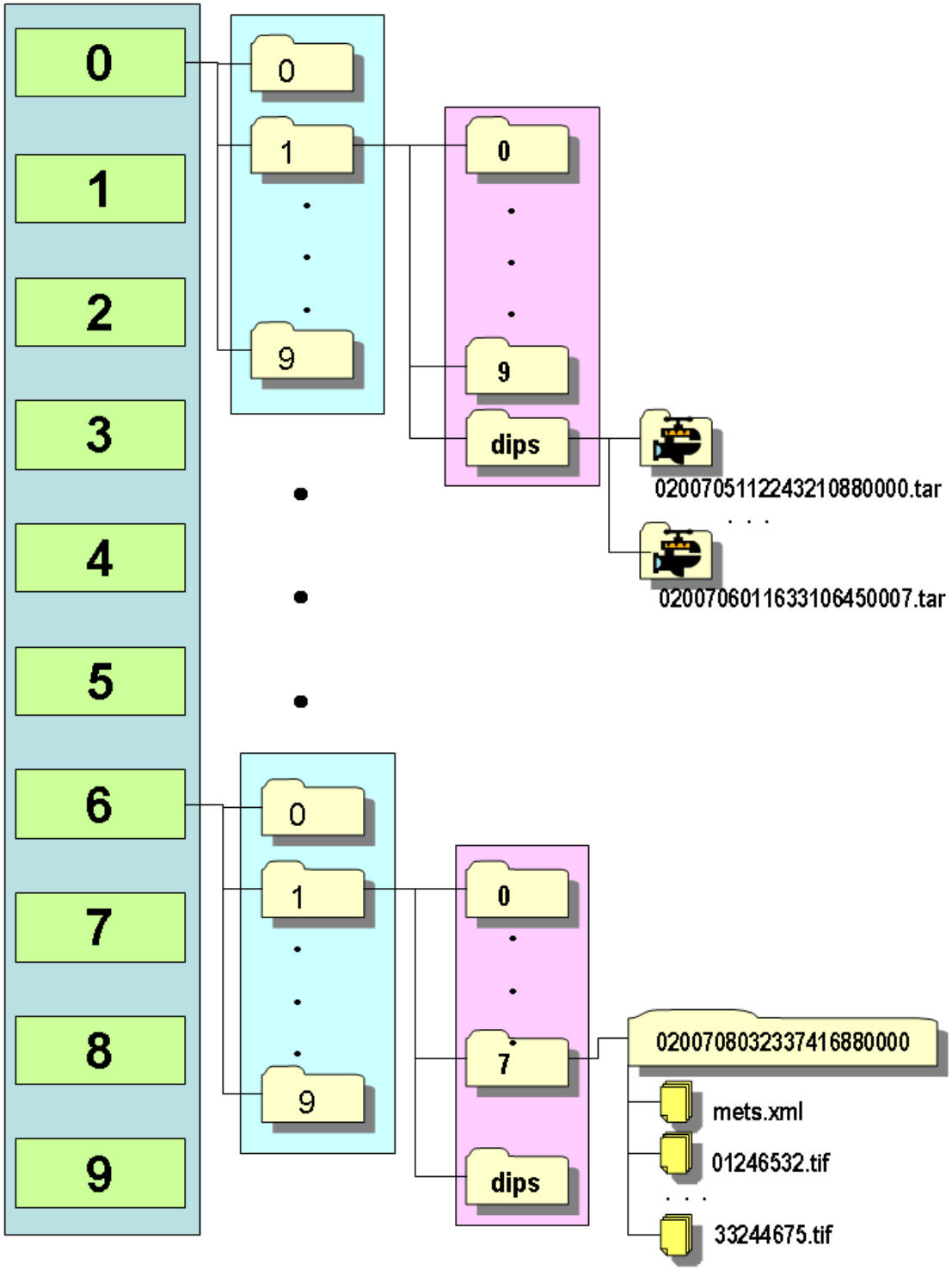

Abbildung 6: Organisation der Verzeichnisstruktur im Cache

einstelligen Zahlen 0...9 als Verzeichnisnamen. Dies basiert auf dem Algorithmus, den DIAS für die Generierung seiner eindeutigen internen IDs benutzt, um den Speicherort einer bestimmten Datei wiederzufinden. DIAS generiert für jede Version aller digitaler Dokumente einen eindeutige Bezeichner im Format

`[prefix]:YYYYYMMDDhhmmssSSSnnnn`

wobei gilt:

- `YYYYY` als Jahr (02007 für dieses Jahr)
- `MM` als Monate (01...12)
- `DD` als Tage (01...31)
- `hh` als Stunden (00...23)
- `mm` als Minuten (00...59)
- `ss` als Sekunden (00...59)
- `SSS` als Millisekunden (000...999)
- `nnnn` als fortlaufende Nummer, falls in derselben Millisekunde mehrere Assets eingespielt wurden.

Der Web-Service generiert aus dieser ID einen eindeutigen Pfad zu einer bestimmten Datei. So ist es jederzeit möglich zu prüfen, ob eine bestimmte Datei sich bereits im Cache befindet. Der Serveradministrator muss diesen Speicherplatz gegen Überlauf kontrollieren. Der Web-Service löscht die alten Dateien nicht selbstständig, sondern bietet die `eraseOldCacheEntries` Methode zum manuellen Abruf.

Eine intelligentere Cacheorganisation wäre zwar möglich, aber würde ohne nennenswerten größeren Nutzwert die Komplexität erhöhen.

9.8 Mögliche Operationen

Der Web-Service bietet folgende Operationen:

- `getFileFormatList`
- `reloadFileFormatList`
- `ingestDocument`
- `ingestSIP`
- `blockNewIngestRequests`

- `allowIngestRequests`
- `getIngestStatistics`
- `resetIngestStatistics`
- `getMETS`
- `getFile`
- `getDIP`
- `getDIPExternal`
- `getModifiedSince`
- `getHistory`
- `blockNewRetrievalRequests`
- `allowRetrievalRequests`
- `getRetrievalStatistics`
- `resetRetrievalStatistics`
- `initCache`
- `eraseOldCacheEntries`
- `deleteCache`
- `setParameters`
- `changePassword`

Alle weitergehenden tiefergehenden Informationen sind im JavaDoc enthalten und soll an dieser Stelle nicht wiederholt werden. Einige dieser Operationen sind für das nächste Release geplant und daher momentan nicht implementiert. Aber deren *Grundgerüst* ist fertig, so dass sie keine Fehler verursachen, sondern fehlerfrei

```
598 Sorry. This service is not implemented at the moment.
```

zurückliefern.

9.9 Weitere Information für Programmierer

An dieser Stelle eine kurze Zusammenfassung für Anwendungsentwickler. Für detaillierte Informationen konsultieren Sie bitte die JavaDoc-Dokumentation und den Quellcode.

9.9.1 WSDL

Der koLibRI Web-Service ist nach dem *contract first*-Ansatz realisiert, d.h., dass zunächst alle Typen, Nachrichten, Ports und Bindings in der Web Services Description Language [17] definiert wurden. Die WSDL-Datei findet sich im Anhang, Kapitel 10.2 auf Seite 58.

9.9.2 Automatisch generierte Java-Packages

Diese Definitionen wurden benutzt, um mit Hilfe des Axis2 Code Generator Plug-Ins von Eclipse die nötigen Java-Klassen generieren zu lassen. Eine Kommandozeilen-Version dieses Generierungswerkzeugs befindet sich bereits im Axis2 Paket. Der maschinengenerierte Code ist für Menschen schwer nachvollziehbar und liegt daher nicht als Quellcode vor, stattdessen wird ein daraus kompiliertes JAR mitgeliefert. Falls Interesse an diesen Klassen besteht, können sie jederzeit aus der WSDL Datei neu generiert werden.

9.9.3 Der Server

Auf der Server-Seite nimmt die Java-Klasse `WebServiceServer`, die aus dem generierten Jar das `KolibriWebServiceSkeletonInterface` implementiert, die SOAP Nachrichten entgegen und extrahiert die benötigten Parameter, um die entsprechende Methode der Zwischenschichtklassen `IngestLayer, AccessLayer und WSUtils` aufzurufen.

9.9.4 Der Client

Auf der Client-Seite nimmt die Java-Klasse `WebServiceClient`, die aus dem generierten Jar eine neue Instanz des `KolibriWebServiceStub` kreiert, die benötigten Parameter und konstruiert SOAP Nachrichten. Sie ist als Hilfsklasse konzipiert, um die clientseitige Programmierung zu vereinfachen, weil die Arbeit mit Java Konstrukten für viele Entwickler einfacher ist als mit komplexen XML-Bäumen.

9.9.5 Die Testklasse

`TestWebServices` ist einerseits ein einfaches Beispiel, um zu demonstrieren, wie der `WebServiceClient` und die Kommunikation mit dem Web-Service funktionieren. Ausserdem ist eine Testklasse ohnehin nötig, um die korrekte Funktionalität der `WebServiceServer` testen zu können. Die Klasse ist nicht für den produktiven Einsatz implementiert. Man sollte eigene Software mit ähnlichen Aufrufen des `WebServiceServer` (mit oder ohne Hilfe von `WebServiceClient`) implementieren.

10 Anhang

10.1 Nutzung von JHOVE in koLibRI

koLibRI nutzt zur Gewinnung technischer Metadaten das JSTOR/Harvard Object Validation Environment [8] (JHOVE) in der Version 1.1f (Release vom 8. Januar 2007) mit einigen Bugfixes, die auch in kommenden Maintainance Releases enthalten sein werden. Die Nutzung von JHOVE funktioniert hierbei vollautomatisch über das ActionModule `MetadataGenerator` und seine interne Klasse `KopalHandler`, die als JHOVE `OutputHandler` fungiert, über den direkt innerhalb des Programms auf die erzeugten Metadaten zugegriffen werden kann. `MetadataGenerator` ist dabei eine Abwandlung der Klasse `JhoveBase`, die auf die Anforderungen von koLibRI angepasst und erweitert wurde.

JHOVE bietet seinerseits ebenfalls ein offenes Modulkonzept, das die Unterstützung zukünftiger Dateiformate gewährleisten soll. Im Prinzip kann JHOVE also technische Metadaten für jegliche Dateiformate erzeugen, vorausgesetzt ein entsprechendes JHOVE Modul existiert.

In der verwendeten Version werden folgende Dateiformate unterstützt:

- AIFF-hul: Audio Interchange File Format
  - AIFF 1.3
  - AIFF-C
- ASCII-hul: ASCII-encoded text
  - ANSI X3.4-1986
  - ECMA-6
  - ISO 646:1991
- BYTESTREAM: Arbitrary bytestreams (always well-formed and valid)
- GIF-hul: Graphics Exchange Format (GIF)
  - GIF 87a
  - GIF 89a
- HTML-hul: Hypertext Markup Language (HTML)
  - HTML 3.2
  - HTML 4.0

  - HTML 4.01
  - XHTML 1.0 and 1.1

- JPEG-hul: Joint Photographic Experts Group (JPEG) raster images

  - JPEG (ISO/IEC 10918-1:1994)
  - JPEG File Interchange Format (JFIF) 1.2
  - Exif 2.0, 2.1 (JEIDA-49-1998), and 2.2 (JEITA CP-3451)
  - Still Picture Interchange File Format (SPIFF, ISO/IEC 10918-3:1997)
  - JPEG Tiled Image Pyramid (JTIP, ISO/IEC 10918-3:1997)
  - JPEG-LS (ISO/IEC 14495)
  - JPEG2000-hul: JPEG 2000
  - JP2 profile (ISO/IEC 15444-1:2000 / ITU-T Rec. T.800 (2000))
  - JPX profile (ISO/IEC 15444-2:2004)

- PDF-hul: Page Description Format (PDF)

  - PDF 1.0 through 1.6

    - Pre-press data exchange
    - PDF/X-1 (ISO 15930-1:2001)
    - PDF/X-1a (ISO 15930-4:2003)
    - PDF/X-2 (ISO 15390-5:2003)
    - PDF/X-3 (ISO 15930-6:2003)

  - Tagged PDF
  - Linearized PDF
  - PDF/A-1 (ISO/DIS 19005-1)

- TIFF-hul: Tagged Image File Format (TIFF) raster images

  - TIFF 4.0, 5.0, and 6.0
  - Baseline 6.0 Class B, G, P, and R
  - Extension Class Y
  - TIFF/IT (ISO 12639:2003)

    - File types CT, LW, HC, MP, BP, BL, and FP, and conformance levels P1 and P2

  - TIFF/EP (ISO 12234-2:2001)
  - Exif 2.0, 2.1 (JEIDA-49-1998), and 2.2 (JEITA CP-3451)
  - GeoTIFF 1.0
  - TIFF-FX (RFC 2301)
  - Profiles C, F, J, L, M, and S

  - Class F (RFC 2306)
  - RFC 1314
  - DNG (Adobe Digital Negative)
- UTF8-hul: UTF-8 encoded text
- WAVE: Audio for Windows
  - PCMWAVEFORMAT
  - WAVEFORMATEX
  - WAVEFORMATEXTENSION
  - Broadcast Wave Format (EBU N22-1997) version 0 and 1
- XML-hul: Extensible Markup Language (XML)
  - XML 1.0

Im Rahmen der Entwicklung von koLibRI wurden zusätzlich zwei JHOVE-Module entwickelt, welche ebenfalls Bestandteil des koLibRI Releases sind. Ein Modul behaldelt *Disc Images nach dem ISO-Standard 9660* (mit Unterstützung für Rock Ridge Dateinamenerweiterungen), das andere untersucht PostScript Dateien.

Die Konfiguration der JHOVE Komponente wird über die JHOVE Konfigurationsdatei vorgenommen, deren Pfad in der Konfigurationsdatei der koLibRI Software angegeben werden muss. Im vorliegenden Release liegt eine Beispieldatei im Ordner `/config` und trägt den Namen `jhove.conf`. In dieser Datei befinden sich mehrere Einträge für JHOVE Module, die wie folgt aussehen:

```
<module>
  <class>de.langzeitarchivierung.kopal.jhove.Iso9660Module</class>
</module>
```

Alle angegebenen Module werden in Reihe aufgerufen, bis ein auf die aktuell bearbeitete Datei passendes Modul ermittelt werden konnte. Dabei ist die Reihenfolge ausschlaggebend. Spezialisierende Module sollten über generischen angeordnet sein. Ein Beispiel: Eine HTML-Datei besteht aus einer bestimmten Folge von Textzeichen, würde jedoch vom Modul für ASCII-Text als Text-Datei identifiziert werden, wenn es noch vor dem HTML-Modul aufgerufen würde.

Das PostScript-Modul kann eine Datei zusätzlich mit Hilfe von Ghostscript [24] validieren, ansonsten wird sie nur auf Well-Formedness untersucht. Hierfür muss der Pfad zur ausführbaren Ghostscript-Applikation in der JHOVE Konfigurationsdatei an folgender Stelle angegeben sein (zwei Beispiel-Einträge sind auskommentiert):

```
<module>
  <class>de.langzeitarchivierung.kopal.jhove.PsModule</class>
  <!--param>/usr/local/bin/gs</param-->
  <!--param>C:\\Programme\\gs\\gs8.54\\bin\\</param-->
</module>
```

Die Aufruf-Parameter von Ghostscript müssen in der Klasse `PSModule` konfiguriert werden.

Zur Einbindung in koLibRI werden die JHOVE JAR-Dateien `jhove.jar`, `jhove-handler.jar` und `jhove-module.jar` im Verzeichnis `/lib` von koLibRI verwendet. Diese Dateien können einfach mit denen eines neuen Maintainance Releases von JHOVE ausgetauscht werden, um von eventuellen Bugfixes und Verbesserungen zu profitieren. Lediglich bei einer Änderung der internen Struktur von JHOVE könnten eventuelle Anpassungen am Quellcode des MetadataGenerators notwenig werden.

10.2 Die WSDL-Datei des koLibRI Web-Services

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions name="kolibri_web_services"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:tns="http://kopal.langzeitarchivierung.de/ws/wsdl2java/"
targetNamespace="http://kopal.langzeitarchivierung.de/ws/wsdl2java/">

<wsdl:documentation>
/**
*  Web Services Description Language (WSDL) file for
*  kopal Library for Retrieval and Ingest (koLibRI) Web Services.
*
*  Please see the koLibRI documentation and source code for details.
*  koLibRI is free software under GNU Public License and sponsored by
*  Federal Republic of Germany, Federal Ministry of Education and Research
*
*  Copyright: Project kopal http://kopal.langzeitarchivierung.de
*  Author: Kadir Karaca Koçer, German National Library
*  June 2007
*/
</wsdl:documentation>

<!-- ***** T Y P E S ***** -->
```

```
<wsdl:types>
<xsd:schema elementFormDefault="qualified"
targetNamespace="http://kopal.langzeitarchivierung.de/ws/wsdl2java/">

<!-- Type USER -->
<xsd:complexType name="userType">
<xsd:sequence>
<xsd:element name="userName" type="xsd:token" />
<xsd:element name="password" type="xsd:token" />
<xsd:element name="institution" type="xsd:int" />
</xsd:sequence>
</xsd:complexType>

<!-- Type Custom XML -->
<xsd:complexType name="customXMLType">
<xsd:sequence>
<xsd:any namespace="##any" processContents="skip" />
</xsd:sequence>
</xsd:complexType>

<!-- Ingest -->
<xsd:element name="getFileFormatListElem">
<xsd:complexType>
<xsd:sequence>
<xsd:element name="user" type="tns:userType" />
</xsd:sequence>
</xsd:complexType>
</xsd:element>

<xsd:element name="reloadFileFormatListElem">
<xsd:complexType>
<xsd:sequence>
<xsd:element name="user" type="tns:userType" />
</xsd:sequence>
</xsd:complexType>
</xsd:element>

<!-- Element OBJECT TO INGEST -->
<xsd:element name="ingestDocumentElem">
<xsd:complexType>
<xsd:sequence>
<xsd:element name="user" type="tns:userType" />
<xsd:element name="policy" type="xsd:token" />
<xsd:element name="metadata" minOccurs="1" maxOccurs="6" type="tns:customXMLType" />
```

```
</xsd:sequence>
</xsd:complexType>
</xsd:element>

<xsd:element name="ingestSIPElem">
<xsd:complexType>
<xsd:sequence>
<xsd:element name="user" type="tns:userType" />
<xsd:element name="urn" type="xsd:token" />
<xsd:element name="linkToSIP" type="xsd:token" />
</xsd:sequence>
</xsd:complexType>
</xsd:element>

<xsd:element name="blockNewIngestRequestsElem">
<xsd:complexType>
<xsd:sequence>
<xsd:element name="user" type="tns:userType" />
</xsd:sequence>
</xsd:complexType>
</xsd:element>

<xsd:element name="allowIngestRequestsElem">
<xsd:complexType>
<xsd:sequence>
<xsd:element name="user" type="tns:userType" />
</xsd:sequence>
</xsd:complexType>
</xsd:element>

<xsd:element name="getIngestStatisticsElem">
<xsd:complexType>
<xsd:sequence>
<xsd:element name="user" type="tns:userType" />
</xsd:sequence>
</xsd:complexType>
</xsd:element>

<xsd:element name="resetIngestStatisticsElem">
<xsd:complexType>
<xsd:sequence>
<xsd:element name="user" type="tns:userType" />
</xsd:sequence>
</xsd:complexType>
```

```
</xsd:element>


<!-- OBJECT TO RETRIEVE -->
<xsd:element name="getMETSElem">
<xsd:complexType>
<xsd:sequence>
<xsd:element name="user" type="tns:userType" />
<xsd:element name="objectId" type="xsd:token" />
</xsd:sequence>
</xsd:complexType>
</xsd:element>

<xsd:element name="getFileElem">
<xsd:complexType>
<xsd:sequence>
<xsd:element name="user" type="tns:userType" />
<xsd:element name="objectId" type="xsd:token" />
<xsd:element name="fileName" type="xsd:token" />
</xsd:sequence>
</xsd:complexType>
</xsd:element>

<xsd:element name="getDIPElem">
<xsd:complexType>
<xsd:sequence>
<xsd:element name="user" type="tns:userType" />
<xsd:element name="objectId" type="xsd:token" />
</xsd:sequence>
</xsd:complexType>
</xsd:element>

<xsd:element name="getDIPExternalElem">
<xsd:complexType>
<xsd:sequence>
<xsd:element name="user" type="tns:userType" />
<xsd:element name="objectId" type="xsd:token" />
</xsd:sequence>
</xsd:complexType>
</xsd:element>

<xsd:element name="getModifiedSinceElem">
<xsd:complexType>
<xsd:sequence>
```

```
<xsd:element name="user" type="tns:userType" />
<xsd:element name="modificationdate" type="xsd:dateTime" nillable="true"/>
</xsd:sequence>
</xsd:complexType>
</xsd:element>

<xsd:element name="getHistoryElem">
<xsd:complexType>
<xsd:sequence>
<xsd:element name="user" type="tns:userType" />
<xsd:element name="objectId" type="xsd:token" />
</xsd:sequence>
</xsd:complexType>
</xsd:element>

<xsd:element name="blockNewRetrievalRequestsElem">
<xsd:complexType>
<xsd:sequence>
<xsd:element name="user" type="tns:userType" />
</xsd:sequence>
</xsd:complexType>
</xsd:element>

<xsd:element name="allowRetrievalRequestsElem">
<xsd:complexType>
<xsd:sequence>
<xsd:element name="user" type="tns:userType" />
</xsd:sequence>
</xsd:complexType>
</xsd:element>

<xsd:element name="getRetrievalStatisticsElem">
<xsd:complexType>
<xsd:sequence>
<xsd:element name="user" type="tns:userType" />
</xsd:sequence>
</xsd:complexType>
</xsd:element>

<xsd:element name="resetRetrievalStatisticsElem">
<xsd:complexType>
<xsd:sequence>
<xsd:element name="user" type="tns:userType" />
</xsd:sequence>
```

```
</xsd:complexType>
</xsd:element>

<xsd:element name="initCacheElem">
<xsd:complexType>
<xsd:sequence>
<xsd:element name="user" type="tns:userType" />
<xsd:element name="rootdir" type="xsd:token" />
</xsd:sequence>
</xsd:complexType>
</xsd:element>

<xsd:element name="eraseOldCacheEntriesElem">
<xsd:complexType>
<xsd:sequence>
<xsd:element name="user" type="tns:userType" />
<xsd:element name="timelimit" type="xsd:int" />
</xsd:sequence>
</xsd:complexType>
</xsd:element>

<xsd:element name="deleteCacheElem">
<xsd:complexType>
<xsd:sequence>
<xsd:element name="user" type="tns:userType" />
</xsd:sequence>
</xsd:complexType>
</xsd:element>

<!-- Change Password -->
<xsd:element name="changePasswordElem">
<xsd:complexType>
<xsd:sequence>
<xsd:element name="user" type="tns:userType" />
<xsd:element name="newPassword" type="xsd:token" />
</xsd:sequence>
</xsd:complexType>
</xsd:element>

<!-- Parameters -->
<xsd:element name="setParametersElem">
<xsd:complexType>
<xsd:sequence>
<xsd:element name="user" type="tns:userType" />
```

```
<xsd:element name="diasRetrieverURL" type="xsd:token" minOccurs="0" maxOccurs="1" />
<xsd:element name="diasLoaderURL" type="xsd:token" minOccurs="0" maxOccurs="1" />
<xsd:element name="fileFormatListURL" type="xsd:token" minOccurs="0" maxOccurs="1" />
<xsd:element name="cacheRootDir" type="xsd:token" minOccurs="0" maxOccurs="1" />
<xsd:element name="protocolRetrievalWS" type="xsd:token" minOccurs="0" maxOccurs="1" />
<xsd:element name="urlRetrievalWS" type="xsd:token" minOccurs="0" maxOccurs="1" />
<xsd:element name="tempDir" type="xsd:token" minOccurs="0" maxOccurs="1" />
</xsd:sequence>
</xsd:complexType>
</xsd:element>

<!-- Type RESPONSE -->
<xsd:element name="responseElem">
<xsd:complexType>
<xsd:sequence>
<xsd:element name="statusCode" type="xsd:int" />
<xsd:element name="responseText" type="xsd:string" />
<xsd:element name="diasId" type="xsd:string" minOccurs="0" maxOccurs="1" />
<xsd:element name="waitTime" type="xsd:int"  minOccurs="0" maxOccurs="1" />
<xsd:element name="linkToFile" type="xsd:string" minOccurs="0" maxOccurs="1" />
<xsd:element name="extraInfo" type="xsd:string" minOccurs="0" maxOccurs="1" />
<xsd:element name="errorMessage" type="xsd:string" minOccurs="0" maxOccurs="1" />
<xsd:element name="customdata" type="tns:customXMLType" minOccurs="0" maxOccurs="1" />
</xsd:sequence>
</xsd:complexType>
</xsd:element>
</xsd:schema>
</wsdl:types>


<!-- ***** M E S S A G E S ***** -->
<!-- Ingest -->
<wsdl:message name="getFileFormatListMsg">
<wsdl:part name="request" element="tns:getFileFormatListElem" />
</wsdl:message>

<wsdl:message name="reloadFileFormatListMsg">
<wsdl:part name="request" element="tns:reloadFileFormatListElem" />
</wsdl:message>

<wsdl:message name="ingestDocumentMsg">
<wsdl:part name="request" element="tns:ingestDocumentElem" />
</wsdl:message>
```

```
<wsdl:message name="ingestSIPMsg">
<wsdl:part name="request" element="tns:ingestSIPElem" />
</wsdl:message>

<wsdl:message name="blockNewIngestRequestsMsg">
<wsdl:part name="request" element="tns:blockNewIngestRequestsElem" />
</wsdl:message>

<wsdl:message name="allowIngestRequestsMsg">
<wsdl:part name="request" element="tns:allowIngestRequestsElem" />
</wsdl:message>

<wsdl:message name="getIngestStatisticsMsg">
<wsdl:part name="request" element="tns:getIngestStatisticsElem" />
</wsdl:message>

<wsdl:message name="resetIngestStatisticsMsg">
<wsdl:part name="request" element="tns:resetIngestStatisticsElem" />
</wsdl:message>

<!-- Retrieval -->
<wsdl:message name="getMETSMsg">
<wsdl:part name="request" element="tns:getMETSElem" />
</wsdl:message>

<wsdl:message name="getFileMsg">
<wsdl:part name="request" element="tns:getFileElem" />
</wsdl:message>

<wsdl:message name="getDIPMsg">
<wsdl:part name="request" element="tns:getDIPElem" />
</wsdl:message>

<wsdl:message name="getDIPExternalMsg">
<wsdl:part name="request" element="tns:getDIPExternalElem" />
</wsdl:message>

<wsdl:message name="getModifiedSinceMsg">
<wsdl:part name="request" element="tns:getModifiedSinceElem" />
</wsdl:message>

<wsdl:message name="getHistoryMsg">
<wsdl:part name="request" element="tns:getHistoryElem" />
</wsdl:message>
```

```
<wsdl:message name="blockNewRetrievalRequestsMsg">
<wsdl:part name="request" element="tns:blockNewRetrievalRequestsElem" />
</wsdl:message>

<wsdl:message name="allowRetrievalRequestsMsg">
<wsdl:part name="request" element="tns:allowRetrievalRequestsElem" />
</wsdl:message>

<wsdl:message name="getRetrievalStatisticsMsg">
<wsdl:part name="request" element="tns:getRetrievalStatisticsElem" />
</wsdl:message>

<wsdl:message name="resetRetrievalStatisticsMsg">
<wsdl:part name="request" element="tns:resetRetrievalStatisticsElem" />
</wsdl:message>

<wsdl:message name="initCacheMsg">
<wsdl:part name="request" element="tns:initCacheElem" />
</wsdl:message>

<wsdl:message name="eraseOldCacheEntriesMsg">
<wsdl:part name="request" element="tns:eraseOldCacheEntriesElem" />
</wsdl:message>

<wsdl:message name="deleteCacheMsg">
<wsdl:part name="request" element="tns:deleteCacheElem" />
</wsdl:message>

<!-- Change password -->
<wsdl:message name="changePasswordMsg">
<wsdl:part name="request" element="tns:changePasswordElem" />
</wsdl:message>

<!-- setParameters -->
<wsdl:message name="setParametersMsg">
<wsdl:part name="request" element="tns:setParametersElem" />
</wsdl:message>

<!-- Response -->
<wsdl:message name="diasResponse">
<wsdl:part name="response" element="tns:responseElem" />
</wsdl:message>
```

```
<!-- ***** P O R T S ***** -->
<wsdl:portType name="KolibriServiceSOAP">
<!-- Ingest -->
<wsdl:operation name="getFileFormatList">
<wsdl:input message="tns:getFileFormatListMsg" />
<wsdl:output message="tns:diasResponse" />
</wsdl:operation>

<wsdl:operation name="reloadFileFormatList">
<wsdl:input message="tns:reloadFileFormatListMsg" />
<wsdl:output message="tns:diasResponse" />
</wsdl:operation>

<wsdl:operation name="ingestDocument">
<wsdl:input message="tns:ingestDocumentMsg" />
<wsdl:output message="tns:diasResponse" />
</wsdl:operation>

<wsdl:operation name="ingestSIP">
<wsdl:input message="tns:ingestSIPMsg" />
<wsdl:output message="tns:diasResponse" />
</wsdl:operation>

<wsdl:operation name="blockNewIngestRequests">
<wsdl:input message="tns:blockNewIngestRequestsMsg"/>
<wsdl:output message="tns:diasResponse"/>
</wsdl:operation>

<wsdl:operation name="allowIngestRequests">
<wsdl:input message="tns:allowIngestRequestsMsg"/>
<wsdl:output message="tns:diasResponse"/>
</wsdl:operation>

<wsdl:operation name="getIngestStatistics">
<wsdl:input message="tns:getIngestStatisticsMsg"/>
<wsdl:output message="tns:diasResponse"/>
</wsdl:operation>

<wsdl:operation name="resetIngestStatistics">
<wsdl:input message="tns:resetIngestStatisticsMsg"/>
<wsdl:output message="tns:diasResponse"/>
</wsdl:operation>
```

```
<!-- Retrieval -->
<wsdl:operation name="getMETS">
<wsdl:input message="tns:getMETSMsg" />
<wsdl:output message="tns:diasResponse" />
</wsdl:operation>

<wsdl:operation name="getFile">
<wsdl:input message="tns:getFileMsg" />
<wsdl:output message="tns:diasResponse"/>
</wsdl:operation>

<wsdl:operation name="getDIP">
<wsdl:input message="tns:getDIPMsg" />
<wsdl:output message="tns:diasResponse"/>
</wsdl:operation>

<wsdl:operation name="getDIPExternal">
<wsdl:input message="tns:getDIPExternalMsg" />
<wsdl:output message="tns:diasResponse"/>
</wsdl:operation>

<wsdl:operation name="getModifiedSince">
<wsdl:input message="tns:getModifiedSinceMsg" />
<wsdl:output message="tns:diasResponse"/>
</wsdl:operation>

<wsdl:operation name="getHistory">
<wsdl:input message="tns:getHistoryMsg" />
<wsdl:output message="tns:diasResponse"/>
</wsdl:operation>

<wsdl:operation name="blockNewRetrievalRequests">
<wsdl:input message="tns:blockNewRetrievalRequestsMsg"/>
<wsdl:output message="tns:diasResponse"/>
</wsdl:operation>

<wsdl:operation name="allowRetrievalRequests">
<wsdl:input message="tns:allowRetrievalRequestsMsg"/>
<wsdl:output message="tns:diasResponse"/>
</wsdl:operation>

<wsdl:operation name="getRetrievalStatistics">
<wsdl:input message="tns:getRetrievalStatisticsMsg"/>
<wsdl:output message="tns:diasResponse"/>
```

```
</wsdl:operation>

<wsdl:operation name="resetRetrievalStatistics">
<wsdl:input message="tns:resetRetrievalStatisticsMsg"/>
<wsdl:output message="tns:diasResponse"/>
</wsdl:operation>

<wsdl:operation name="initCache">
<wsdl:input message="tns:initCacheMsg"/>
<wsdl:output message="tns:diasResponse"/>
</wsdl:operation>

<wsdl:operation name="eraseOldCacheEntries">
<wsdl:input message="tns:eraseOldCacheEntriesMsg"/>
<wsdl:output message="tns:diasResponse"/>
</wsdl:operation>

<wsdl:operation name="deleteCache">
<wsdl:input message="tns:deleteCacheMsg"/>
<wsdl:output message="tns:diasResponse"/>
</wsdl:operation>

<!-- Administration -->
<wsdl:operation name="setParameters">
<wsdl:input message="tns:setParametersMsg" />
<wsdl:output message="tns:diasResponse" />
</wsdl:operation>

<wsdl:operation name="changePassword">
<wsdl:input message="tns:changePasswordMsg"/>
<wsdl:output message="tns:diasResponse"/>
</wsdl:operation>
</wsdl:portType>


<!-- ***** B I N D I N G S  ***** -->
<wsdl:binding name="bindingKolibri" type="tns:KolibriServiceSOAP">
<soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http" />

<wsdl:operation name="getFileFormatList">
<soap:operation soapAction="http://bernstein.d-nb.de/kolibri#getFileFormatList" />
<wsdl:input><soap:body use="literal" /></wsdl:input>
<wsdl:output><soap:body use="literal" /></wsdl:output>
</wsdl:operation>
```

```
<wsdl:operation name="reloadFileFormatList">
<soap:operation soapAction="http://bernstein.d-nb.de/kolibri#reloadFileFormatList" />
<wsdl:input><soap:body use="literal" /></wsdl:input>
<wsdl:output><soap:body use="literal" /></wsdl:output>
</wsdl:operation>

<wsdl:operation name="ingestDocument">
<soap:operation soapAction="http://bernstein.d-nb.de/kolibri#ingestDocument" />
<wsdl:input><soap:body use="literal" /></wsdl:input>
<wsdl:output><soap:body use="literal" /></wsdl:output>
</wsdl:operation>

<wsdl:operation name="ingestSIP">
<soap:operation soapAction="http://bernstein.d-nb.de/kolibri#ingestSIP" />
<wsdl:input><soap:body use="literal" /></wsdl:input>
<wsdl:output><soap:body use="literal" /></wsdl:output>
</wsdl:operation>

<wsdl:operation name="blockNewIngestRequests">
<soap:operation soapAction="http://bernstein.d-nb.de/kolibri#blockNewIngestRequests" />
<wsdl:input><soap:body use="literal" /></wsdl:input>
<wsdl:output><soap:body use="literal" /></wsdl:output>
</wsdl:operation>

<wsdl:operation name="allowIngestRequests">
<soap:operation soapAction="http://bernstein.d-nb.de/kolibri#allowIngestRequests" />
<wsdl:input><soap:body use="literal" /></wsdl:input>
<wsdl:output><soap:body use="literal" /></wsdl:output>
</wsdl:operation>

<wsdl:operation name="getIngestStatistics">
<soap:operation soapAction="http://bernstein.d-nb.de/kolibri#getIngestStatistics" />
<wsdl:input><soap:body use="literal" /></wsdl:input>
<wsdl:output><soap:body use="literal" /></wsdl:output>
</wsdl:operation>

<wsdl:operation name="resetIngestStatistics">
<soap:operation soapAction="http://bernstein.d-nb.de/kolibri#resetIngestStatistics" />
<wsdl:input><soap:body use="literal" /></wsdl:input>
<wsdl:output><soap:body use="literal" /></wsdl:output>
</wsdl:operation>

<!-- Retrieval -->
```

```
<wsdl:operation name="getMETS">
<soap:operation soapAction="http://bernstein.d-nb.de/kolibri#getMETS" />
<wsdl:input><soap:body use="literal" /></wsdl:input>
<wsdl:output><soap:body use="literal" /></wsdl:output>
</wsdl:operation>

<wsdl:operation name="getFile">
<soap:operation soapAction="http://bernstein.d-nb.de/kolibri#getFile" />
<wsdl:input><soap:body use="literal" /></wsdl:input>
<wsdl:output><soap:body use="literal" /></wsdl:output>
</wsdl:operation>

<wsdl:operation name="getDIP">
<soap:operation soapAction="http://bernstein.d-nb.de/kolibri#getDIP" />
<wsdl:input><soap:body use="literal" /></wsdl:input>
<wsdl:output><soap:body use="literal" /></wsdl:output>
</wsdl:operation>

<wsdl:operation name="getDIPExternal">
<soap:operation soapAction="http://bernstein.d-nb.de/kolibri#getDIPExternal" />
<wsdl:input><soap:body use="literal" /></wsdl:input>
<wsdl:output><soap:body use="literal" /></wsdl:output>
</wsdl:operation>

<wsdl:operation name="getModifiedSince">
<soap:operation soapAction="http://bernstein.d-nb.de/kolibri#getModifiedSince" />
<wsdl:input><soap:body use="literal" /></wsdl:input>
<wsdl:output><soap:body use="literal" /></wsdl:output>
</wsdl:operation>

<wsdl:operation name="getHistory">
<soap:operation soapAction="http://bernstein.d-nb.de/kolibri#getHistory" />
<wsdl:input><soap:body use="literal" /></wsdl:input>
<wsdl:output><soap:body use="literal" /></wsdl:output>
</wsdl:operation>

<wsdl:operation name="blockNewRetrievalRequests">
<soap:operation soapAction="http://bernstein.d-nb.de/kolibri#blockNewRetrievalRequests" />
<wsdl:input><soap:body use="literal" /></wsdl:input>
<wsdl:output><soap:body use="literal" /></wsdl:output>
</wsdl:operation>

<wsdl:operation name="allowRetrievalRequests">
<soap:operation soapAction="http://bernstein.d-nb.de/kolibri#allowRetrievalRequests" />
```

```
<wsdl:input><soap:body use="literal" /></wsdl:input>
<wsdl:output><soap:body use="literal"/></wsdl:output>
</wsdl:operation>

<wsdl:operation name="getRetrievalStatistics">
<soap:operation soapAction="http://bernstein.d-nb.de/kolibri#getRetrievalStatistics" />
<wsdl:input><soap:body use="literal" /></wsdl:input>
<wsdl:output><soap:body use="literal" /></wsdl:output>
</wsdl:operation>

<wsdl:operation name="resetRetrievalStatistics">
<soap:operation soapAction="http://bernstein.d-nb.de/kolibri#resetRetrievalStatistics" />
<wsdl:input><soap:body use="literal" /></wsdl:input>
<wsdl:output><soap:body use="literal" /></wsdl:output>
</wsdl:operation>

<wsdl:operation name="initCache">
<soap:operation soapAction="http://bernstein.d-nb.de/kolibri#initCache" />
<wsdl:input><soap:body use="literal" /></wsdl:input>
<wsdl:output><soap:body use="literal" /></wsdl:output>
</wsdl:operation>

<wsdl:operation name="eraseOldCacheEntries">
<soap:operation soapAction="http://bernstein.d-nb.de/kolibri#eraseOldCacheEntries" />
<wsdl:input><soap:body use="literal" /></wsdl:input>
<wsdl:output><soap:body use="literal" /></wsdl:output>
</wsdl:operation>

<wsdl:operation name="deleteCache">
<soap:operation soapAction="http://bernstein.d-nb.de/kolibri#deleteCache" />
<wsdl:input><soap:body use="literal" /></wsdl:input>
<wsdl:output><soap:body use="literal" /></wsdl:output>
</wsdl:operation>

<wsdl:operation name="setParameters">
<soap:operation soapAction="http://bernstein.d-nb.de/kolibri#setParameters" />
<wsdl:input><soap:body use="literal" /></wsdl:input>
<wsdl:output><soap:body use="literal" /></wsdl:output>
</wsdl:operation>

<wsdl:operation name="changePassword">
<soap:operation soapAction="http://bernstein.d-nb.de/kolibri#changePassword" />
<wsdl:input><soap:body use="literal" /></wsdl:input>
<wsdl:output><soap:body use="literal" /></wsdl:output>
```

```
</wsdl:operation>
</wsdl:binding>


<!-- ***** S E R V I C E S  ***** -->
<wsdl:service name="kolibriWebService">
<wsdl:port name="kolibri_ws" binding="tns:bindingKolibri">
<soap:address location="http://bernstein.d-nb.de/kolibri/" />
</wsdl:port>
</wsdl:service>
</wsdl:definitions>
```

10.3 Direkter Zugriff auf die Schnittstellen des DIAS

Für einen direkten Zugriff auf die Access- und Ingest-Schnittstellen von DIAS können die Kommandozeilentools `DiasAccess` und `DiasIngest` verwendet werden. Sie sind durch Main-Methoden in den Klassen `retrieval.DiasAccess` und `ingest.DiasIngest` realisiert. Eine Kenntnis der SIP- und DIP-Interface-Specification [10] [11] wird vorausgesetzt.

Es sei an dieser Stelle nochmal erwähnt, dass wie für koLibRI ingesamt auch für diese beiden Tools üblicherweise die Angabe einer Keystore-Datei notwendig ist, da der Zugang zum DIAS über verschlüsselte Verbindung erfolgen sollte. Eine `known_hosts` Datei muss ebenfalls korrekt angegeben sein.

DiasIngest

```
java -jar DiasIngest.jar -h:
  -hp, --show-properties    Print the system properties and continue.
   -a, --address            The server address.
   -f, --file               The file to submit.
   -h, --help               Print this dialog and exit.
   -n, --port               The server port.
   -p, --password           The CMPassword of the CMUser.
   -t, --testdias           Print dias responses.
   -u, --user               The submitting CMUser.
```

DiasAccess

```
java -jar DiasAccess.jar -h:
   -d, --dip-format         The requested dip format [zip|tar|tar.gz].
   -r, --response-format    The requested response format [xml|html].
   -x, --ext-id             The requested external id.
```

```
 -i, --int-id              The requested internal id.
-hp, --show-properties     Print the system properties and continue.
 -t, --request-type        The type of request [metadata|fullmetadata|asset].
 -a, --address             The server address.
 -f, --file                Parse a file as a Dias response and prints it.
 -g, --download            If no file is specified the file in the dias response is
                             downloaded and saved in the current directory. Else the
                             specified file is downloaded.
 -h, --help                Print this dialog and exit.
 -l, --list                Returns a list of all metadata sets for an external id.
 -n, --port                The server port.
 -p, --print               Prints the dias response.
 -u, --unsecure            Use unsecure access to dias.
```

10.4 Der TIFF Image Metadata Processor

Der TiffIMP kann zum Anzeigen, zur Validation wie auch zur Korrektur von TIFF Header-Metadaten genutzt werden. Diese Klasse ist sowohl über die API als auch als Kommandozeilen-Tool nutzbar. Sie nutzt JHOVE zur Validation der TIFF Images. Einzelheiten über diese Klasse sind in der Javadoc-Dokumentation oder in den Kommentaren des Quellcodes zu erfahren.

Der TiffIMP als Kommandozeilen-Tool

```
usage: tiffimp [options]
    -h, --help                      Very surprising: A help message :-)
    -r, --rewrite-tiff-header       The TIFF header metadata will be rewritten. Please
                                      notice that only the following tasks are able to be
                                      repaired yet:
                                      (a) Word-alignedness of all TIFF ASCII tags.
                                      (b) PageNumber 297 0x0129 set to the valid
                                      TIFF_SHORT count of 2 bytes.
    -i, --input-filename <file>     The filename of the TIFF image to process.
    -o, --output-filename <file>    The filename of the corrected TIFF image. Default is
                                      _cor.tif.
    -j, --jhove-xml-output          Outputs the Jhove XML output using the TIFF-hul module.
    -c, --validate-tiff-image       Validates the TIFF image according to TIFF specification
                                      compliance using the Jhove
                                      (see http://hul.harvard.edu/jhove).
    -s, --show-header-metadata      Outputs the TIFF image header metadata tags and their
                                      content to standard out. This is the default option.
    -f, --force-overwrite           Forces the input image file to be overwritten if header
                                      metadata shall be corrected.
```

```
-n, --rewrite-if-not-valid     Only rewrites the TIFF header if the image file is not
                                 wellformed or not valid according to the Jhove.
-q, --quiet                    Run tiffimp in quiet mode.
-v, --verbose                  Run tiffimp in verbose mode.
```

10.5 Fehlercodes bei Programmende

Allgemein

(0) Reguläres Programmende

kopal.WorkflowTool, kopal.retrieval.DiasAccess

(1) Kommandozeilenparameter falsch – Commandline and its arguments are not correct
Konfiguration des WorkflowTools nicht korrekt – Configuration invalid for Workflow-Tool
Konfiguration eines ProcessStarters nicht korrekt – Configuration invalid for Process-Starter.

(2) Prozess-Starter-Initialisierung schlug fehl – Could not load and initialize process starters. Please check the commandline flag -p or the DefaultProcessStarter field in the main config file.

(3) Programm wurde beendet und es gab Fehler bei der Bearbeitung der Listenelemente – Not everything has finished correctly! Please check the logfile.

(9) Logfile konnte nicht angelegt werden – Could not create logfile.

(12) Fehler beim Initialisieren der Datenbank – Database initialization failed.

kopal.processstarter.MonitorHotfolderBase

(6) Der angegebene Hotfolder Pfad existiert nicht – The given hotfolder path does not exist or is not a directory.

(14) Datei ist keine Datei und kein Verzeichnis – Error processing current File. No file or no directory.

kopal.Policy

(7) Konfigurationsdatei konnte nicht geparst werden – Exception while parsing the policy file.

kopal.util.FormatRegistry.java

(10) Keine Backup-Datei für die Format-Registry vorhanden – No backup file for format registry found! Formats can not be identified.

(13) Fehler beim Parsing der Backup-Datei für die Format-Registry – Error parsing the DIAS format registry backup file.

(16) Fehler beim Zugriff auf die Backup-Datei für die Format-Registry – Error accessing the format registry backup file.

kopal.util.kopalXMLParser, kopal.util.HTMLUtils

(11) XML Parse Error – Parse error in XML file, Parse error in XML string.

kopal.processstarter.MonitorHttpLocationBase

(15) Fehler bei Zugriff auf eine URL – Error accessing URL.

10.6 Fehlerbehandlung und Loglevel

`SEVERE` Das Programm muss abgebrochen werden, da ein schwerwiegender Fehler vorliegt. *Beispiel*: Eine Konfigurations-Datei kann nicht gefunden werden.

`WARNING` Warnungen werden bei Fehlern mitgeloggt, bei denen kein Programmabbruch von Nöten ist. *Beispiel*: Die Bearbeitung eines Listenelements wird abgebrochen, weil das Modul seinen Status auf `ERROR` gesetzt hat, das nächste Element wird bearbeitet.

`INFO` Alle für den Nutzer interessanten Vorgänge werden unter INFO geloggt. *Beispiel*: Das Starten des Server, Erfolgreiches Auslesen einer Konfigurations-Datei, Bearbeitungsschritte der ActionModules, uswusf.

`FINE` Mit `FINE` werden alle Meldungen geloggt, die für den Nutzer wenig Bedeutung haben und die dem Debuggen dienen. *Beispiel*: Interne Meldungen der Listen-Bearbeitungs-Methode `WorkflowTool.process()`.

`FINER/FINEST` Mit `FINER` und `FINEST` könnten allerfeinste Debug-Meldungen mitgeloggt werden, die selbst für das Debuggen selten benötigt werden. *Beispiel*: Meldungen über `notify()` und `wait()` zum Debuggen von Threads.

`ALL/OFF` Alle Meldungen/keine Meldungen werden geloggt.

Literatur

[1] DIAS (Digital Information Archiving System)
http://www-5.ibm.com/nl/dias/

[2] Reference Model for an Open Archival Information System (OAIS)
http://ssdoo.gsfc.nasa.gov/nost/isoas/ref_model.html

[3] Uniform Resource Name; vgl. dazu das Projekt EPICUR
http://www.persistent-identifier.de

[4] METS (Metadata Encoding & Transmission Standard)
http://www.loc.gov/standards/mets/

[5] LMER (Langzeitarchivierungsmetadaten für elektronische Ressourcen)
http://d-nb.de/standards/lmer/lmer.htm

[6] kopal – Kooperativer Aufbau eines Langzeitarchivs digitaler Informationen
http://kopal.langzeitarchivierung.de

[7] nestor – Kompetenznetzwerk Langzeitarchivierung
http://www.langzeitarchivierung.de

[8] JHOVE (JSTOR/Harvard Object Validation Environment)
http://hul.harvard.edu/jhove/

[9] The Free Software Foundation
http://www.fsf.org

[10] DIAS SIP Interface Specification
http://kopal.langzeitarchivierung.de/downloads/kopal_DIAS_SIP_Interface_Specification.pdf

[11] DIAS DIP Interface Specification
http://kopal.langzeitarchivierung.de/downloads/kopal_DIAS_DIP_Interface_Specification.pdf

[12] Hibernate – Relational Persistence for Java
http://www.hibernate.org

[13] SOAP (Simple Object Access Protocol)
http://www.w3.org/TR/soap/

[14] AXIS2
http://ws.apache.org/axis2/

[15] Apache Tomcat
http://tomcat.apache.org

[16] Dublin Core Metadata Initiative
http://dublincore.org

[17] WSDL (Web Service Description Language)
http://www.w3.org/TR/wsdl/

[18] Deutsche Nationalbibliopthek
http://d-nb.de

[19] Niedersächsische Staats- und Universitätsbibliothek Göttingen
http://www.sub.uni-goettingen.de

[20] IBM Deutschland GmbH
http://www.ibm.com/de/

[21] Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen
http://www.gwdg.de

[22] Universelles Objektformat – Ein Archiv- und Austauschformat für digitale Objekte
http://kopal.langzeitarchivierung.de/downloads/kopal_Universelles_Objektformat.pdf

[23] Apache XMLBeans
http://xmlbeans.apache.org/

[24] Ghostscript
http://www.ghostscript.com/awki

[25] Common Query Language (CQL)
http://www.loc.gov/standards/sru/cql/

[26] Global Digital Format Registry (GDFR)
http://hul.harvard.edu/gdfr/

[27] PRONOM – The Online Registry of Technical Information
http://www.nationalarchives.gov.uk/pronom/