

SeManTiK: SEmantic MANagement of aTtachments Inside a wiKi engine

Nicholas Angelo Crespi
Fabio Panozzo

August 2, 2008

Contents

1	Requirement Analysis	3
1.1	The Problem	3
1.2	Establishing The Requirements	3
1.3	Storing Information: The Ontology	4
1.4	File Identification: Who Are You?	5
1.5	Meta-data Extraction: Making File Juice	6
1.6	Meta-data Storing and Updating	7
1.7	Meta-data Visualization	7
1.8	Knowledge Base Query	8
2	Implementation Details	9
2.1	File Identification	9
2.2	Meta-data Extraction	10
2.2.1	Audio	10
2.2.2	Images	11
2.2.3	Video	12
2.3	Meta-data Storing	12
2.4	Meta-data Visualization	12
2.5	Updating Meta-data	13
2.6	Knowledge Base Query	14
3	Issues	17

Preface

The world of Semantic Web is in continuous and constant evolution. Our effort in this project is projected towards the implementation of an innovative method of cataloguing for meta-information. In particular, its main goal is the implementation of an extension of a wiki engine that is required to manage meta-data extracted from attachments.

Chapter 1

Requirement Analysis

1.1 The Problem

The problem that is addressed in this paper is the automatic collection and visualization of attachment files' meta-data in any semantic wiki page, as well as the memorization in a Knowledge base (KB) in which it is possible to perform interrogations from the same wiki pages.

1.2 Establishing The Requirements

At a first glance, the problem can be coarsely subdivided into four main areas of interest:

Distillation: extract meta-data from a multimedia file and store them in a suitable place;

Visualization: meta-data should be printed just below the file link, in the appropriate page;

Filling: the wiki should give the user a chance to fill in any missing meta-data;

Querying: it should be possible to formulate queries about the stored information.

Further analysis of these points reveals that, in order to achieve our goal, we must find a storage method for our data. Furthermore, we need to find a method that identifies the file type (in order to discriminate about which files are multimedia and which are not) and a method for extracting specific meta-data from them. After these two steps, we need to find a method for storing the data we achieved in the aforementioned storage system. Finally, we need to find a way for showing the information gathered and for adding the missing ones.

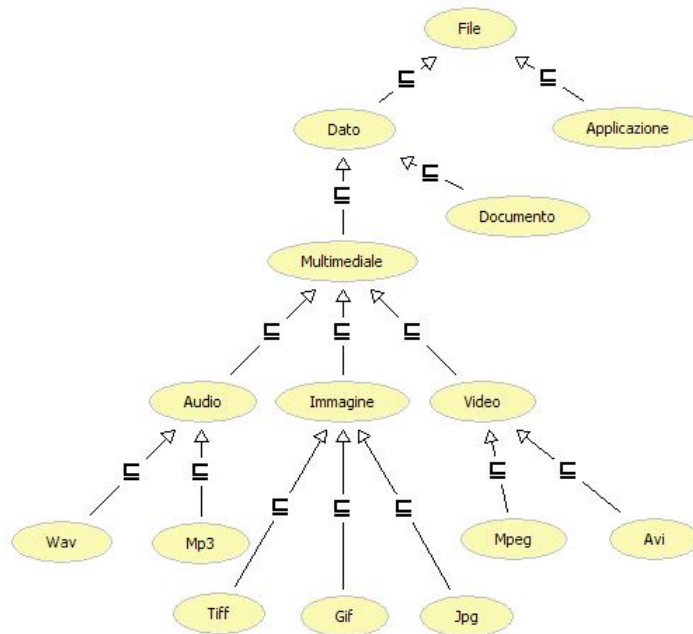


Figure 1.1: Ontology class diagram of the file domain.

1.3 Storing Information: The Ontology

The first question is: "Why an ontology? Why not a database?".

The simplest answer is: "Because (at least for our particular problem) an ontology is better than a database".

Now we explain it in detail: if you use a database, you can do only few operations (like selection and join). Instead with an ontology store in a KB you have in hand not only classes, but also properties which link classes. In any query you can extract information through proprieties.

The first step of any ontology design process is the modelization of the domain. In this case we're asked to store attributes related to special kinds of files: therefore we must first discriminate when a file is a multimedia file. As there isn't a standard taxonomy about files, we based our model on our knowledge, trying to make it as general and as scalable as possible.

The file taxonomy we came out with starts subdividing files in two broad categories: applications and data. Then we concentrated our modelling effort in the data category, which contains all the files we're interested into. The first subdivision of this category differentiates the type of media the format describes. Three categories has been modeled: audio, image and video, and among each one we decided to choose the most common file types.

If from one side this cataloguing has permitted to realize the part of the ontology which deals with files, on the other side we have to model a list of possible metadata possessed by an attachment file, in order to save this information.

Figures 1.1 and 1.2 represent two views of the KB structure.

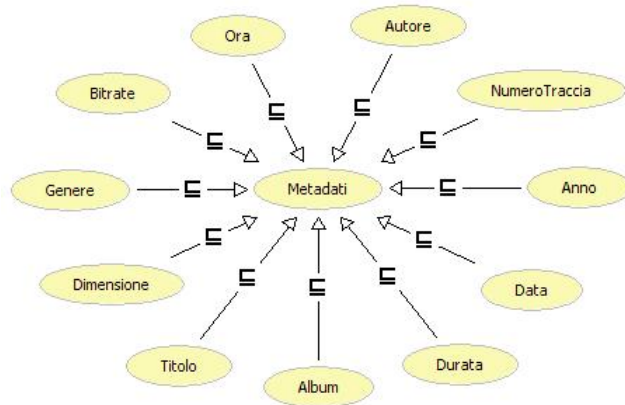


Figure 1.2: Ontology class diagram of the meta-data domain.

1.4 File Identification: Who Are You?

File identification is a tedious issue. Since we're writing an extension to a portable software, we want our identification method to be portable too. It must also discriminate at least the most common multimedia files. Furthermore, its response will be manipulated from inside our software without much effort. Furthermore, and this point is mandatory, it must be released with a free-software license. This requested feature should run every time any user upload an attachment to a page. Moreover, it should not perform the next phases if it detects that a file does not belong to one of the specified file types.

A simple file identification method, mostly used in DOS- based operating systems, discriminates the file type based on a three-char extension placed at the end of the file name, after a special delimitation character (a point). Unfortunately this method has many serious drawbacks. Two of the most important are:

- there can be a file which has no extension;
- the extension is part of the file name, so it could be modified by any person that has write access to it.

This can lead to ambiguity in the extraction phase, so it must be discarded.

Another file identification method, this time used in UNIX systems, looks at the file magic number. A magic number is a known pattern inside a file that identifies it with a specific type. For example, every shell script begins with a she-bang followed by the invocation of an interpreter, so any file that starts with this pattern can be identified as a script. There are several programs that implement this identification method, file being the most common (since it's provided with any Linux distribution). However, it isn't easily available for other operating systems, so we can't use it.

Remembering the portability requirement, we focused our search effort on Java file identification solutions. We found three different frameworks that conforms to that: JmimeMagic, JHOVE and Aperture.

The first, *JmimeMagic*, has a simple detection engine, based on file extension, so we discarded it.

Aperture is “an open source method for crawling and indexing informations”. It performs MIME type file identification by looking at magic numbers, and if anything goes wrong with that it falls back to extension method. It’s written in Java, which matches our portability and manipulation requirements. Another interesting feature of *aperture* is meta-data extraction: we will discuss about it in the next section.

JHOVE is a Java framework that performs many complementary task. It implements modular extraction procedures as *Aperture* does; but it goes even further, encapsulating all the file-related information in it. A *JHOVE* module for a certain file type is thus in charge of:

- keeping the identification pattern by which the file is identified;
- validate the file following some schema;
- extract meaningful information (meta-data), if they’re encoded.

Like *Aperture*, *JHOVE* has a standard list of modules shipped with it, and among these there are almost all of the file types we need; plus, *JHOVE*’s performances are similar to the *Aperture*’s one. So we have, by now, two possible candidates.

1.5 Meta-data Extraction: Making File Juice

Once a file has been identified, we must extract the meta-data provided by it. The fact that nearly every multimedia file type has its own meta-data internal representation makes the whole extraction hard. We need a meta-data extraction algorithm that can support different meta-data structure (at least EXIF tag and ID3 tag). We want it to be written in a portable language, for the above reason. A plus would be to find a modular framework, so that anyone can write a plug in for a new meta-data storage tag system. Finally, as stated for the previous point, it must be released with a free-software license.

Two of the frameworks mentioned in the previous section also provide meta-data extraction: *JHOVE* and *Aperture*.

Aperture has a modular approach for the extraction. Every extractor has its reference MIME type and will perform specific meta-data gathering (i.e.: an extractor associated with the MIME type “audio/MP3” will perform ID3 tag extraction). There’s a standard set of extractors maintained by the *Aperture* development team, and it’s also possible to develop third-party extractors by inheriting a set of interfaces common to all of them. The problem with *Aperture* is that the standard set of extractors does not support most of the files we need to handle, and we could not find third-party extractors that handles them. So we have to discard *Aperture*.

As previously stated, *JHOVE* implements modular extraction procedures as *Aperture* does; but it goes even further, encapsulating all file-related informations in a module. A *JHOVE* module for a certain file type is thus in charge of:

- keeping the identification pattern by which the file is identified;
- validate the file following some schema;
- extract meaningful information (meta-data), if they're encoded.

JHOVE has several standard file type modules, and this set almost completely covers our needed file type taxonomy. It also produces the output in several ways (two of them being text and XML) that are easy to process. Because of this and the properties stated in the previous section, we chose JHOVE.

1.6 Meta-data Storing and Updating

Once we gathered the meta-data, we must insert it in the ontology. We need to find a framework that is portable and free (for previously stated reasons). It must have a method of handling an OWL ontology, either by means of a querying language or in some other way. It must take as input the processed meta-data extracted from JHOVE output, so it must be easy to interface it within Java.

In addition we want to give users the option to complete missing meta-data. This means that the framework we're looking for must provide a way to search for existing instances and to add relations between them and the updated information.

Finally, there's some requirements about the implementation logic. When a user uploads a new version of a file, the system should clear all the information previously stored in the ontology and should create a new entry for the file (thus deleting all the previously inserted meta-data). Furthermore, the name of the file instance should contain the page in which it is attached to, in order to allow for having two pages with the same attachment (or two different attachments with the same name).

The most used framework when it comes to handling ontologies with Java is JENA. It is "...a framework for building semantic web applications". It provides an object-oriented environment for handling OWL ontologies, SPARQL query engine and a rule-based inference engine. Its development is sponsored by the HP semantic lab Web programme and it's carried on under an open-source compatible license (JENA license). Furthermore, we have some knowledge of this framework, as we've already used it in the recent past. So we chose to handle the integration between the knowledge base and the wiki application with it.

1.7 Meta-data Visualization

Once we gathered the meta-data, we must visualize it. Visualization must take place in the appropriate page and close to the file link. Furthermore, the option of completing missing information should be given only to authorized users (and not to anyone).

1.8 Knowledge Base Query

The last feature required in planning is the possibility to execute interrogations to the KB about the attached files and their metadata information.

The current knowledge about user interfaces for semantic search hasn't helped us much: if from one side there has been an enormous progress in information cataloguing and manipulation on semantic level, the research in this field is still moving its first steps.

In planning choice we decided to give the user the maximum possible freedom in interrogation formulation. For this reason we didn't think about a set between which the users can choose; we instead gave them the chance to specify their own queries autonomously, provided they are written in an appropriate language (SPARQL).

Like every choice, ours contains both advantages and disadvantages: user isn't limited from our contains both adv. and a designer decisions and from standard set of queries, but he must have two different kinds of knowledge:

- Knowledge about KB structure;
- Knowledge about syntax and semantic of languages of the query language.

Chapter 2

Implementation Details

Once we've established the requirements of this application, we moved on to the writing phase. The next sections will be about the low-level problems we have faced and the solutions we have come up with. We tried to mirror the previous chapter schema in order to allow for easy referencing between the requirements and the actual implementation. However, some points will be linked together, while some will be split in parts.

2.1 File Identification

The first thing to do, before even thinking about the identification algorithm we want to implement, is to surf the JSPWiki code, in search for the code fragment that handles the attachment of files. We found that there are two different methods that provide attachment handling: *BasicAttachmentProvider* and *CachingAttachmentProvider*. The second one is a "virtual" provider, relying on the first to get the job done, so we have to put our code in that one.

Inside the class, the function that actually does the transaction is *outAttachmentData*. Once the file has been uploaded, we put a call to our identification-extraction-storing class (the class itself has only one static method), *collectAndStoreMetadata*. This method accepts four parameters, which are:

WikiEngine engine: a reference to the running wiki engine;

String pageName: the page that will host the attachment;

String AttName: the name of original attached file;

File AttStore: the full path name under which JSPWiki had stored it.

Notice that, even if the last two parameters can be confused, they are different: *AttName* is the unmangled file name, while *AttStore* is the concatenation of the path where the attachment is stored and the "versioned" file name, automatically changed by the wiki engine.

As we know where the file is stored, we can then dispatch JHOVE on it, and we can save its output in the wiki working directory (extracted from the *engine* parameter). Then we can parse the output in order to identify the file.

The output of JHOVE is a well-structured list of information, each with an associated identifier: that makes it easy to parse it in order to find any piece of information. The file type identifier is “*RepresentationInformation*”. If the associated data is one of the wanted file type (except for MPG or AVI), we proceed to the next step. If it’s not one of them, we must also check if the file is a video, because JHOVE has no plug-in for them. Thus, if the original file extension is AVI or MPG, we handle the attachment as a video file; otherwise, we simply exit.

2.2 Meta-data Extraction

The second step of the insertion algorithm, because of JHOVE’s modularity, is a natural continuation of the first one. The invocation of this part tells us that the format has been identified and checked, and that the file has been validated (which means it’s well formed). The first thing to do is to extract common information from the textual output of JHOVE. As these are provided by the file system, they are always present, even if any other information is missing. They are:

- the file name, that will be used as an identifier (its identification string in the text is “*RepresentationInformation*”);
- the file dimension (its identification string in the text is “*Size*”);
- the date when the file has been last modified (its identification string in the text is “*LastModified*”).

After this first extraction, the algorithm differs depending on the file archetype. We have three basic types of file: audio files, image files and video files.

2.2.1 Audio

For this file archetype we want to extract (if present) information about the bitrate, the title, the author, the album, the genre, the year in which it has been recorded and the album in which it is contained.

Our ontology only models two types of audio file: mp3 and wave. The only information we can extract from wave files is the bitrate, because there’s no tag system that can be embedded in this file type. Conversely, mp3 files have a specific meta-data tagging system that’s also widely used, called ID3. It comes in two version, ID3v1.1 and ID3v2.4, and both they can be extracted using JHOVE.

Mp3

In this case, our algorithm first checks if the file has an ID3 header: if not, it extracts only the bitrate and then skips to the next phase. If the header is found, it searches for all the specific fields listed above. It prefers ID3v2 tags (since they can be accurate for a number of reasons) over ID3v1 if they're both present. The identification strings for the specific meta-data are:

- “*Bitrate Index*” identifies the bitrate of the mp3 file (in *kbps*);
- “*Album*” identifies the album in which the song is published;
- “*Artist*” identifies the author of the file;
- “*Genre*” identifies the genre of the audio file;
- “*Year*” identifies the year in which the audio file has been recorder.

When the end of the output is reached, even if some meta-data are missing, the algorithms begins the insertion procedure.

Wave

In this case, as there isn't any meta-data container that can be embedded into this file, the only information we can get is the bitrate. However, The WAVE module of JHOVE doesn't return this information, but it tells us the sampling frequency and the sample bit depth, so that we can combine them and compute the bitrate. The identifiers of these two meta-data are:

- “*BitDepth*” identifies the size in bits of every sample;
- “*SampleRate*” identifies the sampling rate (in *kHz*).

Once we have these two information, the bitrate is just the factor of them.

2.2.2 Images

Our file taxonomy contains GIF, TIFF and JPEG types of images, which are the most used image types (apart from PNG). The peculiarity of these three is that they all can embed EXIF tags, an image related meta-data tag system. EXIF tags contains useful information about the camera that shoot the image as well as many properties about the picture, such as the shooting date, the canvas size, the color encoding and so on.

If present, we are interested in the shooting date, which is indexed by JHOVE under the identifier “*exif:DateTimeOriginal*”. If the extraction algorithm find this string in the output, it takes the information, separate the hour from the date and replace the file date (one of the three meta-data common to all files) with the shooting date.

2.2.3 Video

In This case, unfortunately we can't extract anything useful, as there's no known meta-data container that can be embedded in them (and JHOVE doesn't have a module for parsing them). So, the only things we can automatically extract are the file size and the file date.

2.3 Meta-data Storing

The third step of the insertion algorithm handles the communication with the ontology. Now we have a file name, its type and a list of meta-data extracted from it. We want to insert them in the ontology with the appropriate relations.

As there's no other way to handle the population of the ontology, we have to use the object-oriented JENA API for ontology representation. It loads the ontology and creates a model of it, referencing it as a Java object (*OntModel*). After that, it's possible to create an individual by simply specifying the class it belongs to and its name. It's also possible to create a link between two instances, by adding properties to objects (using *OntProperty* objects).

The first thing to do is to check if the attachment is a new version of a previously uploaded one. If it's already in the model, we erase all of its previous associations and we proceed to add the newer ones; otherwise, we create the new file instance. After this step, we create one individual for each piece of meta-data extracted in the previous phase (each with the appropriate class type), and then we create the association between the file class and each meta-data class by using *OntProperty* objects. Finally, we save all the changes we made to the ontology, and then the algorithm ends.


2.4 Meta-data Visualization

Now we have to look for the code that writes the attachments information at the bottom of the page. After some search we found that each page is rendered starting by several templates that can be found in the *templates/default* directory of the deployed JSPWiki. The template that renders the attachment list is *PageContent.jsp*. This template makes heavy use of some JSP tags implemented in the JSPWiki engine (*wiki:AttachmentsIterator*, *wiki:LinkTo* and *wiki:PageInfoLink*) in order to iterate over the attachments and, for every file, it creates the link to it.


The simplest way to add semantic information about every attachment is to develop a new JSP tag and to put it in the template, between the iterator tags. In order to query the ontology for the meta-data linked to an attachment, this tag needs two parameters: the attachment name and the page name. Since we can extract the page name automatically, the tag requires only a single parameter.

The developed tag is called *wiki:AddSemantic*, and is implemented in the *AddSemanticTag* class that extends *WikiTagBase*. It uses the object-oriented

Attachments

[Eric Johnson - Cliffs of Dover.mp3](#)  3990836 bytes

Track n°: 2
Title: Cliffs of Dover
Length:
Author: Eric Johnson
Album: Ah Via Musicom
Year: 1990
Genre: 17
Bitrate (kbps): 128
File Size (byte): 3990836 bytes
Upload Date: 2008-01-29

[test.jpg](#)  389202 bytes

Title: Splashin' Lemon
Author: Unknown
Shot Date: 2007-12-30
Shot Hour: 08:28:21
File Size: 389202 bytes

Figure 2.1: A page showing metadata related to its two attachments.

JENA API (the same method used for storing information in the previous section) to read the ontology. After the model has been created it searches for the appropriate file entry and cycles between its properties, in order to retrieve the meta-data associated to it. If a specific meta-data is not present, then it creates an update form (which will be discussed later) that a user can fill.

The last thing we had to do is to put an entry for our JSP tag in *WEB-INF/jspwiki.tld* (in the JSPWiki deployed directory): it's an XML file that implements the JSPWiki tag library.

2.5 Updating Meta-data

While visualizing attachments' information we could incur in an attachment that misses some of the meta-data entries: if that is true, we have to provide a method for completing the entry.

A typical (and most used) way of asking the user for information is an HTML form. We decided to make an input form for every missing piece of meta-data. The forms passes the information gathered to a custom-built JSP page, *MetaUpdater.jsp*, that employs the JENA object-oriented approach (the same method used for insertion and visualization) and inserts the new meta-data in the ontology, linked to the appropriate file.

In order to identify the file entry we need to update, the updater page must know what file it has to look for and which page it belongs to. Furthermore, it also needs to know what kind of meta-data He needs to create and its new value. Between these four parameters only the last one is provided by the user: the other three are already used to search for the file and the meta-data instances in the *AddSemanticTag* code; we could thus reuse them from there.

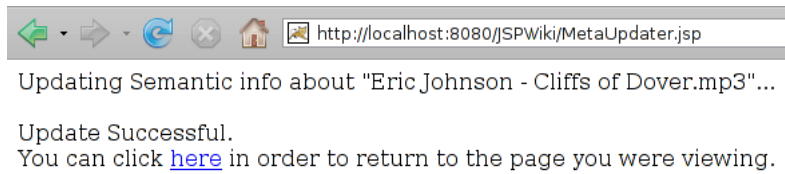


Figure 2.2: The results of a successful update.

The three search parameters are sent to the updater page via hidden input box (an approach widely used in HTML forms). Their identifiers are:

- metaType: refers to the ontology modelled class of the meta-data that will be inserted;
- className: refers to the file name (not the JSPWiki versioned one) of the attachment in question;
- pageName: refers to which wiki page has the file attached to.

The fourth parameters, whose key is *newVal*, is typed by the user in an input box.

The first thing the updater page does is to check if all the parameters are present. Then it loads the ontology creating its model using the JENA API and uses *className* and *pageName* as search keys to look for the object representing the file instance. If the object is found, it creates a new individual of class *metaType* and value *newVal*, links the two instances together and saves the ontology. If there were no errors, it displays a link to the previous page, where the meta-data information displayed will be updated. If not, it visualizes an error message.

2.6 Knowledge Base Query

To implement search in semantic wiki pages we decided to give user the maximum freedom: he can search for every piece of information about every attachment file in every wiki page and from every wiki page.

The solution has been implemented in the most portable mode possible.

First we modified the left menu adding a new link that, once clicked, brings the user in a new wiki page from where he can specify his query. Being the left menu in every page, it gives users the possibility to perform a search operation from any page in any moment.

This new page has been realized creating a file `Query.txt` that is recalled from JSPWiki engine on page upload. This file contains the call to a Java plug-in whose job is to create a text box in the page, in which the user can write the query, and a button that, once clicked, brings in the response page.

To show the query results we created a JSP page called `Query.jsp` (which, being a JSP page, contains Java code mixed with HTML code). This is a replica of `Wiki.jsp` (which is the one used by JSPWiki to show every wiki page) but with

[Main page](#)
[About](#)
[News](#)
[Recent Changes](#)
[WikiEtiquette](#)

[Find pages](#)
[Query for metas](#)
[Unused pages](#)
[Undefined pages](#)
[Page Index](#)

*G'day **root***
(authenticated)

Figure 2.3: The left menu showing the additional *query for metas* link.

Query

Your trail: [Main](#)

```
SELECT ?file ?author
WHERE {?file ONTURI:haAutore ?author}
```

Invia Query

In case you need it, you can use the domain namespace in the form **ONTURI:**

Figure 2.4: The query page, showing a sample query ready to be executed.

AnswerQuery

Your trail: [Main](#), [Query](#)

Domain URI is: <http://www.owl-ontologies.com/Ontology1197022770.owl>

file	author
Main/Eric Johnson - Cliffs of Dover.mp3	Eric Johnson

Figure 2.5: The results given by the execution of the previous query.

some additional routines: in fact, it doesn't take wikitext as an input and converts it into HTML, but it takes the query written by the user and shows the query results. To accomplish this task, inside this page we did not just use the functions provided by the JSPWiki environment, but we also used the ontology management and interrogation methods provided by the Jena libraries.

Chapter 3

Issues

While we were developing this project, several issues were discovered. As we wanted to create only a proof of concept, we didn't correct all of them. Here we will present some issues that are still unresolved:

- Albeit powerful, the object-oriented ontology model provided by JENA is also sensibly slower than the SPARQL engine. We have tested it over a small size ontology (something more than 100 instances) and it's not that slow, but a typical wiki installation (~1 million instances) could represent a serious performance problem;
- When a file is updated we zap all its properties away, but the meta-data instances that were linked to it are still in the ontology. This could pose a dangling reference problem, i.e. after the new version is uploaded and inserted, we could have meta-data instances in the ontology that aren't linked to a file. We think that this problem could be solved by creating inverse relations and crawling through them in the insertion phase, in order to remove all the meta-data instances that have no association;
- The input the user provides when updating a file's meta-data isn't checked at all. This could pose serious security problems (think SQL-injection). Thus, a sanitation function could parse the input in order to check if its content is appropriate for the meta-data type it should represent;
- There could be some synchronization problem when several users attach the same file on the same page, or when several users update the same meta-data field. This can happen because we use the object-oriented JENA API, and should disappear if we switch it to a query-based language (so that its engine would handle it).
- The Java code we have written isn't well engineered (in particular the *collectAndStoreMetadata* method) and it needs a serious refactorization;
- The HTML forms described in the updating section are hard-coded in the *AddSemanticTag* source code, which is an engineering flaw: if we want to change the ontology model we have to modify the source and recompile

it. A possible solution is to extract a template and place it in a separate file, filling it on request;

- JHOVE can't handle information about video files because there aren't plug-ins that can handle them: following the best practices available on JHOVE's site, one could write its own plug-in that manages a file-type (from identification to meta-data extraction).