Morphilo Documentation

Hagen Peukert

Oct 24, 2018

Contents:

CHAPTER 1

Software Design

1.1 MVC Model

A standard architecture for software has become a form of an observer pattern called *Model-View-Controller (MVC)*-Model[3]. This is escpecially true for web-based applications that use some form of a client-server architecture since these systems naturally divide the browser view from the rest of the program logic and, if dynamically set up, also from the data model usually running in an extra server as well. As already implied, the MVC-pattern modularizes the program into three components: model, view, and controller coupled *low* by interfaces. The view is concerned with everything the actual user sees on the screen or uses to interact with the machine. The controller is to recognize and process the events initiated by the user and to update the view. Processing involves to communicate with the model. This may involve to save or provide data from the data base.

From all that follows, MVC-models are especially supportive for reusing existing software and promotes parallel development of its three components. So the data model of an existing program can easily be changed without touching the essentials of the program logic. The same is true for the code that handles the view. Most of the time view and data model are the two components that need to be changed so that the software appearance and presentation is adjusted to the new user group as well as the different data is adjusted to the needs of the different requirements of the new application. Nevertheless, if bugs or general changes in the controller component have to be done, it usually does not affect substantially the view and data model.

Another positive consequence of MVC-models is that several views (or even models) could be used simultaneously. It means that the same data could be presented differently on the user interface.

1.2 Morphilo Architecture

The architecture of a possible *take-and-share* approach for language resources is visualized in figure 1. Because the very gist of the approach becomes clearer if describing a concrete example, the case of annotating lexical derivatives of Middle English with the help of the Morphilo Tool[1] using a MyCoRe repository is given as an illustration. However,

[3] Butz, Andreas; Antonio Krüger (2017): Mensch-Maschine-Interaktion, De Gruyter, pp. 93.

[1] Peukert, H. (2012): From Semi-Automatic to Automatic Affix Extraction in Middle English Corpora: Building a Sustainable Database for Analyzing Derivational Morphology over Time, Empirical Methods in Natural Language Processing, Wien, Scientific series of the ÖGAI, 413-23.

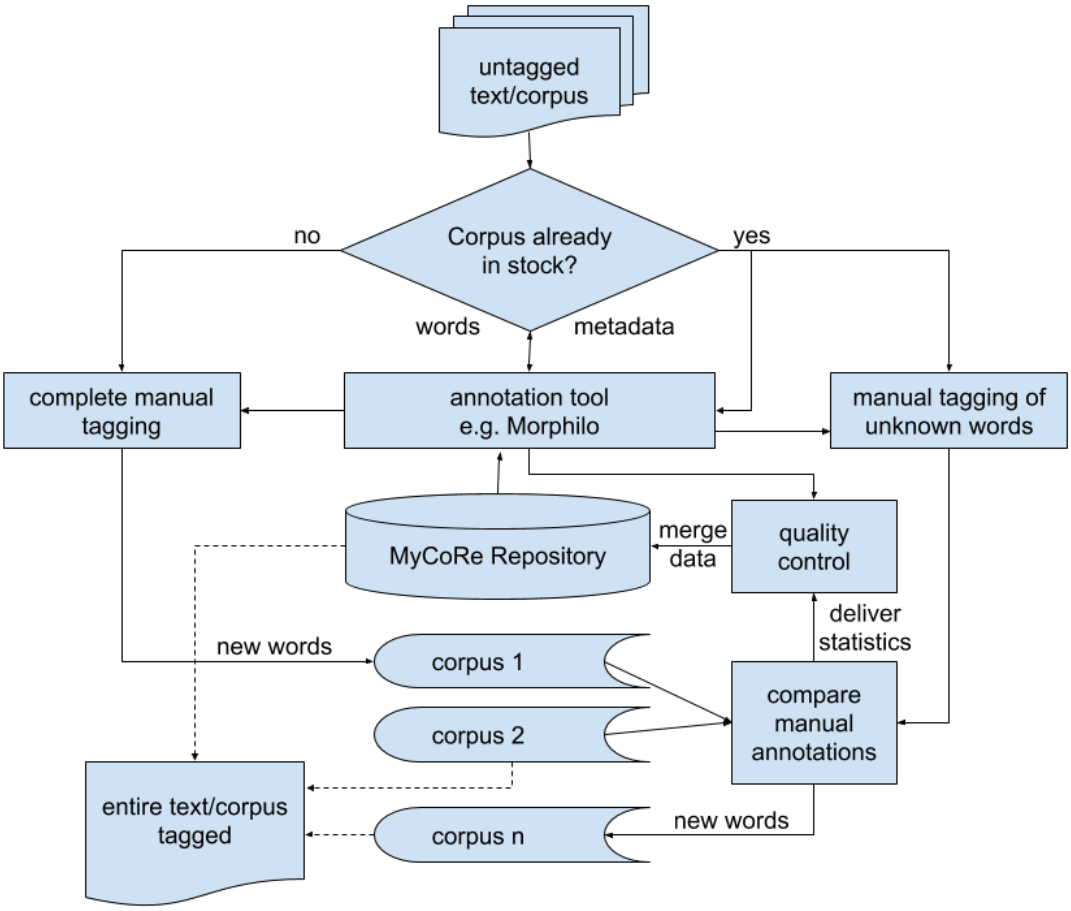

Fig. 1: Figure 1: Basic Architecture of a Take-&-Share-Approach

any other tool that helps with manual annotations and manages metadata of a corpus could be substituted here instead.[2]

After inputting an untagged corpus or plain text, it is determined whether the input material was annotated previously by a different user. This information is usually provided by the metadata administered by the annotation tool; in the case at hand, the *Morphilo* component. An alternative is a simple table look-up for all occurring words in the datasets Corpus 1 through Corpus n. If contained completely, the *yes*-branch is followed up further – otherwise *no* succeeds. The difference between the two branches is subtle, yet crucial. On both branches, the annotation tool (here *Morphilo*) is called, which, first, sorts out all words that are not contained in the master database (here *MyCoRe* repository) and, second, makes reasonable suggestions on an optimal annotation of the items. The suggestions made to the user are based on simple string mapping of a saved list of prefixes and suffixes whereas the remainder of the mapping is defined as the word root. The annotations are linked to the respective items (e.g. words) in the text, but they are also persistently saved in an extra dataset, i.e. in figure 1 in one of the delineated Corpus 1 through n, together with all available metadata.

The difference between the two branches in figure 1 is that in the *yes*-branch a comparison between the newly created dataset and all of the previous datasets of this text is carried out while this is not possible if a text was not annotated before. Within this unit, all deviations and congruencies of the annotated items are marked and counted. The underlying assumption is that with a growing number of comparable texts the correct annotations approach a theoretic true value of a correct annotation while errors level out provided that the sample size is large enough. How the distribution of errors and correct annotations exactly looks like and if a normal distribution can be assumed is still object of the ongoing research, but independent of the concrete results, the component (called *compare manual annotations* in figure 1) allows for specifying the exact form of the sample population. In fact, it is necessary at that point to define the form of the distribution, sample size, and the rejection region. To be put it simple here, a uniform distribution in form of a threshold value of e.g. 20 could be defined that specifies that a word has to be annotated equally by 20 different users before it enters the master database.

Continuing the information flow in figure 1 further, the threshold values or, if so defined, the results of the statistical calculation of other distributions respectively are delivered to the quality-control-component. Based on the statistics, the respective items together with the metadata, frequencies, and, of course, annotations are written to the master database. All information in the master database is directly used for automated annotations. Thus it is directly matched to the input texts or corpora respectively through the *Morphilo*-tool. The annotation tool decides on the entries looked up in the master which items are to be manually annotated.

The processes just described are all hidden from the user who has no possibility to impact the set quality standards but by errors in the annotation process. The user will only see the number of items of the input text he or she will process manually. The annotator will also see an estimation of the workload beforehand. On this number, a decision can be made if to start the annotation at all. It will be possible to interrupt the annotation work and save progress on the server. And the user will have access to the annotations made in the respective dataset, correct them or save them and resume later. It is important to note that the user will receive the tagged document only after all items are fully annotated. No partially tagged text can be output.

1.3 Repository Framework

To specify the repository framework, the morphilo application logic will have to be implemented, a data model specified, and the input, search and output mask programmed.

There are three directories which are important for adjusting the MyCoRe framework to the needs of one's own application.

These three directories correspond essentially to the three components in the MVC model as explicated above. Roughly, they are also envisualized in figure 2 in the upper right hand corner. More precisely, the view (*Layout* in figure 2) and the model layer (*Datenmodell* in figure 2) can be done completely via the *interface*, which is a directory with a predefined structure and some standard files. For the configuration of the logic an extra directory is

[2] The source code of a possible implementation is available on https://github.com/amadeusgwin/morphilo. The software runs in test mode on https://www.morphilo.uni-hamburg.de/content/index.xml.

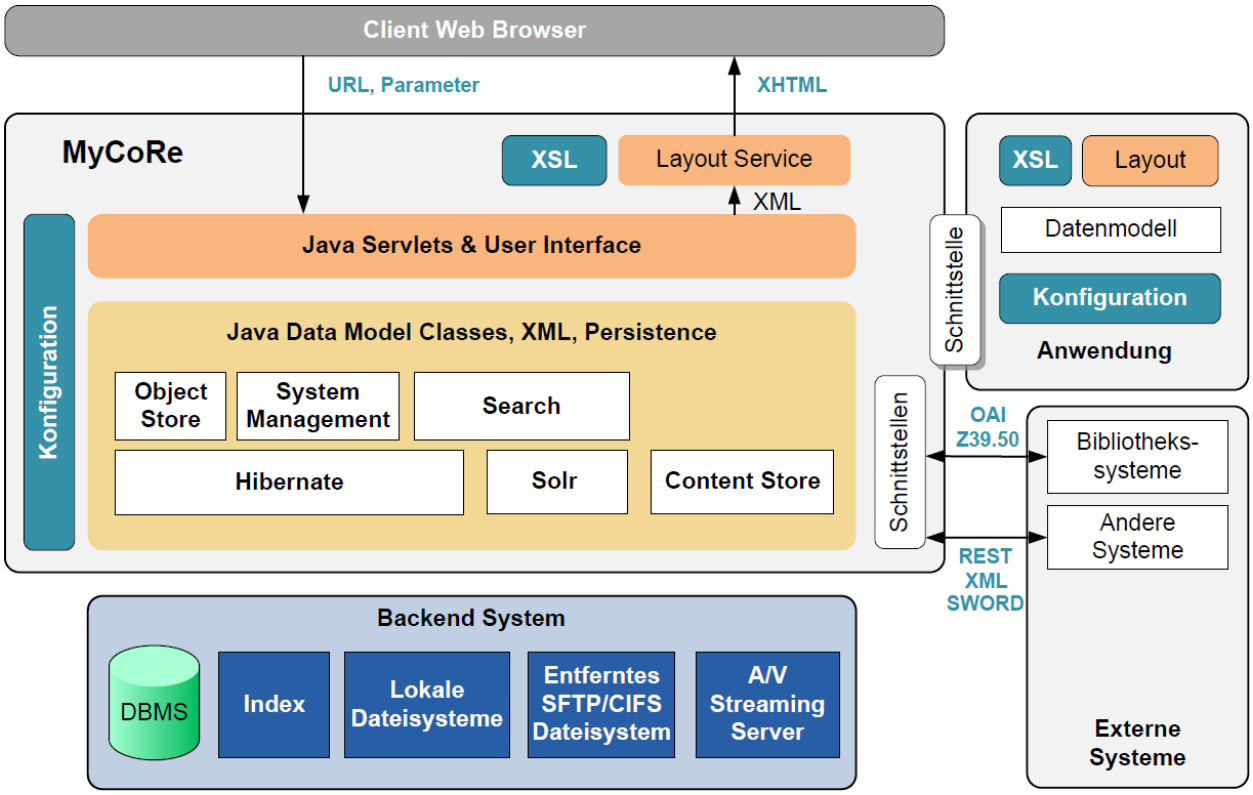

Fig. 2: Figure 2: MyCoRe-Architecture and Components

offered (*/src/main/java/custom/mycore/addons/*). Here all, java classes extending the controller layer should be added. Practically, all three MVC layers are placed in the *src/main/*-directory of the application. In one of the subdirectories, *datamodel/def,* the datamodel specifications are defined as xml files. It parallels the model layer in the MVC pattern. How the data model was defined will be explained in the section Data Model.

Notes

CHAPTER 2

Data Model

2.1 Conceptualization

From both the user and task requirements one can derive that four basic functions of data processing need to be carried out. Data have to be read, persistently saved, searched, and deleted. Furthermore, some kind of user management and multi-user processing is necessary. In addition, the framework should support web technologies, be well documented, and easy to extent. Ideally, the MVC pattern is realized.

The guidelines of the TEI standard on the word level are defined in line with the defined word structure. In listing *TEI-example for comfortable* an example is given for a possible markup at the word level for comfortable

Listing 1: TEI-example for *comfortable*

```
<w type="adjective">
 <m type="base">
  <m type="prefix" baseForm="con">com</m>
  <m type="root">fort</m>
 </m>
 <m type="suffix">able</m>
</w>
```

This data model reflects just one theoretical conception of a word structure model. Crucially, the model emanates from the assumption that the suffix node is on par with the word base. On the one hand, this implies that the word stem directly dominates the suffix, but not the prefix. The prefix, on the other hand, is enclosed in the base, which basically means a stronger lexical, and less abstract, attachment to the root of a word. Modeling prefixes and suffixes on different hierarchical levels has important consequences for the branching direction at subword level (here right-branching). Left the theoretical interest aside, the choice of the *TEI*-standard is reasonable with view to a sustainable architecture that allows for exchanging data with little to no additional adjustments.

The negative account is that the model is not eligible for all languages. It reflects a theoretical construction based on Indo-European languages. If attention is paid to which language this software is used, it will not be problematic. This is the case for most languages of the Indo-European stem and corresponds to the overwhelming majority of all research carried out (unfortunately).

2.2 Implementation

It is advantageous to use established standardsn and it makes sense to keep the meta data of each corpus separate from the data model used for the words to be analyzed.

For the present case, the *TEI*-standard was identified as an appropriate markup for words. In terms of the implementation this means that the *TEI*-guidelines have to be implemented as an object type compatible with the chosen repository framework. However, the TEI standard is not complete regarding the diachronic dimension, i.e. information on the development of the word. To be compatible with the elements of the TEI standard on the one hand and to best meet the requirements of the application on the other hand, some attributes are added. This solution allows for processing the xml files according to the *TEI*-standard by ignoring the additional attributes and at the same time, if needed, additional markup can be extracted. The additional attributes comprise a link to the corpus meta data, but also *position* and *occurrence* of the affixes. Information on the position and some quantification thereof are potentially relevant for a wealth of research questions, such as predictions on the productivity of derivatives and their interaction with the phonological or syntactic modules. So they were included with respect to future use.

For reasons of efficiency in subsequent processing, the historic dates *begin* and *end* were included in both the word data model and the corpus data model. The result of the word data model is given in listing *Word Data Model*. Whereas attributes of the objecttype are specific to the repository framework, the TEI structure can be recognized in the hierarchy of the meta data element starting with the name *w* (line 17).

Listing 2: Word Data Model

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <objecttype
3   name="morphilo"
4   isChild="true"
5   isParent="true"
6   hasDerivates="true"
7   xmlns:xs="http://www.w3.org/2001/XMLSchema"
8   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
9   xsi:noNamespaceSchemaLocation="datamodel.xsd">
10  <metadata>
11   <element name="morphiloContainer" type="xml" style="dontknow"
12  notinherit="true" heritable="false">
13    <xs:sequence>
14     <xs:element name="morphilo">
15      <xs:complexType>
16       <xs:sequence>
17        <xs:element name="w" minOccurs="0" maxOccurs="unbounded">
18         <xs:complexType mixed="true">
19          <xs:sequence>
20           <!-- stem -->
21           <xs:element name="m1" minOccurs="0" maxOccurs="unbounded">
22            <xs:complexType mixed="true">
23             <xs:sequence>
24              <!-- base -->
25              <xs:element name="m2" minOccurs="0" maxOccurs="unbounded">
26               <xs:complexType mixed="true">
27                <xs:sequence>
28                 <!-- root -->
29                 <xs:element name="m3" minOccurs="0" maxOccurs="unbounded">
30                  <xs:complexType mixed="true">
31                   <xs:attribute name="type" type="xs:string"/>
32                  </xs:complexType>
33                 </xs:element>
34                 <!-- prefix -->
```

(continues on next page)

(continued from previous page)

```
                <xs:element name="m4" minOccurs="0" maxOccurs="unbounded">
                 <xs:complexType mixed="true">
                  <xs:attribute name="type" type="xs:string"/>
                  <xs:attribute name="PrefixbaseForm" type="xs:string"/>
                  <xs:attribute name="position" type="xs:string"/>
                 </xs:complexType>
                </xs:element>
               </xs:sequence>
               <xs:attribute name="type" type="xs:string"/>
              </xs:complexType>
             </xs:element>
             <!-- suffix -->
             <xs:element name="m5" minOccurs="0" maxOccurs="unbounded">
              <xs:complexType mixed="true">
               <xs:attribute name="type" type="xs:string"/>
               <xs:attribute name="SuffixbaseForm" type="xs:string"/>
               <xs:attribute name="position" type="xs:string"/>
               <xs:attribute name="inflection" type="xs:string"/>
              </xs:complexType>
             </xs:element>
            </xs:sequence>
            <!-- stem-Attribute -->
            <xs:attribute name="type" type="xs:string"/>
            <xs:attribute name="pos" type="xs:string"/>
            <xs:attribute name="occurrence" type="xs:string"/>
           </xs:complexType>
          </xs:element>
         </xs:sequence>
         <!-- w -Attribute auf Wortebene -->
         <xs:attribute name="lemma" type="xs:string"/>
         <xs:attribute name="complexType" type="xs:string"/>
         <xs:attribute name="wordtype" type="xs:string"/>
         <xs:attribute name="occurrence" type="xs:string"/>
         <xs:attribute name="corpus" type="xs:string"/>
         <xs:attribute name="begin" type="xs:string"/>
         <xs:attribute name="end" type="xs:string"/>
        </xs:complexType>
       </xs:element>
      </xs:sequence>
     </xs:complexType>
    </xs:element>
   </xs:sequence>
  </element>
  <element name="wordtype" type="classification" minOccurs="0" maxOccurs="1">
   <classification id="wordtype"/>
  </element>
  <element name="complexType" type="classification" minOccurs="0" maxOccurs="1">
   <classification id="complexType"/>
  </element>
  <element name="corpus" type="classification" minOccurs="0" maxOccurs="1">
   <classification id="corpus"/>
  </element>
  <element name="pos" type="classification" minOccurs="0" maxOccurs="1">
   <classification id="pos"/>
  </element>
  <element name="PrefixbaseForm" type="classification" minOccurs="0"
  maxOccurs="1">
```

(continues on next page)

(continued from previous page)

```
    <classification id="PrefixbaseForm"/>
   </element>
   <element name="SuffixbaseForm" type="classification" minOccurs="0"
   maxOccurs="1">
    <classification id="SuffixbaseForm"/>
   </element>
   <element name="inflection" type="classification" minOccurs="0" maxOccurs="1">
    <classification id="inflection"/>
   </element>
   <element name="corpuslink" type="link" minOccurs="0" maxOccurs="unbounded" >
    <target type="corpmeta"/>
   </element>
  </metadata>
</objecttype>
```

Additionally, it is worth mentioning that some attributes are modeled as a *classification*. All these have to be listed as separate elements in the data model. This has been done for all attributes that are more or less subject to little or no change. In fact, all known suffix and prefix morphemes should be known for the language investigated and are therefore defined as a classification. The same is true for the parts of speech named *pos* in the morphilo data model above. Here the PENN-Treebank tagset was used. Last, the different morphemic layers in the standard model named *m* are changed to *m1* through *m5*. This is the only change in the standard that could be problematic if the data is to be processed elsewhere and the change is not documented more explicitly. Yet, this change was necessary for the MyCoRe repository throws errors caused by ambiguity issues on the different *m*-layers.

The second data model describes only very few properties of the text corpora from which the words are extracted. Listing *Corpus Data Model* depicts only the meta data element. For the sake of simplicity of the prototype, this data model is kept as simple as possible. The obligatory field is the name of the corpus. Specific dates of the corpus are classified as optional because in some cases a text cannot be dated reliably.

Listing 3: Corpus Data Model

```
<metadata>
  <!-- Pflichtfelder -->
  <element name="korpusname" type="text" minOccurs="1" maxOccurs="1"/>
  <!-- Optionale Felder -->
  <element name="sprache" type="text" minOccurs="0" maxOccurs="1"/>
  <element name="size" type="number" minOccurs="0" maxOccurs="1"/>
  <element name="datefrom" type="text" minOccurs="0" maxOccurs="1"/>
  <element name="dateuntil" type="text" minOccurs="0" maxOccurs="1"/>
  <!-- number of words -->
  <element name="NoW" type="text" minOccurs="0" maxOccurs="1"/>
  <element name="corpuslink" type="link" minOccurs="0" maxOccurs="unbounded">
    <target type="morphilo"/>
  </element>
</metadata>
```

As a final remark, one might have noticed that all attributes are modelled as strings although other data types are available and fields encoding the dates or the number of words suggest otherwise. The MyCoRe framework even provides a data type *historydate*. There is not a very satisfying answer to its disuse. All that can be said is that the use of data types different than the string leads later on to problems in the convergence between the search engine and the repository framework. These issues seem to be well known and can be followed on github.

CHAPTER 3

View

3.1 Conceptualization

The MyCoRe-directory (*src/main/resources*) contains all code needed for rendering the data to be displayed on the screen. So this corresponds to the view in an MVC approach. It is done by xsl-files that (unfortunately) contain some logic that really belongs to the controller. Thus, the division is not as clear as implied in theory. I will point at this issue more specifically in the relevant subsection below. Among the resources are all images, styles, and javascripts.

3.2 Implementation

The view component handles the visual representation in the form of an interface that allows interaction between the user and the task to be carried out by the machine. As a webservice in the present case, all interaction happens via a browser, i.e. webpages are visualized and responses are recognized by registering mouse or keyboard events. More specifically, a webpage is rendered by transforming xml documents to html pages. The MyCoRe repository framework uses an open source XSLT processor from Apache, Xalan. This engine transforms document nodes described by the XPath syntax into hypertext making use of a special form of template matching. All templates are collected in so called xml-encoded stylesheets. Since there are two data models with two different structures, it is good practice to define two stylesheet files one for each data model.

As a demonstration, in the listing below a short extract is given for rendering the word data.

Listing 1: word data rendering in morphilo.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xalan="http://xml.apache.org/xalan"
  xmlns:i18n="xalan://org.mycore.services.i18n.MCRTranslation"
  xmlns:acl="xalan://org.mycore.access.MCRAccessManager"
  xmlns:mcr="http://www.mycore.org/" xmlns:xlink="http://www.w3.org/1999/xlink"
  xmlns:mods="http://www.loc.gov/mods/v3"
```

(continues on next page)

(continued from previous page)

```
  xmlns:encoder="xalan://java.net.URLEncoder"
  xmlns:mcrxsl="xalan://org.mycore.common.xml.MCRXMLFunctions"
  xmlns:mcrurn="xalan://org.mycore.urn.MCRXMLFunctions" exclude-result-prefixes=
↪"xalan xlink mcr i18n acl mods mcrxsl mcrurn encoder" version="1.0">
  <xsl:param name="MCR.Users.Superuser.UserName"/>
   <xsl:template match="/mycoreobject[contains(@ID,'_morphilo_')]">
    <head>
     <link href="{$WebApplicationBaseURL}css/file.css" rel="stylesheet"/>
    </head>
    <div class="row">
     <xsl:call-template name="objectAction">
      <xsl:with-param name="id" select="@ID"/>
      <xsl:with-param name="deriv" select="structure/derobjects/derobject/@xlink:href
↪"/>
     </xsl:call-template>
     <xsl:variable name="objID" select="@ID"/>
     <!-- Hier Ueberschrift setzen -->
     <h1 style="text-indent: 4em;">
      <xsl:if test="metadata/def.morphiloContainer/morphiloContainer/morphilo/w">
       <xsl:value-of select="metadata/def.morphiloContainer/morphiloContainer/
↪morphilo/w/text()[string-length(normalize-space(.))>0]"/>
      </xsl:if>
     </h1>
     <dl class="dl-horizontal">
     <!-- (1) Display word -->
      <xsl:if test="metadata/def.morphiloContainer/morphiloContainer/morphilo/w">
       <dt>
        <xsl:value-of select="i18n:translate('response.page.label.word')"/>
       </dt>
       <dd>
        <xsl:value-of select="metadata/def.morphiloContainer/morphiloContainer/
↪morphilo/w/text()[string-length(normalize-space(.))>0]"/>
       </dd>
      </xsl:if>
     <!-- (2) Display lemma -->
      ...
   </xsl:template>
    ...
   <xsl:template name="objectAction">
    ...
   </xsl:template>
...
</xsl:stylesheet>
```

This template matches with the root node of each *MyCoRe object* ensuring that a valid MyCoRe model is used and checking that the document to be processed contains a unique identifier, here a *MyCoRe-ID*, and the name of the correct data model, here *morphilo*. Then, another template, *objectAction*, is called together with two parameters, the ids of the document object and attached files. In the remainder all relevant information from the document is accessed by XPath, such as the word and the lemma, and enriched with hypertext annotations it is rendered as a hypertext document. The template *objectAction* is key to understand the coupling process in the software framework. It is therefore separately listed in *template ObjectAction*.

Listing 2: template ObjectAction

```
1 <xsl:template name="objectAction">
2 <xsl:param name="id" select="./@ID"/>
```

(continues on next page)

(continued from previous page)

```
<xsl:param name="accessedit" select="acl:checkPermission($id,'writedb')"/>
<xsl:param name="accessdelete" select="acl:checkPermission($id,'deletedb')"/>
<xsl:variable name="derivCorp" select="./@label"/>
<xsl:variable name="corpID" select="metadata/def.corpuslink[@class='MCRMetaLinkID']/corpuslink/@xlink:href"/>
<xsl:if test="$accessedit or $accessdelete">
<div class="dropdown pull-right">
  <xsl:if test="string-length($corpID) &gt; 0 or $CurrentUser='administrator'">
    <button class="btn btn-default dropdown-toggle" style="margin:10px" type="button" id="dropdownMenu1" data-toggle="dropdown" aria-expanded="true">
      <span class="glyphicon glyphicon-cog" aria-hidden="true"></span> Annotieren
      <span class="caret"></span>
    </button>
  </xsl:if>
  <xsl:if test="string-length($corpID) &gt; 0">
    <xsl:variable name="ifsDirectory" select="document(concat('ifs:/',$derivCorp))"/>
      <ul class="dropdown-menu" role="menu" aria-labelledby="dropdownMenu1">
        <li role="presentation">
          <a href="{$ServletsBaseURL}object/tag{$HttpSession}?id={$derivCorp}&amp;objID={$corpID}" role="menuitem" tabindex="-1">
            <xsl:value-of select="i18n:translate('object.nextObject')"/>
          </a>
        </li>
        <li role="presentation">
          <a href="{$WebApplicationBaseURL}receive/{$corpID}" role="menuitem" tabindex="-1">
            <xsl:value-of select="i18n:translate('object.backToProject')"/>
          </a>
        </li>
      </ul>
    </xsl:if>
    <xsl:if test="$CurrentUser='administrator'">
     <ul class="dropdown-menu" role="menu" aria-labelledby="dropdownMenu1">
      <li role="presentation">
       <a role="menuitem" tabindex="-1" href="{$WebApplicationBaseURL}content/publish/morphilo.xed?id={$id}">
        <xsl:value-of select="i18n:translate('object.editWord')"/>
       </a>
      </li>
      <li role="presentation">
       <a href="{$ServletsBaseURL}object/delete{$HttpSession}?id={$id}" role="menuitem" tabindex="-1" class="confirm_deletion option" data-text="Wirklich loeschen">
        <xsl:value-of select="i18n:translate('object.delWord')"/>
       </a>
      </li>
     </ul>
    </xsl:if>
    </div>
    <div class="row" style="margin-left:0px; margin-right:10px">
     <xsl:apply-templates select="structure/derobjects/derobject[acl:checkPermission(@xlink:href,'read')]">
      <xsl:with-param name="objID" select="@ID"/>
     </xsl:apply-templates>
    </div>
   </xsl:if>
  </xsl:template>
```

The *objectAction* template defines the selection menu appearing – once manual tagging has started – on the upper right hand side of the webpage entitled *Annotieren* and displaying the two options *next word* or *back to project*. The first thing to note here is that in line 7 a simple test excludes all guest users from accessing the procedure. After ensuring that only the user who owns the corpus project has access (line 15), s/he will be able to access the drop down menu, which is really a url, e.g. line 19. The attentive reader might have noticed that the url exactly matches the definition in the *web-fragment.xml* as shown in listing *Servlet Registering in the web-fragment.xml*, line 17, which resolves to the respective java class there. Really, this mechanism is the data interface within the MVC pattern. The url also contains two variables, named *derivCorp* and *corpID*, that are needed to identify the corpus and file object by the java classes (see section *Implementation*).

The morphilo.xsl stylesheet contains yet another modification that deserves mention. In listing *derobject template*, line 18, two menu options – *Tag automatically* and *Tag manually* – are defined. The former option initiates ProcessCorpusServlet.java as can be seen again in listing *Servlet Registering in the web-fragment.xml*, line 7, which determines words that are not in the master data base. Still, it is important to note that the menu option is only displayed if two restrictions are met. First, a file has to be uploaded (line 19) and, second, there must be only one file. This is necessary because in the annotation process other files will be generated that store the words that were not yet processed or a file that includes the final result. The generated files follow a certain pattern. The file harboring the final, entire TEI-annotated corpus is prefixed by *tagged*, the other file is prefixed *untagged*. This circumstance is exploited for manipulating the second option (line 27). A loop runs through all files in the respective directory and if a file name starts with *untagged*, the option to manually tag is displayed.

Listing 3: derobject template

```
1  <xsl:template match="derobject" mode="derivateActions">
2   <xsl:param name="deriv" />
3   <xsl:param name="parentObjID" />
4   <xsl:param name="suffix" select="''" />
5   <xsl:param name="id" select="../../../@ID" />
6   <xsl:if test="acl:checkPermission($deriv,'writedb')">
7    <xsl:variable name="ifsDirectory" select="document(concat('ifs:',$deriv,'/'))" />
8    <xsl:variable name="path" select="$ifsDirectory/mcr_directory/path" />
9   ...
10    <div class="options pull-right">
11     <div class="btn-group" style="margin:10px">
12      <a href="#" class="btn btn-default dropdown-toggle" data-toggle="dropdown">
13       <i class="fa fa-cog"></i>
14       <xsl:value-of select="' Korpus'"/>
15       <span class="caret"></span>
16      </a>
17     <ul class="dropdown-menu dropdown-menu-right">
18      <!-- Anpasssungen Morphilo -->|\label{ln:morphMenu}|
19      <xsl:if test="string-length($deriv) &gt; 0">|\label{ln:1test}|
20       <xsl:if test="count($ifsDirectory/mcr_directory/children/child) = 1">|\label
↪{ln:2test}|
21        <li role="presentation">
22         <a href="{$ServletsBaseURL}object/process{$HttpSession}?id={$deriv}&amp;objID=
↪{$id}" role="menuitem" tabindex="-1">
23          <xsl:value-of select="i18n:translate('derivate.process')"/>
24         </a>
25        </li>
26       </xsl:if>
27       <xsl:for-each select="$ifsDirectory/mcr_directory/children/child">|\label
↪{ln:loop}|
28        <xsl:variable name="untagged" select="concat($path, 'untagged')"/>
29        <xsl:variable name="filename" select="concat($path,./name)"/>
30        <xsl:if test="starts-with($filename, $untagged)">
31         <li role="presentation">
```

(continues on next page)

(continued from previous page)

```
32          <a href="{$ServletsBaseURL}object/tag{$HttpSession}?id={$deriv}&amp;objID={
    ↪$id}" role="menuitem" tabindex="-1">
33            <xsl:value-of select="i18n:translate('derivate.taggen')"/>
34          </a>
35        </li>
36       </xsl:if>
37      </xsl:for-each>
38     </xsl:if>
39     ...
40    </ul>
41   </div>
42  </div>
43 </xsl:if>
44 </xsl:template>
```

Besides the two stylesheets *morphilo.xsl* and *corpmeta.xsl*, other stylesheets have to be adjusted. They will not be discussed in detail here for they are self-explanatory for the most part. Essentially, they render the overall layout (*common-layout.xsl*, *skeleton_layout_template.xsl*) or the presentation of the search results (*response-page.xsl*) and definitions of the solr search fields (*searchfields-solr.xsl*). The former and latter also inherit templates from *response-general.xsl* and *response-browse.xsl*, in which the navigation bar of search results can be changed. For the use of multilinguality a separate configuration directory has to be created containing as many *.property*-files as different languages want to be displayed. In the current case these are restricted to German and English (*messages_de.properties* and *messages_en.properties*). The property files include all *i18n* definitions. All these files are located in the *resources* directory.

Furthermore, a search mask and a page for manually entering the annotations had to be designed. For these files a specially designed xml standard (*xed*) is recommended to be used within the repository framework.

CHAPTER 4

Controller Adjustments

4.1 General Principle of Operation

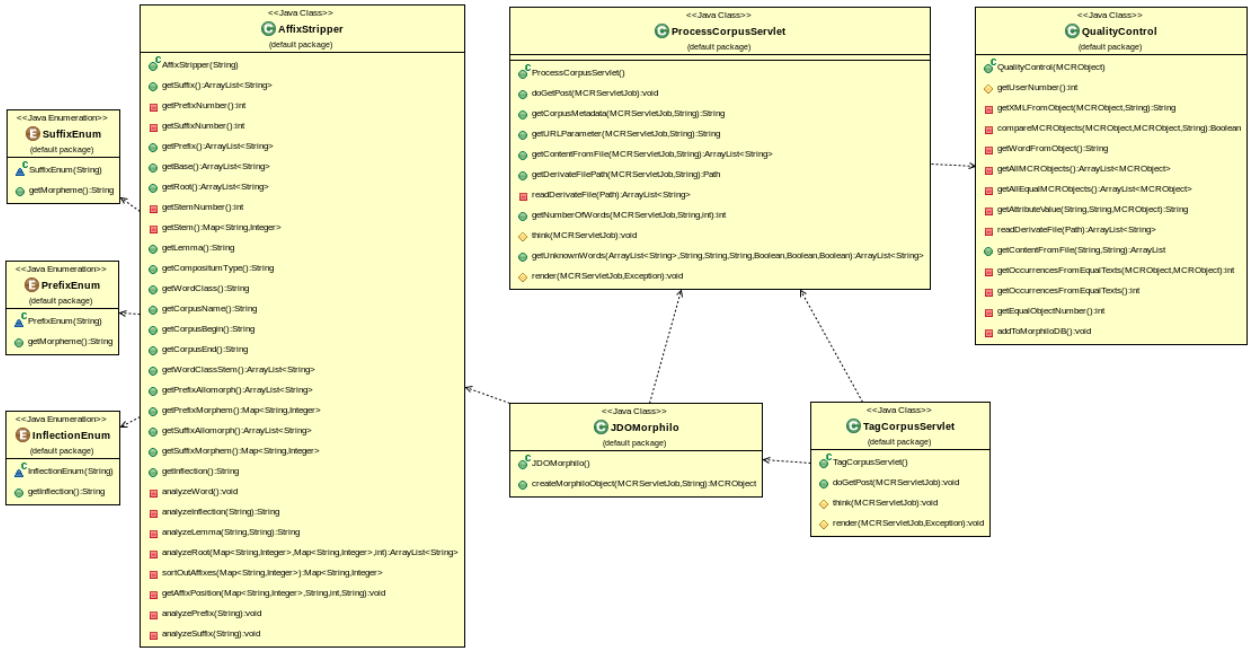

Fig. 1: Figure 3: Morphilo UML Diagramm

Figure *Figure 3: Morphilo UML Diagramm* illustrates the dependencies of the five java classes that were integrated to add the morphilo functionality defined in the default package *custom.mycore.addons.morphilo*. The general principle of operation is the following. The handling of data search, upload, saving, and user authentification is fully left to the MyCoRe functionality that is completely implemented. The class *ProcessCorpusServlet.java* receives a request from the webinterface to process an uploaded file, i.e. a simple text corpus, and it checks if any of the words are available in the master database. All words that are not listed in the master database are written to an extra file. These are the

words that have to be manually annotated. At the end, the servlet sends a response back to the user interface. In case of all words are contained in the master, an xml file is generated from the master database that includes all annotated words of the original corpus. Usually this will not be the case for larger textfiles. So if some words are not in the master, the user will get the response to initiate the manual annotation process.

The manual annotation process is processed by the class *TagCorpusServlet.java*, which will build a JDOM object for the first word in the extra file. This is done by creating an object of the *JDOMorphilo.java* class. This class, in turn, will use the methods of *AffixStripper.java* that make simple, but reasonable, suggestions on the word structure. This JDOM object is then given as a response back to the user. It is presented as a form, in which the user can make changes. This is necessary because the word structure algorithm of *AffixStripper.java* errs in some cases. Once the user agrees on the suggestions or on his or her corrections, the JDOM object is saved as an xml that is only searchable, visible, and changeable by the authenicated user (and the administrator), another file containing all processed words is created or updated respectively and the *TagCorpusServlet.java* servlet will restart until the last word in the extra list is processed. This enables the user to stop and resume her or his annotation work at a later point in time. The *TagCorpusServlet* will call methods from *ProcessCorpusServlet.java* to adjust the content of the extra files harboring the untagged words. If this file is empty, and only then, it is replaced by the file comprising all words from the original text file, both the ones from the master database and the ones that are annotated by the user, in an annotated xml representation.

Each time *ProcessCorpusServlet.java* is instantiated, it also instantiates *QualityControl.java*. This class checks if a new word can be transferred to the master database. The algorithm can be freely adopted to higher or lower quality standards. In its present configuration, a method tests at a limit of 20 different registered users agreeing on the annotation of the same word. More specifically, if 20 JDOM objects are identical except in the attribute field *occurrences* in the metadata node, the JDOM object becomes part of the master. The latter is easily done by changing the attribute *creator* from the user name to *administrator* in the service node. This makes the dataset part of the master database. Moreover, the *occurrences* attribute is updated by adding up all occurrences of the word that stem from different text corpora of the same time range.

4.2 Conceptualization

The controller component is largely specified and ready to use in some hundred or so java classes handling the logic of the search such as indexing, but also dealing with directories and files as saving, creating, deleting, and updating files. Moreover, a rudimentary user management comprising different roles and rights is offered. The basic technology behind the controller's logic is the servlet. As such all new code has to be registered as a servlet in the web-fragment.xml (here the Apache Tomcat container).

Listing 1: Servlet Registering in the web-fragment.xml

```
<servlet>
 <servlet-name>ProcessCorpusServlet</servlet-name>
 <servlet-class>custom.mycore.addons.morphilo.ProcessCorpusServlet</servlet-class>
</servlet>
<servlet-mapping>
 <servlet-name>ProcessCorpusServlet</servlet-name>
 <url-pattern>/servlets/object/process</url-pattern>
</servlet-mapping>
<servlet>
 <servlet-name>TagCorpusServlet</servlet-name>
 <servlet-class>custom.mycore.addons.morphilo.TagCorpusServlet</servlet-class>
</servlet>
<servlet-mapping>
 <servlet-name>TagCorpusServlet</servlet-name>
 <url-pattern>/servlets/object/tag</url-pattern>
</servlet-mapping>
\end{lstlisting}
```

Now, the logic has to be extended by the specifications. Some classes have to be added that take care of analyzing words (*AffixStripper.java, InflectionEnum.java, SuffixEnum.java, PrefixEnum.java*), extracting the relevant words from the text and checking the uniqueness of the text (*ProcessCorpusServlet.java*), make reasonable suggestions on the annotation (*TagCorpusServlet.java*), build the object of each annotated word (*JDOMorphilo.java*), and check on the quality by applying statistical models (*QualityControl.java*).

4.3 Implementation

Having taken a bird's eye perspective in the previous chapter, it is now time to take a look at the specific implementation at the level of methods. Starting with the main servlet, *ProcessCorpusServlet.java*, the class defines four getter method:

1. *public String getURLParameter(MCRServletJob, String)*
2. *public String getCorpusMetadata(MCRServletJob, String)*
3. *public ArrayList<String> getContentFromFile(MCRServletJob, String)*
4. *public Path getDerivateFilePath(MCRServletJob, String)*
5. *public int getNumberOfWords(MCRServletJob job, String)*

Since each servlet in MyCoRe extends the class MCRServlet, it has access to MCRServletJob, from which the http requests and responses can be used. This is the first argument in the above methods. The second argument of method (in 1.) specifies the name of an url parameter, i.e. the object id or the id of the derivate. The method returns the value of the given parameter. Typically *MyCoRe* uses the url to exchange these ids. The second method provides us with the value of a data field in the xml document. So the string defines the name of an attribute. *getContentFromFile(MCRServletJob, String)* returns the words as a list from a file when given the filename as a string. The getter listed in 4., returns the Path from the MyCoRe repository when the name of the file is specified. And finally, method (in 5.) returns the number of words by simply returning *getContentFromFile(job, fileName).size()*.

There are two methods in every MyCoRe-Servlet that have to be overwritten, *protected void render(MCRServletJob, Exception)*, which redirects the requests as *POST* or *GET* responds, and *protected void think(MCRServletJob)*, in which the logic is implemented. Since the latter is important to understand the core idea of the Morphilo algorithm, it is displayed in full length in source code *The overwritten think method*.

Listing 2: The overwritten think method

```
protected void think(MCRServletJob job) throws Exception
{
 this.job = job;
 String dateFromCorp = getCorpusMetadata(job, "def.datefrom");
 String dateUntilCorp = getCorpusMetadata(job, "def.dateuntil");
 String corpID = getURLParameter(job, "objID");
 String derivID = getURLParameter(job, "id");

 //if NoW is 0, fill with anzWords
 MCRObject helpObj = MCRMetadataManager.retrieveMCRObject(MCRObjectID.
↪getInstance(corpID));
 Document jdomDocHelp = helpObj.createXML();
 XPathFactory xpfacty = XPathFactory.instance();
 XPathExpression<Element> xpExp = xpfacty.compile("//NoW", Filters.element());
 Element elem = xpExp.evaluateFirst(jdomDocHelp);
 //fixes transferred morphilo data from previous stand alone project
 int corpussize = getNumberOfWords(job, "");
 if (Integer.parseInt(elem.getText()) != corpussize)
 {
  elem.setText(Integer.toString(corpussize));
```

(continues on next page)

(continued from previous page)

```
20   helpObj = new MCRObject(jdomDocHelp);
21   MCRMetadataManager.update(helpObj);
22  }
23
24  //Check if the uploaded corpus was processed before
25  SolrClient slr = MCRSolrClientFactory.getSolrClient();
26  SolrQuery qry = new SolrQuery();
27  qry.setFields("korpusname", "datefrom", "dateuntil", "NoW", "id");
28  qry.setQuery("datefrom:" + dateFromCorp + " AND dateuntil:" + dateUntilCorp + " AND␣
    ↪NoW:" + corpussize);
29  SolrDocumentList rslt = slr.query(qry).getResults();
30
31  Boolean incrOcc = true;
32  // if resultset contains only one, then it must be the newly created corpus
33  if (slr.query(qry).getResults().getNumFound() > 1)
34  {
35   incrOcc = false;
36  }
37
38  //match all words in corpus with morphilo (creator=administrator) and save all words␣
    ↪that are not in morphilo DB in leftovers
39  ArrayList<String> leftovers = new ArrayList<String>();
40  ArrayList<String> processed = new ArrayList<String>();
41
42  leftovers = getUnknownWords(getContentFromFile(job, ""), dateFromCorp, dateUntilCorp,
    ↪ "", incrOcc, incrOcc, false);
43
44  //write all words of leftover in file as derivative to respective corpmeta dataset
45  MCRPath root = MCRPath.getPath(derivID, "/");
46  Path fn = getDerivateFilePath(job, "").getFileName();
47  Path p = root.resolve("untagged-" + fn);
48  Files.write(p, leftovers);
49
50  //create a file for all words that were processed
51  Path procWds = root.resolve("processed-" + fn);
52  Files.write(procWds, processed);
53 }
```

Using the above mentioned getter methods, the *think*-method assigns values to the object ID, needed to get the xml document that contains the corpus metadata, the file ID, and the beginning and starting dates from the corpus to be analyzed. Lines 10 through 22 show how to access a mycore object as an xml document, a procedure that will be used in different variants throughout this implementation. By means of the object ID, the respective corpus is identified and a JDOM document is constructed, which can then be accessed by XPath. The XPath factory instances are collections of the xml nodes. In the present case, it is save to assume that only one element of *NoW* is available (see corpus datamodel listing *Corpus Data Model* with *maxOccurs='1'*). So we do not have to loop through the collection, but use the first node named *NoW*. The if-test checks if the number of words of the uploaded file is the same as the number written in the document. When the document is initially created by the MyCoRe logic it was configured to be zero. If unequal, the *setText(String)* method is used to write the number of words of the corpus to the document.

Lines 25–36 reveal the second important ingredient, i.e. controlling the search engine. First, a solr client and a query are initialized. Then, the output of the result set is defined by giving the fields of interest of the document. In the case at hand, it is the id, the name of the corpus, the number of words, and the beginnig and ending dates. With *setQuery* it is possible to assign values to some or all of these fields. Finally, *getResults()* carries out the search and writes all hits to a *SolrDocumentList* (line 29). The test that follows is really only to set a Boolean encoding if the number of occurrences of that word in the master should be updated. To avoid multiple counts, incrementing the word frequency is only done if it is a new corpus.

In line 42 *getUnknownWords(ArrayList, String, String, String, Boolean, Boolean, Boolean)* is called and returned as a list of words. This method is key and will be discussed in depth below. Finally, lines 45–48 show how to handle file objects in MyCoRe. Using the file ID, the root path and the name of the first file in that path are identified. Then, a second file starting with *untagged* is created and all words returned from the *getUnknownWords* is written to that file. By the same token an empty file is created (in the last two lines of the *think*-method), in which all words that are manually annotated will be saved.

In a refactoring phase, the method *getUnknownWords(ArrayList, String, String, String, Boolean, Boolean, Boolean)* could be subdivided into three methods: for each Boolean parameter one. In fact, this method handles more than one task. This is mainly due to multiple code avoidance. In essence, an outer loop runs through all words of the corpus and an inner loop runs through all hits in the solr result set. Because the result set is supposed to be small, approximately between 10-20 items, efficiency problems are unlikely to cause a problem, although there are some more loops running through collection of about the same sizes. Since each word is identified on the basis of its projected word type, the word form, and the time range it falls into, it is these variables that have to be checked for existence in the documents. If not in the xml documents, *null* is returned and needs to be corrected. Moreover, user authentification must be considered. There are three different XPaths that are relevant.

- *//service/servflags/servflag[@type='createdby']* to test for the correct user
- *//morphiloContainer/morphilo* to create the annotated document
- *//morphiloContainer/morphilo/w* to set occurrences or add a link

As an illustration of the core functioning of this method, listing *Mode of Operation of getUnknownWords Method* is given.

Listing 3: Mode of Operation of getUnknownWords Method

```
1  public ArrayList<String> getUnknownWords(
2   ArrayList<String> corpus,
3   String timeCorpusBegin,
4   String timeCorpusEnd,
5   String wdtpe,
6   Boolean setOcc,
7   Boolean setXlink,
8   Boolean writeAllData) throws Exception
9   {
10   String currentUser = MCRSessionMgr.getCurrentSession().getUserInformation().
    ↪getUserID();
11   ArrayList lo = new ArrayList();
12
13   for (int i = 0; i < corpus.size(); i++)
14   {
15    SolrClient solrClient = MCRSolrClientFactory.getSolrClient();
16    SolrQuery query = new SolrQuery();
17    query.setFields("w","occurrence","begin","end", "id", "wordtype");
18    query.setQuery(corpus.get(i));
19    query.setRows(50); //more than 50 items are extremely unlikely
20    SolrDocumentList results = solrClient.query(query).getResults();
21    Boolean available = false;
22    for (int entryNum = 0; entryNum < results.size(); entryNum++)
23    {
24     ...
25     // update in MCRMetaDataManager
26     String mcrIDString = results.get(entryNum).getFieldValue("id").toString();
27     //MCRObjekt auslesen und JDOM-Document erzeugen:
28     MCRObject mcrObj = MCRMetadataManager.retrieveMCRObject(MCRObjectID.
    ↪getInstance(mcrIDString));
29     Document jdomDoc = mcrObj.createXML();
```

(continues on next page)

(continued from previous page)

```
    ...
    //check and correction for word type
    ...
    //checkand correction time: timeCorrect
    ...
    //check if user correct: isAuthorized
   ...
   XPathExpression<Element> xp = xpfac.compile("//morphiloContainer/morphilo/w",␣
↪Filters.element());
   //Iterates w-elements and increments occurrence attribute if setOcc is true
   for (Element e : xp.evaluate(jdomDoc))
   {
    //wenn Rechte da sind und Worttyp nirgends gegeben oder gleich ist
      if (isAuthorized && timeCorrect
       && ((e.getAttributeValue("wordtype") == null && wdtpe.equals(""))
       || e.getAttributeValue("wordtype").equals(wordtype))) // nur zur␣
↪Vereinheitlichung
     {
        int oc = -1;
        available = true;
      try
        {
         //adjust occurrence Attribut
         if (setOcc)
       {
        oc = Integer.parseInt(e.getAttributeValue("occurrence"));
              e.setAttribute("occurrence", Integer.toString(oc + 1));
         }

       //write morphilo-ObjectID in xml of corpmeta
         if (setXlink)
         {
              Namespace xlinkNamespace = Namespace.getNamespace("xlink", "http://www.
↪w3.org/1999/xlink");
              MCRObject corpObj = MCRMetadataManager.retrieveMCRObject(MCRObjectID.
↪getInstance(getURLParameter(job, "objID")));
              Document corpDoc = corpObj.createXML();
              XPathExpression<Element> xpathEx = xpfac.compile("//corpuslink",␣
↪Filters.element());
              Element elm = xpathEx.evaluateFirst(corpDoc);
              elm.setAttribute("href" , mcrIDString, xlinkNamespace);
         }
         mcrObj = new MCRObject(jdomDoc);
         MCRMetadataManager.update(mcrObj);
         QualityControl qc = new QualityControl(mcrObj);
        }
        catch(NumberFormatException except)
        {
         // ignore
        }
       }
      }
    if (!available) // if not available in datasets under the given conditions
    {
     lo.add(corpus.get(i));
    }
   }
```

(continues on next page)

(continued from previous page)

```
82     return lo;
83   }
```

As can be seen from the functionality of listing *Mode of Operation of getUnknownWords Method*, getting the unknown words of a corpus, is rather a side effect for the equally named method. More precisely, a Boolean (line 47) is set when the document is manipulated otherwise because it is clear that the word must exist then. If the Boolean remains false (line 77), the word is put on the list of words that have to be annotated manually. As already explained above, the first loop runs through all words (corpus) and the following lines a solr result set is created. This set is also looped through and it is checked if the time range, the word type and the user are authorized. In the remainder, the occurrence attribute of the morphilo document can be incremented (*setOcc* is true) or/and the word is linked to the corpus meta data (*setXlink* is true). While all code lines are equivalent with what was explained in listing *The overwritten think method*, it suffices to focus on an additional name space, i.e. *xlink* has to be defined (line 60). Once the linking of word and corpus is set, the entire MyCoRe object has to be updated. This is done by the functionality of the framework (lines 67–69). At the end, an instance of *QualityControl* is created.

The class *QualityControl* is instantiated with a constructor depicted in listing *Constructor of QualityControl.java*.

Listing 4: Constructor of QualityControl.java

```
private MCRObject mycoreObject;
/* Constructor calls method to carry out quality control, i.e. if at least 20
 * different users agree 100% on the segments of the word under investigation
 */
public QualityControl(MCRObject mycoreObject) throws Exception
{
 this.mycoreObject = mycoreObject;
 if (getEqualObjectNumber() > 20)
 {
  addToMorphiloDB();
 }
}
```

The constructor takes an MyCoRe object, a potential word candidate for the master data base, which is assigned to a private class variable because the object is used though not changed by some other java methods. More importantly, there are two more methods: *getEqualNumber()* and *addToMorphiloDB()*. While the former initiates a process of counting and comparing objects, the latter is concerned with calculating the correct number of occurrences from different, but not the same texts, and generating a MyCoRe object with the same content but with two different flags in the *//service/servflags/servflag*-node, i.e. *createdby='administrator'* and *state='published'*. And of course, the *occurrence* attribute is set to the newly calculated value. The logic corresponds exactly to what was explained in listing *The overwritten think method* and will not be repeated here. The only difference are the paths compiled by the XPathFactory. They are

- *//service/servflags/servflag[@type='createdby']* and
- *//service/servstates/servstate[@classid='state']*.

It is more instructive to document how the number of occurrences is calculated. There are two steps involved. First, a list with all mycore objects that are equal to the object which the class is instantiated with (*mycoreObject* in listing *Constructor of QualityControl.java*) is created. This list is looped and all occurrence attributes are summed up. Second, all occurrences from equal texts are substracted. Equal texts are identified on the basis of its meta data and its derivate.

Listing 5: Occurrence Extraction from Equal Texts

```
1  /* returns number of Occurrences if Objects are equal, zero otherwise
2   */
```

(continues on next page)

(continued from previous page)

```
3  private int getOccurrencesFromEqualTexts(MCRObject mcrobj1, MCRObject mcrobj2) throws␣
   ↪SAXException, IOException
4  {
5   int occurrences = 1;
6   //extract corpmeta ObjectIDs from morphilo-Objects
7   String crpID1 = getAttributeValue("//corpuslink", "href", mcrobj1);
8   String crpID2 = getAttributeValue("//corpuslink", "href", mcrobj2);
9   //get these two corpmeta Objects
10  MCRObject corpo1 = MCRMetadataManager.retrieveMCRObject(MCRObjectID.
   ↪getInstance(crpID1));
11  MCRObject corpo2 = MCRMetadataManager.retrieveMCRObject(MCRObjectID.
   ↪getInstance(crpID2));
12  //are the texts equal? get list of 'processed-words' derivate
13  String corp1DerivID = getAttributeValue("//structure/derobjects/derobject", "href",␣
   ↪corpo1);
14  String corp2DerivID = getAttributeValue("//structure/derobjects/derobject", "href",␣
   ↪corpo2);
15
16  ArrayList result = new ArrayList(getContentFromFile(corp1DerivID, ""));
17  result.remove(getContentFromFile(corp2DerivID, ""));
18  if (result.size() == 0) // the texts are equal
19  {
20   // extract occurrences of one the objects
21   occurrences = Integer.parseInt(getAttributeValue("//morphiloContainer/morphilo/w",
   ↪"occurrence", mcrobj1));
22  }
23  else
24  {
25   occurrences = 0; //project metadata happened to be the same, but texts are different
26  }
27  return occurrences;
28 }
```

In this implementation, the ids from the *corpmeta* data model are accessed via the xlink attribute in the morphilo documents. The method *getAttributeValue(String, String, MCRObject)* does exactly the same as demonstrated earlier (see from line 60 on in listing *Mode of Operation of getUnknownWords Method*). The underlying logic is that the texts are equal if exactly the same number of words were uploaded. So all words from one file are written to a list (line 16) and words from the other file are removed from the very same list (line 17). If this list is empty, then the exact same number of words must have been in both files and the occurrences are adjusted accordingly. Since this method is called from another private method that only contains a loop through all equal objects, one gets the occurrences from all equal texts. For reasons of confirmability, the looping method is also given:

Listing 6: Occurrence Extraction from Equal Texts (2)

```
1  private int getOccurrencesFromEqualTexts() throws Exception
2  {
3   ArrayList<MCRObject> equalObjects = new ArrayList<MCRObject>();
4   equalObjects = getAllEqualMCRObjects();
5   int occurrences = 0;
6   for (MCRObject obj : equalObjects)
7   {
8    occurrences = occurrences + getOccurrencesFromEqualTexts(mycoreObject, obj);
9   }
10  return occurrences;
11 }
```

Now, the constructor in listing *Constructor of QualityControl.java* reveals another method that rolls out an equally

complex concatenation of procedures. As implied above, *getEqualObjectNumber()* returns the number of equally annotated words. It does this by falling back to another method from which the size of the returned list is calculated (*getAllEqualMCRObjects().size()*). Hence, we should care about *getAllEqualMCRObjects()*. This method really has the same design as *int getOccurrencesFromEqualTexts()* in listing *Occurrence Extraction from Equal Texts (2)*. The difference is that another method (*Boolean compareMCRObjects(MCRObject, MCRObject, String)*) is used within the loop and that all equal objects are put into the list of MyCoRe objects that are returned. If this list comprises more than 20 entries,[1] the respective document will be integrated in the master data base by the process described above. The comparator logic is shown in listing *Comparison of MyCoRe objects*.

Listing 7: Comparison of MyCoRe objects

```
private Boolean compareMCRObjects(MCRObject mcrobj1, MCRObject mcrobj2, String xpath)␣
↪throws SAXException, IOException
{
 Boolean isEqual = false;
 Boolean beginTime = false;
 Boolean endTime = false;
 Boolean occDiff = false;
 Boolean corpusDiff = false;

 String source = getXMLFromObject(mcrobj1, xpath);
 String target = getXMLFromObject(mcrobj2, xpath);

 XMLUnit.setIgnoreAttributeOrder(true);
 XMLUnit.setIgnoreComments(true);
 XMLUnit.setIgnoreDiffBetweenTextAndCDATA(true);
 XMLUnit.setIgnoreWhitespace(true);
 XMLUnit.setNormalizeWhitespace(true);

 //differences in occurrences, end, begin should be ignored
 try
 {
  Diff xmlDiff = new Diff(source, target);
  DetailedDiff dd = new DetailedDiff(xmlDiff);
  //counters for differences
  int i = 0;
  int j = 0;
  int k = 0;
  int l = 0;
  // list containing all differences
  List differences = dd.getAllDifferences();
  for (Object object : differences)
  {
   Difference difference = (Difference) object;
   //@begin,@end,... node is not in the difference list if the count is 0
   if (difference.getControlNodeDetail().getXpathLocation().endsWith("@begin")) i++;
   if (difference.getControlNodeDetail().getXpathLocation().endsWith("@end")) j++;
   if (difference.getControlNodeDetail().getXpathLocation().endsWith("@occurrence"))␣
↪k++;
   if (difference.getControlNodeDetail().getXpathLocation().endsWith("@corpus")) l++;
   //@begin and @end have different values: they must be checked if they fall right␣
↪in the allowed time range
   if ( difference.getControlNodeDetail().getXpathLocation().equals(difference.
↪getTestNodeDetail().getXpathLocation())
      && difference.getControlNodeDetail().getXpathLocation().endsWith("@begin")
      && (Integer.parseInt(difference.getControlNodeDetail().getValue()) < Integer.
↪parseInt(difference.getTestNodeDetail().getValue())) )
```

(continues on next page)

[1] This number is somewhat arbitrary. It is inspired by the sample size n in t-distributed data.

(continued from previous page)

```
42     {
43        beginTime = true;
44     }
45     if (difference.getControlNodeDetail().getXpathLocation().equals(difference.
↪getTestNodeDetail().getXpathLocation())
46        && difference.getControlNodeDetail().getXpathLocation().endsWith("@end")
47        && (Integer.parseInt(difference.getControlNodeDetail().getValue()) > Integer.
↪parseInt(difference.getTestNodeDetail().getValue())) )
48     {
49        endTime = true;
50     }
51     //attribute values of @occurrence and @corpus are ignored if they are different
52     if (difference.getControlNodeDetail().getXpathLocation().equals(difference.
↪getTestNodeDetail().getXpathLocation())
53        && difference.getControlNodeDetail().getXpathLocation().endsWith("@occurrence"))
54     {
55        occDiff = true;
56     }
57     if (difference.getControlNodeDetail().getXpathLocation().equals(difference.
↪getTestNodeDetail().getXpathLocation())
58        && difference.getControlNodeDetail().getXpathLocation().endsWith("@corpus"))
59     {
60        corpusDiff = true;
61     }
62   }
63   //if any of @begin, @end ... is identical set Boolean to true
64   if (i == 0) beginTime = true;
65   if (j == 0) endTime = true;
66   if (k == 0) occDiff = true;
67   if (l == 0) corpusDiff = true;
68   //if the size of differences is greater than the number of changes admitted in␣
↪@begin, @end ... something else must be different
69   if (beginTime && endTime && occDiff && corpusDiff && (i + j + k + l) == dd.
↪getAllDifferences().size()) isEqual = true;
70   }
71   catch (SAXException e)
72   {
73    e.printStackTrace();
74   }
75   catch (IOException e)
76   {
77    e.printStackTrace();
78   }
79  return isEqual;
80 }
```

In this method, XMLUnit is heavily used to make all necessary node comparisons. The matter becomes more complicated, however, if some attributes are not only ignored, but evaluated according to a given definition as it is the case for the time range. If the evaluator and builder classes are not to be overwritten entirely because needed for evaluating other nodes of the xml document, the above solution appears a bit awkward. So there is potential for improvement before the production version is to be programmed.

XMLUnit provides us with a list of the differences of the two documents (see line 29). There are four differences allowed, that is, the attributes *occurrence*, *corpus*, *begin*, and *end*. For each of them a Boolean variable is set. Because any of the attributes could also be equal to the master document and the difference list only contains the actual differences, one has to find a way to define both, equal and different, for the attributes. This could be done by ignoring these nodes. Yet, this would not include testing if the beginning and ending dates fall into the range of the master

document. Therefore the attributes are counted as lines 34 through 37 reveal. If any two documents differ in some of the four attributes just specified, then the sum of the counters (line 69) should not be greater than the collected differences by XMLUnit. The rest of the if-tests assign truth values to the respective Booleans. It is probably worth mentioning that if all counters are zero (lines 64–67) the attributes and values are identical and hence the Boolean has to be set explicitly. Otherwise the test in line 69 would fail.

Once quality control (explained in detail further down) has been passed, it is the user's turn to interact further. By clicking on the option *Manual tagging*, the *TagCorpusServlet* will be callled. This servlet instantiates *ProcessCorpusServlet* to get access to the *getUnknownWords*-method, which delivers the words still to be processed and which overwrites the content of the file starting with *untagged*. For the next word in *leftovers* a new MyCoRe object is created using the JDOM API and added to the file beginning with *processed*. In line 16 of listing *Manual Tagging Procedure*, the previously defined entry mask is called, with which the proposed word structure could be confirmed or changed. How the word structure is determined will be shown later in the text.

Listing 8: Manual Tagging Procedure

```
1  ...
2  if (!leftovers.isEmpty())
3  {
4   ArrayList<String> processed = new ArrayList<String>();
5   //processed.add(leftovers.get(0));
6   JDOMorphilo jdm = new JDOMorphilo();
7   MCRObject obj = jdm.createMorphiloObject(job, leftovers.get(0));
8   //write word to be annotated in process list and save it
9   Path filePathProc = pcs.getDerivateFilePath(job, "processed").getFileName();
10  Path proc = root.resolve(filePathProc);
11  processed = pcs.getContentFromFile(job, "processed");
12  processed.add(leftovers.get(0));
13  Files.write(proc, processed);
14
15  //call entry mask for next word
16  tagUrl = prop.getBaseURL() + "content/publish/morphilo.xed?id=" + obj.getId();
17 }
18 else
19 {
20  //initiate process to give a complete tagged file of the original corpus
21  //if untagged-file is empty, match original file with morphilo
22  //creator=administrator OR creator=username and write matches in a new file
23  ArrayList<String> complete = new ArrayList<String>();
24  ProcessCorpusServlet pcs2 = new ProcessCorpusServlet();
25  complete = pcs2.getUnknownWords(
26   pcs2.getContentFromFile(job, ""), //main corpus file
27   pcs2.getCorpusMetadata(job, "def.datefrom"),
28   pcs2.getCorpusMetadata(job, "def.dateuntil"),
29   "", //wordtype
30   false,
31   false,
32   true);
33
34  Files.delete(p);
35  MCRXMLFunctions mdm = new MCRXMLFunctions();
36  String mainFile = mdm.getMainDocName(derivID);
37  Path newRoot = root.resolve("tagged-" + mainFile);
38  Files.write(newRoot, complete);
39
40  //return to Menu page
41  tagUrl = prop.getBaseURL() + "receive/" + corpID;
42 }
```

At the point where no more items are in *leftsovers* the *getUnknownWords*-method is called whereas the last Boolean parameter is set true. This indicates that the array list containing all available and relevant data to the respective user is returned as seen in the code snippet in listing ref{src:writeAll}.

Listing 9: Code snippet to deliver all data to the user

```
...
// all data is written to lo in TEI
if (writeAllData && isAuthorized && timeCorrect)
{
 XPathExpression<Element> xpath = xpfac.compile("//morphiloContainer/morphilo",␣
↪Filters.element());
 for (Element e : xpath.evaluate(jdomDoc))
 {
  XMLOutputter outputter = new XMLOutputter();
  outputter.setFormat(Format.getPrettyFormat());
  lo.add(outputter.outputString(e.getContent()));
 }
}
...
```

The complete list (*lo*) is written to yet a third file starting with *tagged* and finally returned to the main project webpage.

The interesting question is now where does the word structure come from, which is filled in the entry mask as asserted above. In listing *Manual Tagging Procedure* line 7, one can see that a JDOM object is created and the method *createMorphiloObject(MCRServletJob, String)* is called. The string parameter is the word that needs to be analyzed. Most of the method is a mere application of the JDOM API given the data model in *Conceptualization* and listing *Word Data Model*. That means namespaces, elements and their attributes are defined in the correct order and hierarchy.

To fill the elements and attributes with text, i.e. prefixes, suffixes, stems, etc., a Hashmap – containing the morpheme as key and its position as value – are created that are filled with the results from an AffixStripper instantiation. Depending on how many prefixes or suffixes respectively are put in the hashmap, the same number of xml elements are created. As a final step, a valid MyCoRe id is generated using the existing MyCoRe functionality, the object is created and returned to the TagCorpusServlet.

Last, the analyses of the word structure will be considered. It is implemented in the *AffixStripper.java* file. All lexical affix morphemes and their allomorphs as well as the inflections were extracted from the Oxford English Dictionary and saved as enumerated lists (see the example in listing *Enumeration Example for the Prefix over*). The allomorphic items of these lists are mapped successively to the beginning in the case of prefixes (see listing *Method to recognize prefixes*, line 7) or to the end of words in the case of suffixes (see listing *Cut-off mechanism for suffixes*). Since each morphemic variant maps to its morpheme right away, it makes sense to use the morpheme and so implicitly keep the relation to its allomorph.

Listing 10: Enumeration Example for the Prefix *over*

```
package custom.mycore.addons.morphilo;
public enum PrefixEnum {
...
 over("over"), ufer("over"), ufor("over"), uferr("over"), uvver("over"), obaer("over
↪"), ober("over)"), ofaer("over"),
 ofere("over"), ofir("over"), ofor("over"), ofer("over"), ouer("over"),oferr("over"),␣
↪offerr("over"), offr("over"), aure("over"),
 war("over"), euer("over"), oferre("over"), oouer("over"), oger("over"), ouere("over
↪"), ouir("over"), ouire("over"),
 ouur("over"), ouver("over"), ouyr("over"), ovar("over"), overe("over"), ovre("over"),
↪ovur("over"), owuere("over"), owver("over"),
 houyr("over"), ouyre("over"), ovir("over"), ovyr("over"), hover("over"), auver("over
↪"), awver("over"), ovver("over"),
```

(continues on next page)

(continued from previous page)

```
 hauver("over"), ova("over"), ove("over"), obuh("over"), ovah("over"), ovuh("over"),␣
↪ofowr("over"), ouuer("over"), oure("over"),
 owere("over"), owr("over"), owre("over"), owur("over"), owyr("over"), our("over"),␣
↪ower("over"), oher("over"),
 ooer("over"), oor("over"), owwer("over"), ovr("over"), owir("over"), oar("over"),␣
↪aur("over"), oer("over"), ufara("over"),
 ufera("over"), ufere("over"), uferra("over"), ufora("over"), ufore("over"), ufra(
↪"over"), ufre("over"), ufyrra("over"),
 yfera("over"), yfere("over"), yferra("over"), uuera("over"), ufe("over"), uferre(
↪"over"), uuer("over"), uuere("over"),
 vfere("over"), vuer("over"), vuere("over"), vver("over"), uvvor("over") ...
 private String morpheme;
 //constructor
 PrefixEnum(String morpheme)
 {
  this.morpheme = morpheme;
 }
 //getter Method

 public String getMorpheme()
 {
  return this.morpheme;
 }
}
```

As can be seen in line 12 in listing *Method to recognize prefixes*, the morpheme is saved to a hash map together with its position, i.e. the size of the map plus one at the time being. In line 14 the *analyzePrefix* method is recursively called until no more matches can be made.

Listing 11: Method to recognize prefixes

```
1  private Map<String, Integer> prefixMorpheme = new HashMap<String,Integer>();
2  ...
3  private void analyzePrefix(String restword)
4  {
5   if (!restword.isEmpty()) //Abbruchbedingung fuer Rekursion
6   {
7    for (PrefixEnum prefEnum : PrefixEnum.values())
8    {
9     String s = prefEnum.toString();
10    if (restword.startsWith(s))
11    {
12       prefixMorpheme.put(s, prefixMorpheme.size() + 1);
13       //cut off the prefix that is added to the list
14       analyzePrefix(restword.substring(s.length()));
15    }
16    else
17    {
18       analyzePrefix("");
19    }
20   }
21  }
22 }
```

The recognition of suffixes differs only in the cut-off direction since suffixes occur at the end of a word. Hence, line 14 in listing *Method to recognize prefixes* reads in the case of suffixes.

Listing 12: Cut-off mechanism for suffixes

```
analyzeSuffix(restword.substring(0, restword.length() - s.length()));
```

It is important to note that inflections are suffixes (in the given model case of Middle English morphology) that usually occur at the very end of a word, i.e. after all lexical suffixes, only once. It follows that inflections have to be recognized at first without any repetition. So the procedure for inflections can be simplified to a substantial degree as listing *Method to recognize inflections* shows.

Listing 13: Method to recognize inflections

```
private String analyzeInflection(String wrd)
{
 String infl = "";
 for (InflectionEnum inflEnum : InflectionEnum.values())
 {
  if (wrd.endsWith(inflEnum.toString()))
  {
   infl = inflEnum.toString();
  }
 }
 return infl;
}
```

Unfortunately the embeddedness problem prevents a very simple algorithm. Embeddedness occurs when a lexical item is a substring of another lexical item. To illustrate, the suffix *ion* is also contained in the suffix *ation*, as is *ent* in *ment*, and so on. The embeddedness problem cannot be solved completely on the basis of linear modelling, but for a large part of embedded items one can work around it using implicitly Zipf's law, i.e. the correlation between frequency and length of lexical items. The longer a word becomes, the less frequent it will occur. The simplest logic out of it is to assume that longer suffixes (measured in letters) are preferred over shorter suffixes because it is more likely tha the longer the suffix string becomes, the more likely it is one (as opposed to several) suffix unit(s). This is done in listing *Method to workaround embeddedness*, whereas the inner class *sortedByLengthMap* returns a list sorted by length and the loop from line 17 onwards deletes the respective substrings.

Listing 14: Method to workaround embeddedness

```
1  private Map<String, Integer> sortOutAffixes(Map<String, Integer> affix)
2  {
3   Map<String,Integer> sortedByLengthMap = new TreeMap<String, Integer>(new Comparator
   ↪<String>()
4    {
5     @Override
6     public int compare(String s1, String s2)
7     {
8      int cmp = Integer.compare(s1.length(), s2.length());
9      return cmp != 0 ? cmp : s1.compareTo(s2);
10    }
11   }
12  );
13  sortedByLengthMap.putAll(affix);
14  ArrayList<String> al1 = new ArrayList<String>(sortedByLengthMap.keySet());
15  ArrayList<String> al2 = al1;
16  Collections.reverse(al2);
17  for (String s2 : al1)
18  {
19   for (String s1 : al2)
```

(continues on next page)

(continued from previous page)

```
20     if (s1.contains(s2) && s1.length() > s2.length())
21     {
22        affix.remove(s2);
23     }
24    }
25   return affix;
26  }
```

Finally, the position of the affix has to be calculated because the hashmap in line 12 in listing *Method to recognize prefixes* does not keep the original order for changes taken place in addressing the affix embeddedness (listing *Method to workaround embeddedness*). Listing *Method to determine position of the affix* depicts the preferred solution. The recursive construction of the method is similar to *private void analyzePrefix(String)* (listing *Method to recognize prefixes*) only that the two affix types are handled in one method. For that, an additional parameter taking the form either *suffix* or *prefix* is included.

Listing 15: Method to determine position of the affix

```
private void getAffixPosition(Map<String, Integer> affix, String restword, int pos,␣
↪String affixtype)
{
 if (!restword.isEmpty()) //Abbruchbedingung fuer Rekursion
 {
  for (String s : affix.keySet())
  {
   if (restword.startsWith(s) && affixtype.equals("prefix"))
   {
      pos++;
      prefixMorpheme.put(s, pos);
    //prefixAllomorph.add(pos-1, restword.substring(s.length()));
      getAffixPosition(affix, restword.substring(s.length()), pos, affixtype);
   }
   else if (restword.endsWith(s) && affixtype.equals("suffix"))
   {
      pos++;
      suffixMorpheme.put(s, pos);
      //suffixAllomorph.add(pos-1, restword.substring(s.length()));
      getAffixPosition(affix, restword.substring(0, restword.length() - s.length()),␣
↪pos, affixtype);
   }
   else
   {
      getAffixPosition(affix, "", pos, affixtype);
   }
  }
 }
}
```

To give the complete word structure, the root of a word should also be provided. In listing *Method to determine roots* a simple solution is offered, however, considering compounds as words consisting of more than one root.

Listing 16: Method to determine roots

```
private ArrayList<String> analyzeRoot(Map<String, Integer> pref, Map<String, Integer>␣
↪suf, int stemNumber)
{
 ArrayList<String> root = new ArrayList<String>();
```

(continues on next page)

(continued from previous page)

```
 int j = 1; //one root always exists
 // if word is a compound several roots exist
 while (j <= stemNumber)
 {
  j++;
  String rest = lemma;

  for (int i=0;i<pref.size();i++)
  {
   for (String s : pref.keySet())
   {
    //if (i == pref.get(s))
      if (rest.length() > s.length() && s.equals(rest.substring(0, s.length())))
      {
       rest = rest.substring(s.length(),rest.length());
    }
   }
  }

  for (int i=0;i<suf.size();i++)
  {
   for (String s : suf.keySet())
   {
      //if (i == suf.get(s))
      if (s.length() < rest.length() && (s.equals(rest.substring(rest.length() - s.
↪length(), rest.length()))))
      {
       rest = rest.substring(0, rest.length() - s.length());
      }
   }
  }
  root.add(rest);
 }
 return root;
}
```

The logic behind this method is that the root is the remainder of a word when all prefixes and suffixes are substracted. So the loops run through the number of prefixes and suffixes at each position and substract the affix. Really, there is some code doubling with the previously described methods, which could be eliminated by making it more modular in a possible refactoring phase. Again, this is not the concern of a prototype. Line 9 defines the initial state of a root, which is the case for monomorphemic words. The *lemma* is defined as the wordtoken without the inflection. Thus listing *Method to determine lemma* reveals how the class variable is calculated

Listing 17: Method to determine lemma

```
/*
 * Simplification: lemma = wordtoken - inflection
 */
private String analyzeLemma(String wrd, String infl)
{
 return wrd.substring(0, wrd.length() - infl.length());
}
```

The constructor of *AffixStripper* calls the method *analyzeWord()* whose only job is to calculate each structure element in the correct order (listing *Method to determine lemma*. All structure elements are also provided by getters.

Listing 18: Method to determine all word structure

```
private void analyzeWord()
{
 //analyze inflection first because it always occurs at the end of a word
 inflection = analyzeInflection(wordtoken);
 lemma = analyzeLemma(wordtoken, inflection);
 analyzePrefix(lemma);
 analyzeSuffix(lemma);
 getAffixPosition(sortOutAffixes(prefixMorpheme), lemma, 0, "prefix");
 getAffixPosition(sortOutAffixes(suffixMorpheme), lemma, 0, "suffix");
 prefixNumber = prefixMorpheme.size();
 suffixNumber = suffixMorpheme.size();
 wordroot = analyzeRoot(prefixMorpheme, suffixMorpheme, getStemNumber());
}
```

To conclude, the Morphilo implementation as presented here, aims at fulfilling the task of a working prototype. It is important to note that it neither claims to be a very efficient nor a ready software program to be used in production. However, it marks a crucial milestone on the way to a production system. At some listings sources of improvement were made explicit; at others no suggestions were made. In the latter case this does not imply that there is no potential for improvement. Once acceptability tests are carried out, it will be the task of a follow up project to identify these potentials and implement them accordingly.

Notes

CHAPTER 5

Indices and tables

- genindex
- modindex
- search