k o L i b R I

kopal Library for Retrieval and Ingest

Documentation

State: February 2007

Table of contents

1. koLibRI Overview

koLibRI is a framework for the integration of a long-term archiving system, like the IBM Digital Information Archiving System (DIAS)[6], into the infrastructure of an institution. In particular, it organizes the creation and ingest of archival packages into DIAS and provides functionalities to retrieve and manage these packages.

This document describes the installation and configuration of a fully functional koLibRI system, as well as it's basic internal design to allow individual developments and extensions. A certain amount of basics about long-term archiving[1], as well as about the standards OAIS[2], URN[3], METS[4], LMER[5] und IBM DIAS[6] is essential for fully understanding the system. The necessary knowledge is, however, freely available via the Internet.

It has to be concidered that koLibRI is still in a very early state of development and will not reach it's full functionality and a stable state until the succesful end of the kopal project [8]. The early release of a so called "beta version" shall only be used for research and development issues, and not for the use in a productive environment. Incidentally, the terms of use and liability as stated on the kopal website[1] [8], apply.

1.1 Functionality

In short, koLibRI generates a XML file according to the METS schema out of the metadata, provided with the object to archive or generated by JHOVE[12], bundles it into an archive file together with the object (zip or tar) and delivers this file (SIP / Asset) to the DIAS system.

From that point of view, koLibRI was developed to provide a full implemented long-term archiving solution together with the IBM DIAS. However, koLibRI can also be used as an independent software to create METS files or whole SIPs according to the Universal Object Format (UOF)[7], completely without the need of the DIAS system. XML metadata files or SIPs generated in this way, can be used for data exchange between several institutions; a feature which was one of the leading aspects in the development of the UOF. Because of its modular design and its open specified interfaces, koLibRI can alternatively be adapted to another archival system or metadata format with affordable effort.

Functionality: Ingest

It has to be defined which working steps are necessary to create and ingest an archival package. These steps can differ for different types of digital objects.
For instance, the sources of objects (CD-ROM within a local machine, document server in the intranet, etc.) and their descriptive metadata (XML file attached to the object, OAI interface of a catalog system, etc.) strongly vary for electronic theses, digitized books and even within these object classes. koLibRI was designed modular, so that these different steps can be integrated. It allows the notation of these steps within a XML file called **`policies.xml`**. The working steps are referred to as **`steps`**, which are implemented by **`action modules`**.
Typical modules process tasks like the download of files for the archival package, the validation of the files, extraction of technical metadata, generation of the metadata file that belongs to the UOF and the ingest of the final archival package into DIAS. There is also the possibility to scale these steps to different machines, as to create archival packages within several departments, but to use a central instance for quality assurance, entries into catalog systems and the final ingest.

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

Functionality: koLibRI-Database

For many user scenarios, it is useful to harvest and store data about processed objects during the ingest process. To do this, koLibRI provides database support, so that own identifiers, Dublin Core metadata or some technical metadata for instance can be stored file based. This way, the data is available also for functionalities outside of koLibRI, e.g. for statistics, identifier resolving or an OAI interface.

Functionality: Retrieval

The functionality of retrieving archive objects is limited to the command line tool **`DiasAccess`**, and the direct use of the according Java methods of the same class within own software respectively. Further functionality is in development.

1.2 Project scale

koLibRI has been developed within the scale of project kopal "Co-operative Development of a Long-Term Digital Information Archive"[8] by the German National Library and the Goettingen State and University Library for the use of the system kopal solution. Subject of the 3-year project kopal is the practical proving and implementation of a cooperative developed and operated long-term archiving system for digital publications. As partner, the German National Library, the Goettingen State and University Library and IBM Germany are implementing a cooperative and reusable solution for the long-term preservation of digital resources. The technical maintenance is done by the Gesellschaft für wissenschaftliche Datenverarbeitung mbH Goettingen (GWDG).

1.3 Distribution

The precompiled binaries and the source codes of koLibRI can be downloaded from http://kopal.langzeitarchivierung.de/kolibri - free of charge or registration.

Please send an email to the following address when facing any problems: **`kolibri@kopal.langzeitarchivierung.de`**

Everyone interested in the software is free to modify or extend it and can pass it on to others. However, the licence agreements that are being provided by kopal have to be acknowledged.

The software libraries and service programs that have been developed of the kopal project, and are only being used by koLibRI, partially apply to other licences which are available in the corresponding license files. Any questions regarding these software libraries are only to be addressed to the respective authors of the according software package.

2. Installation and Configuration

2.1 Requirements

koLibRI has been implemented in Java only, that means it should be usable on any platform that provides a Java Virtual Machine in version 1.5 (However, even alternating Java environments are not always compatible. Problems with alternating XML parsers where sporadically observed). The development team has tested koLibRI on various processor architectures and operating systems.

koLibRI is free software in the sense of the Free Software Foundation[9]. It can likely be ported to other platforms in favour. In such cases, the kopal team would be pleased to be informed about any kind of customization.

2.2 Installation

The installation of the precompiled distribution consists of the following components:

- **`kolibri.jar`** - The precompiled program package
- **`/lib`** - Directory with linked software packages
- **`/config`** - Example and main configuration files

In addition there are several batch files for MS Windows and Unix to provide an easy way of the invocation of the included software tools.

These components simply have to be unpacked from the packed archive file into a favoured directory.

If one wishes to recompile koLibRI from the source files by himself, the packed **`.java`** files simply have to be unpacked and compiled - preferential with the included Ant script.

After **`kolibri.jar`** is available on the hard disk, the batch files **`workflowtool.bat`**, **`diasingest.bat`** and **`diasaccess.bat`** (for MS Windows), respectively their pendants for Unix have to be customized with the help of a text editor. Once all paths and parameters have been set correctly, the different functionalities of the program are available by the invocation of the according batch files.

The important thing at this point is to know, which paths and parameters are appropriate for the according system. In cases of uncertainness, the according system administrator should be contacted. It is recommended - if available - to reserve 512 megabytes of heap size through the parameters.

Listed below are the main parts of the **`workflowtool.bat`** file. They are followed by comments to the lines that have to be customized.

```
@ECHO OFF
REM Copyright 2005-2007 by Project kopal
REM -- License info taken out --
REM Usage: workflowtool
REM [-c,--config-file] [-s,--show-properties] [-p,--process-starter]
REM [-h,--help] [-i,--input]
```

```
REM
REM where
REM -c,--config-file <filename> The config file to use.
REM  -s,--show-properties      Prints the system properties and
continues.
REM -p,--process-starter <classname>
REM The process starter module which chooses items for processing.
REM -h,--help Prints this dialog
REM -i,--input <source> The input source for the process starter.
REM It depends on the process starter whether if this
REM mandatory or not.
REM
REM Configuration constants:
REM KOLIBRI_HOME koLibRI installation directory
REM JAVA_HOME  Java JRE directory
REM JAVA       Java interpreter
REM EXTRA_JARS Extra jar files to add to CLASSPATH

SET KOLIBRI_HOME="C:\path_to_your_directory"

SET JAVA_HOME="C:\Programme\java\jre1.5.0_09"
SET JAVA=%JAVA_HOME%\bin\java

SET EXTRA_JARS=

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

SET CP=%KOLIBRI_HOME%\koLibRI.jar
IF "%EXTRA_JARS%"=="" GOTO FI
  SET CP=%CP%;%EXTRA_JARS%
:FI

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

SET ARGS=
:WHILE
IF "%1"=="" GOTO LOOP
  SET ARGS=%ARGS% %1
  SHIFT
  GOTO WHILE
:LOOP

REM Set the CLASSPATH and invoke the Java loader

%JAVA% -classpath %CP% -Xmx512m \
       de.langzeitarchivierung.kopal.WorkflowTool %ARGS%
```

KOLIBRI_HOME - the directory in which the kolibri.jar file is located.

JAVA_HOME - the directory of the local Java installation.

JAVA - the directory within **JAVA_HOME** which contains the executable Java Virtual Machine. The default value stated above should be appropriate for most default Java installations.

EXTRA_JARS - complete paths and filenames to optional software libraries to be linked, which are to be separated by a semicolon (MS Windows) or a colon (Unix). In example, this can be institution specific software extensions for koLibRI.

For the installation of the koLibRI database please see the part "koLibRI database".

2.3 Configuration of koLibRI

To configure koLibRI, the files **config.xml** (an alternative file name or path can be used through the command line parameter "**-c**") and the policy file (default name is **policies.xml**) have to be customized. The config file contains global and module / class specific configuration parameters. Configuration values for individual working steps can be set within the policy file.
The most specific value has the highest priority, that means the Configurator first tries to fnd a parameter for the actual step, then for the actual module or class and finally it uses global configuration parameters.

The configuration file has a structure as follows (for the formal specification please see the file **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>
```

The **common** block defines the global parameters, the **modules** block the module and class specific ones. Each configuration parameter is embedded within **property** tags. The **field** tag defines the name of the parameter (case sensitivity is not relevant) and the according values are set within one ore more **value** tags. The repeatability of a **value** tag is dependent of the according parameter, whose meaning is documented within the **description** tags. The **class** tag bundles all configuration parameters for a specific module, where the name attribute is stating the name of the according Java module (either by the full qualified package name or by abandonment of the usual "**de.langzeitarchivierung.kopal.**" prefix).
Setting configuration values for individual working steps of the policy file is done in the following way (for the formal specification please see the file **policies.xsd**, for a description of the policy file please see the part "Using koLibRI for the ingest workflow"):

```
<step class="MetadataGenerator">
      <config>
```

```
            <property>
                <field>showPdfPages</field>
                <value>false</value>
            </property>
        </config>
        ...
    </step>
```

If a **config** tag is present one hierarchical level below a **step** tag, all configuration parameters contained in the config tag are used for the specific class of the **step** tag.
A basic configuration of koLibRI needs the following global configuration parameters in particular (for documentation please see example configuration files):

- **destinationDir**
- **workDir**
- **tempDir**
- **logfileDir**
- **maxNumberOfThreads**
- **useDatabase**
- **logLevel**
- **defaultProcessStarter**
- **policyFile**
- **defaultPolicyName**
- **mdClassName**
- **mdTemplateFile**
- **mdWrapPrefix**
- **techMdPrefix**
- **dmdPrefix**

It is suggested to use the provided examples as a basis for individual configurations.

3. Using koLibRI for the ingest workflow

To use koLibRI for the ingest of archival packages it is essential to specify the working steps in the configuration file policies.xml[2]. The formal syntax is defined in the example file **policies.xsd** within the config directory, the functionality of the example workflows, together with the example configuration files, is explained in the appendix under "Information about the example configuration". Workflows are seen as trees in the sense of the graph theory, that is processed in the direction from the root to the leafs. Each node is a step, whose child nodes are only processed, if the parent node was processed successfully. The child nodes are processed parallel if applicable, this can be very useful for time consuming tasks (like burning a CD-ROM or the transfer to another archive)[3]. An example workflow would process the following six steps in a row (for more examples please see appendix "Information about the example configurations"):

1. Copy selective files to a process directory
   (ActionModule: **FileCopyBase**),
2. extract descriptive metadata for the files
   (ActionModule: **MetadataExtractorDmd**),
3. generate technical metadata for the files
   (ActionModule: **MetadataGenerator**, please see appendix for JHOVE),
4. generate metadata file mets.xml
   (ActionModule: **MetsBuilder**),
5. compress all files into an archival package
   (ActionModule: **Zip**), and finally
6. ingest the archival package into DIAS (or another archive system)
   (ActionModule: **SubmitDummySipToArchive**).

```
<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>
```

[2] Which steps are needed is also dependent of the individual archiving requirements. Each institution has to declare these requirements for itself; the process of finding a long-term archiving policy can not be replaced by a technical solution (please see partner project nestor[1] for further informations).

[3] This functionality is not yet realized in koLibRI 0.7

Besides the working steps itself, it has to be defined, how new working steps are being started and which initialization values they get. This is the meaning of the so called **ProcessStarters**, which are set at the start of the program either through the command line parameter **-p**, or in the configuration file through the parameter **defaultProcessStarter**. For basic usage, there are the ProcessStarters **MonitorHotfolderExample** and **MonitorHttpLocationExample**, which generate archival packages from files and folders beneath a given hotfolder or URL. A more special one would be a server (**Server** and **ClientLoaderServerThread**), that makes one machine listening to retrieve archival packages from another machine for further processing. It is also possible to run multiple ProcessStarters at the same time by setting more than one value in the configuration parameter **defaultProcessStarter**.

Starting a WorkflowTool instance:

The batch files, which are included within this release, are used to execute an instance of the WorkflowTool after a correct installation .

To do this, the batchfiles simply have to be customized with the respective local configurations. Additionally, optional module packages, developed by other institutions can be included and used through the batchfiles.

workflowtool -h explains the command line options:

```
-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
-i,--input            The input source for the process starter.
                      It depends on the processstarter whether
                      this is mandatory or not.
```

Optional to the execution of the batch file, a WorkflowTool instance can also be started through following command:

```
java -jar kolibri.jar OPTIONS
```

Because the access to DIAS is usually realized through an encrypted connection, the parameter **-Djavax.net.ssl.trustStore=KEYSTORE_LOCATION** will be necessary after the installation of the certificate of the DIAS hosting partner with the Java keytool. The path to the keystore file, as well as the path to the "known hosts" file, can be set in the configuration file.

Overview over more ActionModules and ProcessStarters:

ProcessStarters

`Server:`
The koLibRI instance is listening for network connections and starts a new thread for each request, that executes the necessary actions. This ServerThread has to be set in the configuratiopn file (as serverClassName), for example ClientLoaderServerThread (takes archival packages from other koLibRI instances).
Important configuration values:

- **`serverClassName`**
- **`defaultConnectionPort`**

`MonitorHotfolderExample:`
Example ProcessStarter that builds an archival package (SIP - Submission Information Package) for every subdirectory of a given "hotfolder" directory. It monitors the directory, stated as **`hotfolderDir`** and processes all existing subfolders first. Then it processes all added subfolders. The names of the subfolders are interpreted as persistent identifiers for the SIPs.
Important configuration values:

- **`hotfolderDir`**
- **`startingOffset`**
- **`checkingOffset`**
- **`addingOffset`**

`MonitorHttpLocationExample:`
Same as MonitorHotfolderExample, but files and directories can be retrieved by an URL. This URL is also monitored and added directories are processed. Directory names are interpreted as persistent identifiers as before.
Important configuration values:

- **`urlToMonitor`**
- **`readDirectoriesOnly`**
- **`startingOffset`**
- **`checkingOffset`**
- **`addingOffset`**

ActionModules

`AddDataToDb:`
Adds configurable information to the database, if **`<field>useDatabase</field>`** is set in the configuration file.
Important configuration values:

- **`storeFileData`**
- **`storeFileDataTechMd`**
- **`storeDc`**
- **`storeIds`**

`CleanPathToContentFiles:`
Deletes the files and folders in the temporary processing directory.

`CleanUpFiles:`
Deletes created ZIP files and METS files from the destination directory.

`Executor:`

Executor provides the functionality to invoke system based command line tools. The static arguments have to be set in the configuration file.
Important configuration values:

- **command**

FileCopyBase:
Files and folders of the object are copied to a temporary processing directory, to work with these copies, without altering the original files. The ActionModul can be extended by own modules, i.e. to change the names of the copied files or ignore certain files for the archival package.

FileCopyHttpBase:
Copies files and folders over HTTP. This allows archiving files from a webserver. For more, see FileCopyBase.

MetadataExtractorDmd:
Includes descriptive Metadata into the archival package. This module has to be customized for individual needs. The given example only shows how descriptive metadata is embedded into a SIP. The source of this metadata is dependant of the structures of the individual institutions.

MetadataGenerator:
Generates technical metadata for all content files of the archival package with the use of the JHOVE toolkit from the Harvard University Library (please see part JHOVE for further information).
Important configuration values:

- **acceptedUnknownFileFormat**
- **showGifGlobalColorTable**
- **showPdfPages**
- **showPdfImages**
- **showPdfOutlines**
- **showPdfFonts**
- **showTiffMixColorMap**
- **showTiffPhotoshopProperties**
- **showWaveAESAudioMetadata**
- **showHtmlLinks**
- **showHtmlImages**
- **showIso9660FileStructure**

MetsBuilder:
Creates a METS file for the archival package out of the gathered informations. Responsible for the creation is the class, stated in **field>mdClassName</field>**.
For kopal, the implementation **Uof.java** of the class MetadataFormat creates a METS file according to the UOF of the kopal project.

SubmitDummySipToArchive:
Example module for demonstration purposes.

SubmitSipToClientLoader:
Submits a SIP to a client loader. This module can be used by more than one asset builder at the same time, that send their archival packages to a central WorkflowTool instance. Please see "Informations about the example configuration" for further information.
Important configuration values:

- **clientLoaderServer**
- **clientLoaderPort**

- **submitPolicyName**

SubmitSipToDias:
Transfers a SIP to the DIAS, used by kopal, and is the implementation of the SIP specification[10] for ingests into kopal-DIAS, provided by IBM.
Important configuration values:
- **cmUser**
- **cmPwd**
- **ingestPort**
- **ingestServer**
- **knownHostsFile**
- **preloadArea**

TiffImageMetadataProcessor:
Validates all TIFF image files within the archival package with the use of JHOVE first. Two types of errors within the TIFF headers can be directly corrected. These are first wrong offsets of TIFF-ASCII fields, and second false parameter length of TIFF tag 297 (PageNumber). After the correction, the file is validated once more.
Important configuration values:
(TiffImageMetadataProcessor extends the class MetadataGenerator, so, also those configuration values apply):
- **storeBackupFiles**
- **correctInvalidTiffHeaders**
- **separateLogging**
- **separateLogfileDir**
- **verbose**

Utf8Controller:
With the help of this module, a created METS file, that is always UTF-8 encoded, can be checked for invalid UTF-8 characters and correct them. The source of the used **Utf8FilterInputStream** is the **utf8conditioner** by Simeon Warner (**simeon@cs.cornell.edu**), which was adapted to JAVA for the use in koLibRI.
Important configuration values:
- **filename**
- **storeOriginalFile**
- **originalPostfix**

XorFileChecksums:
Generates an XOR checksum from all existing file checksums and puts it into the database. This XOR checksum can be used to check if an identical SIP has already been ingested into the archive.
Important configuration values:
- **gotoErrorIfAlreadyIngested**

Zip:
Compresses the content files together with the METS file (mets.xml) into a compact package.
Important configuration values:
- **compressionLevel**

Additional ActionModules and ProcessStarters are currently available by request from the kopal project partner The German National Library (DNB) and the Goettingen State and University Library (SUB) or can be developed by everyone himself.

4. Using koLibRI for retrieval

The functionality to retrieve archival packages is limited to the command line tool **`DiasAccess`**, or respectively to the use of the according Java methods of the according class for own programs, at the moment.

For better understanding of the **`DiasAccess`** command line options, knowledge of the DIP interface specification[11] is important.

An overview over these options can be found in the appendix "Direct usage of the interfaces of DIAS".

5. koLibRI database

5.1 Current limitations

1. Parallel database access from different machines has not been sufficiently tested yet.
2. koLibRI uses the functionality of Hibernate for all database access. Foreign keys, transactions and auto-incremental functionalities are used in particurla, which limits the choice of the table engine to **`innoDB`** when using mySQL.

5.2 Installation

1. Required software/jars

The class path has to include:

- db-beans.jar
- hibernate3.jar (plus packages required by Hibernate)
- a jar of the database driver, e.g. mysql-connector-java-3.1.13-bin.jar for mySQL

All these jars can be found in kopal/lib. Hibernate[13] and the mysql-connector are also available from the websites of the according companies.

2. Installation of a database

The koLibRI database was tested with a mySQL database in version 4.1.11 and the InnoDB table engine, contained in this version. With "mySQL Administrator" and "mySQL Query Browser" there are easy to use and free managing tools available, that can do many configuration and query tasks.

Please read the according installation guides for all databases.

3. Creation of a database schema

The database schema for mySQL can be created with help of the SQL script createDB.sql (under config). "kopalDbTest" is used as default database name, this can customized by search and replace functionalities in the script. For executing the script, either the Query Browser can be uesed (Menu "File", "Load Script...", Button "Execute"), or you can connect through command line by "mysql -u [USER] -h [HOST] -p" and execute "source [PATH]/createDB.sql". To use the database, it is important to fill the tables **`owner`** and **`server`** with at least one entry each, which then have to be set in the configuration file as defaultOwner and defaultServer.

For other databases, a specific version of the createDB script has to be created.

4. Configuring the database

A database user and password for koLibRI must be created, that is allowed to access the database with full rights (except DROP) from all machines running koLibRI. This can be done under "User Administration" in the mySQL Administrator. Firewall settings do also have to be considered for remote access.

5. Configuring koLibRI for the database

For the use of the database, koLibRI has to be configured at various points. In addition to the setting if the database is even used (and not only one ore more log files), there also have to be set class specific values for util.db.HibernateUtil in the configuration file (see below).
Besides that, the moment and type of the stored data has to be declared for the according policies. This is done by addition of the ActionModule de.langzeitarchivierung.kopal.actionmodule.sub.AddDataToDb. The following boolean options are available:

- **`storeFileData`** (fills tables "file" and "fileformat" for each file in the metadata class)
- **`storeFileDataTechMd`** (fills table "techmd" for each file in the metadata class)
- **`storeDc`** (stores all Dublin Core metadata of the metadata class in table "dublincore")
- **`storeIds`** (stores all entries in ProcessData.customIds in table "identifier")

The ActionModules de.langzeitarchivierung.kopal.actionmodule.AlreadyIngestedChecker (before ingest) and de.langzeitarchivierung.kopal.actionmodule.DiasIngestFeedback (after ingest) can be used to add entries into the "dias" table. These do some checking and only ingest the packages, or respectively add data to the database, if these checks were successful. These ActionModules are not being used in the example files. They can only be used to document the functionality, as direct access to DIAS is required for their functionality.

Parameter in the configuration file:

Common:

```
<property>
  <field>useDatabase</field>
  <value>true</value>
  <description>
    If a database should be used for logging and bookkeeping
    of the SIPs, etc. See the full documentation.
  </description>
</property>
```

Class:

```
<class name="util.db.HibernateUtil">
  <property>
    <field>hibernateDbHost</field>
    <value>10.0.3.123</value>
    <description>
      The IP-address of the server hosting the database.
    </description>
  </property>

  <property>
    <field>hibernateDbName</field>
    <value>kopalDbTest</value>
    <description>
      The name of the database schema to be used.
    </description>
  </property>

  <property>
    <field>hibernateConnectionUsername</field>
```

```
      <value>notroot</value>
      <description>
        Username which koLibRI can use to connect
        to database.</description>
    </property>

    <property>
      <field>hibernateConnectionPassword</field>
      <value>!#ee34ydg4</value>
      <description>
        Password for hibernateConnectionUsername
      </description>
    </property>

    <property>
      <field>hibernateConnectionDriver_class</field>
      <value>org.gjt.mm.mysql.Driver</value>
      <description>
        The java driver class hibernate can use to access
        the database.
      </description>
    </property>

    <property>
      <field>hibernateDialect</field>
      <value>org.hibernate.dialect.MySQLInnoDBDialect</value>
      <description>
        The hibernate class specific for the used
        database/table type. A class from the package
        org.hibernate.dialect.
      </description>
    </property>

    <property>
      <field>hibernateLogLevel</field>
      <value>WARNING</value>
      <description>
        The lowest level for which you want to see the hibernate
        log messages. For possible values see
        java.util.logging.Level
      </description>
    </property>
```

Class:

```
  <class name="actionmodule.sub.AddDataToDb">
    <property>
      <field>storeFileData</field>
      <value>true</value>
      <description>
        If set to TRUE the file table is filled
        if this class is called.
      </description>
    </property>

    <property>
      <field>storeFileDataTechMd</field>
```

```
    <value>true</value>
    <description>
      If set to TRUE the file's technical metadata is added to
      the database.
    </description>
  </property>

  <property>
    <field>storeDc</field>
    <value>true</value>
    <description>
      If set to TRUE the SIPs Dublin Core descriptive metadata
      is added to the database.
    </description>
  </property>

  <property>
    <field>storeIds</field>
    <value>true</value>
    <description>
      If set to TRUE all custom identifier are added
      to the database.
    </description>
  </property>
</class>
```

5.3 Design of the database schema

- **input** contains files, that koLibRI received at "first contact" with the source material: Who is the owner of the object (reference to **owner**), where does the request or files come from (referende to **server**) and when (column **startdate**, given in milliseconds since 1970, see java.lang.System.currentTimeMillis()). If there are files present in the ProcessData object even before the execution of the policy (that is after the ProcessStarter), an XORed SHA-1 checksum of all files is stored. In addition, the object can have multiple files, Dublin Core metadata, identifier or extra information (references to **file**, **dublincore, identifier** or **various** on the **input** entry).

- Warnings and errors may appear while processing the policy. Such events are listed in table **event** and refer to the **input** entry, that the problem is related to. There are references to the **modulestatus** and the **description** of the error. In addition, the time of the event (**eventdate**) and the module name (**module**) are being logged. There is also the possibility for a reference to the file that was responsible for the event.

- Finally, the archival package gets successfully ingested into DIAS, and an entry into **dias** is made. Next to the external and internal DIAS IDs and the time if the ingest, the internal ID of the parent object is also logged for a migration. If the files were changed since their entry to the input table, the altered XOR SHA-1 checksum of all files is stored here. The deleted-flag is used to also be able to log deletions.

koLibRI -- DB (mysql) -- 05.02.2007
Server: MySQL 4.1.11-Debian_4sarge7-log
Client: MySQL Client Version 5.0.11

primary
unique
foreign 1 → n

int8 is MySQL-alias für bigint and used for all dates.

The numbering describes the order of the create table.

All ids (primary keys) are auto-increment (besides modulstatus.modulestatusid

file (15)

fileid	**int**	NN	
inputid	*int*	*NN*	*FS (input.inputid)*
checksumtypeid	*int*	*NN*	*FS(checksumtype.checksumtypeid)*
checksum	vc	NN	
filedate	int8	NN	
size	int8	NN	in byte
filename	vc	NN	

techmd (17)

techmdid	**int**	NN	
fileid	*int*	*NN*	*FS (file.fileid)*
techmd	longtext	NN	

Unique is not possible because of the longtext type.

fileformat (16)

fileformatid	**int**	**NN**	
fileid	*int*	*NN*	*FS (file.fileid)*
formatid	*int*	*NN*	*FS(format.formatid)*

format (1)

formatid	**int**	NN	
format	vc	NN	
version	vc	NN	

dublincore (14)

dublincoreid	**int**	NN	
inputid	*int*	*NN*	*FS (input.inputid)*
dcelementid	*int*	*NN*	*FS(dcelement.dcelementid)*
dcvalue	text	NN	indexlength 767

dcelement (7)

dcelementid	**int**	NN	
dcelement	vc	NN	

Values: (1,'title'), (2,'creator'), (3,'subject'), (4,'description'), (5,'publisher'), (6,'contributor'), (7,'date'), (8,'type'), (9,'format'), (10,'identifier'), (11,'source'), (12,'language'), (13,'relation'), (14,'coverage'), (15,'rights')

input (9)

inputid	**int**	NN	
ownerid	*int*	*NN*	*FS (owner.ownerid)*
serverid	*int*	*NN*	*FS (server.serverid)*
checksumtypeid	*int*		*FS(checksumtype.checksumtypeid)*
checksum	vc	NN	
startdate	int8	NN	
serialdata	text		

identifier (13)

identifierid	**int**	**NN**	
inputid	*int*	*NN*	*FS (input.inputid)*
idtypeid	*int*	*NN*	*FS(idtype.idtypeid)*
idvalue	vc	NN	

idtype (8)

idtypeid	**int**	NN	
idtype	vc	NN	
text	vc	NN	

owner (2)

ownerid	**int**	NN	
owner	vc	NN	
pass	vc	NN	

server (3)

serverid	**int**	NN	
server	vc	NN	

various (12)

variousid	**int**	NN	
inputid	*int*	*NN*	*FS(input.inputid)*
type	vc	NN	
value	vc	NN	

modulestatus (4)

modulestatusid	**int**	NN	**not** autoinc
modulestatus	vc(45)	NN	

Values: (2, 'running'), (3, 'error'), (4, 'cancelled'), (5, 'warning')

event (10)

eventid	**int**	NN	
inputid	*int*	*NN*	*FS (input.inputid)*
modulestatusid	*int*	*NN*	*FS (modulstatus.modulestatusid)*
descriptionid	*int*	*NN*	
eventdate	int8	NN	
module	vc	NN	
filename	vc		

checksumtype (5)

checksumtypeid	**int**	NN	
checksumtype	vc	NN	

Values: (1, 'xor of sha1 file checksums'), (2, 'SHA-1'), (3, 'MD5')

description (6)

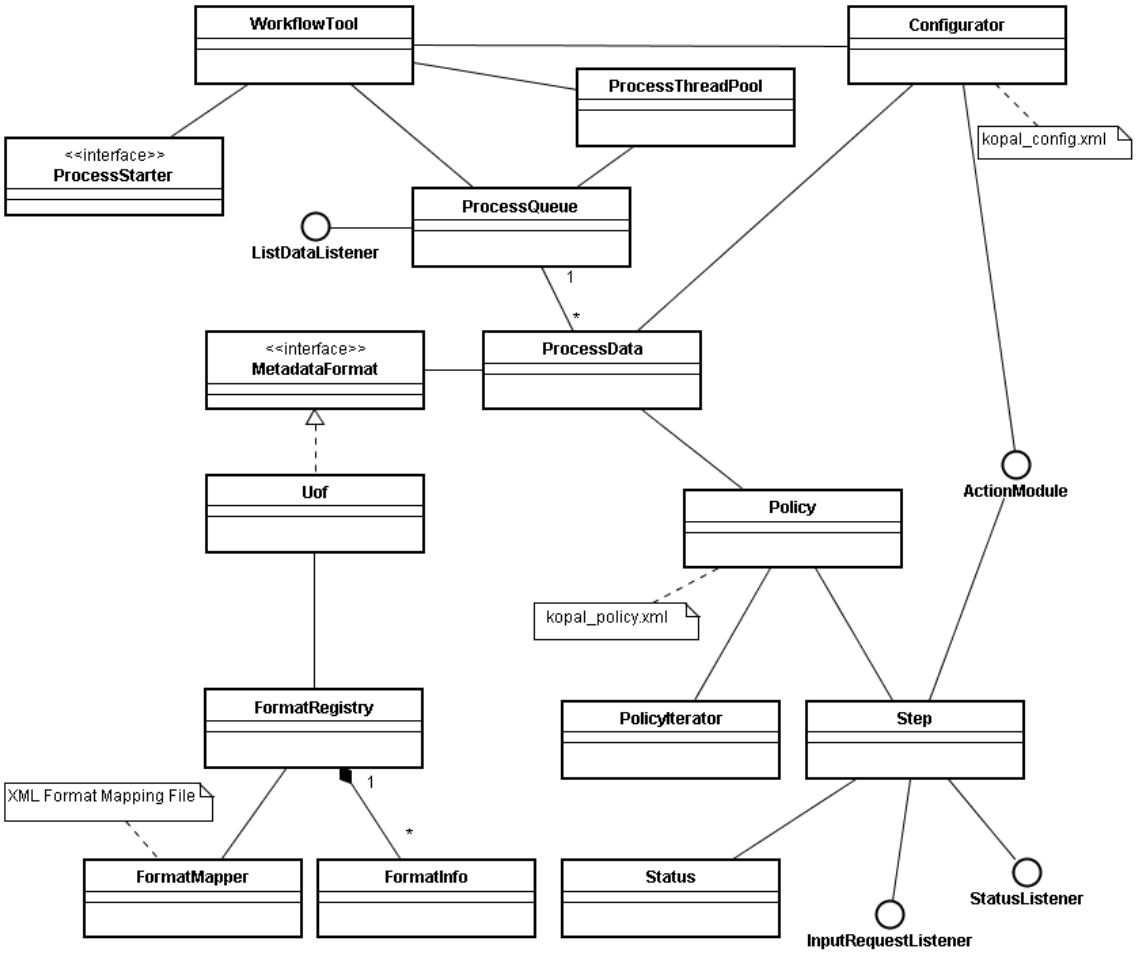
descriptionid	**int**	NN	
description	text	NN	indexlength 767

dias (11)

diasid	**int**	NN	
inputid	*int*	*NN*	*FS(input.inputid)*
checksumtypeid	*int*		*FS(checksumtype.checksumtypeid)*
diasintid	vc		kriegts erst ganz am schluß
diasdate	int8	NN	ingested / deleted im DIAS
deleted	bool	NN	deleted-flag
urn	vc	NN	ext. Id, brauchts DIAS
checksum_new	vc		
diasoldobjid	vc		

6. Customized koLibRI extensions

For further extensions and customizations, the interfaces ActionModule and ProcessStarter become more interesting in particular. A certain familiarity with the structure of koLibRI and the classes of the workflow framework **`de.langzeitarchivierung.kopal`** is, however, necessary. At this point, a functional overview shall be provided. Please consult the javadoc API documentation for more detailed information.



Class diagramm koLibRI Overview.

6.1 The Structure of koLibRI

The package structure of koLibRI is lying within de.langzeitarchivierung.kopal. This package contains the central classes of the workflow framework. Underneath lie the following packages:

- **`actionmodule:`** Contains the ActionModule interface and it's implementing classes
- **`formats:`** Contains the interface MetadataFormat in its implementing class Uof for the "Universal Object Format" of the kopal project.
- **`ingest:`** Classes for the ingest into DIAS
- **`processstarter:`** Contains the ProcessStarter interface and it's implementing classes
- **`retrieval:`** Classes for the access to assets within DIAS
- **`ui:`** Classes for user interfaces and event handling
- **`util:`** miscellaneous helper classes

It is possible to create subpackages for institution specific extentions.

The workflow framework **de.langzeitarchivierung.kopal is** consisting of eight classes at the moment:

- **Policy:** Manages a workflow and builds it from ist XML representation.
- **PolicyIterator:** Iterates over the working steps of a workflow tree
- **ProcessData:** Contains the central information of an asset: Name of the process (e.g. the assets URN), policy, metadata, file list. The **run** method contains the logic for processing the policy.
- **ProcessQueue:** A queue of all ProcessData objects to be processed
- **ProcessThreadPool:** A ThreadPool, that processes ProcessStarter and ProcessData objects.
- **Status:** Contains status value and description for the actual state of the ActionModule. The processing of the workflow is controlled by the status values.
- **Step:** Single working step within the workflow of an asset. Contains – among others – functions to load and start action modules and to set their status values.
- **WorkflowTool:** Provides the command-line interface, starts the process starters and is working off the processes. Working steps of a process are only processed if it's value is '**Status.TODO**' and its predecessor ended with the status '**Status.DONE**'.

ProcessStarters

For each new asset, the ProcessStarter has to process the following actions:

- Create a new ProcessData object
- initialize start values of the metadata if necessary (if ActionModules need those for further processing)
- add the ProcessData object to the process queue (**ProcessQueue.addElement()**)

As long as at least one ProcessStarter is running, or the ProcessQueue still contains elements respectively, koLibRI does not end, and instead waits for new processes or the end of the actual processed one.

ActionModules

An ActionModule must:

- set its status value to **Status.RUNNING** upon invocation
- set ist status value to **Status.DONE**, when it was finished successfully.

An ActionModule should:

- set it's status value to **Status.ERROR** if an error occurs. The error message will be logged seperately and modules with this status are not processed again. It is *not* equal with the throwing of a **Java-Exception**, that means the module is not interrupted in it's processing and could, for example, retry the erroneous action and reset it's status at a success. A usual procedure would be the setting of **Status.ERROR** on a timeout, directly followed by the **Status.TODO**. The module would then return temporarily to the workflow framework. This way, the error will be logged and the processing will be retried later.

- use the staus value **`Status.WARNING`**, if an event appears, that should be logged as warning (also in the database), but not lead to the abortion of the modules.

An ActionModule can:

- especially work with and update the metadata and files of the ProcessData object.
- store customData in the HashMap of the ProcessData object for other working steps, which has no place in the structure behind UOF.
- access the configuration data.

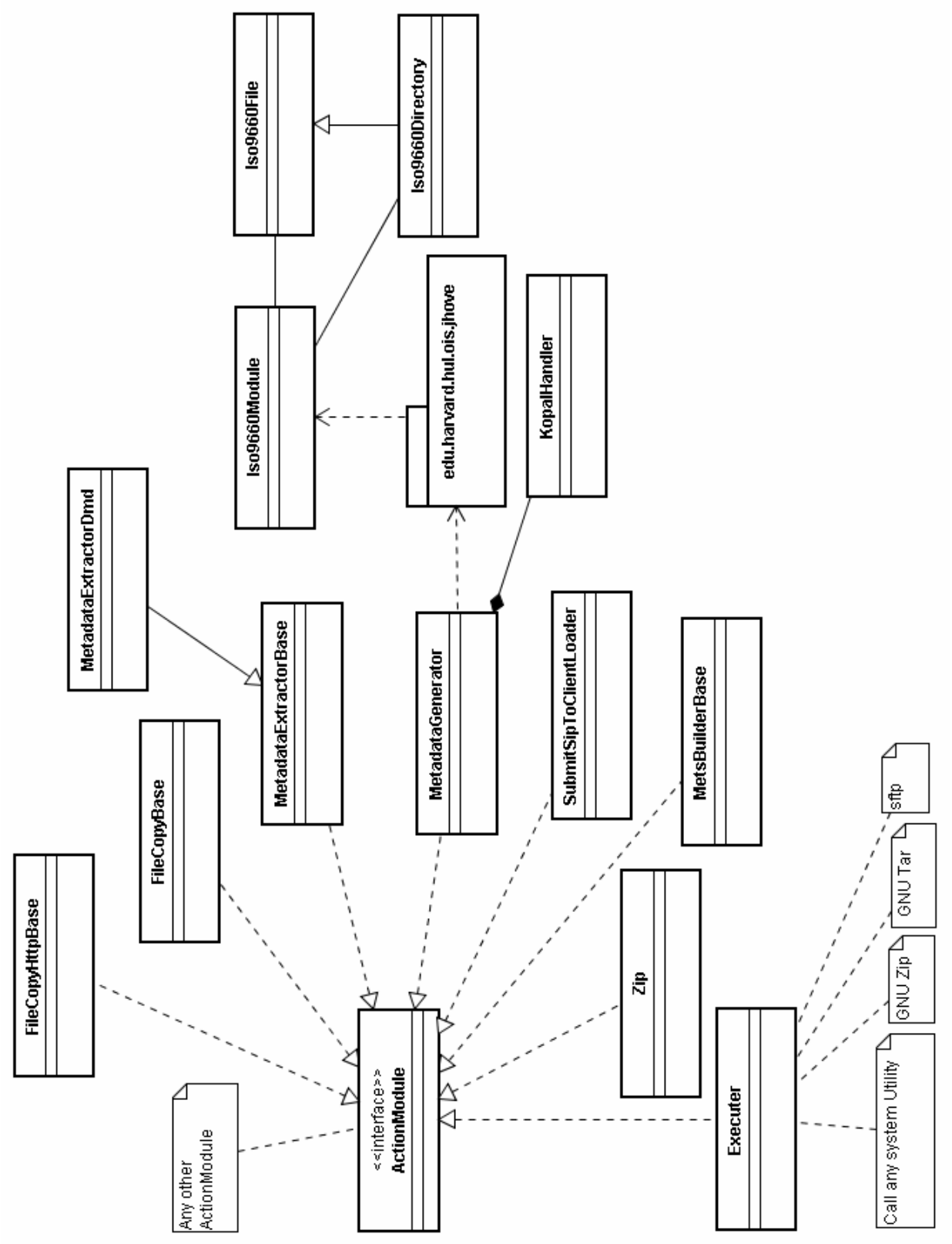
6.2 Configuring classes

The configuration of classes is done through the class **`Configurator`**. It gets initialized with the help of a XML file by the method **`setConfigDocument()`** and does then set the required configuration values of classes and objects, especially ActionModules and ProcessStarters, over various overridden configure methods. This is possible, because the classes and objects to configure are providing setter methods according to the JavaBeans specification. If an ActionModule requires the value **`useDatabase`** from the configuration file for instance, it just provides a **`setUseDatabase`** method, that is called with the according parameter by the **`Configurator`** while execution of the program (see also "Configuring koLibRI"). Also possible ist the transfer of configuration values between objects and classes through getter and setter methods. Please see the API documentation for more details.

6.3 Metadata formats

The metadata specific functionalities and object structures are mostly concentrated in the **`formats`** package. The interface **`MetadataFormat`** declares the basic functionalities that are required for processing. These are not always specific enough to provide full independency of a concrete metadata format like the UOF.
In version 0.7 of koLibRI, this in particular is valid for mapping the metadata for the database by the class **`MappToHib`**.

For the implementation of the interface MetadataFormat, a Java-XML-Binding through XML-Beans[14] was used. The XML schema, according to the UOF, was transferred to Java classes by the XML-Beans scomp compiler. This allows the structured and consistent usage of the schema in Java. In particular, it provides functionalities for writing, reading and validating of according XML files. To integrate individual schemas or metadata formats, according Java classes can be analogous generated through the XML-Beans scomp compiler. The main work for customization is then primary the implementation of the interface MetadataFormat and additional helper methods, customization of some ActionModules and of the class **`MappToHib`**, if the database is used.



Class diagram ActionModules.

7. Appendix

7.1 Information about the example configuration

There are three ways to use the example configuration files. The first is to configure a single machine, so that it creates and ingests archival packages into an archival system itself - a combination of SIP creation and SIP transfer with one instance of the **`WorkflowTool`** class.

There is also the possibility to configure multiple machines as so called AssetBuilders and let them create archival packages resulting of different workflows and submit them to another central machine. This central machine, the so called ClientLoader, receives all these archival packages and ingests them into an archival system - a separation of the SIP creation and the SIP transfer with one instance of the WorkflowTool as ClientLoader, and multiple instances as AssetBuilders.

The third way is to use a database for bookkeeping over the ingested and created archival packages. This has currently only been tested further with a single instance of the WorkflowTool in standalone mode (see point 1).

ATTENTION: The use of the database has not yet been tested for the parallel use of multiple WorkflowTools as standalone AssetBuilders. Usage for multiple independent instances of the Workflowtool in standalone mode is theoretically possible, but must be used at one own's risk. Usage of the database through a ClientLoader is not yet possible.

1. Creation and ingest of archival packages with a single instance of the WorkflowTool (standalone mode)

The file **`example_config_standalone.xml`** is used for a single instance of the class **`WorkflowTool`**. To execute this example, all that must be done, is to simply fill all tags enclosed by stars ******* with the right values. These are four path values for the work and temporary directories, a URL or path value depending on the **`ProcessStarter,`** and maybe a path for more logfiles. The **`<description>`** tags in the configuration file give a description of the according values.

There is the choice out of two ProcessStarters at the moment. "MonitorHotfolderExample" takes all subfolders of the directory stated in **`<field>hotfolderDir</field>`** and further processes all subfolders added later as an archival package to ingest (SIP), by adding the subfolders to the **`ProcessQueue`** and processing them. "MonitorHttpLocationExample" does the same, but checks files and subfolders of an URL stated in **`<field>urlToMonitor</field>`**.
Of course the stated directories must exist and contain files and folders, of which JHOVE can extract technical metadata, to get meaningful and usable results. Any kind of file structure and files can be used to test this module. See "The use of JHOVE in koLibRI" for further information.

The policy form the policy configuration file **`policies.xml`** that is used here, is **`example_standalone`** and executes the following ActionModules for each step. More informations for each module can be obtained through the documentation of the Java classes or the <**`description`**> tags in the configuration file.

> **`FileCopyBase/FileCopyHttpBase:`**
> These modules copy the wanted subfolders to the given temporary directory **`<field>tempDir</field>`**, depending on the used ProcessStarter either by file copy or HTTP download.

Please consider the correct combination of ProcessStarter and FileCopy module.

XorFileChecksums:
This module creates another checksum from all existing file checksums, that can be used to check, whether an identical archival package has already been ingested into the archival system or already has been processed.

TiffImageMetadataProcessor:
All TIFF image files in the directory to process are validated with the use of JHOVE. Some errors in the TIFF header metadata (only few at the moment) can be automatically corrected.

MetadataExtractorDmd:
The "MetadataExtractorDmd" is an example for integrating descriptive metadata into the METS file. This module can be customized to extract descriptive metadata in individual workflows.

MetadataGenerator:
Generates technical metadata for all existing content files of the archival package with the use of the JHOVE-Toolkit by the Harvard University Library.

MetsBuilder:
Creates a METS file for the archival package out of the gathered information. Responsible for the creation is the class, stated in **<field>mdClassName</field>**. This class creates a METS file according to the Universal Object Format (UOF) of the kopal project.

Utf8Controller:
With the help of this module, a created METS file, that is always UTF-8 encoded, can be checked for invalid UTF-8 characters and correct them. The source of the used `Utf8FilterInputStream` is the **utf8conditioner** by Simeon Warner (simeon@cs.cornell.edu) which was adapted to JAVA for the use in koLibRI.

Zip:
Compresses the content files together with the METS file (mets.xml) into a compact package.

SubmitDummySipToArchive:
Transfers the finished archival package to an archival system of choice. This module is just a dummy, like MetadataExtractorDmd, that has to be customized for the individual used archival system.

AddDataToDb:
Adds configurable information to the database, if **<field>useDatabase</field>** is set in the configuration file. See example 3 in part "koLibRI database".

CleanPathToContentFiles:
Deletes the files and folders in the temporary processing directory.

CleanUpFiles:
Deletes created ZIP and METS files from the destination directory. Commented out in the example for better understanding.

The command to run the standalone example from a command line is as follows:

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

or better (if "workflowtool.bat" or the "workflowtool" script has been configured), to set some more, maybe important parameters:

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

2. Separation of the creation and ingest of archival packages with two or more instances of the WorkflowTool (Usage of ClientLoader /AssetBuilder constellation)

The file "**example_config_clientloader.xml**" is used with a single instance of the class WorkflowTool, the file "**example_config_assetBuilder.xml**" can be used for one or more instances of the WorkflowTool each as AssetBuilder. For the communication between the instances, some more configuration values are needed in the configuration files. The division of work is described below.

The AssetBuilders can be started on different machines. This way, each machine can process an individual workflow for the creation of archival packages. For individual workflows, also individual policies and configuration files have to be created. The AssetBuilders in this example process the same steps as in the last example. Again, both ProcessStarters stated above, can be used. The difference is, that the created SIPs are not directly transferred to an archival system by each AssetBuilder himself, but are submitted to a central ClientLoader first. To do this, the AssetBuilders use the ActionModule "SubmitSipToClientLoader". The policy is "example_assetBuilder".

A WorkflowTool instance with the ProcessStarter "Server" is started as ClientLoader. It is then waiting for request on a port, defined by the field **defaultConnectionPort** in the configuration file.
When the server receives a request from an AssetBuilder, it starts a ClientLoaderServerThread, configured by **<field>serverClassName</field>**. These Threads are listening for the transfer of a SIP from the requesting AssetBuilder. Multiple SIPs from more than one AssetBuilder can be received at the same time.

The policy here is "example_clientloader" and contains two ActionModules:

SubmitDummySipToArchive:
Is the same ActionModule as in the above example. The separation of AssetBuilders and ClientLoaders is suggestive if multiple AssetBuilders shall be used, e.g. to process different workflows at the same time and create SIPs from different sources. The ClientLoader gathers all archival packages from all AssetBuilders centrally, and can also apply additional actions on them, e.g. burning CD-ROMs, re-validating the METS files or inform other institutions about newly ingested SIPs.

CleanUpFiles:
After successfull ingest into the archive system, the archival packages, delivered by the AssetBuilders, are deleted. Commented out in the example for better understanding.

The command to start the AssetBuilder / ClientLoader example on a command line is as follows:

Step 1, Starting the ClientLoader:
```
java -jar kolibri.jar -c config/example_config_clientloader.xml
```

Step 2, Starting an AssetBuilder:

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

Step 3, Starting more AssetBuilder's:

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

or better (if "workflowtool.bat" or the "workflowtool" script has been configured), to set some more, maybe important parameters:

Step 1, Starting the ClientLoader:

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

Step 2, Starting an AssetBuilder:

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

Step 3, Starting more AssetBuilder's (optional with different configuration files):

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

3. Creation and ingest of archival packages with a single instance of the WorkflowTool using the database.

At first, a database has to be created, according to the database documentation. If this was successful, the following steps have to be done:

- The WorkflowTool has to be configured as described in point 1.
- The tag **<useDatabase>** of the configuration file has to be set to TRUE .
- All database related values have to be filled with the right contents. These values are marked by **###** in the configuration file.
- The WorkflowTool can now be started as described in point 1. The database is now updated with useful information during the ingest process. See the database documentation for further information about the use of the database.

If the database shall not be used, all that has to be done, is to set the value of **<field>useDatabase</field>** to **FALSE**. The modules that use the database can remain in the policy file, because all database modules check this parameter.

7.2 The use of JHOVE in koLibRI

For the extraction of technical metadata, koLibRI uses the JSTOR/Harvard Object Validation Environment[12] (in short: JHOVE) version 1.1f (release of 01-08-2007) with some bugfixes which will be also contained in the upcoming maintenance release. JHOVE is used fully automatic by the ActionModule **MetadataGenerator** and its internal class **KopalHandler**, which works as an JHOVE OutputHandler. This handler enables the direct access to the created metadata within the program. MetadataGenerator is a customization of the **JhoveBase** class, which was matched and extended according to the given requirements.

JHOVE provides a very open module concept, that shall guarantee the support for future file formats. So in principal, JHOVE can create technical metadata for virtually any file format, preconditioned that an applying JHOVE module exists for it.

The used version of JHOVE supports the following file formats:

- 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

In the context of the development of koLibRI, there was developed another JHOVE module for

- Disc Images according to ISO9660 (with support for Rock Ridge filename extensions)

which is also a part of the koLibRI release, and available as version 1.0. However, it has to be considered, that this release not necessarily reflects the final design of the module.

The configuration of a JHOVE component is done with help of the JHOVE configuration file, whose path must be stated in the configuration file of koLibRI. In the current release, an example file is present within the /config directory and is named 'jhove.conf'. Within this file, various entries for JHOVE modules exist, which look as followed:

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

All stated modules are invoked sequential, until a module was found, which applies to the file that is actually processed. The order of the modules is absolutely essential here. Specialized modules should be placed above more generic ones. An example: A HTML file consists of a certain sequence of characters, but would be (correctly) recognized as a text file by the module for ASCII text, if it was invoked prior to the HTML module.

To include the JHOVE functionalities in koLibRI, the JHOVE JAR files **`jhove.jar`**, **`jhove-handler.jar`** and **`jhove-module.jar,`** within the **`/lib`** directory of the koLibRI software, are used. These files can easily exchanged by the ones of upcoming maintenance releases of JHOVE, to profit from possible bugfixes and enhancements. Solely internal, structural changes within a new version of JHOVE could make possible changes to the source code of MetadataGenerator necessary.

7.3 Errorcodes at System.exit

Miscellaneous:

(0)
Regular ending of the program

kopal.WorkflowTool, kopal.retrieval.DiasAccess:

(1)
Commandline and its arguments are not correct
Configuration invalid for WorkflowTool
Configuration invalid for ProcessStarter

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

(3)

Program was terminated and there was an error while processing some of the lists elements: Not everything has finished correctly! Please check the logfile

(9)
Could not create logfile

(12)
Database initialization failed

kopal.processstarter.MonitorHotfolderBase:

(6)
The given hotfolder path does not exist or is not a directory

(14)
Error processing current File. No file or no directory!

kopal.Policy:

(7)
Configuration file could not be parsed: Exception while parsing the policy file

kopal.util.FormatRegistry.java:

(10)
No backup file for format registry found! Formats can not be identified

kopal.util.kopalXMLParser, kopal.util.HTMLUtils:

(11)
Parse error in XML file, Parse error in XML string

kopal.processstarter.MonitorHttpLocationBase:

(15)
Error accessing URL

7.4 Error handling und loglevels

SEVERE:

The program has to be terminated because of an severe failure.
Example: A configuration file cannot be found.

WARNING:

Warnings are logged for failures that do not require the termination of the program.
Example: The processing of a list element is stopped, because the module set the status 'ERROR'. The next module will be processed.

INFO:

All information, relevant to the user, are logged within 'INFO'.
Example: The start of a Server, successful parsing of a configuration file, addition of an element to the ProcessQueue, etc.

FINE:

FINE loggs all informations which have little relevance to the user and are only interesting for debugging purposes.
Example: Internal messages of the list processing method WorkflowTool.process().

FINER / FINEST:

With FINER and FINEST, the finest debug messages can be logged, which are seldom needed, even for debugging.
Example: Messages about notify() and wait() for the debugging of threads.

ALL / OFF:

All messages, respectively no messages, are logged.

7.5 Direct usage of the interfaces of DIAS

For the direct use of the access and ingest interfaces of DIAS, the command-line tools DiasAccess and DiasIngest can be used. Knowledge of the DIP and SIP interface specifications[10][11] are preconditioned. The command-line tools DiasAccess and DiasIngest are realized through main() methods within the classes retrieval.DiasAccess and ingest.DiasIngest.

It has to be stated here again, that, as for the WorkflowTool also for these tools, the configuration of a keystore file is typically necessary, as the access to DIAS should be realized over an encrypted connection. A 'known hosts' file also has to be stated correctly.

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

8. References

[1] For an introduction, the homepage of project nestor is recommended:
http://www.langzeitarchivierung.de

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

[3] Uniform Resource Name; see also project EPICUR:
http://www.persistent-identifier.de/

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

[5] LMER Long-term preservation Metadata for Electronic Resources
http://ddb.de/standards/lmer/lmer.htm

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

[7] Universal Object Format - An archiving and exchange format for digital objects:
http://kopal.langzeitarchivierung.de/medien_presse/kopal_Universelles_Objektformat.pdf

[8] kopal - Co-operative Development of a Long-Term Digital Information Archive:
http://kopal.langzeitarchivierung.de/

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

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

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

[12] JHOVE - JSTOR/Harvard Object Validation Environment
http://hul.harvard.edu/jhove/

[13] Hibernate
http://www.hibernate.org/

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