Extraction of technical Metadata for longterm preservation :

The work with JHOVE in the “kopal” project

by Matthias Neubauer,
Die Deutsche Bibliothek,
Software developer for project “kopal“
January 2006

Table of contents

Introduction

The “kopal”[1] project, founded by the BMBF[2], has the goal to create a consistant long term preservation system for digital informations. This system is based on the gathering and creation of descriptive and technical metadata for the objects to archive. The software koLibRI®[3], which is a result of the work in project “kopal”, integrated the JHOVE[4] software to obtain the technical metadata.

[1] Kooperativer Aufbau eines Langzeitarchivs digitaler Informationen - http://kopal.langzeitarchivierung.de
[2] Bundesministerium für Bildung und Forschung – http://www.bmbf.de
[3] **ko**pal **Lib**rary for **R**etrieval and **I**ngest
[4] http://hul.harvard.edu/jhove/

Some facts about JHOVE

JHOVE stands for **J**STOR[1] / **H**arvard **O**bject **V**alidation **E**nvironment and has been developed from 2003 to 2005 by a collaboration of JSTOR and the Harvard University Library (HUL). It was released to the public as an open source project on may 26th, 2005. That release contains filetype specific modules for the following formats:

- PDF
- TIFF
- JPEG
- JPEG2000
- GIF
- AIFF
- WAVE
- XML
- HTML
- UTF8
- ASCII
- BYTESTREAM

[1] **J**ournal **Stor**age – The Scholarly Journal Archive – http://www.jstor.org

What JHOVE is

JHOVE itself is a skeleton, providing an environment for the filetype specific Java modules that are doing the main work by parsing a given file for its specific metadata.

It offers a great potentiality to extract technical metadata from all kinds of files in a consistent output format (e.g. XML).

To do this, JHOVE provides an open module interface that allows everybody to develop his own specific Java modules for a file format. Once developed, a module can be used by anybody without additional customization.

JHOVE is Open Source Software and was released under the LGPL[1]. That means it is

- freely extensible
- fixable by everybody, independent of the HUL
- and (independently of the LGPL) free of charge

[1] GNU **L**esser **G**eneral **P**ublic **L**icence **-** http://www.gnu.org/copyleft/lesser.html

What JHOVE is NOT

Against many expectations, JHOVE is NOT a wondrous tool, that can recognize all filetypes by instance. For each format that shall be recognized, a specific module is needed, and as stated before, very few modules exist at the moment.

Initially, JHOVE was developed to match the Harvard University Library`s requirements, and, as they can limit the formats in which contents are to be submitted to them, they had only needs for the present modules.

To match wider requirements of many institutions and to provide a stable environment for the validation of all kinds of objects, many modules will have to be developed in the future.

It should be added that the HUL ist not developing JHOVE any further at the moment. They support the current release and try to fix found bugs, but there are no plans for new features in the next maintainance releases.

According to Stephen Abrams, JHOVE project leader, there are plans for a JHOVE 2.0, but as resources are very limited (he alone is working on JHOVE now, as the 2-year project has come to its end), that are plans for a distant future only.

How JHOVE is used in koLibRI®

Intentionally, JHOVE is used as a command line or Swing-based Java application. In this way, JHOVE uses output handlers to present the created metadata on screen, or to write it directly into a text or XML file.

We use a combination of the METS[1] and LMER[2] metadata concepts to describe our assets. We encapsulate all available metadata into the METS container format, and write it into a file called “mets.xml”, which is then stored within the asset, together with the object relevant files.

For our ingest-software, we needed to access the created metadata *before* it is written to an output handler, and we decided to integrate JHOVE directly into our software. We included the Java JAR archive that is provided with the public release of JHOVE, in our project.

The functionality of the “JhoveBase” class was replaced by one of our own classes which implements our module interface. This way, we could optimize some methods for our needs, and delete unnecessary code. In short, this class manages the invocation of the JHOVE modules and the storage of the created metadata.

At the moment, our class works exactly like the original JHOVE strategy. That means, it invokes all modules in row, until it can find a matching module to the given file. We are planning to improve this method by checking the file extension of the file, and explicitly invoking the module that should match the according filetype. Only if the validation process fails for this chosen module, we are invoking all modules in row again, to find the real matching type. That would also be a workaround for a problem we found with invalid files of some types, e.g. TIFF images. If a TIFF image is not completely valid, the according JHOVE module rejects it, and JHOVE invokes all following modules again. This leads to a recognition by the BYTESTREAM module, instead of a concrete error message by the TIFF module. But as it is our intention to preserve only valid files, we would like to recognize these invalid files, and report its failure to the user, who can then correct the according file by hand.

This is also an issue if it comes to validation for a specific file. To truly validate the file to its format, JHOVE has to be invoked with the according module explicitly. Then it can tell if the given file is valid (assuming that the JHOVE module works correctly according to the formats specification).

JHOVE does include a concept for a “signature match” method, so that the modules should not try to identify a files format by themselves, but only rely on the signature[3] inside of the file. But sadly, this method is not well supported, and many modules do not even implement the necessary Java method for this functionality.

To be able to grab the created metadata directly, and map it to our own internal datastructure, we implemented our own output handler that inherits the

[1] **M**etadata **E**ncoding & **T**ransmission **S**tandard - http://www.loc.gov/standards/mets/
[2] **L**ong-term preservation **M**etadata for **E**lectronic **R**esources - http://www.ddb.de/eng/standards/lmer/lmer.htm
[3] Also known as „Magic Number“

functionality of the XML output handler. This “LmerHandler” is registered to JHOVE as the output handler in charge, and stores the created metadata into an internal datastructure that can be easily accessed by the software parts which create the “mets.xml” file later.

We have also improved the speed of processing a file by de-activating JHOVEs internal checksum calculation. After all, we are just using the SHA-1 checksum, so we calculate that in our own software. We also used a higher buffer for the calculation, and had impressive performance boosts after these improvements.

JHOVE might create lots of metadata, depending on the format of the file, and the used features of this format. In cases of assets with a high amount of included files, the “mets.xml” metadata file can become really big (we had files up to 120 megabytes). Thus we implemented some options to our class, that allow us to ignore some specific metadata sections. We do not use these options at the moment, though, as the currently created assets do not contain such large amounts of files.

The back side of customizing the JHOVE package is that upgrading to a newer version could mean changes to the code of our own software. But this applies only to a change in the architecture of JHOVE, and not to the expected maintainance releases.

Experiences with JHOVE in project “kopal”

It should be said in advance, that the support from the HUL for JHOVE is absolutely excellent. Reported bugs were fixed mostly within hours after sending them to the JHOVE mailing list.

But on the other hand, there were many bugs, especially in the module for PDF documents. Often, these bugs were related to the validation of the XML-output that JHOVE created for a specific file. The XML-output contained characters which are not allowed in XML, but were read in the PDF file itself. In one case, JHOVE completely hung up on a PDF-file, which could be tracked back to the “Outlines” section by debugging the Java sourcecode. In our fixed recompiled version of JHOVE, this section has now been taken out completely, that means it is not created by the module anymore. The eventuality that JHOVE could hang up our whole system, was considered much more unconvenient than the temporary loss of this section for some files. The solution for this problem is yet to be found, but Stephen Abrams has promised to investigate on this problem.

As for invalid characters in the XML-output of the module for PDF files, there were made some changes to the Java sourcecode by ourselves, for most of the time with support from the HUL and the JHOVE mailing list. Checks for these characters were added before writing it to the output, and if illegal characters appear, they are simply ignored. That way, the output stays valid, and no information is really lost, as these illegal characters would not have been readable anyway.

Another problem that led to invalid XML-output, was the existence of empty property-lists. The JHOVE XML schema insists, that a list of properties must at least contain one property. Some modules did create lists for metadata sections, before they even knew whether there would be something to fill the list with, or not. As a result, the XML-output contained some empty property-lists. By checking the content of these lists before writing it to the output, this problem could be successfully solved in the XML and PDF modules, where this error occurred.

A third major problem was the creation of a “subMessage” tag within the output - found in the HTML module for example – although this element was not defined in the JHOVE XML schema. This problem has already been fixed by the HUL, by adding the appropriate parts to the official JHOVE XML schema.

We also had (and still have) problems with wrong recognitions of file types. We had several images that were recognized as BYTESTREAM, as well as HTML files that where recognized as ASCII or UTF-8 files and vice versa. There would be a workaround, as stated in the description on how we use JHOVE, but an improvement of the modules file type recognition could also be useful.

Some of these problems may still exist at some other modules or sections, though, as we could not test all aspects of each filetype. In fact, PDF files were the format we have used most extensively with JHOVE yet, and so it must be considered that there might be an unknown amount of yet undiscovered problems in the remaining JHOVE modules.

Supported file formats and future plans

The “kopal” project uses the DIAS[1] longtime preservation system developed by IBM Netherlands. This system has its own set of filetype identifiers, which are listed in the following table.

Matching the existing JHOVE modules	**Unmatched**
“Unknown File Type“ (Bytestream)	Bitmap Image (BMP)
Adobe Acrobat Document (PDF)	PNG Image
JPEG	PKZIP File
GIF	TAR File
TIFF	Executable File
TXT File (ASCII)	Postscript
Hypertext Document (HTML)	Movie Clip (MPEG)
	Video Clip (Avi)
	PQI Image File
	Cascading Style Sheet Document (CSS)
	MS Word
	MS Excel
	MS PowerPoint
	MS Access

As it can easily be seen, there is a need of many more modules, than those existent at the moment. In fact, the DIAS filetype list is also far from complete yet. More needed file formats have to be added in the near future, to match the needs of the long time preservation plans in Die Deutsche Bibliothek.

There are plans for development of some JHOVE modules in certain institutions.

Stephen Abrams told us that the HUL is planning to develop a module for ZIP files. He did not say, though, if it will also cover other compression modes like GNU TAR. Also, this development is in an early state and might not be released in the near future.

Within project “kopal” we are planning to develop the most needed JHOVE modules first. See the following list for a short overview over the filetypes, for which we are planning to develop JHOVE modules in the near future.

- Disc Images according to ISO 9660
- Microsoft Office formats
    - MS Word
    - MS Excel
    - MS Powerpoint
- Postscript
- MP3
- PNG

[1] **D**igital **I**nformation **A**rchiving **S**ystem - http://www-5.ibm.com/nl/dias/

The module for ISO 9660 DISC IMAGES is already in development, and at a state of approximately 80% of completion. The modules for MP3 FILES and PNG IMAGES are going into development in early 2006.

We expect the development of a module for the MICROSOFT OFFICE FORMATS to be somewhat difficult, as the detailed aspects of these formats are kept corporate secrets by Microsoft. We are currently examining already existent open source JAVA tools, that already work with Microsoft Office formats, to see if we can use them to get an easier access into these formats.

The SUB Göttingen agreed on developing the POSTSCRIPT module, but did not yet start the development.

Summarizing, it would be very helpful, if a bigger development community for JHOVE modules would form. There are no other modules in development, nor have any been made public currently. The concept of JHOVE stands and falls with its modules, and all JHOVE users would benefit from sharing the development of modules. That could make JHOVE the mighty tool that it was made to be.

Another aspect to make JHOVE even mightier, would be the support for unique file format identifiers. It is rather difficult to match two (or even more) different identifiers for one single file format. In DIAS, for example, every file format is identified by a URN[1], so that a reference to this format cannot be mistaken for another format. As JHOVE uses only names like “PDF” together with a version number, e.g. “1.5”, these information easily can be misinterpreted. The lack of a global list of unique format identifiers, makes it rather difficult to map two different descriptions for one single format.

[1] **U**niform **R**esource **N**ame - http://www.persistent-identifier.de/

Conclusions

Using JHOVE for the creation of technical metadata has spared us a lot of time and work for developing a similar functionality by ourselves. In advance, an environment used by many users is always better than an individual solution, as it grows on its community. For JHOVE, that means especially development of new filetype modules that can be used by all other users afterwards.

But it has to be considered that JHOVE is still a very young project which suffers from undiscovered bugs, and not yet tested unexpectable circumstances.

If JHOVE is used in an every-day workflow, and one of these undiscovered bugs appears, it can even cost a high amount of time (and thus money) to identify and fix the problem, although the support from the HUL is very good, as stated before.

The solutions for the problems we found and fixed, will be included in the next maintainance release of JHOVE, so that every user can take advantage of them.

At the moment, we are using a version of JHOVE in our software that has been cleaned from the problems stated above, and will be replaced by the version from the next maintainance release, after it is made public and has been tested by us.