chist-era

Image Retrieval Prototype

Ralf Bierig**, Alexandru L. Ginsca*, Adrian Iftene†, Pinar Duygulu Sahin‡

*CEA, LIST, LVIC

** Vienna University of Technology, ISIS, IMP

† "Al. I. Cuza" University

‡ Bilkent University

Contacts: lupu@ifs.tuwien.ac.at, adrian.popescu@cea.fr

MUCKE Project, Deliverable 5.3

18 December 2015

Contents

Abstract

This document describes the architecture of the MUCKE Image Retrieval Prototype. We present the prototype from the viewpoint of the information flows that are covered within the prototype. Descriptions for this part are provided by high-level flow diagrams for general understanding about the different use cases, how these interact, and how information travels within the prototype framework. The system architecture shows the prototype from a more static perspective and describes the packages that implement its functionality in a more detailed form.

1 OVERVIEW

This document describes the architecture of the MUCKE Image Retrieval Prototype[1] as one of the deliverables of the CHIST-ERA MUCKE project. We present the prototype from two different viewpoints.

1. Section 2 below documents the general flow of information through the MUCKE system, such as the process of indexing text, its retrieval and its output. The descriptions are provided by high-level flow diagrams and are aimed to provide a general understanding to the reader about how different use case processes interact and how information moves within the prototype framework.

2. Section 3 highlights the prototype from a static perspective and describes the packages that implement its functionality, i.e. indexing, concepts and search, as a more detailed view on its implementation with some information and suggestions of how it may be extended. The descriptions in this section use UML-styled high level descriptions that omit some of the many details contained in the code to enable the reader to gain an overview. The interested reader can access the full JavaDoc documentation and the code at `https://github.com/mucke/mucke`.

2 SYSTEM FLOW: OVERVIEWING THE PROCESSES

The purpose of the system is to enable users to create and retrieve multimedia content. This section highlights the general flow of information through the MUCKE system for a set of use cases that are important for MUCKE. The summary of all process flows are depicted in figure 1. The subsections that follow will highlight individual parts in more detail.

2.1 INPUT

The input for the system may come from two sources:

- The users, who may:
  - add new images and their associated metadata into the system (i.e. the raw data cloud on the diagram). All this information will then be processed and indexed;
  - query the system for images (i.e. the query cloud on the diagram);
- Various collections, which contain:
  - Images and associated metadata, as shown by the raw data cloud in figure 2 that is processed and indexed by the prototype system

[1] Based on the context in which it is described, we will also refer to it as the prototype or the framework.

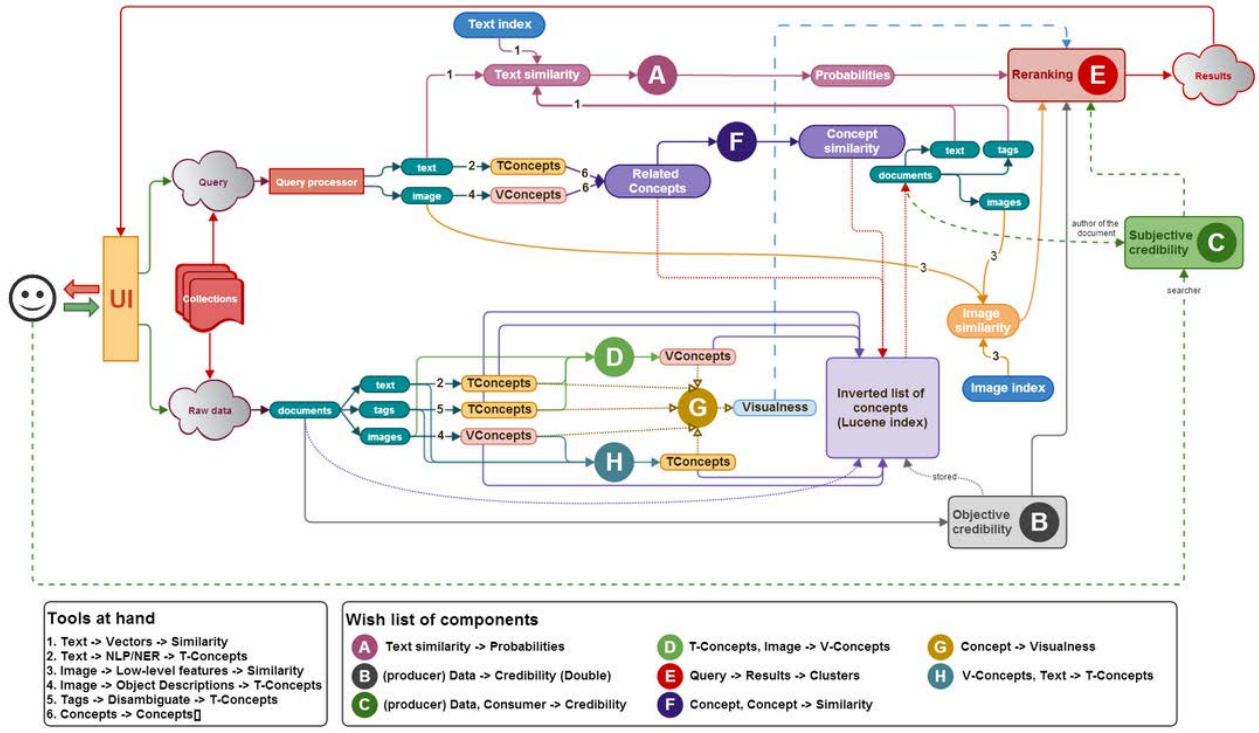

FIGURE 1: PROCESS OVERVIEW OF THE MUCKE IMAGE RETRIEVAL PROTOTYPE

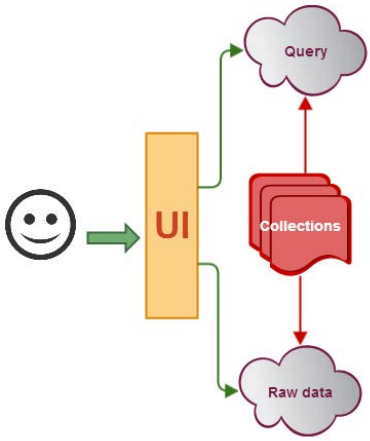

FIGURE 2: PROCESS OF SYSTEM INPUT BY QUERY OR TEST COLLECTION

– Topics from which queries can be extracted (i.e. the query cloud on the diagram).

This highlights the two modes with which the prototype may be used — an *interactive mode* that originates from a user interface and a *batch mode* that operates from a program and is aimed to support researchers to conduct classic IR experiments.

2.2 CREATING THE CONCEPT INDEX

Raw data data is organized as documents which comprise of images, text, and tags. These documents are processed using tools 2, 5 and 4, which extract text concepts from text and tags, and visual concepts from images. Although tags are also text, they are processed differently since they

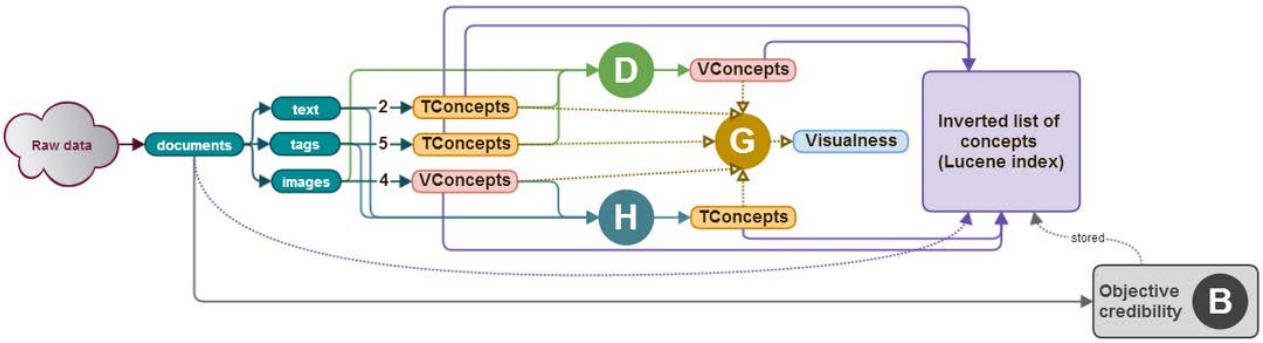

FIGURE 3: PROCESS OF CREATING THE CONCEPT INDEX

are very specific, short words that describe an image, unlike large blocks of text that may comprise of various phrases and thus may require more complicated processing techniques.

The documents along with the extracted Concepts are further used by component D, which uses machine learning techniques to produce visual concepts by analyzing text concepts associated with images, and component H, which uses machine learning techniques to produce text concepts by analyzing visual concepts associated with text. Component G uses all the concepts extracted so far and computes their visualness, an indicator of how easily the concept can be represented visually. A simple physical object, such as a chair, would therefore have a high visualness whereas an abstract idea, such as 'success' would have a low visualness. Visualness of concepts are later used for re-ranking. The extracted concepts and the documents are then indexed. The inverted list of concepts is stored as a Lucene index that contain, on the left side, concepts, and, on the right side, documents. For each document, objective credibility is computed by component B, based on the producer of the document. The value of the objective credibility is stored in the index (as a field in each document) and later it is used for re-ranking.

2.3 SEARCHING THE CONCEPT INDEX

A query processing component prepares the query from text before searching. A query may consist of text and/or an image. Tools 2 and 4 will be used for extracting the text concepts and the visual concepts from the query. Using tool 6, concepts related to those extracted from the query are found. Component F computes the concept similarity between them. These values will act as weights for the concepts when searching the index. The result of the search will be a list of ranked documents, which consist of text, tags and images.

2.4 TEXT SIMILARIY

Text similarity between the text from the query and the text in the result set is computed using tool 1. The values for similarity are transformed into probabilities by component A.

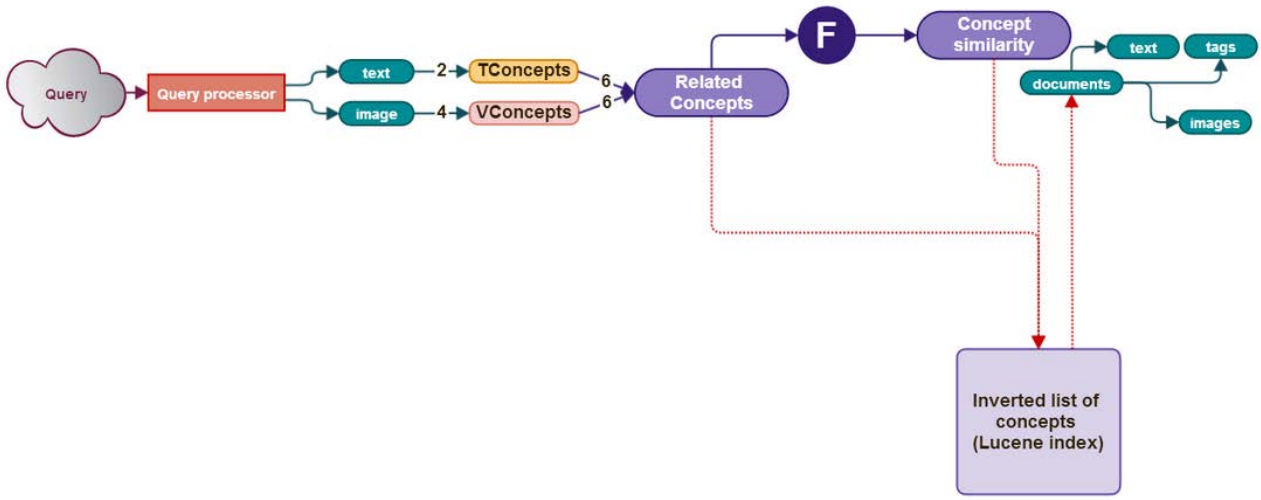

FIGURE 4: PROCESS OF SEARCHING THE CONCEPT INDEX

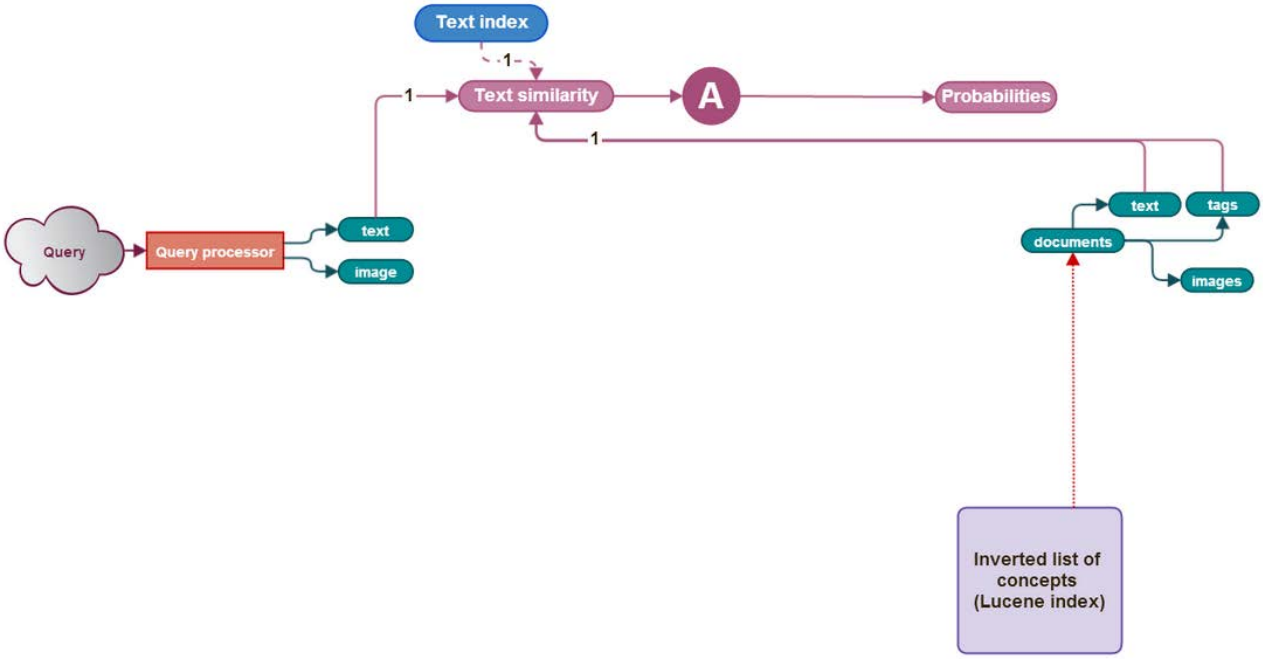

FIGURE 5: PROCESS OF TEXT SIMILARITY

2.5 IMAGE SIMILARIY

If the query contains a visual element, image similarity with the images in the result set is computed, using tool 3.

2.6 SUBJECTIVE CREDIBILITY

Subjective credibility is the result of the relationship between the searcher and the author of a document. Component C is in charge of computing a value for subjective crediblity.

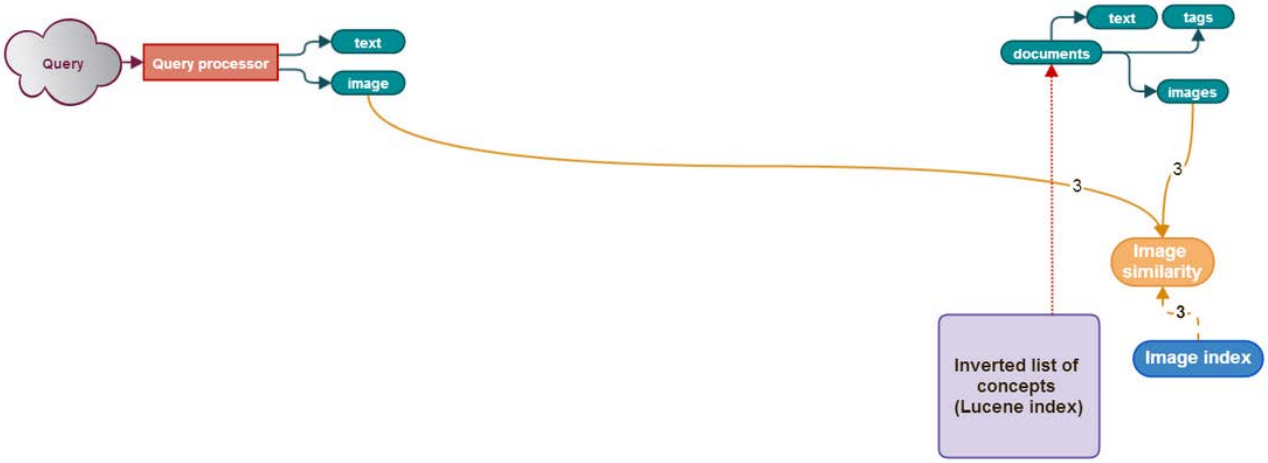

FIGURE 6: PROCESS OF IMAGE SIMILARITY

FIGURE 7: PROCESS OF IMAGE SIMILARITY

2.7 RE-RANKING

Because looking for related concepts might have broadened the scope of the search too much, re-ranking of the search results is necessary. Re-ranking takes objective and subjective credibility, visualness, text similarity and image similarity into account. Component E clusters the results before returning them to the user.

2.8 OUTPUT

Results are returned to the user interface as the output of the prototype.

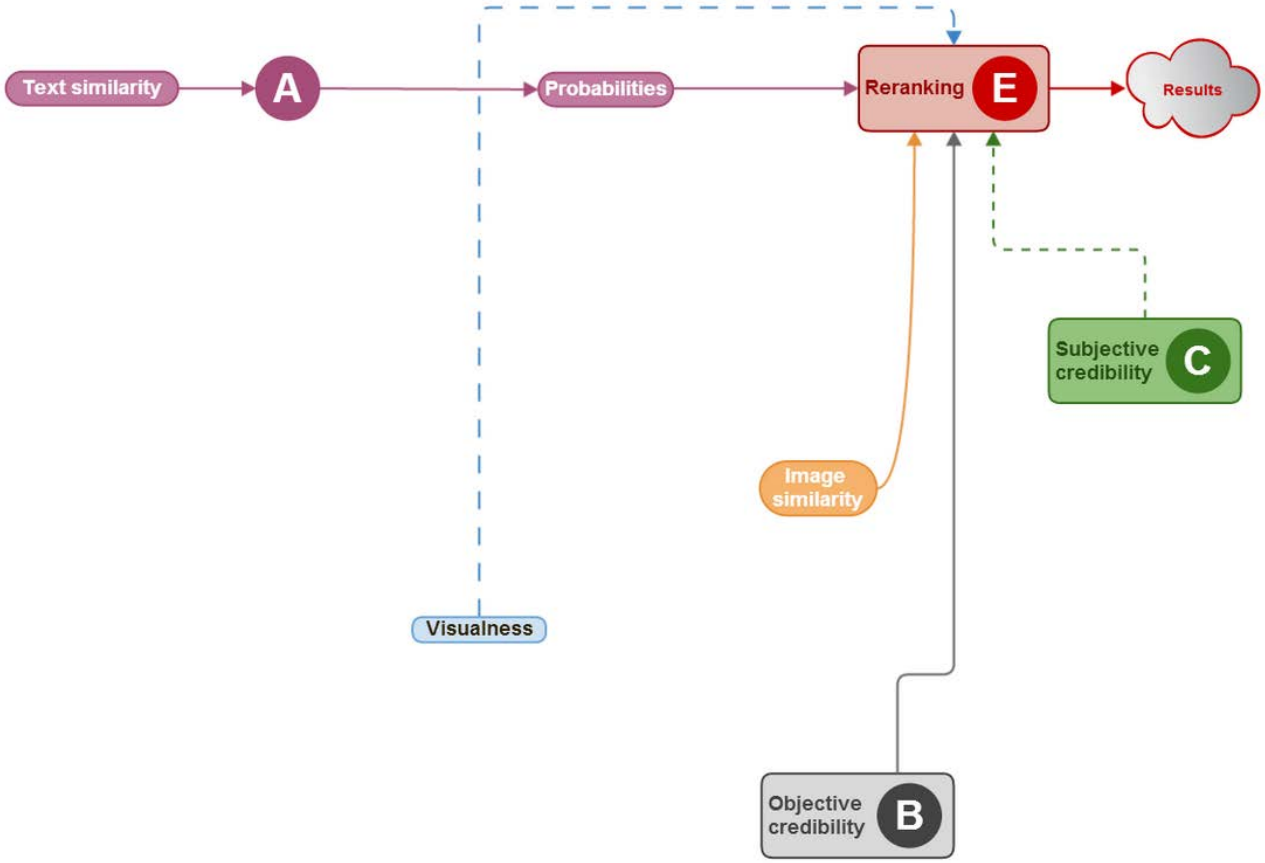

FIGURE 8: PROCESS OF RE-RANKING SEARCH RESULTS

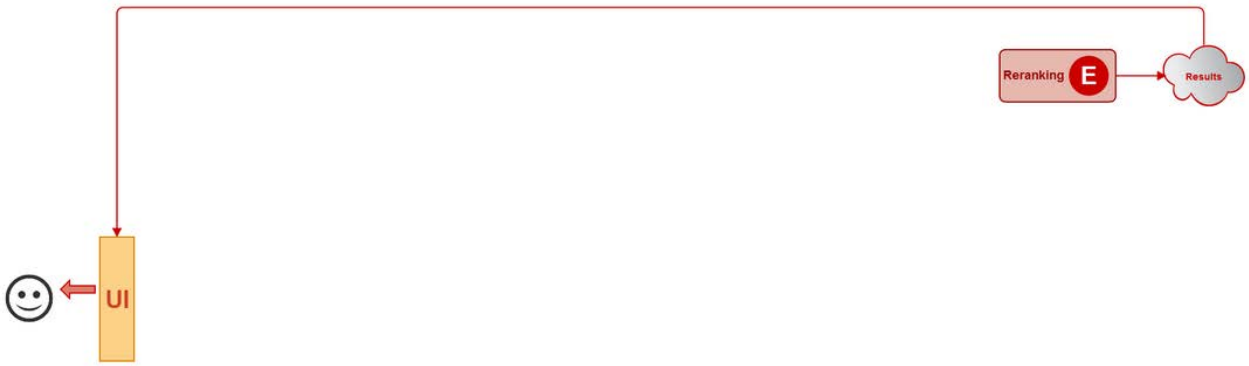

FIGURE 9: PROCESS OF OUTPUT

3 SYSTEM PACKAGES: THE COMPONENTS OF THE PROTOTYPE FRAMEWORK

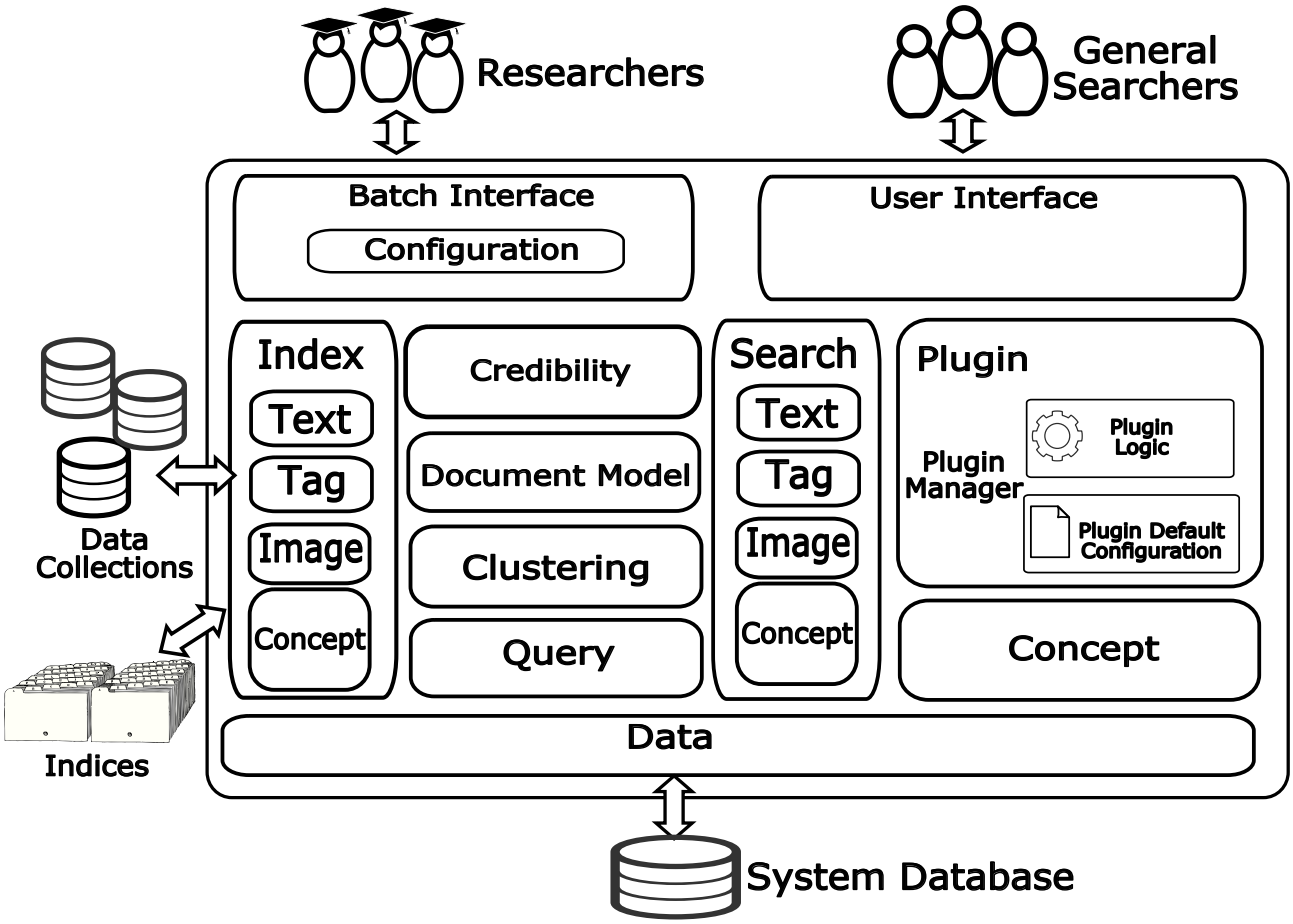

FIGURE 10: PACKAGE VIEW OF THE PROTOTYPE FRAMEWORK

The project proposal describes the purpose of the MUCKE multimedia system as enabling users to create and retrieval multimedia content. Based on the different processes described in the previous section, this part of the documentation describes the system from the perspective of its functional packages. This section highlights the prototype from a static perspective and describes the packages that implement its functionality, i.e. indexing, concepts and search, as a more detailed view on its implementation with some information and suggestions of how it may be extended. The descriptions in this section use UML-styled high level descriptions that deliberatly omit some of the details contained in the code. Interested readers may access both code and the full JavaDoc documentation at `https://github.com/mucke/mucke`. Figure 10 depicts an overview to all packages of the MUCKE prototype. The subsections that follow describe each package in more detail using UML-style class diagrams.

The MUCKE prototype architecture supports two types of users, as indicated in section 2, based on the type of input it supports. General searchers can operate it with a user interface and the framework can be accessed by such a user interface to inject queries and collect results for display.

Researchers, as the second user category, can access the system in batch mode to perform classic IR experiments. This is described in more detail in section 3.1. The *Indexing* package works with multiple collections and provides facet indices for text, tags, images and a document index that allows facets to be associated. Likewise, the *Search* package searches these indices. The *Document Model* package formalizes the document and its facets as a data structure for indexing and search and whenever content is used within the MUCKE framework. The *Credibility* package manages credibility scores that are applied for re-ranking in the Clustering package. The *Query* package manages queries to be transformed into query objects before being used for search. The *Concept* package models concepts and their generation. The *Plugin* package allows specific applications to use parts of the system by configuring it for its specific purposes or extending its base functionality with more specialized classes (e.g. writing an own indexer).

The following subsection describe each of the packages in more detail.

3.1 SYSTEMMANAGER, CONFIGURATION AND PLUGINS

The prototype framework is able to be applied for different corpora and IR test collections (although the full IR evaluation process is not yet supported within the framework). It provides tools that help assemble, process, index and search these collections as part of an IR evaluation process. It is not a general library, such as Lucene or Solr, but a more specialized high-level set of classes for the specific requirements of the project. These components can be configured to fulfill some of the research requirements of MUCKE with a potential for re-use afterward.

The architecture of the prototype maintains two configurations (each stored in a file) that each provide settings. Figure 11 depicts the basic structure of how these configurations interact. The central access point for the batch-mode is the SystemManager that coordinates both types of configurations. The next two subsections describe them each in more detail.

3.1.1 SYSTEM CONFIGURATION

MUCKE requires a small set of basic system settings that are needed to generally operate the prototype. These are the database setting for the unique system database, as shown in figure 10. These settings are kept in the system configuration file *system.properties* located at *mucke/mucke-backend/conf*. This *conf* is also the location for all other configuration files are are applied by the prototype. The listing below show one example of such a system.properties file.

```
#######################
# MUCKE primary properties file
# ============================
# This property file defines all general system settings
#######################

# a comma-separated list of property files
```

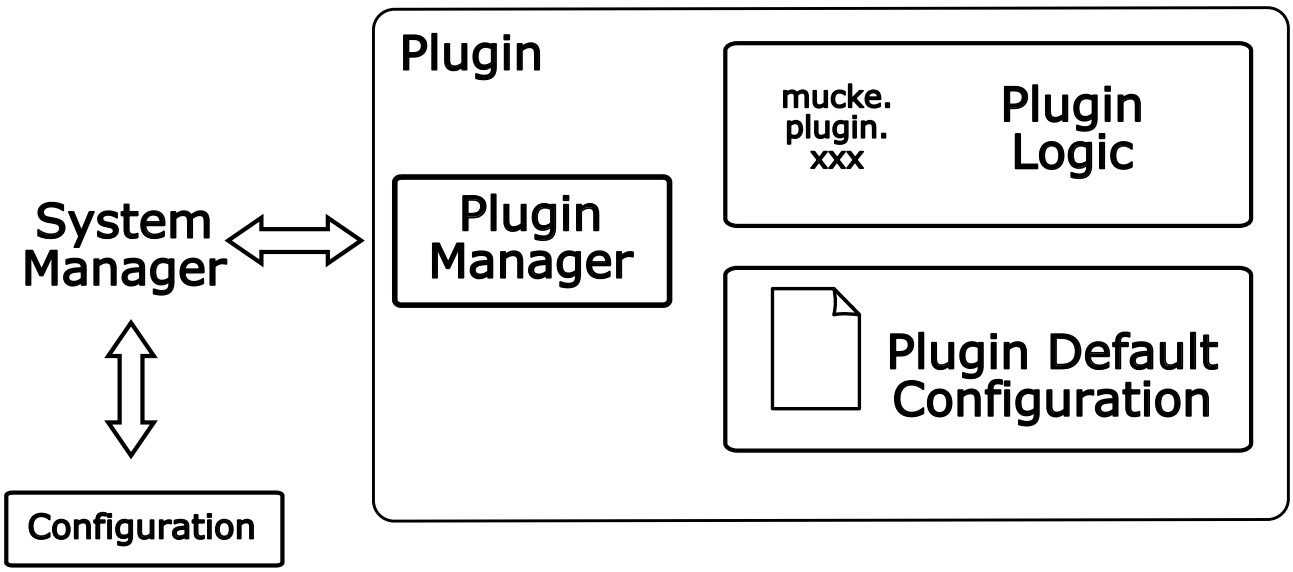

FIGURE 11: PLUGIN MECHANISM

```
# each property file creates a separate run
run.properties = prototypetest-crediblity-run1.properties

# connection details to the general system database
driver = jdbc:mysql://localhost:3306/mymuckeDB
user = mymuckeDB
pass = myPassword123
```

The parameter driver defines the Java database drive that is used to access and manipulate the database. During the project time, only MySQL was applied for this purpose, but the architecture should support any other type of database that provides a driver for Java. The next two parameters, *user* and *pass* define the credentials for accessing the database.

The parameter *run.properties* supports the researcher use of the prototype by connecting the general settings of system with specific settings for a plugin, as described in the next subsection. Each properties file must be located in *mucke/mucke-backend/conf*. The parameter allows multiple property files to be listed, comma separated, which allows them to be executed as so-called runs in batch mode.

3.1.2 PLUGINS AND PLUGIN CONFIGURATION

A specific extension of the MUCKE prototype architecture is called a *plugin*. Here, 'plugin' should be understood as an application or an extension as it either uses the general features of the prototype with a specific set of configuration or it extends its functionality with additional packages and classes to introduce additional features (e.g. adding a new indexer or a new query collection reader) as showed in figure 11. Plugins may handle, for example, specific collections and may focus on particular features of these collections. Each plugin has an additional configuration file that contains the settings of the plugin. Plugins run in their own environment and in isolation, thus one plugin

cannot access the configuration environment of another. Each plugin has a manager class that is inherited from PluginManager to provide a generic access point and a minimal set of features for the SystemManager.

One needs to distinguish plugin configuration in terms of default and an active instance; called a run. Each plugin occupies a subfolder in *mucke/mucke-backend/src/main/java/at/tuwien/mucke/plugin*. The properties file in this folder is the default confirmation for the plugin und named after the plugin. For example prototypetest.properties is the default confirmation file for the plugin prototypetest. The meaning and function of each property inside the property file depends on the plugin and is documented inside this file. To turn a plugin into a active (run) instance, one needs to copy this configuration file into *mucke/mucke-backend/conf* and adapt its parameters. Then, one needs to add the name to the run.properties of system.properties. Here, multiple active instances can be added to create a sequence of runs for the batch mode of the prototype.

By copying the default plugin properties file into *mucke/mucke-backend/conf*, one can create multiple altered versions of one plugin. These multiple variations of the same plugins can be used to test alternative parameters as experimental runs. The system can run them in a ordered list that allows them to be executed in a batch-like fashion. This offers the following flexibilities:

Specificity: Configurations can be made specific to collections (e.g. Wiki CLEF 2011 collection vs. flickr 2013 collection), different tasks (e.g. different research questions) or technical qualities (e.g. testing a plugin versus a finished demo of a plugin).

Partitioning: Configuration settings can be clearly separated from each other and exchanged without touching the logic of the source code.

Batch Mode and Workflows: When using the SystemManager and configuration files, MUCKE performs in batch mode. Then it executes a configuration as a so-called „Run“. Multiple of such configurations can be queued which means that MUCKE chains them and executes them as a workflow.

An example of a plugin configuration is provided in the listing below:

```
#############################
# Prototypetest plugin properties file
# ====================
# This property file defines the parametrization
# of the plugin with default values that will be
# used when the plugin is used in batch-mode.
#############################

# plugin manager
class = at.tuwien.mucke.plugin.prototypetest.PrototypetestPluginManager
```

```
##############
# index configuration
##############

# document index links facets with a document ID

# defines the location of the content that is parsed
docindex.contentfolder = D:/Data/collections/UserCredibilityImages/imageLists
# defines the document id: FILENAME, XPATH or REGEX
docindex.docid = FILENAME
# defines the expression facet signatures: XPATH or REGEX
#docindex.facetsignatureformat = XPATH
# defines which facets are used
docindex.facets = credtext
# defines wikidoc facet: location of facet id, facet type
docindex.facet.credtext = XPATH; /image/text/caption/@article; TEXT
# defines wikiimage facet: location of facet id, facet type
docindex.facet.credimage = XPATH; /image/@id; IMAGE
```

The configuration file is abbreviated in this document but available in full as part of the source code.

Shortcomings The high flexibility of the configuration comes at the cost of a rather complex layering of configuration with too many configuration parameters. In future releases, the amount of configuration parameters should be drastically reduced to a much smaller set where sensible default decision are defined (e.g. for the indexers) and set fix in the code without any options. This would reduce the settings to a much smaller set and simplify its use considerably. The design should gradually be moved toward more code extension and less parametrization but with a strong default set that covers a large proportion of application cases.

3.2 DOCUMENTMODEL PACKAGE

The DocumentModel package defines the structure of a MUCKE document. A MUCKE document may consist of any number of facets, that represent the types of content that MUCKE handles. There are three types of facets:

- *TextFacet:* This defines normal text in any proportion, such as documents, paragraphs, sentences and words.
- *TagFacet:* This defines text as short semantic descriptions with a controlled vocabulary. This is semantically different from the TextFacet, however, is indexed with the same technology as the TextFacet.

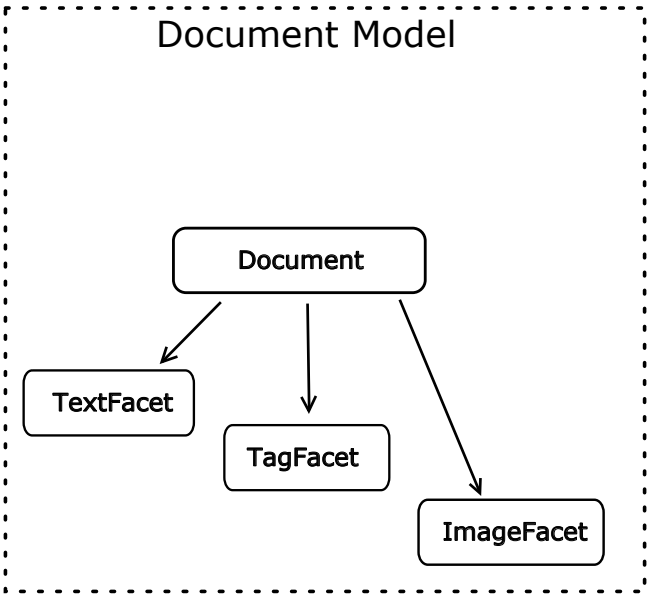

FIGURE 12: DOCUMENT MODEL

- *ImageFacet:* This facet defines images and in a multitude of binary data format, such as JPG, PNG or BMP.

A Document has an arbitrarily long list of Facets of either of the three types. This allows documents to have a flexible amount of different text, tag and image modalities in any order to enable creating a large number of possible types of documents. The document model, at the same time, deliberately does not support embedded documents or does not support any type of hierarchical structures with documents and facets. This would unnecessarily complicate the design of the overall prototype. Other components of the protoype architecture, when handling content, also handle these Document and Facet objects. For example the index package, as described on page 16, provides indices for each of the different types of facets plus an index for document itself.

3.3 INDEX PACKAGE

The index package provides interfaces and standard implementations for different types of indices that have emerged to be of significant value for the purpose of MUCKE. Two types of indexes are provided with the prototype architecture:

- *Facet indices* represents specialized indices for the different modalities that MUCKE supports. These are text, tags and images. The text and the tag facet index index are represented by a classic Lucene inverted text index where content is represented by a list of tokenized text fields. The image index is implemented using Lucene Lire that incorporates image features, such as color distributions, and binds them in specialized fields.
- A *document index* that enables developers to connect the previously mentioned facet indices together into more complex document structures. This document structure is described in more detail in the DocumentModel package on page 14.

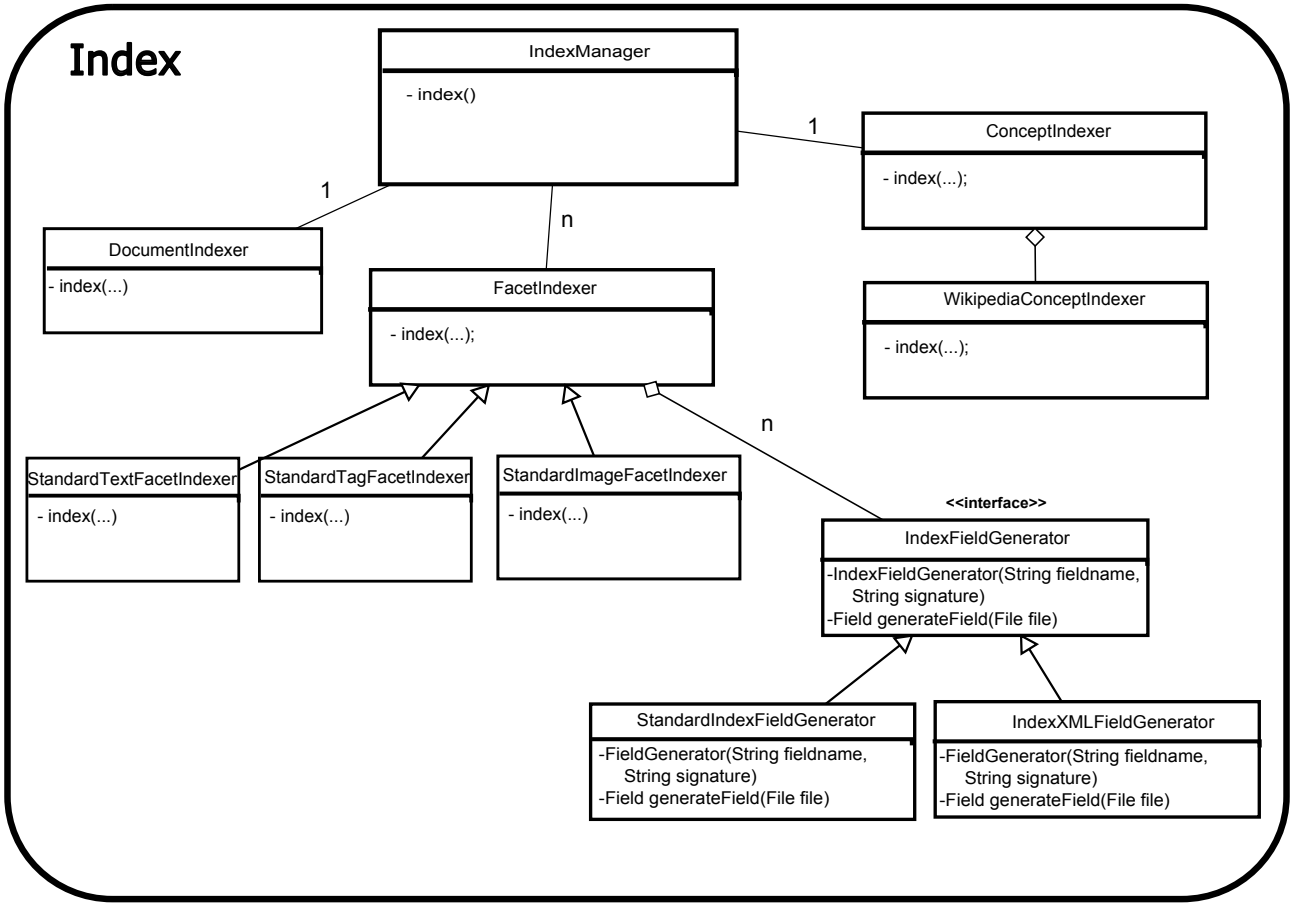

FIGURE 13: INDEX PACKAGE

The index package is controlled by an IndexManager class that controls a maximum of one DocumentIndex. That means, the DokumentIndex is optional and can be avoided, for example if only one facet is applied. The DokumentIndex is not meant to be extended. There can be one or more facet indices that each handle one of the three modalities of MUCKE: text, tag,

and image. Each FacetIndex can use any collection location to point to different content sources (called *contentDirectory*) which is processed and stored as an index in a specific location (called *indexDirectory*). The DocumentIndexer extracs information from a file that contains information about the entire collection document structure and creates a Database table that links the document identifier with identifiers of the facets. The database is accessed via the *Data* package. The DocumentIndexer performs therefore similar tasks like the TextIndexer w.r.t. the information it needs to know, but it stores it in a database and is does not employ FieldGenerators since it does not need Lucene Fields with their connection to the Lucene Analyzers (e.g. tokenization, stopword generation, stemming, linguistic processing etc.) but instead simply extracts ID information. Some of what IndexFieldGenerators do, it also uses, but in a more direct fashion to simply get to the information in the file. The DocumentIndexer is a core class of the index package and not meant to be modified or overwritten. The FacetIndexer has three standard implementations: StandardTextFacetIndexer, StandardTagIndexer, and StandardImageFacetIndexer that each present the default behavior for the three types of facets that MUCKE handles. Any FacetIndexer has an indexing method (called: *index*) that inherits the minimal indexing information (contentDirectory, indexDirectory and a list of IndexFieldGenerators). An IndexFieldGenerator takes a *file* and extracts a selection of content by accessing the file and extracting content based on a *signature* that defines what is going to be extracted. Signatures must be xpaths. The IndexFieldGenerator writes the result in a Lucene field object with the given *fieldname*. IndexFieldGenerators can be extended to implement different types of special parsers for generating fields. This means, if it is not possible to find an xpath expression that extracts the required selection of content, than one must overwrite die IndexFieldGenerator.

3.4 CREDIBILITY PACKAGE

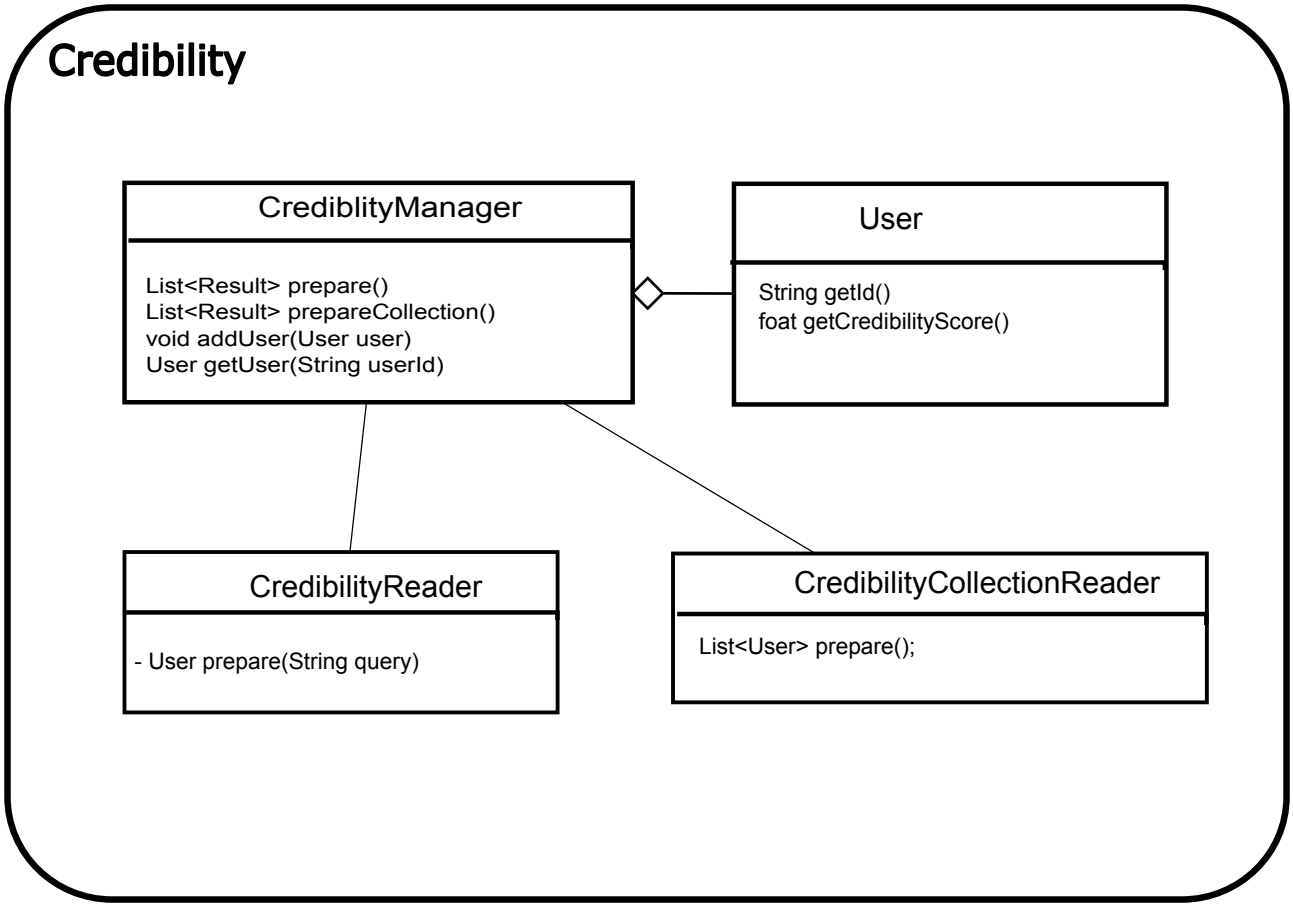

FIGURE 14: CREDIBILITY PACKAGE

The credibility package provides an interface to determine the credibility of a user. This feature is primarily used for the re-ranking process in the clustering package described in section 3.5. The credibility package has a CredibilityManager that provides access to all key features of the package. The manager class maintains and list of *User* objects that are uniquely identified with an *id* and a that have a *credibilityScore*. The manager triggers a CrediblityCollectionReader and a CredibilityReader. The prototype provides only interfaces to ensure a common format of its implementation but leaves the specifics to the developer. By implementing these two interfaces and its prepare method, the CredibilityManager can transfer a list of user information, either from a collection or individually, into a list of User objects that provide credibility scores.

3.5 CLUSTERING PACKAGE

The clustering package provides tools for merging results into ranked lists, basically tools for re-ranking search results of different modalities and considering the content similarity and the credibility of the results. The ClusteringManager is the main access point to this package that controls many possible ResultMergers based on the setting in the configuration (see 3.1). Merge results with a particular strategy requires implementing the ResultMerger interface. Currently, there is one

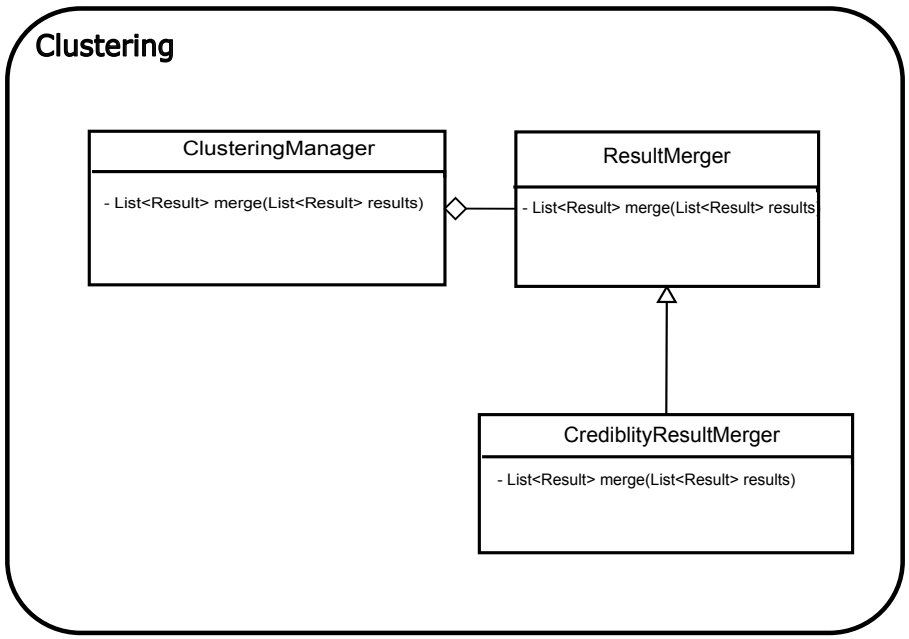

FIGURE 15: CLUSTERING PACKAGE

implementation: CredibilityResultMerger. This ResultMerger re-ranks results based on the credibility scores of the content producers of results. It does that by first discovering the owner of content (i.e. a Facet) and then accessing the credibility of that content producer via the Credibility package. Then it combines the search score with the content producer's credibility score as a harmonic mean and returns the list of re-ranked results to the caller.

3.6 QUERY PACKAGE

The query package provides an interface to process user input strings or a set of query strings from a collection, when applied in batch-mode. The QueryManager handles all access to the query package. A Query consists of Facets, based on the DocumentModel package, and can be a TextFacet, a TagFacet or an ImageFacet. These facets can be added and retrieved from the object and are created by either the QueryReader or the QueryCollectionReader (for sets of queries). A QueryReader takes a string and returns a single Query object by using the *prepare* method. A collection of queries are processed by a QueryCollectionReader. The prototype framework provides implementations for these interfaces for the case of XML queries, that is when the query has followed the well defined XML format. The XMLQueryReader and XMLQueryCollectionReader represent these implementations. The specifics of how queries are detected and organized are defined in framework configuration, as shown below.

```
########################
# Prototypetest plugin properties file
# =====================
# This property file defines the parameterization
# of the plugin with their default values that will
```

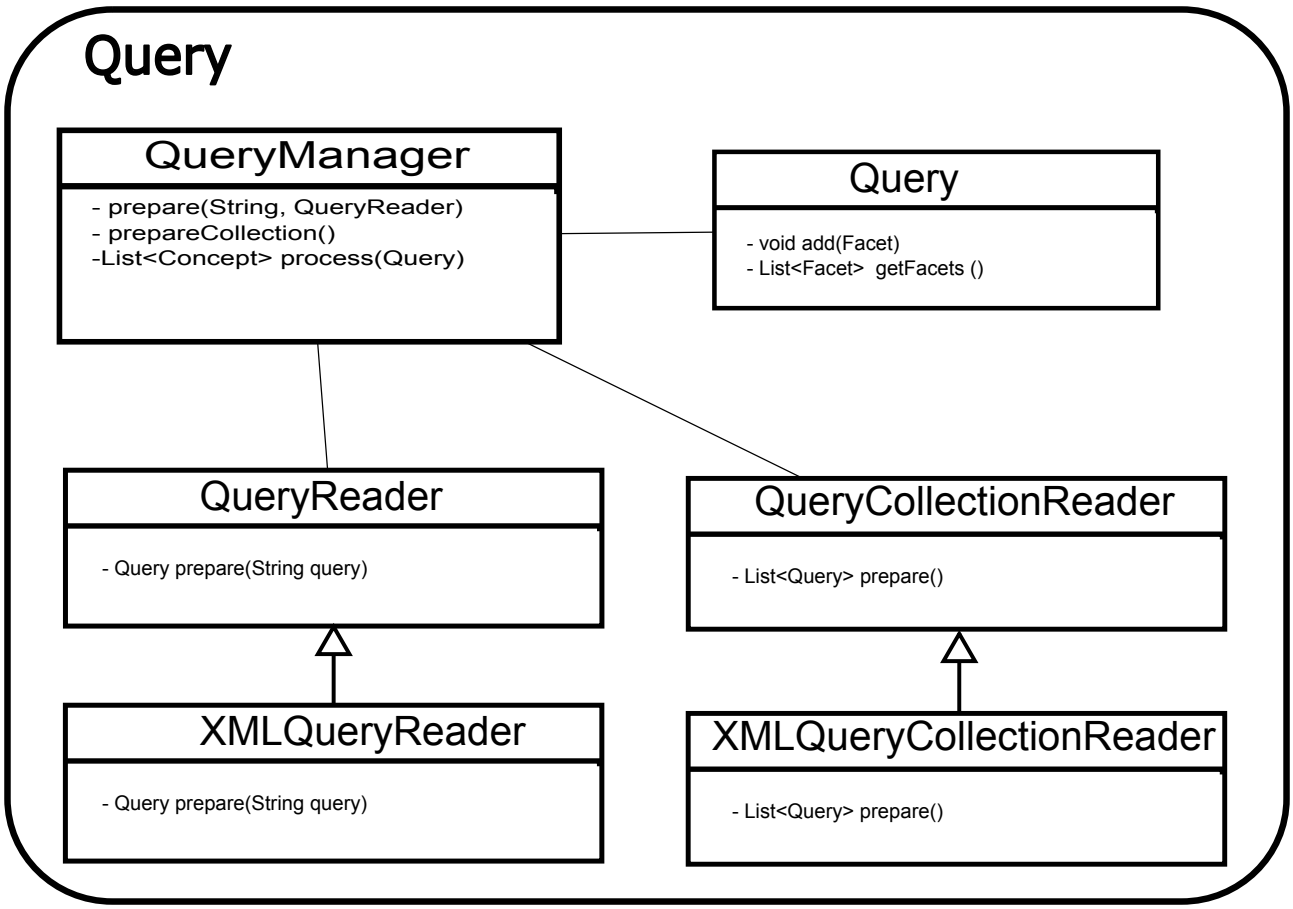

FIGURE 16: QUERY PACKAGE

```
# be used when the plugin is used in batch-mode.
##########################

# plugin manager
class = at.tuwien.mucke.plugin.prototypetest.PrototypetestPluginManager

.... lots of index configuration ....

#####################
# query configuration
#####################

# defines the query collection file
querycollection.file =
  D:/Data/collections/UserCredibilityImages/topics
# defines the query collection reader:
# a collection-specific narrow reader without parameters
querycollection.reader =
  at.tuwien.mucke.plugin.prototypetest.query.FileQueryCollectionReader
```

As you can see from this configuration file, the plugin extension 'prototypetest' uses a specific implementation of a QueryCollectionReader. If the developmer wants to read queries in a format other than XML, the developer must implement code that extends QueryReader and QueryCollectionReader that handles these particular formats (e.g. CSV or binary formats).

3.7 SEARCH PACKAGE

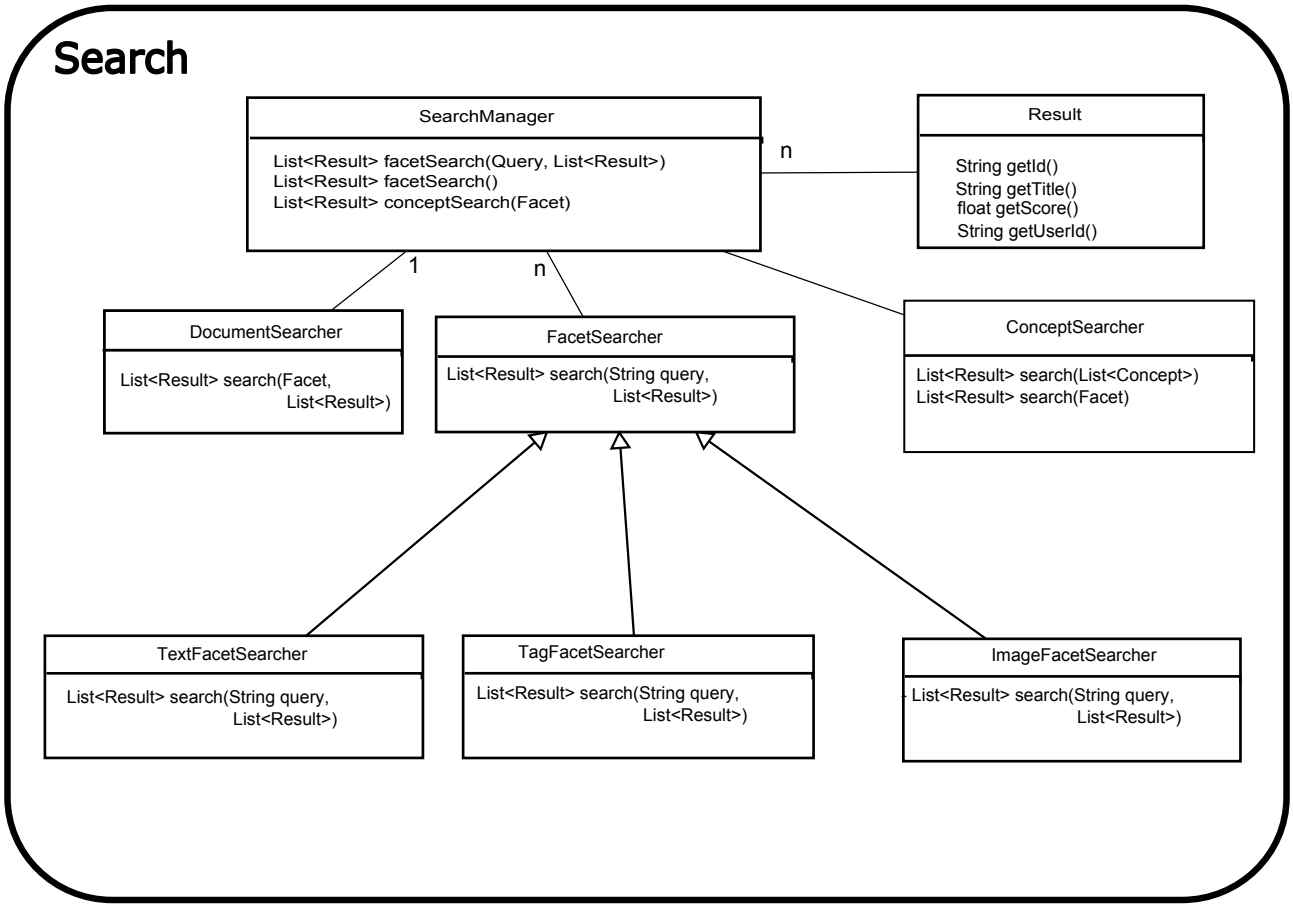

FIGURE 17: SEARCH PACKAGE

The search package provides the search interface for the MUCKE architecture via the SearchManager. Two types of searches are possible: DocumentSearcher allows searching for a Document based on one of the Facets (as described in the documentmodel package) and FacetSearcher allows to search facet indices. These FacetSearchers come in three flavours: TextFacetSearcher, TagFacetSearcher and ImageFacetSearcher. Every FacetSearcher works with a query string and an optional list of already existing results which will be used to filter the final results. The DocumentSearcher is based on a Facet instead of a query text and is fundamentally a straightforward lookup from the Document index as described in section 3.3.

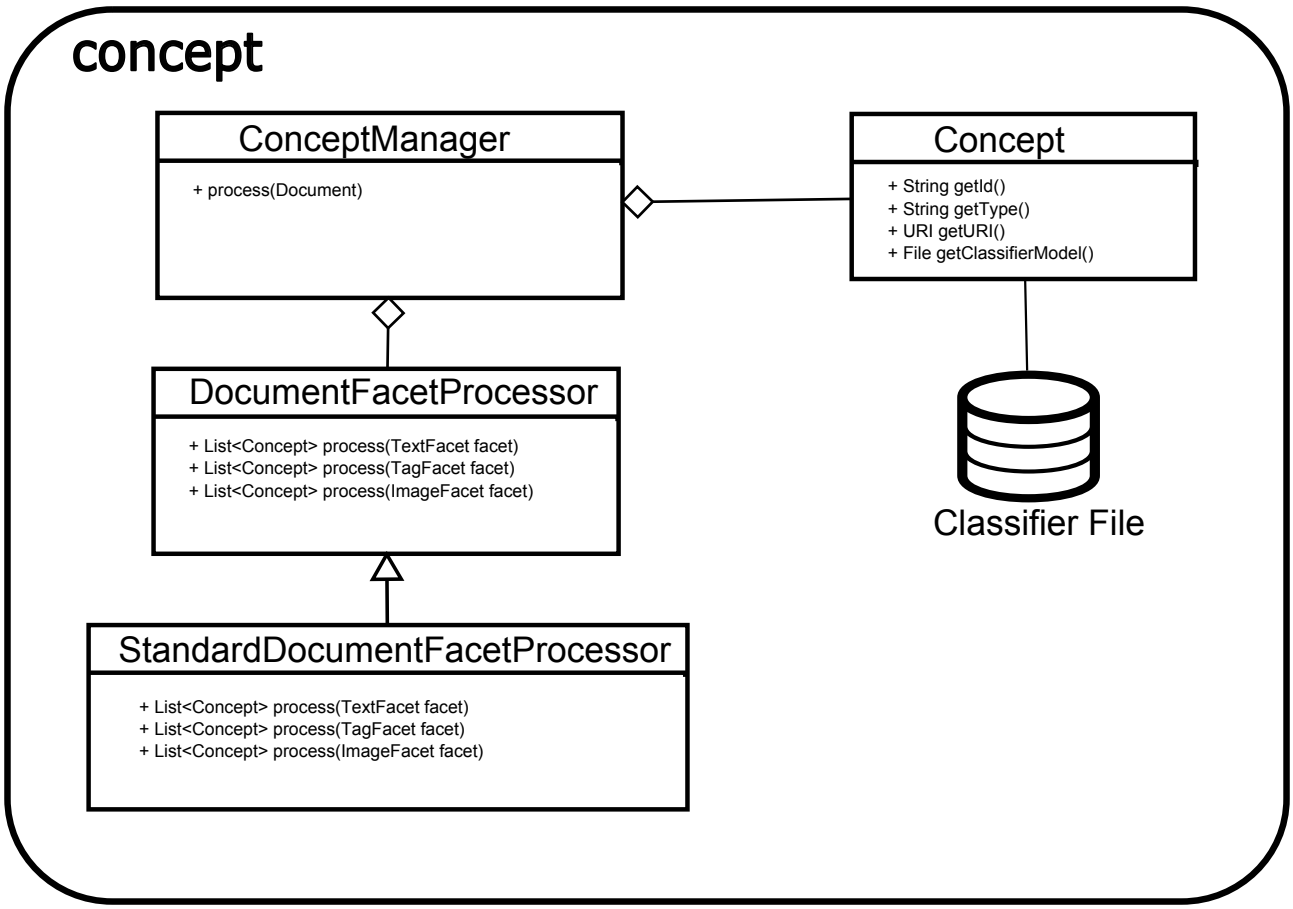

FIGURE 18: CONCEPT PACKAGE

3.8 CONCEPT PACKAGE

The concept package contains the classes used to describe the concept model. Every indexed document consists of one or more types of facets: TextFacets, TagFacets and ImageFacets corresponding to the different types of data a document can contain. By processing these fields, a list of concept objects is obtained.

A concept can be of either two types: textual and visual. These two types, however, are united in one concept class and distinguished with a type attribute. Every concept has an unique identifier and an optional link to a file that contains a classifier (e.g. a Markov model or a neural network). Sometimes models can be too complex to express and standardize. They are linked and not stored locally for reasons of simplicity and data size. It makes also very little sense to include their structure in a Java class since their processing will most likely not be performed in a uniform development environment but more likely with specific languages and environments (e.g. Python or Matlab). If there is no implicit classifier model that describes the concept, then this file link is set to null. This also represents the most common case.

A concept might often refer to a Wikipedia article or another point of reference that offers a unique representation[2]. Should a concept not have such a unique reference, than it can be created.

[2]As agreed in the MUCKE S3 meeting

Is the concept very fine-grained and hard to pin down (e.g. a particular texture of oak wood), then is shall be defined by pointing to the concept "oak" or "oak texture" and further refined by linking to a classifier model that identifies this particular type of oak texture. With this combination, it is possible to define very fine differences. Should an application require less explicit forms of concept representation, then this architecture allows representing any kind of concept with a unique identifier (as a URI) plus an optional pointer to a file that defines its implicit knowledge structure.

In order to extract concepts from facets, the visitor pattern is used to extract concepts from three document facets (for text, tag and image facets) with alternative implementations of how to identify concepts. DocumentFacetProcessors can be implemented to define how concepts are identified from facets (e.g. how text is turned into a lists of concepts). It is most likely that this part of the framework turns out to be rather suboptimal for practical use. Instead it is more likely that complex concept detection is performed completely outside the framework for issues of performance and required resources. This will result in DocumentFacetProcessors not to be used as frequently. The concept package however does model the general concept object representation to build a standard for internal use when used within the prototype framework.

3.9 DATA PACKAGE

The Data package provides a centralized access point to the underlying MySQL database for the MUCKE prototype. The data package contains a DBManager that provides methods of instantiating all DB tables that are required by MUCKE and methods for creating and maintaining entries for the document index as described in section 3.3.

3.10 USER INTERFACE PACKAGE

The user interface package provides the web interface that a user can use it in order to look up for images matching his query. The current query box also allows users to select if they prefer credible sources (meaning that content from credible users is favored). For more detailed instructions how to install the demos and the framework, please see appendix 3.10 and 3.10.

FIGURE 19: EXAMPLE USER INTERFACE

Here an example when searching for Castles in Austria:

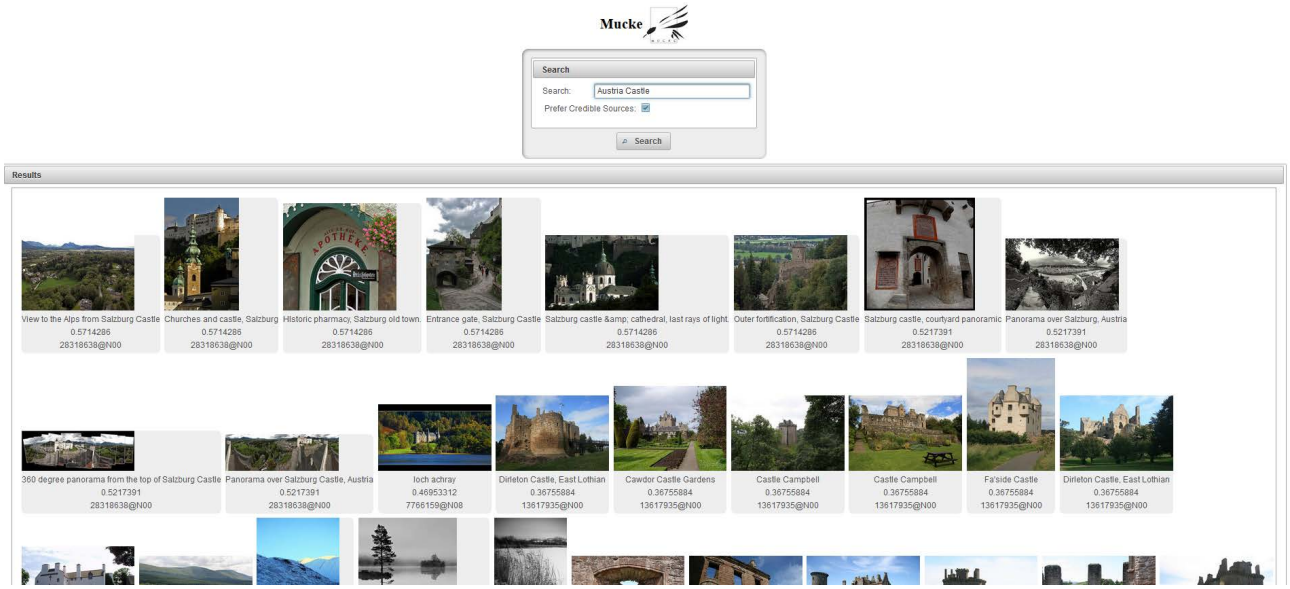

FIGURE 20: EXAMPLE USER INTERFACE WHEN SEARCHING FOR AUSTRIAN CASTLES

APPENDIX A: USER GUIDE

PURPOSE

This user guide describes the software environment that is needed to install and run the MUCKE framework and its demos (mucke-ui and mucke-concept-ui). The document also documents troubleshooting items that have been discovered in the process. This guide will however not teach you how to install IDEA, Maven or Git. There are better guides for all those tools that you find on their respective websites.

WINDOWS SETUP

BASIC DEVELOPMENT SETUP

- Install an IDE. We recommend IntelliJ IDEA Community edition because of its seamless integration with Maven. It is available here: `http://www.jetbrains.com/idea`. You can also use NetBeans. Eclipse does not integrate well with Maven and is harder to use.
- Install Maven from here: `http://maven.apache.org`. We provide specific build and deploy commands as part of this documentation.
- Install Git. If you use Windows, you can use TortouseGit from here: `https://code.google.com/p/tortoisegit`. All source and source documentation (e.g. system diagrams) are depolyed on GitHub. More later.
- Install MySQL. (Linux users, please check appendix 3.10). If you use windows, you should install XAMPP (`https://www.apachefriends.org`). Configure it so that you have basic security and a database called ‚mucke' with a root user and a password. You will need it later for the configuration of both demos.

BASIC DEVELOPMENT SETUP (WHEN USING NETBEANS)

- Install Netbeans from `https://netbeans.org/downloads` (choose Java EE option)
- Install MySQL from `http://dev.mysql.com/downloads/installer` (choose the default installation). In order to configure MySql, open the MySql Command Line and follow the instructions from section 3.10, step 4 - step 7.
- Install Git. If you use Windows, you can use TortouseGit from here: `https://code.google.com/p/tortoisegit`. All source and source documentation (e.g. system diagrams) are depolyed on GitHub.

GET CODE AND DOCUMENTATION

- Clone the MUCKE respository on GitHub (see `https://github.com/mucke/mucke.git`) to a directory (e.g. C:/mucke, we call this directory from now on *MUCKE code directory*) on your machine. This folder will then contain a sub-folder ‚mucke' with the folders
  - mucke-backend (the core MUCKE framework, or backend code)
  - mucke-ui (the image search engine demo user interface shown at ICMR 2014)
  - mucke-concept-ui (the concept demo user interface)
  - doc (documentation files, e.g. system architecture diagrams and user guides).

- IntelliJ IDEA: Open the pom.xml with as a project. Do this for each mucke-backend, mucke-ui and mucke-concept-ui separately as they represent separate projects.
- Netbeans: File → Open project (go to the place you cloned the mucke project and select mucke-backend, mucke-concept-ui, mucke-ui pom files).

GET THE DATA

- Create a ‚data' directory on your local files system with two subfolders:
  - collections (for various collections (only needed if you build own indices))
  - indices (for various index folders)

To run the demos (no index building) you need the following from the data disk. Note that each folder contains zipped archives that you need to unpack first.

- */data/indices/ConceptDemo* folder for the concept demo
- */data/indices/UserCredibilityImages/prototypetest* for the image search (prototypetest) demo
- */data/collections/UserCrediblityImages* for the image search (prototypetest) demo. Later, you put the /data/collections/UserCrediblityImages/images folder into the Tomcat/web-apps/mucke-ui-SNAPSHOT-1.0/ folder when you run the image search demo.

GENERAL CONFIGURATION

A configuration consits of one system.properties file and one, more specific, properties file for the plugin with arbitrary naming — in our case 'conceptdemo.properties' (for the concept demo plugin) and 'prototypetest-interactive.properties' for the interactive image search demo plugin.

- Access MUCKE-CODE/mucke-backend/conf [3] and open 'system.properties'. This the main properties file and contains the database login details to your local SQL database.
- Make sure you set them right (database user and password for a local database).
- Your database needs to run on the default port 3306

Configure and Run Interactive Search Demo Configuration (mucke-ui)

- Open system.properties and set the property „run.properties" so that it reads: „run.properties = prototypetest-interactive.properties"
- Open prototypetest-interactive.properties and edit the property 'credtext.indexfolder' so that it points to the Wikipedia index folder (as described in section 3.10 so that it reads for example:

[3] Note that MUCKE-CODE represents your MUCKE code root folder

  - credtext.indexfolder = D:/data/indices/UserCredibilityImages/prototypetest/credtext

- Maven is used for the code compilation and the creation of the .war file. Clean and compile with Maven:
  - *mucke-backend:* Go to MUCKE-CODE/mucke/mucke-backend folder and execute 'mvn clean install' to build the jar and install it in the Maven repository on your machine.
  - *mucke-ui:* Go to MUCKE-CODE/mucke/mucke-ui and compile and package the code as a .war file with the following command: „mvn clean package“. You should now find a /target directory with a .war file in it (mucke-ui-SNAPSHOT-1.0.war)
- Copy the .war file to your tomcat webapps directory
- Modify the ImageServlet.java file so it point to your image collection.
- Start Tomcat (tomcat should now create the folder of your demo webapp)
- Start MySQL (via XAMPP or otherwise)
- Point your browser to `http://localhost:8080/mucke-ui-1.0-SNAPSHOT`

LINUX: CONFIGURE AND RUN INTERACTIVE SEARCH DEMO (MUCKE-UI)

- *Step 1.* Install software requirements (example commands for a Debian based distribution):
  - git (sudo apt-get install git)
  - tomcat (sudo apt-get install tomcat7)
  - maven (sudo apt-get install maven)
  - mysql (sudo apt-get install mysql-server mysql-client)
- *Step 2.* Configure git installation
  - git config –global user.name "your name here"
  - git config –global user.email username@example.com
- *Step 3.* Specify in Tomcat's default configuration file which java sdk to use (in Debian, the file is “/etc/default/tomcat7”), by writing the following line:
  - JAVA_HOME=/path/to/java/sdk
- *Step 4. (Optional)* Configure Tomcat and MySQL to start on system boot (if you don't want to manually start them after every system boot in order to access the framework)

- *Step 5.* Start MySQL and Tomcat services (by default MySQL listens on port 3306, and Tomcat on 8080)
  - sudo service mysql start
  - sudo service tomcat7 start
- *Step 6.* Create the MySQL database for MUCKE framework (see appendix 3.10)
- *Step 7.* Get the data collection (see "Get the Data" section) and put it somewhere where you and Tomcat can access it.
- *Step 8.* Set the permissions accordingly on the folder containing the data collection (and all of its subfolders and files). The Tomcat installation must be able to read this data. One solution is to set 644 permissions for files and 755 for folders.
- *Step 9.* Clone the MUCKE repository somewhere in your user's home directory
  - git clone `https://github.com/mucke/mucke.git`
- *Step 10.* Go to "mucke-backend/conf" folder and open "system.properties" file.
- *Step 11.* Set the database connection details
  - driver = jdbc:mysql://localhost:3306/mucke # your database name
  - user = mucker # your username
  - pass = 123456 # your password
- *Step 12.* Set the property "run.properties" so that it reads:
  - run.properties = prototypetest-interactive.properties
- *Step 13.* Close 'system.properties' file and open the 'prototypetest-interactive.properties' file.
- *Step 14.* Edit the property 'credtext.indexfolder', so that it points to the Wikipedia index folder (as described in 3.10), so that it reads for example:
  - credtext.indexfolder = D:/data/indices/UserCredibilityImages/prototypetest/credtext
- *Step 15.* Close 'prototypetest-interactive.properties' file, go to 'mucke-ui/src/main/webapp/WEB-INF' folder and open 'system.properties' file.
- *Step 16.* Edit the property 'credimage.contentfolder' so that it points to the images folder, so that it reads for example:
  - credimage.contentfolder = D:/data/collections/UserCredibilityImages/images
- *Step 17.* Close the 'system.properties' file.

- *Step 18.* Set the *JAVA_HOME* environment variable by editing the file '.profile' and adding the following line (note: the path must be the same as the one that you set in Tomcat's configuration file at step 3):
  - export JAVA_HOME=/path/to/java/sdk
- *Step 18a.* Only when you first edit the '.profile' file execute also the following command; for the following logins, you don't need to do this anymore (note: specify the same path as above):
  - export JAVA_HOME=/path/to/java/sdk
- *Step 19.* Go to "mucke-backend" directory. In there run the following command (it will build a .jar file, store it in 'target' subfolder and also copy it into the local repository of Maven (by default this is located in '$HOME/.m2/repository/at/tuwien/mucke/mucke-backend/1.0')).
  - mvn clean install
- *Step 16a. (Optional)* If your 'mvn clean install' fails with errors finding other libraries than the backend, it might be that it does not find the above folder. In this case, add the following lines to the 'pom.xml', replacing the repository path to your local 'm2' path
  - ```
    <repository>
    <id>local-libraries</id>
    <url>file://<your home folder>/.m2</url>
    </repository>
    ```
- *Step 16b. (Optional)* If the above command doesn't copy the .jar file into Maven repository, you have to do that manually:
  - cp target/mucke-backend-1.0.jar ~/.m2/repository/at/tuwien/mucke/mucke-backend/1.0/
- *Step 17.* Go to 'mucke-ui' folder and execute the following command (this will create the mucke-ui-1.0-SNAPSHOT.war package in 'target' subfolder):
  - mvn clean package
- *Step 18.* Copy the .war file into your Tomcat's 'webapps' folder (on Debian, this is found in '/var/lib/tomcat7', which we consider to be $TOMCAT_HOME):
  - cp target/mucke-ui-1.0-SNAPSHOT.war $TOMCAT_HOME/webapps/
- *Step 19.* Go to the 'webapps' folder and set 644 permissions on the .war file.
- *Step 20.* Change the owner and group for the .war file to the user and group that Tomcat uses (in Debian these are 'tomcat7'). This is needed for Tomcat to be able to automatically deploy the app.
    ```

  - chown tomcat7.tomcat7 mucke-ui-1.0-SNAPSHOT.war

- *Step 21.* Restart Tomcat server.
- *Step 22.* Access the app: `http://localhost:8080/mucke-ui-1.0-SNAPSHOT`

ACCESSING THE TOMCAT'S INSTALLED APPS (SPECIFIC STEPS IN THE CASE OF IASI SERVER)

- *Step 1.* Create a SSH tunnel by connection to the console:
  - from Linux, use the command: ssh -C -L 8080:localhost:8080 -p 3223 <username>@85.122.23.93
  - from Windows, using Putty, go to Connection → SSH → Tunnels, in 'Source port' field insert 8080 and in 'Destination' field insert 'localhost:8080'
- *Step 2.* Open your browser and load the URL: http://localhost:8080/your-app-name.

UPDATING/DEPLOYING A TOMCAT APP (SPECIFIC STEPS IN THE CASE OF IASI SERVER)

- *Step 1.* By default, each account contains the MUCKE repository in '$HOME/git_repositories/' folder. Fetch the new sources by making a git pull.
- *Step 2.* Configure necessary properties files to point to correct local paths where you have placed different data. *Note:* The data collection for the mucke-ui demo is located in '/opt/-mucke_framework/data'
- *Step 3.* Build the .war file.
- *Step 4.* Copy the .war file to the Tomcat's webapps folder (/var/lib/tomcat7/webapps). *Note:* When copying, please remember to rename your .war file to a unique name so that it doesn't conflict with other existing apps; for example:
  - cp target/mucke-ui-1.0-SNAPSHOT.war /var/lib/tomcat7/webapps/new-app-name.war
- *Step 5.* Open the app in the browser (Tomcat will automatically deploy the .war file):
  - http://localhost:8080/new-app-name

MYSQL INSTALLATION ON LINUX MACHINES (TESTED WITH UBUNTU 12.X)

- *Step 1.* mySQL installation: The most straightforward way to install mySQL is through the Ubuntu package manager:

  - sudo aptitude update
  - sudo apt-get install mysql-server

- *Step 2.* Launch mySQL
  - sudo service mysql start

- *Step 3.* Get into the mySQL shell. At the command prompt, run:
  - /usr/bin/mysql -u root -p
  - Then you are prompted for a root password; if you don't want to set it, just hit Enter to submit no password.

- *Step 4.* Create a user. MySQL stores its user information in its own table 'user' of the database called 'mysql'. We can list all available users by querying: 'SELECT User, Host, Password FROM mysql.user;'

- *Step 5.* Create MUCKE database. The MUCKE framework will not work unless a database named mucke is already created and MySQL service is running.
  - Log into the mySQL shell, and then run: 'CREATE DATABASE mucke;'
  - Checking that this database was created with 'SHOW DATABASES;'

- *Step 6.* Add a database user. If you do not want the MUCKE framework accesses the database using root privilege, let's create another user that can connect to the database (here the username is mucker with the password 123456):
  - 'INSERT INTO mysql.user (User, Host, Password) VALUES('mucker', 'localhost', PASSWORD('123456'));'
  - 'FLUSH PRIVILEGES;'
  - Check that the user was created: 'SELECT User, Host, Password FROM mysql.user;'

- *Step 7.* Grant database user permissions. Make sure that the user 'mucker' can access the database 'mucke':
  - 'GRANT ALL PRIVILEGES ON mucke.* to mucker@localhost;'
  - 'FLUSH PRIVILEGES'
  - Check the operation: 'SHOW GRANTS FOR 'mucker'@'localhost';'

- Step 8. mySQL is listening at the port 3306? Make sure that mySQL is running prior deploying the MUCKE framework by verifying on the result returned by:
  - 'netstat -tanp | grep LISTEN'

- Otherwise, we need to set it:
  - iptables -I INPUT -p tcp –dport 3306 -m state –state NEW,ESTABLISHED -j ACCEPT
  - iptables -I OUTPUT -p tcp –sport 3306 -m state –state ESTABLISHED -j ACCEPT
- *Step 9.* Remember the username and password. We need them to fill in the file 'system.properties' of the project 'mucke-backend'.

APPENDIX B: INTEGRATION GUIDE

PURPOSE

This appendix describes all required steps when integrating with more specific code from MUCKE partners. This was exercised during the first integration meeting in Ankara, Turkey in August 2014, but has general value for the future whenever the MUCKE Framework is applied and extended. In the following, we will describe how to set up the system to handle additional coding requirements (mainly Scientific Python (SciPy) with additional graphics libraries) and how to write a plugin for the MUCKE Framework that integrates code. In particular, we will document how a small code example from Bilkent University was integrated with the MUCKE Framework.

PYTHON IMAGE PROCESSING DEVELOPMENT SETUP

Based on licensing and general technical difficulties with MatLab we decided to select Python as an alternative based on its free and open nature. It appears that Python and Matlab share a lot which makes it easy to translate logic between the two. Here, we describe the additional steps for setting up the environment that was used to allow the MUCKE Framework integrating with image processing code.

- Download and install Python 2.7.x (`https://www.python.org/download`).
- Download and install the SciPy Extension stack (`http://www.scipy.org/install.html`). For windows, you can use Anaconda (`http://continuum.io/downloads`). Make sure you enter the path in your environment so that the native call from the MUCKE Framework can reach the SciPy stack.
- Install OpenCV (`http://opencv.org/downloads.html`).

CREATING A NEW PLUGIN

We assume that you have installed and ran the MUCKE Framework and the two demos (mucke-ui and mucke-concept-ui) as described in the User Guide in section 3.10. This section explains what is needed to create a new plugin. A plugin is code and configuration that runs on top of the basic

MUCKE framework and, while using its functionality, it has separate configuration and may have separated/additional code that is specific to the plugin. Therefore, it allows a programmer to write new features that are not part of the core MUCKE framework while providing means to connect it with the core MUCKE framework.

The plugin folder (part of the code folder structure) has one folder for each existing plugin and each of them represents an example that can be inspected. in the following, we will describe the bilkentplugin as an example.

The bilkentdemo has a BilkentDemoPluginManager that extends the at.tuwien.mucke.pluginPluginManager class and overwrites the run method. This method contains all the code that is executed when the plugin is ran by the SystemManager. The bilkentdemo.properites file contains the default configuration of the plugin. The ConfigConstants class maps the namings of the parameters to Java so that they can be used inside the code without hardcoded names. The plugin executs two Python scripts that perform two kinds of feature classifications: BasicClassification.py and FullClassification.py. The latter creates a log file for debugging use (Full_Classification_LOG.txt). The plugin currently also contains the data and the result folder within, which would normally be stored elsewhere for reasons of efficiency, i.e. avoiding storing gigabytes of pictures on GitHub. The data and the result folder separates the basic and the full classification in sub-folders.

CREATING A NEW PLUGIN

Based on the previously described example, that was created during the Ankara Integration Meeting in August 2014, we now summarize the steps necessary to create a plugin.

- Create a new folder in the plugin package folder of the code based.
- Create a class MyDemoPluginManager.java that extends the at.tuwien.mucke.pluginPluginManager class. When extending this class, your plugin will become executable by the SystemManager as a Run.
- Put a plugin configuration file in the folder that contains the standard configuration parameter of your plugin.
- Put the external code files (e.g. MatLab, R or Python) in the folder.
- Put data and result directories in your file system and configure their locations in the properties file accordingly. Since this is accessed from the MUCKE framework, which acts as a backend, it can be located anywhere in your filesystem with general access. This is in fact advisable because data is potentially too big for an upload to GitHub
- Implement a ConfigConstants class that maps your parameter names to Java variables
- Implement the run method by calling the external files (if there are any) and load the configuration parameters via the ConfigConstants class when needed.

Naturally, these steps are kept very general simply because plugins differ hugely based on what they try to archive.

STEPS TO RUN A NEW PLUGIN

Based on the previously described example, as it was created during the Ankara Integration Meeting in August 2014, we now summarize the steps necessary to run a plugin inside the backend.

- Copy the plugin configuraiton file into the mucke-backend/conf folder to make it accessible as an instance for a system run (e.g. mydemoplugin.properites)
- Edit system.properties of the MUCKE framework and add the properites filename
- Execute the SystemManager to run the plugin, if you want to operate the plugin in batch-mode. Execute it through a frontend, if you want to operate the plugin in interactive mode. For this, please review the mucke-ui and the mucke-concept-ui frontend that each call the SystemManager.