**GLOTTAL**

**Design specification and user guide.**

1. **General design considerations.**

Since Java has an extensive support for internationalization and localization it was picked as a language of choice for the implementation of the GLOTTAL tool.

Since there is no need to support legacy code the JDK 1.6 (the latest version at the time of writing) was chosen as the target JDK.

Since there is not much of a user interaction in a process of creation of the glossaries it is reasonable to build the tool as a command line application with a set of execution options available for the user to set.

The following frameworks and libraries are used in the tool:

* Spring, for configuration of the tool’s objects;
* Jaxb, for de-serialization of the glossary in memory representation in to XML;
* Jakarta Commons, for various useful utility functionality, e.g. command line parameters parsing and handling;
* Log4J, for logging the debug and info output of the tool;

1. **Language specific collation for sorting.**

Since we have to deal with language specific sorting the special collator has to be used for a particular language. These collators can be directly instantiated if there is correspondent Java locale (based on ISO 639-1 two letter code) exists. Unfortunately not all of the languages that TL needs to support have ISO-639-1 codes. Luckily there is a way in Java to build rule based collators. These collators accept language specific comparison rules as the constructor parameter (see <http://java.sun.com/javase/6/docs/api/java/text/RuleBasedCollator.html>). This way even if the language doesn’t have correspondent ISO-639-1 code it can be sorted and compared according to the rules of the language (essentially according to the alphabet).

1. **Source code structure.**

The GLOTTAL tool’s source code is split in to the following packages:

|  |  |  |
| --- | --- | --- |
| **Package** | **Classes** | **General description** |
| com.transparent.glossary | CliClient  CliClientShell | Classes that act as main entry point of the tool and provide command line execution interface. |
| com.transparent.glossary.io | GlossaryInputDataReader  GlossaryOutputWriter  PatternFileFilter  XmlContentFileFilter | Classes responsible for reading the XML input content data and generating the glossary XML output. |
| com.transparent.glossary.model | GlossaryAnnotation  GlossaryEntryBase  GlossaryKey  GlossaryLetter  GlossaryPhraseEntry  GlossaryTranslation  GlossaryWordEntry  SortObject | The **GlossartXXX** classes represent in memory DOM of the glossary. The **SortObject** class represents a raw data element for sorting that contains all of the collected input content information. |
| com.transparent.glossary.processor | ContentProcessingRule  GlossaryProcessor  GlossaryProcessorEntry  ISortObjectsFilter  SortObjectsL1L2DuplicateFilter  SortObjectsL1L2FileMergeFilter | Classes that used to process the sort objects in various ways before they can be sorted. |
| com.transparent.glossary.sort | CollatorData  CollatorDataHelper  GlossaryComparator  SortDirection  SortType | Classes that are responsible for sorting of the sort objects according to the language rules. |

The majority of the classes have JavaDoc comments. The JavaDoc can be generated by executing **ant doc** build target. All of the classes also have source code level comments.

1. **Glossary structure.**

The glossary structure conforms to the XML schema described in the **glossary.xsd** located in the **/src** folder of the tool. The sample XML file generated according to that XSD would look like the following:

**<glossary language="string">**

**<letters>**

**<!--Zero or more repetitions:-->**

**<letter name="string">**

**<words>**

**<!--Zero or more repetitions:-->**

**<word L2="string" L1="string">**

**<key language="string">string</key>**

**<translation language="string">string</translation>**

**<annotations>**

**<!--Zero or more repetitions:-->**

**<annotation name="string">string</annotation>**

**</annotations>**

**</word>**

**</words>**

**<phrases>**

**<!--Zero or more repetitions:-->**

**<phrase L2="string" L1="string">**

**<key language="string">string</key>**

**<translation language="string">string</translation>**

**<annotations>**

**<!--Zero or more repetitions:-->**

**<annotation name="string">string</annotation>**

**</annotations>**

**</phrase>**

**</phrases>**

**</letter>**

**</letters>**

**</glossary>**

The tool will always generate two glossary files per processed language, one that is sorted by the L1 language and another by L2. It means that one will have all its letter entries sorted by the rules of L1 and have L1 language as a key and another will have that as L2 respectively.

1. **How glossary gets created.**

The tool starts from the given root folder that suppose to hold complete content data for a given language. Each file (that represents complete sort object information) in each folder, e.g., B4X is processed and sort objects are created. Files that contain partial sort object information, e.g., individual TT files, are processed in pairs and then merged. All sort fields are normalized to be in lower case and also stripped off of all of the stop words if applicable. Duplicates of the sort objects are removed (for how duplicates are found see below). The phrase sort objects are marked. In the context of GLOTTAL phrase is a sequence of words ended with punctuation. All sort objects are sorted according to their L1 and then separately L2 language. Glossaries for L1 and L2 then written to the given output folder.

1. **Configuration.**

It is possible to configure almost every aspect of the functionality via XML configuration file without changing the Java code. There is a Spring configuration file **glottal-config.xml** that describes all internal wiring of the tool’s objects (located under **/src**). This file can be copied and edited in any text editor that support UTF-8 encoding. There is a command line option that allows one to use custom **glottal-config.xml** instead of the one packaged with the tool (see “**How to run GLOTTAL**” section).

* 1. **Collators’ configuration.**

The first section of the **glottal-config.xml** contains the collators’ information (**collatorData** bean). The map entries for that bean can be adjusted or added. The map entry configuration comes in a pretty simple format, see below:

**<entry key="">**

**<bean class="com.transparent.glossary.sort.CollatorData">**

**<property name="tplLanguageCode" value=""/>**

**<property name="javaLocaleCode" value="" />**

**<property name="collationRules" value=""/>**

**<property name="stopWords" value=""/>**

**<property name="alphabet" value=""/>**

**</bean>**

**</entry>**

The **key** and **tplLanguageCode** elements must contain a TL language code that this collator is supposed to be used with.

The **javaLanguageCode** must contain either language code that conforms to the rules defined by the Java Locale or special value **\_NA\_**, which indicates that **collationRules** property must be used instead.

The **collationRules** property, if specified, must contain the string that describes collation rules. This string must conform to the format specified by the Java **RuleBasedCollator** class (<http://java.sun.com/javase/6/docs/api/java/text/RuleBasedCollator.html>).

The **stopWords** parameter if specified should contain a list of stop words for a given language, this may include punctuation signs. Stop words are listed as a space separated string of regular-expression-oriented tokens. This means that in order to, for example, specify stop words: **“a”, “an”** and **“?”** it has to be written as: **\ba\b \ban\b \?** Note that all words are enclosed in to the **\b** regular expression word boundary marker and special characters like:**? !,** etc. are escaped with **\**

The **alphabet** parameter if specified should contain a full alphabet for a given language. It is a string of space separated letters of a given alphabet.

* 1. **Advanced configuration.**

The following section describes how to configure some advanced aspects of the GLOTTAL tool. Most useful here is probably configuration of XPath expressions for the input content data retrieval purposes.

* + 1. **The xxxReader beans’ configuration.**

There are a few beans with names ending in **xxxReader**, e.g. **b4xReader**. At the moment there are two types of reader beans used, the B4X reader and TT reader. They are used by processors (see below) to read input content data. There are two ways to configure readers; one is to change the existing ones to adjust their behavior and another to add a new one to read from a different input source.

Here is the bean configuration XML and description of the customizable properties:

**<bean id="" class="com.transparent.glossary.io.GlossaryInputDataReader">**

**<property name="collatorHelper" ref="collatorData"/>**

**<property name="languageCodeL1XPath" value=""/>**

**<property name="languageCodeL2XPath" value=""/>**

**<property name="dataObjectCollectionXPath" value=""/>**

**<property name="sortFieldsL1XPaths">**

**<list value-type="java.lang.String">**

**<value></value>**

**...**

**</list>**

**</property>**

**<property name="sortFieldsL2XPaths">**

**<list value-type="java.lang.String">**

**<value></value>**

**...**

**</list>**

**</property>**

**<property name="fieldsToInclude">**

**<map key-type="java.lang.String" value-type="java.lang.String">**

**<entry key="L1" value=" "/>**

**<entry key="L2" value=" "/>**

**...**

**</map>**

**</property>**

**</bean>**

If creating a new bean the **id** property must be specified and unique in the context of the configuration file.

The **languageCodeL1XPath** must contain a valid XPath expression that when evaluated against target input source will yield a text representing TL language code for L1 language of the sort object, e.g., for B4X file it will be: **/list/head/side1\_language\_key/text()**

The **languageCodeL2XPath** must contain a valid XPath expression that when evaluated against target input source will yield a text representing TL language code for L2 language of the sort object, e.g., for B4X file it will be: **/list/head/side2\_language\_key/text()**

The **dataObjectCollectionXPath** must contain a valid XPath expression that when evaluated against target input source will yield a collection of nodes containing data elements that can be used to build sort objects, e.g., for B4X file it will be: **/list/cards/\***

The **sortFieldsL1XPaths** must contain a list of valid XPath expressions that when individually evaluated against the single element from the collection retrieved by the **dataObjectCollectionXPath** will yield an L1 text which will be used as the sort key for L1, e.g., for B4X file it will be: **side1\_phrase/text().** The order in which XPaths are listed defines the order in which the text retrieved by these XPaths will be sorted (this essentially represents a complex sort key).

The **sortFieldsL2XPaths** must contain a list of valid XPath expressions that when individually evaluated against the single element from the collection retrieved by the **dataObjectCollectionXPath** will yield an L2 text which will be used as the sort key for L2, e.g., for B4X file it will be: **side2\_phrase/text()**. The order in which XPaths are listed defines the order in which the text retrieved by these XPaths will be sorted (this essentially represents a complex sort key).

The **fieldsToInclude** must contain a list of valid XPath expressions that when individually evaluated against the single element from the collection retrieved by the **dataObjectCollectionXPath** will yield a text which will be included in the glossary. Note that there are always have to be fields with names L1 and L2. These predefined fields define the key/translation pairs in the glossary output. In addition to L1 and L2 there could be any number of other fields. Those additional fields will end up in the annotation section on the glossary. The example set of XPaths for B4X input source would be:

**<entry key="L1" value="side1\_phrase/text()"/>**

**<entry key="L2" value="side2\_phrase/text()"/>**

**<entry key="hint1" value="side1\_hint/text()"/>**

**<entry key="comment" value="comment/text()"/>**

It is important to note that reader can be used to read just a part of the sort object as it is the case with TT input readers. Since there are always two TT files that represent complete sort object we must configure separate readers (see original **glottal-config.xml** for details) to read just a “half” of a sort objects per TT file. These incomplete sort objects then merged later by the processors’ filters (see below).

* + 1. **Processing filters.**

There are currently only two types of processing filters available.

One that has no customization options is a duplicate filter. This filter just clears duplicate sort objects from the originally collected object collection. The rule goes like the following: two sort objects considered duplicated if and only if their correspondent L1 and L2 pairs are equal.

Second filter is merge filter. Although it is currently applied to the TT files it can be used with any files that have similar logical structure as TT files, meaning that every input content file contains only half of the sort object language content (as it is in case of TT files). This filter has file patterns for the name of the files and also a regular expression for the common part of the file name. See JavaDoc and source code for more information on this filter’s functionality.

* + 1. **Processor entry configuration.**

A processor entry represents a processing unit that ties together input source (via its reader) the input source file content or pattern filter and processing filter. Only input sources that satisfy the input source filter will be processed. The configuration of the processor entry is somewhat complex please refer to the original **glottal-config.xml** for an example. Here are just a few key points.

Each processor entry can be configured to process one or all entries. This is controlled by the **processingRule** parameter which can be set to one of the two values: **SINGLE** or **ALL**. This means that when glossary processor works with its entries it examines this property and either collects sort objects for all of the files located at the same level (**ALL**) or just for every individual file (**SINGLE**) at that level before running them through sort object filters. This way, for example, the TT files’ content gets processed (merged) correctly.

Each processor can configure multiple input readers. This is done via **inputReadersMap** parameters. This map property is keyed by the special implementation of the **FileFilter** interface (see below) and has a specific input reader bean referenced as a correspondent value of the map entry.

Currently there are two types of file filters that are used in configuration of the processor entry.

First one is **XmlContentFileFilter** which filters (accepts) files based on their content. It uses XPath to retrieve an element of each file it examines and special expression syntax to compare value retrieved via XPath with the expected result. The expressions that are currently supported are following: **length(*<op><value>*)** and **value(<*value>*)**. Here is an example of the configuration of the **XmlContentFileFilter**:

**<bean class="com.transparent.glossary.io.XmlContentFileFilter">**

**<property name="XPath" value="/list/cards/\*"/>**

**<property name="expectedResult" value="length(!=0)"/>**

**</bean>**

This means that the filter (in order to accept the file) expects to find non-zero length collection of the elements at a given XPath in that file.

Second file filter is a simple file name pattern based filter. It has single configuration parameter **patterns -** the list of regular-expression-conforming file name patterns that this filter accepts.

The last configuration property of the processor entry is **sortObjectsFilters** that defines what filters and in what order needs to be applied to the sort objects read by this processor entry’s input reader (the two available filters described above in “**Processing filters**” section).

* + 1. **Glossary writers’ configuration.**

There are two glossary writer beans. The only field that is configurable for a writer is the **annotationFields**. This is the list of the additional (in addition to L1 and L2) fields that will be included in to the glossary as annotations. The names must correspond to ones defined in the input readers’ **fieldsToIncludeMap** fields.

* + 1. **Glossary processor configuration.**

The glossary processor bean (**glossaryProcessor**) is the core of the GLOTTAL tool functionality. It ties together processor entries and output writes in order to create a glossary.

The **processorEntries** property lists all of the processor entry beans’ references used to process (read and filter) the sort objects.

The **inputDataFinalFilters** property is there to allow for final filtering of the sort objects collected at different levels of the folders hierarchy.

The **glossaryOutputWriters** property lists writers for the L1 and L2 types of glossary.

The **phraseMarkers** property lists a set of punctuation signs that demarcate the phrase.

1. **Distribution.**

The distribution package of GLOTTAL tool can be built by executing **ant kit** build command. Upon successful build it is placed in the **/kit** folder and contains the following:

**/lib** - all of the libraries that tool needs;

**glottal.jar** - executable code of the tool itself;

**glottal.bat** - simple windows batch file to run the tool;

The batch file assumes that you run it from the same folder where **glottal.jar** is located, also it assumes that **JAVA\_HOME** environment variable is set and also %**JAVA\_HOME%/bin** is included in **PATH** environment variable.

1. **How to run GLOTTAL.**

There are two required and four optional parameters for executing GLOTTAL on a command line.

|  |  |  |
| --- | --- | --- |
| **Name** | **Required** | **Description** |
| inputFolderRoot | Yes | Full path to the folder where single language content is located. |
| outputFolder | Yes | Full path to the folder that will hold the tool output. |
| configFile | No | The configuration file name including full path. This configuration file will override the original glottal-config.xml. |
| prefix | No | Prefix of the name of the glossary file. By default the “**glossary**” is used. |
| suffixL1 | No | Suffix of the L1 glossary file. By default the “**L1**” will be used. |
| suffixL2 | No | Suffix of the L2 glossary file. By default the “**L2**” will be used. |
| swType | No | The type of stop word processing. The default is ALL, which means that for words and phrases stop words will be removed from the sort keys before doing sort. The other available are: WORDS (processes stop word only for vocabulary words); PHRASES (processes stop word only for phrases); NONE (do not process stop words). |

There is one more optional parameter called “**version**” if it is specified then GLOTTAL tool prints a help on parameters and exits.