



k o L i b R I

kopal Library for Retrieval and Ingest

Documentation

© Project kopal
Die Deutsche Bibliothek / Staats- und Universitätsbibliothek Göttingen

State: March 2006

Table of Contents

Introduction	3
Requirements	4
Overview over koLibRI (kopal Library for Retrieval and Ingest).....	5
Ingest	5
Retrieval.....	5
Start of a WorkflowTool instance:	6
Overview over additional ActionModules and ProcessStarters:.....	6
Extending koLibRI	7
The Structure of koLibRI.....	8
Appendix	10
The use of JHOVE in koLibRI	10
Errorcodes at System.exit.....	12
Error handling and loglevels	12
Direct use of the access- and ingest interfaces of DIAS.....	13
Classdiagrams	15
Informations about the example configurations.....	17

Introduction

Subject of the 3-year project kopal ist the practical proving and implementation of a cooperatively developed and operated long-term archiving system for digital publications. Die Deutsche Bibliothek, the Niedersächsische Staats- und Universitätsbibliothek Göttingen (SUB Göttingen) and IBM Germany want to implement a cooperatively operated and re-usable solution for the long-term preservation of digital resources as allied partners in this project. The technical system operation is managed by the Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen.

Die Deutsche Bibliothek and the SUB Göttingen had the need of a software for the use of the system kopal-DIAS-Solution¹. Its scopes are the ingest of informations, the access to these informations, the administration of the system and the integration into their existing infrastructures.

This document is meant to describe the software solution of Die Deutsche Bibliothek and the SUB Göttingen for the use of the kopal-DIAS-Solution.

¹ DIAS is a registered trademark of the IBM

Requirements

The project scheme of kopal (version 1.2) states on page 41, the requirements of implementation phase E2:

“Support of an universal objecttype: A universal objecttype will be defined, in which all electronic resources of the partners shall be transferable, and which contains all necessary metadata. This objecttype has to be accepted by the system as an archival object, and must be accessible from it.

Embedding of the depot system in existing system structures: There is a need for a comprehensive conversion- and interface software, to provide the connection of the existing structures of the project partners and the data producers to the depot system. The electronic resources, provided (at least in the medium-term) in heterogeneous formats, shall be converted into the agreed generic object format. Additionally, interfaces to inquiry systems are to be build, in which the resources are made available to the end-user.”

Precisely, on page 43 it says about implementation phase E2:

“The partners Die Deutsche Bibliothek and SUB Göttingen will implement flexible interfaces for the system DIAS-CORE, which allow the integration of the system into several utilization and service systems. The foundations for an automated transfer of the actual and future stock of electronic resources in different types into the system must be accomplished. Basis is at first the object format, defined in P2, and further data formats or object structures that are existing or expected in the future (Arrangements with publishers and others). The generated software and data interfaces are aimed for the re-use through third parties and are therefore aligned by international standards and made public. Same applies for the design of the reading access to the long-time archive, through which a direct use over retrieval systems, under considerations of legal issues, shall be available. Also, different ways of access are set up within the implementation of the interfaces together with the GWDG, under certain consideration of security aspects.”

Overview over koLibRI (kopal Library for Retrieval and Ingest)

koLibRI is organizing the construction and ingest for assets into DIAS and provides functionalities to retrieve them again.

Ingest

To construct and ingest an asset, it has to be defined which policies are necessary to do that. These policies can be absolutely different for miscellaneous types of digital objects. For example, the sources of the objects (CD-ROM within the local machine, document server within the intranet, etc.) and their descriptive metadata (provided with the object in form of xml, OAI interface of a catalogue system, etc.) for electronic dissertations, digitized books and even within these object types can vary considerably. To integrate various policies, koLibRI provides a corresponding framework. It allows the notation of the policies within a xml file, the so called 'policies.xml', where the individual policies are realized by so called 'action modules'. Typical modules fulfil tasks like the gathering of data for an archive object, the validation of files, the extraction of metadata, the creation of the metadata file according to the Universal Object Format and the ingest of the finally created archive object into DIAS. It is also possible to assign the workflows to different machines, i.e. to create assets by various departments, while using a single central instance for quality control, maintenance of verification systems and for the ingest of the packages.

Retrieval

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

The use of koLibRI for the ingest workflow

The main requirement to use koLibRI for the creation and ingest of archive objects is the definition of policies within the configuration file 'policies.xml'².

Its syntax is explained in the example file 'policies.xml' which is available within the 'config' directory. The functionality of the example policies is explained within the same directory in 'example_config_README.txt'.

Policies are presumed as trees in the sense of the graph theory, which are worked off from the root to its leafs. Each node is a working step, whose childnodes are only executed if itself was executed successfully. The childnodes themselves are executed parallel, this is an advantage for time consuming side tasks (e.g. additional transfer of the data to another archive or the burning of a CD-ROM)³.

A minimal policy would contain the following 5 steps in succession:

² Which working steps are necessary is dependant of the Universal Object Format (see Document "Universal Object Format" from the kopal website) and also of individual requirements to the archiving.. These requirements have to be cleared for each institution itself; the process of declaring an own long-term archiving policy can not be done by a technical solution (for more information see project nestor: www.langzeitarchivierung.de).

³ This functionality is not yet realized in koLibRI 0.5.

1. initialize some values of the metadata format (ActionModule: LmerInitializer),
2. extract technical metadata from the included files
(ActionModule: MetadataGenerator, see the appendix concerning JHOVE),
3. generate the metadata file 'mets.xml' (ActionModule: MetsBuilderBase),
4. compress all necessary files into an asset (ActionModule: Zip) and
5. ingest the asset into the DIAS system (ActionModule: SubmitAssetToArchive).

Besides the policies themselves, there also has to be stated in which way new steps shall be started and which parameters they should be given on their start. This is the task of the so called "ProcessStarter`s", which are selected at the execution of the software by a command-line parameter. The ProcessStarter ReadDirToCreateAssets would be an easy example, which builds assets from all files and subdirectories of a given directory. A more challenging would be the ClientLoaderServer, through which one machine is waiting to receive an asset from another machine, to apply further treatments on it.

Start of a WorkflowTool instance:

The batchfiles, which are included within this release, are used to execute an instance of the WorkflowTool after a correct installation⁴.

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.
-m,--multi-threaded  Runs the processStarter process as a daemon
                      thread.
-h,--help            Prints this dialog
-i,--input            The input source for the process starter. It
                      depends on the process starter whether this is
                      mandatory or not."

```

Optional to the execution of the batchfile, a WorkFlow 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. If the batchfiles are used, the path to the keystore file can be stored there.

For the direct use of the JAR file, the applying command is the following:

```
java -Djavax.net.ssl.trustStore=KEYSTORE_LOCATION -jar kolibri.jar OPTIONS.
```

Also a 'known hosts' file for ssh must be specified within the configuration file correctly.

Overview over additional ActionModule`s and ProcessStarter`s:

⁴ see the according documentation for further informations

ActionModule:

Embed descriptive metadata:

- Define a source: The dummy module GetDmdFromSomewhere
- Generate mets file with descriptive metadata: Use MetsBuilderDmd Instead of MetsBuilderBase.

Execute other programs:

- execute program with static parameters: Executor with command in form of a parameter within policies.xml
- execute program with dynamic parameters: Executor with superposed customized ActionModule`s

Deleting the source files:

- CleanPathToContentFiles

Forward asset to another koLibRI machine:

- The forwarding machine: SubmitAssetToClientLoader
- The receiving machine: The koLibRI instance has to be started with the ProcessStarter ClientLoaderServer and option -m (multi-threaded)

ProcessStarter:

Use koLibRI instance as ClientLoader (middle instance): Use ProcessStarter ClientLoaderServer and start koLibRI instance with option -m (multi-threaded).

ReadDirToCreateAssets: Example-ProcessStarter, which generates an asset For each subdirectory existent in a given directory.

It interprets the names of the subdirectories as the URN`s of the assets.

Additional ActionModule`s and ProcessStarter`s are currently available from the kopal project partners Die Deutsche Bibliothek (DDB) and the SUB Göttingen or can be developed by everyone himself.

Extending koLibRI

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.

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 its implementing classes
- formats: Classes for metadata formats
 - metslmer: Classes for the metadata format 'Universal Object Format'
- ingest: Classes for the ingest into DIAS
- processtarter: Contains the ProcessStarter interface and its implementing classes
- retrieval: Classes for the access to assets within DIAS
- ui: Classes for user interfaces and eventhandling
- util: miscellaneous helper classes

It is possible to create subpackages for institution specific extensions ('ddb' and 'sub' are available at the moment).

The workflow framework de.langzeitarchivierung.kopal

At the moment, seven classes exist:

- Policy: Manages a workflow and builds it from its xml representation
- PolicyIterator: Iterates over the working steps of a workflow tree
- ProcessData: Contains the central informations of an asset:
 - Name of the process (e.g. the assets URN), policy, metadata
- ProcessList: List of all assets to process
- 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 ActionModule's and to set their status values.
- WorkflowTool: Provides the command-line interface, starts the ProcessStarter's and is working off the processes. Working steps of a process are only processed if its status value is 'Status.TODO' and its predecessor ended with the status 'Status.DONE'.

ProcessStarter:

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

- Create a new metadata object LmerSIP
- initialize start values of the metadata if necessary (if ActionModule's need those for further processing)
- Add the metadata object with process and policy names to the list of processes (ProcessList.addElement())

The method ProcessStarter.hasMoreProcesses() has to be implemented in a way, that it returns false if the ProcessStarter will add no more processes. In that way, no other threads have to wait unnecessarily or can quit their processing.

ActionModule:

An ActionModule must:

- set its status value to Status.RUNNING upon invocation.
- set its status value to Status.DONE, when it was finished successfully.
- declare all its construction parameters as strings.

An ActionModule should:

- set its status value to Status.Error if an error occurs. The error message will be logged separately 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 its processing and could, for example retry the erroneous action and reset its 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.

An ActionModule can:

- especially work on the metadata and files within the LmerSIP. After all, this is its main task.
- store customData in the HashMap of the ProcessData object for other working steps, which has no place in the structure behind LmerSIP.
- access the configuration data.

Appendix

The use of JHOVE in koLibRI

For the extraction of technical metadata, koLibRI uses the JSTOR/Harvard Object Validation Environment (in short: JHOVE) version 1.0 (release of 05-26-2005) 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 'LmerHandler', 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 nach dem ISO-Standard 9660 (with support for Rock Ridge filename extensions), which is also a part of the koLibRI release, and available as version 0.9 BETA. However, it has to be considered, that this beta release has been added for testing purposes only, and not necessarily reflects the final design of the module.

The configuration of 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 direcotry 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.

Errorcodes at System.exit

Miscellaneous:

- (0) Regular ending of the program

kopal.WorkflowTool.java:

- (1) false command-line parameter
- (2) initialization of the ProcessStarter failed
- (3) program was terminated and there was an error while processing some of the lists elements
- (8) no policy file stated
- (9) logfile could not be created

kopal.ingest.DiasIngest.java:

- (1) false command-line parameter

kopal.ProcessList.java:

- (4) unknown Policy within Policy HashMap

kopal.Policy.java:

- (5) Policy tree could not be created

kopal.processStarter.ReadDirToCreateAssets.java:

- (6) The given path is no directory or contains no subdirectories

kopal.util.KopalConfigHandler.java:

- (7) error while parsing the configuration file

kopal.util.FormatRegistry.java:

- (10) no backup file present for the format registry

Error handling and 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 informations, relevant to the user, are logged within 'INFO'

Example: The start of a ClientLoaderServer, successful parsing of a configuration file, Addition of an element to the ProcessList, etc.

FINE:

FINE logs 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.

Direct use of the access- and ingest 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 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 parameter `-Djavax.net.ssl.trustStore=KEYSTORE_LOCATION` is typically necessary, as the access to DIAS should be realized over an encrypted connection. A 'known hosts' file also has to be stated.

DiasIngest

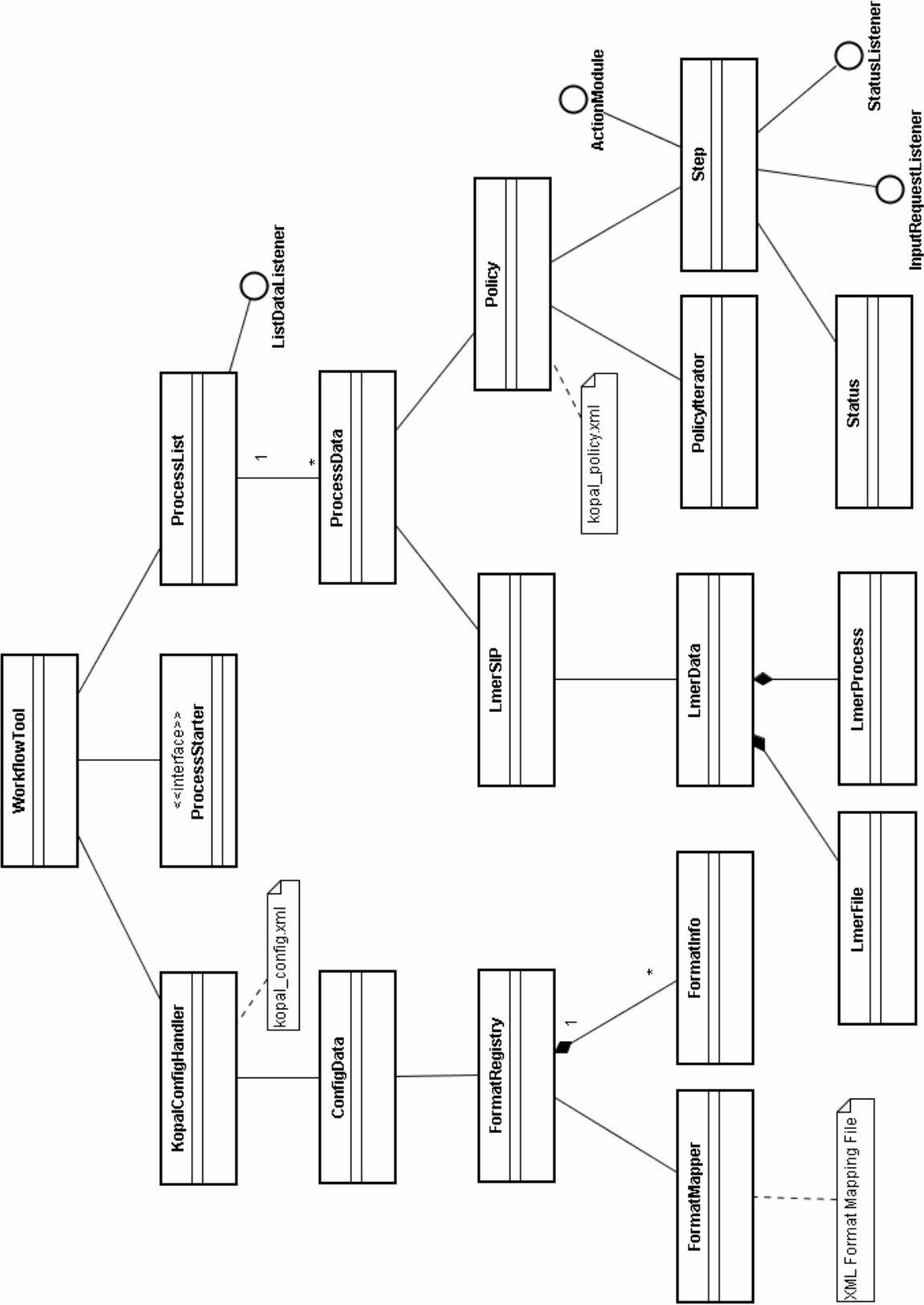
```
java -jar DiasIngest.jar -h:
  -hp,--show-properties  Print the system properties and continue
  -a,--address           The server address
  -f,--file             The file to submit
  -h,--help             Print this dialog and exit
  -n,--port            The server port
  -p,--password        The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

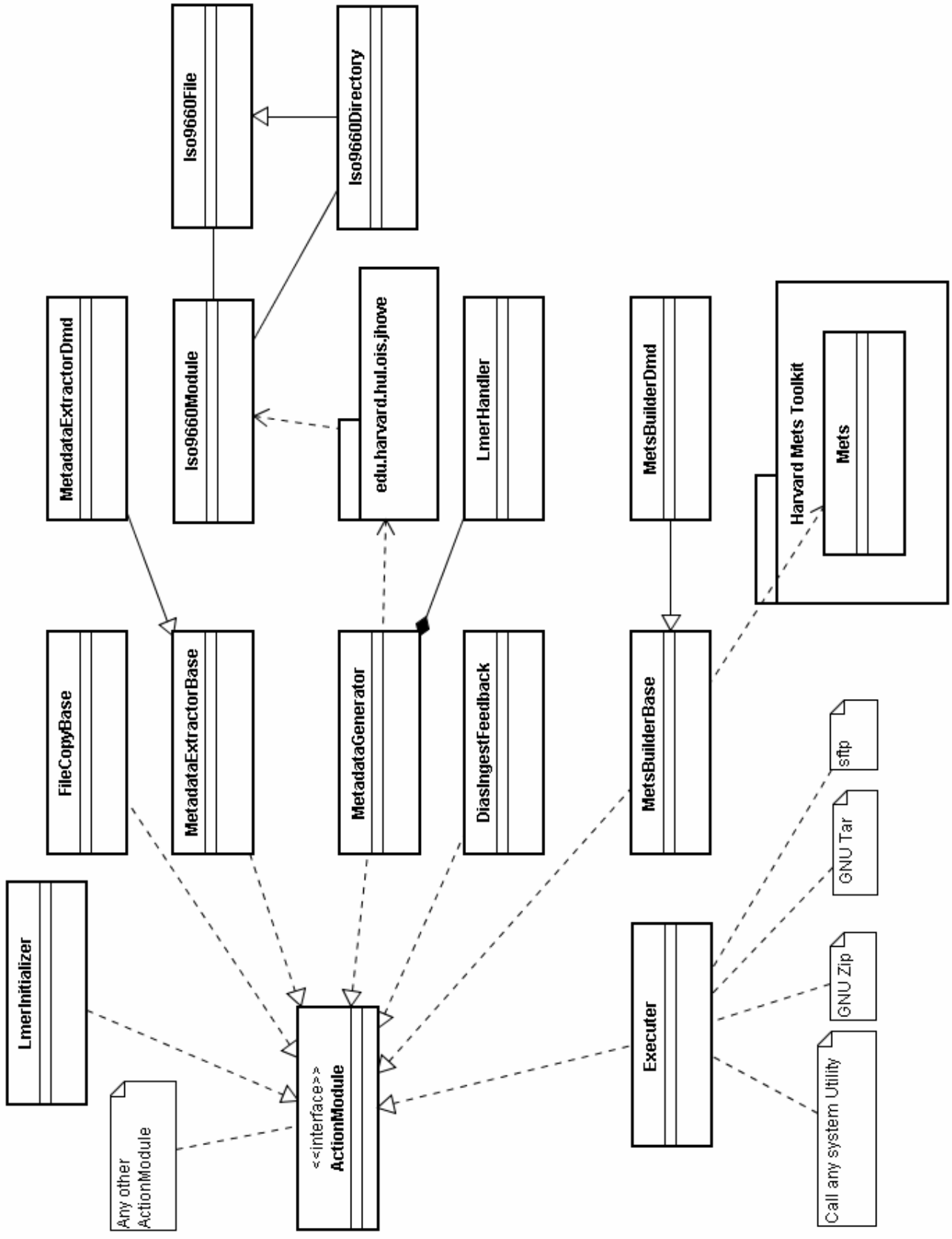
DiasAccess

```
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

Classdiagrams





Informations about the example configurations

There are two ways of using the example configuration files. For the first, a single machine can be configured to both create assets and submit them to an archiving system (1st Combination of asset creation and archive transfer). And further there is the possibility to configure multiple machines to create assets (from various workflows) and submit them to another central machine. This central machine receives all assets and transfers them to an archiving system (2nd Separation of asset creation and archive transfer).

1. Combination of asset creation and archive transfer

The file 'example_config_standalone.xml' is used with only one instance of the WorkflowTool class. To execute the example, simply the four paths to the work and temporary directories have to be supplemented (all tag values marked with '***' within <dirs>).

The used ProcessStarter is called 'ReadDirToCreateAssets'. He treats all subdirectories of the directory stated in <work_dir> as assets, and adds these directories to the ProcessList to process them. Of course, these subdirectories have to exist and contain files from which JHOVE can extract technical metadata to gain significant and useful results. Any file structures and any files can be contained within the subdirectories to test this module. For further informations please see the part 'The use of JHOVE in koLibRI'.

The policy from the 'policies.xml' configuration file is called 'standalone' here and invokes the following Actionmodule`s for each asset:

- a. ActionModule: MetadataExtractorDmd
Assumes some dummy values for some DC data. This module can be customized to own import workflows for descriptive metadata
- b. ActionModule: LmerInitializer
Initializes some data for the LMER section of the METS object.
- c. ActionModule: MetadataGenerator
Generates the technical Metadata. The JHOVE toolkit by the Harvard University is used for this.
- d. ActionModule: MetsBuilderDmd
Builds the METS file out of the gathered informations. The METS toolkit by the Harvard University is used for this.
- e. ActionModule: Zip
Compresses the content files including the METS file (mets.xml) into a manageable package.
- f. ActionModule: SubmitAssetToArchive
Transfers the completed asset to an archive of choice. This module (as well as 'GetDmdFromSomewhere') is a dummy module, which has to be customized for the respectively used system.

The command to start the standalone example looks as follows:
java -jar kolibri.jar -c config/example_config_standalone.xml -m

2. Separation of asset creation and archive transfer

The files 'example_config_assetBuilder' and 'example_config_clientLoader' are used with respectively one instance of the WorkflowTool class, where the division of work is as follows:

The so called AssetBuilder can be started on various machines (thus more than once) and process different workflows for the creation of assets (different workflows would surely require various policies and maybe according configuration files). In the present example, it processes the working steps of the first example first. 'ReadDirToCreateAssets' is used as a ProcessStarter here, also, with the difference, that the build asset is not transferred directly to the archive, but instead to the so called 'ClientLoader'. To do this, he uses the ActionModule 'SubmitAssetToClientLoader'. 'example_config_assetBuilder' is used as the configuration file and the used policy is 'assetBuilder'.

For the ClientLoader, a WorkFlowTool is started with the ProcessStarter 'ClientLoaderServer'. He then waits for requests on a defined port (<interaction_port> in the configuration file). If he receives such a request, je awaits to receive an asset from the AssetBuilder. These transferred assets are then added to the ProcessList and processed. Multiple assets form different AssetBuilder`s can be received at the same time.

The policy here is called "clientLoader" und and contains only one Actionmodule:

- a. ActionModule: SubmitAssetToArchive
The same ActionModule as in 1.f. and would transfer the according asset to an archive of choice (this function is however not contained in the given example).

The separation into AssetBuilder and ClientLoader is reasonable if more than one AssetBuilder is used to realize various workflows at the same time and create assets from various sources. The ClientLoader gathers all assets centrally from all AssetBuilder`s and can even do some optional tasks additionally for these assets (like burning them to a CD, re-validate the METS files, or maybe manage a central statistic of assets).

The command to start the AssetBuilder/ClientLoader example looks as follows:

Step 1, starts the ClientLoader:

```
java -jar kolibri.jar -c config/example_config_clientLoader.xml -m
```

Schritt 2, starts the AsserBuilder:

```
java -jar kolibri.jar -c config/example_config_assetBuilder.xml -m
```