# DKPro Core™ Guide and Reference

**Written and maintained by the DKPro Core Development Community**

**Version 1.7.0**

Publication date Dezember, 2014

# Table of Contents

# Chapter 1. Introduction

Many powerful and state-of-the-art NLP tools are already freely available in the NLP research community. New and improved tools are being developed and released continuously. The tools cover the whole range of NLP-related processing tasks. DKPro Core provides UIMA components wrapping these tools so they can be used interchangeably in processing pipelines. DKPro Core builds heavily on uimaFIT which allows for rapid and easy development of UIMA-based NLP processing pipelines.

# Chapter 2. Type System

The DKPro Core type system forms the interface between all the integrated components. Components store and retrieve their data from the UIMA CAS based on this type system. The type system design is using a rather flat hierarchy and a mostly loose coupling between annotations. It is offered as a set of modules, not as a single monolithic type system.

## 2.1. Meta data

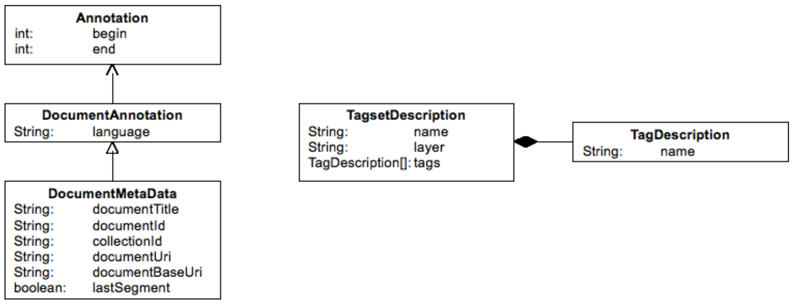

*Figure 2.1. Metadata types*

FIXME: Describe TagDescription and TagsetDescription

### 2.1.1. DocumentMetaData

The DocumentMetaData annotation stores information about a single processed document. There can only be one of these annotations per CAS. The annotation is created by readers and contains information to uniquely identify the document from which a CAS was created. Writer components use this information when determining under which filename a CAS is stored.

There are two principle ways of identifying a document:

- collection id / document id: this simple system identifies a document within a collection. The ID of the collection and the document are each simple strings without any further semantics such as e.g. a hierarchy. For this reason, this identification scheme is not well suited to preserve information about directory structures.
- document base URI / document URI: this system identifies a document using a URI. The base URI is used to derive the relative path of the document with respect to the base location from where it has been read. E.g. if the base URI is `file:/texts` and the document URI is `file:/texts/english/text1.txt`, then the relativ path of the document is `english/text1.txt`. This information is used by writers to recreate the directory structure found under the base location in the target location.

It is possible and indeed common for a writer to initialize both systems of identification. If both systems are present, most writers default to using the URI-based systems. However, most writers also allow forcing the use of the ID-based systems.

### 2.1.2. TagsetDescription and TagDescription

**Note:** Recording tagset and tag descriptions in the CAS is still a feature under development. It is not supported by all components and it is not yet well defined. Expect changes and enhancements to this feature in future versions of DKPro Core.

FIXME

## 2.2. Segmentation

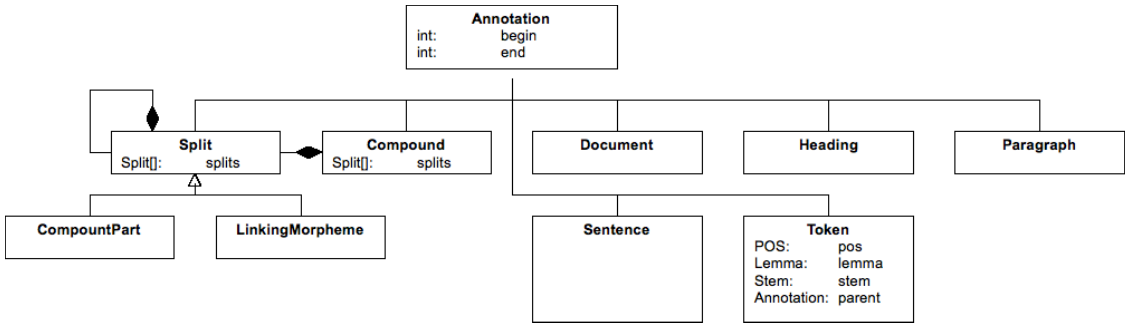

*Figure 2.2. Segmentation types*

FIXME: Describe Sentence & Token

FIXME: Describe Document, Heading, and Paragraph

FIXME: Describe Compound, Split, CompoundPart, and LinkingMorpheme

## 2.3. Morphology

## 2.4. Phonetics

## 2.5. Coreference

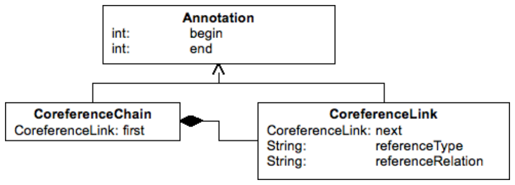

*Figure 2.3. Coreference types*

## 2.6. Syntax

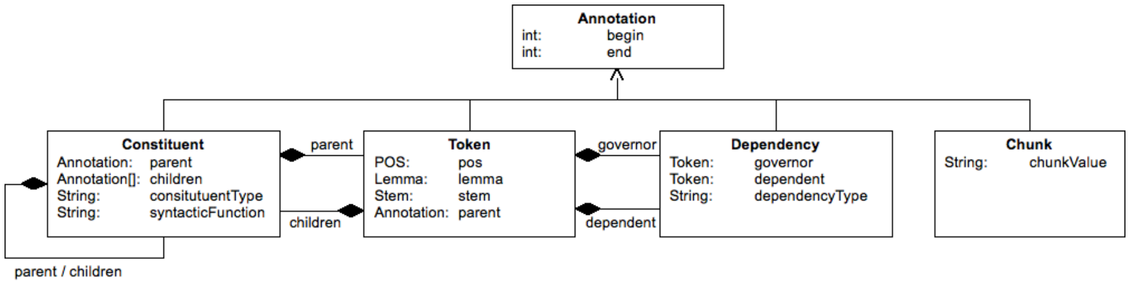

*Figure 2.4. Syntax types*

## 2.7. Semantics and Named Entities

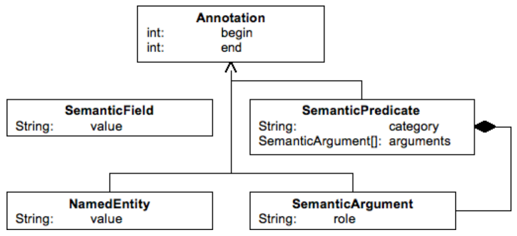

*Figure 2.5. Segmentation types*

### 2.7.1. Named entities

Named entities refer e.g. to persons, locations, organizations and so on. They often consist of multiple tokens.

### 2.7.2. Semantic fields

The SemanticField is a coarse-grained semantic category that can be attached to nouns, verbs or adjectives. Semantic field information is present e.g. in WordNet as lexicographer file names. Previously, this kind of semantic information has also been called supersenses or semantic types.

### 2.7.3. Semantic argument structure

The SemanticPredicate annotation can be attached to predicates in a sentences. Semantic predicates express events or situations and take semantic arguments expressing the participants in these events ore situations. All forms of main verbs can be annotated with a SemanticPredicate. However, there are also many nouns and adjectives that take arguments and can thus be annotated with a SemanticPredicate, e.g. event nouns, such as "suggestion" (with arguments what and by whom), or relational adjectives, such as "proud" (with arguments who and of what). The SemanticArgument annotation is attached to semantic arguments of semantic predicates. Semantic arguments are characterized by their semantic role, e.g. Agent, Experiencer, Topic. The semantic role of an argument is related to its semantic type (for communication verbs, the Agent can be a person or an organization, but typically not food). The semantic type of arguments is not yet covered by the SemanticType.

# Chapter 3. Analysis Components

This section provides an overview of the analysis componens that can be deployed at the different levels of analysis.

## 3.1. Coreference Resolution

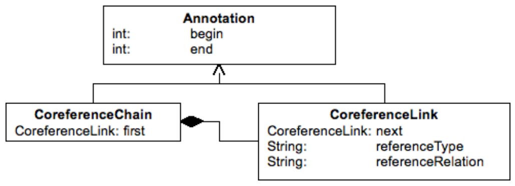

*Figure 3.1. Coreference types*

*Table 3.1. Coreference components*

| **Component** | **Comments** |
|---|---|
| StanfordCoreferenceResolver | |

## 3.2. Chunking

*Table 3.2. Chunking components*

| **Component** | **Comments** |
|---|---|
| OpenNlpChunker | |
| TreeTaggerChunker | (uses native binary) |

## 3.3. Decompounding

*Table 3.3. Decompounding components*

| **Component** | **Comments** |
|---|---|
| CompoundAnnotator | |

## 3.4. Dictionary-Based Annotation

*Table 3.4. Dictionary-based annotation components*

| **Component** | **Comments** |
|---|---|
| DictionaryAnnotator | |

| Component | Comments |
|---|---|
| SemanticFieldAnnotator | |

## 3.5. Language Identification

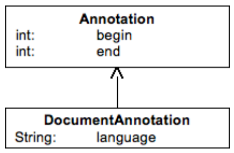

*Figure 3.2. DocumentAnnotation type*

*Table 3.5. Language identification components*

| Component | Comments |
|---|---|
| LangDetectLanguageIdentifier | Character n-grams |
| LanguageDetectorWeb1T | Token n-grams |
| TextCat | |

## 3.6. Lemmatization

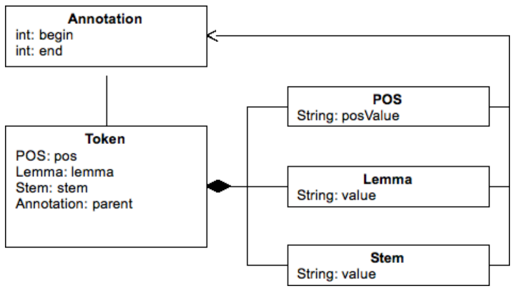

*Figure 3.3. Lemma type*

*Table 3.6. Lemmatization components*

| Component | Comments |
|---|---|
| ClearNlpLemmatizer | |
| GateLemmatizer | |
| LanguageToolLemmatizer | |
| MateLemmatizer | |
| MorphaLemmatizer | |

| Component | Comments |
|---|---|
| StanfordLemmatizer | (variant of Morpha) |
| TreeTaggerPosTagger | (uses native binary) |

## 3.7. Morphological Analysis

*Table 3.7. Morphological analysis components*

| Component | Comments |
|---|---|
| MateMorphTagger | |
| SfstAnnotator | (uses native binary) |

## 3.8. Named-Entity Recognition

*Table 3.8. Named-entity recognition components*

| Component | Comments |
|---|---|
| OpenNlpNameFinder | |
| StanfordNamedEntityRecognizer | |

## 3.9. Parsing

*Table 3.9. Parsing components*

| Component | Comments |
|---|---|
| BerkeleyParser | constituents |
| ClearNlpDependencyParser | dependencies |
| OpenNlpParser | constituents |
| MaltParser | dependencies |
| MateParser | dependencies |
| MstParser | dependencies |
| StanfordParser | constituents, dependencies (for some languages) |

## 3.10. Part-of-Speech Tagging

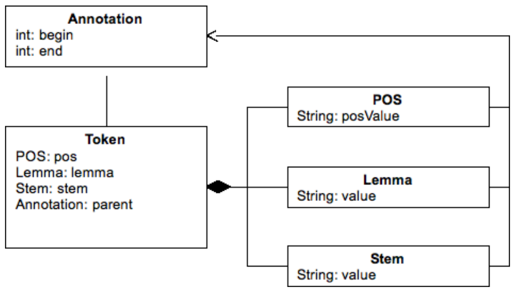

*Figure 3.4. Part-of-speech type*

*Table 3.10. Part-of-speech tagging components*

| Component | Comments |
|---|---|
| ArktweetTagger | (not in release) |
| ClearNlpPosTagger | |
| HepplePosTagger | |
| HunPosTagger | (uses native binary) |
| MatePosTagger | |
| MeCabTagger | (uses native binary) |
| OpenNlpPosTagger | |
| StanfordPosTagger | |
| TreeTaggerPosTagger | also does lemmatization (uses native binary) |

## 3.11. Segmentation

Segmenter components identify sentence boundaries and tokens. The order in which sentence splitting and tokenization are done differs between the integrated the NLP libraries. Thus, we chose to integrate both steps into a segmenter component to avoid the need to reorder the components in a pipeline when replacing one segmenter with another.

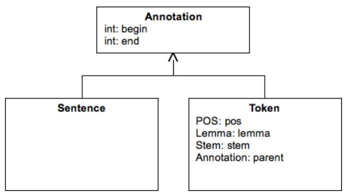

*Figure 3.5. Segmentation types*

*Table 3.11. Segmentation components*

| Component | Comments |
|---|---|
| BreakIteratorSegmenter | |

| Component | Comments |
|---|---|
| ClearNlpSegmenter | |
| LanguageToolSegmenter | |
| OpenNlpSegmenter | |
| StanfordSegmenter | |

## 3.12. Semantic Role Labelling

*Table 3.12. Semantic role labelling components*

| Component | Comments |
|---|---|
| ClearNlpSemanticRoleLabeler | |

## 3.13. Spell Checking

*Table 3.13. Spell checking components*

| Component | Comments |
|---|---|
| LanguageToolChecker | |
| NorvigSpellingCorrector | |
| SpellChecker | using Jazzy |

## 3.14. Stemming

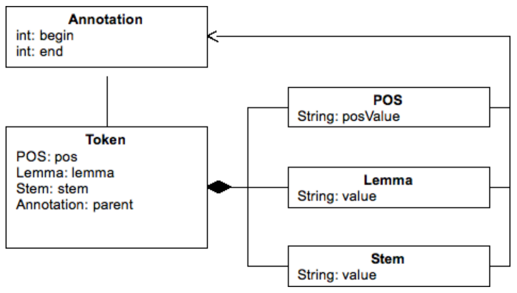

*Figure 3.6. Stem type*

*Table 3.14. Stemming components*

| Component | Comments |
|---|---|
| SnowballStemmer | |

## 3.15. Choosing components

Sometimes we get asked which parser, tagger, etc. is the best and which should be used. We currently do not make any evaluations of the integrated tools. Also, building a pipeline just of the "best" components may not actually yield the best results, because of several reasons:

- components or models may expect different tokenizations or tagsets
- components or models may be good for one domain (e.g. news) but not for another (e.g. twitter data)

We recommend that you try various combinations and stick with the one that gives the best result for *your* data.

### 3.15.1. Compatibility of Components

When selecting components for your pipeline you should make sure that the components are compatible regarding the annotation types they expect or offer.

- if a component expects an annotation type that is not provided by the preceding component, that may lead to an error or simply to no results
- if a component (e.g. a reader which adds sentence annotations) provides an annotation that is added again by a subsequent component (e.g. a segmenter), that will result in undefined behaviour of other components when you iterate over the annotation that has been added more than once.

To check whether components are compatible, you can look at the `@TypeCapability` annotation which is available in most DKPro Core components. Mind that many components can be configured with regards to which types they consume or produce, so the `@TypeCapability` should be taken as a rough indicator, not as a definitive information. It is also important to note, that the `@TypeCapability` does say anything about the tagset being consumed or produced by a component. E.g. one if a POS-tagger uses a model that produces POS-tags from the tagset X and a dependency-parser uses a model that requires POS-tags from the tagset Y, then the two models are not semantically compatible - even though the POS-tagger and dependency-parser components are compatible on the level of the type system.

### 3.15.2. Particular Settings

#### 3.15.2.1. Dictionaries and other lexical resources

If you use components in your pipeline that access dictionaries or other lexical resources, it might be essential to include a Lemmatizer in your pipeline: Many dictionaries and well-known lexical resources such as WordNet require at minimum a lemma form as a search word in order to return information about that word. For large-scale lexical resources, e.g. for Wiktionary, additional information about POS is very helpful in order to reduce the ambiguity of a given lemma form.

#### 3.15.2.2. Lemmatizing multiwords

If you use lemma information in your pipeline, you should bear in mind that multiword expressions, in particular discontinuous multiwords, might not be lemmatized as one word (or expression), but rather each multiword part might be lemmatized separately. In languages such as

German, there are verbs with separable particle such as _anfangen_ (_an_ occurs separate from _fangen_ in particular sentence constructions). Therefore - depending on your use case - you might consider postprocessing the output of the lemmatizer in order to get the true lemmas (which you might need, e.g. in order to look up information in a lexical resource).

#### 3.15.2.3. Morphologically Rich Languages

- Parsing: Morphologically rich languages (e.g. Czech, German, and Hungarian) pose a particular challenge to parser components (Tsarfaty et al. 2013).
- Morphological analysis: for languages with case syncretism (displaying forms that are ambiguous regarding their case, e.g. _Frauen_ in German can be nominative or genitive or dative or accusative), it might be better to leave case underspecified at the morphosyntactic level and leave disambiguation to the components at the syntactic level. Otherwise errors might be introduced that will then be propagated to the next pipeline component (Seeker and Kuhn 2013).

#### 3.15.2.4. Domain-specific and other non-standard data

Most components (sentence splitters, POS taggers, parsers ...) are trained on (standard) newspaper text. As a consequence, you might encounter a significant performance drop if you apply the components to domain specific or other non-standard data without adaptation.

- Tokenizing: adapting the tokenizer to your specific domain is crucial, since tokenizer errors propagate to all subsequent components in the pipeline and worsen their performance. For example, you might adapt your tokenizer to become aware of emoticons or chemical formulae in order to process social media data or text from the biochemical domain.

#### 3.15.2.5. Shallow processing and POS tagsets

While more advanced semantic processing (e.g. discourse analysis) typically depends on the output of a parser component, there might be settings where you prefer to perform shallow processing (i.e. POS tagging and chunking).

For shallow processing, it might be necessary to become familiar with the original POS tagsets of the POS taggers rather than relying on the uniform, but coarse-grained DKPro Core POS tags (because the original fine-grained POS tags carry more information).

Although many POS taggers in a given language are trained on the same POS tagset (e.g. the Penn TreeBank Tagset for English, the STTS Tagset for German), the individual POS Taggers might output variants of this tagset. You should be aware of the fact that in the DKPro Core version of the tagger, the original POS tagger output possibly has been mapped to a version that is compatible with the corresponding original tagset. (Example)

## 3.15.3. Further references

Here are some further references that might be helpful when deciding which tools to use:

- Giesbrecht, Eugenie and Evert, Stefan (2009). Part-of-speech tagging - a solved task? An evaluation of POS taggers for the Web as corpus. In I. Alegria, I. Leturia, and S. Sharoff, editors, Proceedings of the 5th Web as Corpus Workshop (WAC5), San Sebastian, Spain. PDF[1]

[1] http://purl.org/stefan.evert/PUB/GiesbrechtEvert2009_Tagging.pdf

- Reut Tsarfaty, Djamé Seddah, Sandra Kübler, and Joakim Nivre. 2013. Parsing morphologically rich languages: Introduction to the special issue. Comput. Linguist. 39, 1 (March 2013), 15-22. PDF[2]
- Wolfgang Seeker and Jonas Kuhn. 2013. Morphological and syntactic case in statistical dependency parsing. Comput. Linguist. 39, 1 (March 2013), 23-55. PDF[3]Using

## 3.16. Adding components as dependencies (Maven)

As an example, we take the OpenNlpPosTagger component. To make it available in a pipeline, we add the following dependency to our POM file:

```
<properties>
  <dkpro.core.version>1.7.0</dkpro.core.version>
</properties>
<dependencies>
  <dependency>
    <groupId>de.tudarmstadt.ukp.dkpro.core</groupId>
    <artifactId>de.tudarmstadt.ukp.dkpro.core.opennlp-asl</artifactId>
  </dependency>
</dependencies>
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>de.tudarmstadt.ukp.dkpro.core</groupId>
      <artifactId>de.tudarmstadt.ukp.dkpro.core-asl</artifactId>
      <version>${dkpro.core.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

The dependency on DKPro Core declared in the dependency management section fixes the version of all DKPro Core dependencies that are added to the POM. Hence, it is not necessary to declare the version for each dependency. When upgrading to a new DKPro Core version, it is sufficient to change the value of the `dkpro.core.version` property in the properties section.

## 3.17. Adding resources as dependencies (Maven)

Most components require resources such as models in order to operate. Since components and resources are versioned separately, it can be non-trivial to find the right version of a resource for a particular version of a component. For this reason, DKPro Core components each maintain a list of resources known to be compatible with them. This information can be accessed in a Maven POM, thus avoiding the need to manually specify the version of the models. Consequently, when you upgrade to a new version of DKPro Core, all models are automatically upgraded as well. This is usually the desired solution, although it can mean that your pipelines may produce slightly different results.

As an example, we take the OpenNlpPosTagger component. In the previous section, we have seen how to make it available in a pipeline. Now we also add the model for English.

```
<dependencies>
```

[2] https://aclweb.org/anthology/J/J13/J13-1003.pdf
[3] http://aclweb.org/anthology//J/J13/J13-1004.pdf

```
  <dependency>
    <groupId>de.tudarmstadt.ukp.dkpro.core</groupId>
    <artifactId>
      de.tudarmstadt.ukp.dkpro.core.opennlp-model-tagger-en-maxent
    </artifactId>
  </dependency>
</dependencies>
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>de.tudarmstadt.ukp.dkpro.core</groupId>
      <artifactId>de.tudarmstadt.ukp.dkpro.core.opennlp-asl</artifactId>
      <version>${dkpro.core.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

The dependency on the DKPro Core OpenNLP module declared in the dependency management section fixes the version of all known OpenNLP models. Thus, it is not necessary to declare a version on each model dependency. When upgrading to a new DKPro Core version, it is sufficient to change the value of the `dkpro.core.version` property in the properties section.

# Chapter 4. Models and Resources

This section explains how resources, such as models, are packaged, distributed, and used within DKPro Core.

## 4.1. Architecture

The architecture for resources (e.g. parser models, POS tagger models, etc.) in DKPro is still work in progress. However, there are a couple of corner points that have already been established.

- *REQ-1* - Addressable by URL: Resources must be addressable using an URL, typically a classpath URL (`classpath:/de/tudarmstadt/.../model.bin`) or a file URL (`file:///home/model.bin`). Remote URLs like HTTP should not be used and may not be supported.
- *REQ-2* - Maven compatible: Resources are packaged in JARs and can be downloaded from our Maven repositories (if the license permits).
- *REQ-3* - Document-sensitive: A component should dynamically determine at runtime which resource to use based on properties of a processed document, e.g. based on the document language. This may change from one document to the next.
- *REQ-4* - Overridable: The user should be able to override the model or provide additional information as to what specific variant of a resource should be used. E.g. if there are two resources for the language de, `de-fast` and `de-accurate`, the component could use `de-fast` per default unless the user specifies to use variante accurate or specifies a different model altogether.
- *REQ-5* - Loadable from classpath: Due to REQ-1, REQ-2, and REQ-3 models must be resolvable from the classpath.
    - `ResourceUtils.resolveLocation(String, Object, UimaContext)`
    - Resource Providers (see below)
    - `PathMatchingResourcePatternResolver`

### 4.1.1. Versioning scheme

To version our packaged models, we use a date (yyyymmdd) and a counter (x). We use a date, because often no (reliable) upstream version is available. E.g. with the Stanford NLP tools, the same model is sometimes included in different pacakges with different versions (e.g. parser models are included with the CoreNLP package and the parser package). TreeTagger models are not versioned at all. With the OpenNLP version, we are not sure if they are versioned - it seems they are just versioned for compatibility with a particular OpenNLP version (e.g. 1.5.) but have no proper version of their own. If we know it, we use the date when the model was last changed, otherwise we use the date when we first package a new model and update it when we observe a model change.

We include additional metadata with the packaged model (e.g. which tagset is used) and we sometimes want to release packaged models with new metadata, although the upstream model itself has not changed. In such cases, we increment the counter. The counter starts at 0 if a new model is incorporated.

Thus, a model version has the format "yyyymmdd.x".

### 4.1.2. Versioning scheme

## 4.2. Packaging resources as JARs

Resources needed by DKPro components (e.g. parser models or POS tagger models) are not packaged with the corresponding analysis components, but as separate JARs, one per language and model variant.

Due to license restrictions, we may not redistribute all of these resources. But, we offer Ant scripts to automatically download the resources and package them as DKPro-compatible JARs. When the license permits, we upload these to our public Maven repository.

If you need a non-redistributable resource (e.g. TreeTagger models) or just want to package the models yourself, here is how you do it.

### 4.2.1. Installing Ant in Eclipse

Our build.xml scripts require Ant 1.8.x. If you use an older Eclipse version, you may have to manually download and register a recent Ant version:

- Download the latest Ant binaries from the website and unpack them in a directory of your choice.
- Start Eclipse and go to *Window* > *Preferences* > *Ant* > *Runtime* and press *Ant Home...*.
- Select the Ant directory you just unpacked, then confirm.

### 4.2.2. Implementing the build.xml Ant script

Models are usually large and we therefore package them separately from the components that use them. Each model becomes a JAR that is uploaded to our Maven repositories and added as a dependency in the projects that use them.

Often, models are single files, e.g. serialize Java objects that represent a parser model, POS tagger model, etc. The simplest case is that these files are distributed from some website. We use an Ant script then to download the file and package it as a JAR. We defined custom Ant macros like install-model-file that make the process very convenient. The following code shows how we import the custom macros and define two targets, local-maven and separate-jars. The first just sets a property to cause install-model-file to copy the finished JAR into the local Maven repository (`~.m2/repository`).

The versioning scheme for models is "yyyymmdd.x" where "yyyymmdd" is the date of the last model change (if known) or the date of packaging and "x" is a counter unique per date starting a 0. Please refer to the versioning scheme documentation for more information.

The model building ANT script goes to `src/scripts/build.xml` with the project.

```
<project basedir="../.." default="separate-jars">
  <import>
```

```
    <url url="http://dkpro-core-asl.googlecode.com/svn/built-ant-macros/
      tags/0.6.0/ant-macros.xml"/>
  </import>

  <!--
      - Output package configuration
    -->
  <property name="outputPackage"
     value="de/tudarmstadt/ukp/dkpro/core/opennlp/lib"/>

  <target name="local-maven">
    <property name="install-artifact-mode" value="local"/>
    <antcall target="separate-jars"/>
   </target>

  <target name="remote-maven">
    <property name="install-artifact-mode" value="remote"/>
    <antcall target="separate-jars"/>
  </target>

  <target name="separate-jars">
    <mkdir dir="target/download"/>

    <!-- FILE: models-1.5/en-pos-maxent.bin - - - - - - - - - - - - - - -
      - 2012-06-16 | now        | db2cd70395b9e2e4c6b9957015a10607
      -->
    <get
      src="http://opennlp.sourceforge.net/models-1.5/en-pos-maxent.bin"
      dest="target/download/en-pos-maxent.bin"
      skipexisting="true"/>
    <install-stub-and-upstream-file
      file="target/download/en-pos-maxent.bin"
      md5="db2cd70395b9e2e4c6b9957015a10607"
      groupId="de.tudarmstadt.ukp.dkpro.core"
      artifactIdBase="de.tudarmstadt.ukp.dkpro.core.opennlp"
      upstreamVersion="20120616"
      metaDataVersion="1"
      tool="tagger"
      language="en"
      variant="maxent"
      extension="bin" >
        <metadata>
          <entry key="pos.tagset" value="ptb"/>
        </metadata>
    </install-model-file>
  </target>
</project>
```

The model file `en-pos-maxent.bin` is downloaded from the OpenNLP website and stored in a local cache directory `target/download/tagger/da-pos-maxent.bin`. From there, install-stub-and-upstream-file picks it up and packages it as two JARs, 1) a JAR containing the DKPro Core meta data and a POM referencing the second JAR, 2) a JAR containing the actual model file(s). The JAR file names derive from the artifactIdBase, tool, language, variant, upstreamVersion and metaDataVersion parameters. These parameters along with the extension parameter are also used to determine the package name and file name of the model in the JAR. They are determined as follows (mind that dots in the artifactBase turn to slashes, e.g. "de.tud" turns "de/tud":

```
    {artifactIdBase}/lib/{tool}-{language}-{variant}.{extension}
```

The following values are commonly used for *tool*:

- `token` - tokenizer
- `sentence` - sentence splitter
- `lemmatizer` - lemmatizer
- `tagger` - part-of-speech tagger
- `morphtagger` - morphological analyzer
- `ner` - named-entity recognizer
- `parser` - syntactic or dependency parser
- `coref` - coreference resolver

The values for *variant* are very tool-dependent. Typically, the variant encodes parameters that were used during the creation of a model, e.g. which machine learning algorithm was used, which parameters it had, and on which data set is has been created.

An md5 sum for the remote file must be specified to make sure we notice if the remote file changes or if the download is corrupt.

The metadata added for the models currently used to store tagset information, which is used to drive the tag-to-DKPro-UIMA-type mapping. The following values are commonly used as keys:

- `pos.tagset` - part-of-speech tagset (ptb, ctb, stts, ...)
- `dependency.tagset` - dependency relation labels, aka. syntactic functions (negra, ancora, ...)
- `constituent.tagset` - constituent labels, aka. syntactic categories (ptb, negra, ...)

### 4.2.3. Using the build.xml Ant script

For those modules where we support packaging resources as JARs, we provide an Ant script called `build.xml` which is located in the corresponding module in the SVN.

`build.xml` is a script that can be run with Apache Ant (version 1.8.x or higher) and requires an internet connection.

You can find this script in the `src/scripts` folder of the module.

Depending on the script, various build targets are supported. Three of them are particularly important: *separate-jars*, *local-maven*, and *remote-maven*:

- *separate-jars* downloads all resource from the internet, validates them against MD5 checksums and packages them as DKPro-compatible JARs. The JARs are stored to the target folder. You can easily update them to an Artifactory Maven repository. Artifactory automatically recognizes their group ID, artifact ID and version. This may not work with other Maven repositories.
- *local-maven* additionally installs the JARs into your the local Maven repository on your computer. It assumes the default location of the repository at `~/.m2/repository`. If you keep your repository in a different folder, specify it via the *alt.maven.repo.path* system property.

- *remote-maven* additionally installs the JARS into a remote Maven repository. The repository to deploy to can be controlled via the system property alt.maven.repo.url. If the remote repo also requires authentication, use the system property *alt.maven.repo.id* to configure the credentials from the settings.xml that should be used. An alternative settings file can be configured using *alt.maven.settings*.

  **Note:** This target requires that you have installed `maven-ant-tasks-2.1.3.jar`[1] in `~/.ant/lib`.

It is recommended to open the `build.xml` file in Eclipse, run the *local-maven* target, and then restart Eclipse. Upon restart, Eclipse should automatically scan your local Maven repository. Thus, the new resource JARs should be available in the search dialog when you add dependencies in the POM editor.

### 4.2.4. Example: how to package TreeTagger binaries and models

TreeTagger and its models cannot be re-distributed with DKPro Core, you need to download it yourself. For your convenience, we included an Apache Ant script called `build.xml` in the `src/scripts` folder of the TreeTagger module. This script downloads the TreeTagger binaries and models and packages them as artifacts, allowing you to simply add them as dependencies in Maven.

To run the script, you need to have Ant 1.8.x installed and configured in Eclipse. This is already the case with Eclipse 3.7.x. If you use an older Eclipse version, please see the section below on installing Ant in Eclipse.

Now to build the TreeTagger artifacts:

- Locate the Ant build script (`build.xml`) in the scripts directory (`src/scripts`) of the *de.tudarmstadt.ukp.dkpro.core.treetagger* module.
- Right-click, choose *Run As* > *External Tools Configurations*. In the *Target* tab, select local-maven, run.
- Read the license in the Ant console and - if you care - accept the license terms.
- Wait for the build process to finish.
- Restart Eclipse

To use the packaged TreeTagger resources, add them as Maven dependencies to your project (or add them to the classpath if you do not use Maven).

Note that in order to use TreeTagger you must have added at least the JAR with the TreeTagger binaries and one JAR with the model for the language you want to work with.

### 4.2.5. Which build.xml file to use? The trunk version or a release version?

For any given module supporting packaged resources, there is always the build.xml in SVN trunk and the ones in previous releases (tags folder) in SVN. Which one should you use?

[1] http://repo1.maven.org/maven2/org/apache/maven/maven-ant-tasks/2.1.3/maven-ant-tasks-2.1.3.jar

For TreeTagger, you should always use the version from SVN trunk. Here, it is least likely that the MD5 checksums are outdated and you will always get the latest and greatest version of TreeTagger.

We do not ship the `build.xml` files in any other way than via SVN.

## 4.3. Updating a model

Whenever one existing model have a new release, it is good to update the build.xml changing the:

- URL for retrieving the model (if it has changed)
- The version from the model (the day when the model was created in the yyyymmdd format)

After that, run the ant script with the *local-maven* target, add the jars to your project classpath and check if the existing unit tests work for the up to date model. If they do, then run the script again, this time with the *remote-maven* target. Then, change the versions from the models in the dependency management section from the project's pom file, commit those changes and move these new models from staging into model repository on zoidberg.

### 4.3.1. MD5 checksum check fails

Not all of the resources are properly versioned by their maintainers (in particular TreeTagger binaries and models). We observed that resources changed from one day to the next without any announcement or increase of the version number (if present at all). Thus, we validate all resources against an MD5 checksum stored in the `build.xml` file. This way, we can recognize if a remote resource has been changed. When this happens, we add a note to the `build.xml` file indicating, when we noticed the MD5 changed and update the version of the corresponding resource.

Since we do not test the build.xml files every day, you may get an MD5 checksum error when you try to package the resources yourself. If this happens, open the build.xml file with a text editor, locate the MD5 checksum that fails, update it and update the version of the corresponding resource. You can also tell us on the DKPro Core User Group and we will update the `build.xml` file.

## 4.4. Using Resource Providers

The `CasConfigurableProviderBase` class provides some support for the above requirements. The following code is taken from the `OpenNlpPosTagger` component. It shows how the POS Tagger model is addressed using a parametrized classpath URL with parameters for language and variant. The `produceResource()` method is called with the URL of the model once it has been located by `CasConfigurableProviderBase`.

```
modelProvider = new CasConfigurableStreamProviderBase<POSTagger>() {
    {
        // These are the default values
        setDefault(LOCATION, "classpath:/de/tudarmstadt/ukp/dkpro/core/" +
                "opennlp/lib/tagger-${language}-${variant}.bin");
        setDefault(VARIANT, "maxent");

        // These are parameters the user may have set on the component,
        // they may be null
        setOverride(LOCATION, modelLocation);
        setOverride(LANGUAGE, language);
        setOverride(VARIANT, variant);
    }
    @Override
    protected POSTagger produceResource(InputStream aStream)
      throws Exception
    {
        POSModel model = new POSModel(aStream);
        return new POSTaggerME(model);
    }
};

// Here the language is picked up from the CAS.
modelProvider.configure(cas);

// Here we get the tagger according to the language, variant and location
// chosen. A new instance is only created if necessary (e.g. if the
// current CAS has a different language than the previous).
POSTagger tagger = modelProvider.getResource();
```

**Note:** This is an illustrative code example. See `OpenNlpPosTagger`[2] for the complete code.

### 4.4.1. Mapping Providers

The DKPro type system design provides two levels of abstraction on most annotations:

- a generic annotation type, e.g. POS (part of speech) with a feature value containing the original tag produced by an analysis component, e.g. TreeTagger
- a set of high-level types for very common categories, e.g. N (noun), V (verb), etc.

DKPro maintains mappings for commonly used tagsets, e.g. in the module *de.tudarmstadt.ukp.dkpro.core.api.lexmorph-asl*. They are named:

```
{language}-{layer}.map
```

The following values are commonly used for *layer*:

- `pos` - part-of-speech tag mapping

Mapping providers are a convenient way of fetching a mapping between the original tag value and the high-level types.

```
// General setup of the mapping provider in initialize()
mappingProvider = new MappingProvider();
mappingProvider.setDefault(MappingProvider.LOCATION,
    "classpath:/de/tudarmstadt/ukp/dkpro/" +
    "core/api/lexmorph/tagset/${language}-pos.map");
mappingProvider.setDefault(MappingProvider.BASE_TYPE, POS.class.getName());
mappingProvider.setOverride(MappingProvider.LOCATION, mappingLocation);
mappingProvider.setOverride(MappingProvider.LANGUAGE, language);

// Document-specific configuration in process()
mappingProvider.configure(cas);

// Resolve an original tag value to a high-level type
Type posTag = mappingProvider.getTagType(tags[i]);
```

[2] http://code.google.com/p/dkpro-core-asl/source/browse/de.tudarmstadt.ukp.dkpro.core-asl/trunk/de.tudarmstadt.ukp.dkpro.core.opennlp-asl/src/main/java/de/tudarmstadt/ukp/dkpro/core/opennlp/OpenNlpPosTagger.java

```
POS posAnno = (POS) cas.createAnnotation(posTag, t.getBegin(), t.getEnd());
```

**Note:** This is an illustrative code example. See `OpenNlpPosTagger`[3] for the complete code.

## 4.4.2. Language-dependent default variants

It is possible a different default variant needs to be used depending on the language. This can be configured by placing a properties file in the classpath and setting its location using `setDefaultVariantsLocation(String)`. The key in the properties is the language and the value is used a default variant. These file should always reside in the libsub-package of a component and use the naming convention:

```
{tool}-default-variants.map
```

```
// General setup of the mapping provider in initialize()
mappingProvider.setDefaultVariantsLocation("de/tudarmstadt/ukp/dkpro/" +
    "core/stanfordnlp/lib/tagger-default-variants.map");
```

**Note:** This is an illustrative code example. See `StanfordPosTagger`[4] for the complete code.

The `tagger-default-variants.map` is a Java properties file which defines for each language which variant should be assumed as default.

```
ar=fast
de=fast
en=bidirectional-distsim-wsj-0-18
zh=default
```

[3] http://code.google.com/p/dkpro-core-asl/source/browse/de.tudarmstadt.ukp.dkpro.core-asl/trunk/de.tudarmstadt.ukp.dkpro.core.opennlp-asl/src/main/java/de/tudarmstadt/ukp/dkpro/core/opennlp/OpenNlpPosTagger.java

[4] http://code.google.com/p/dkpro-core-gpl/source/browse/de.tudarmstadt.ukp.dkpro.core-gpl/trunk/de.tudarmstadt.ukp.dkpro.core.stanfordnlp/src/main/java/de/tudarmstadt/ukp/dkpro/core/stanfordnlp/StanfordPosTagger.java

# Chapter 5. I/O

This section gives an overview on the I/O components. The components are organized into one module per file type. These modules contain typically one reader and/or writer component.

All readers initialize the CAS with a DocumentMetaData annotation.

Most readers and writers do not support all features of the respective formats. Additionally, readers and writers may onle support a specific variant of a format.

## 5.1. Common parameters

DKPro Core aims to provide a consistent API for reading and writing annotated data. Most of our readers are resource readers (RR) or file writers (FW) and support a common set of parameters which are explained below.

*Table 5.1. Resource reader parameters*

| **Parameter** | **Description** | **Default** |
|---|---|---|
| PARAM_SOURCE_LOCATION | Location to read from. | optional |
| PARAM_PATTERNS | Include/exclude patterns. | optional |
| PARAM_USE_DEFAULT_EXCLUDES | Enable default excludes for versioning systems like Subversion, git, etc. | true |
| PARAM_INCLUDE_HIDDEN | Include hidden files | false |
| PARAM_LANGUAGE | Two letter ISO code | optional |

Either PARAM_SOURCE_LOCATION or PARAM_PATTERNS or both must be set. Examples:

- Read all files in the folder `files/texts`
  - PARAM_LOCATON: "files/texts"
- Recursively read all `.txt` files in the folder `files/texts`:
  - Pattern embedded in the location: PARAM_LOCATION: "files/texts/**/*.txt"
  - Pattern separate: PARAM_LOCATION: "files/texts", PARAM_PATTERNS: "*.txt"
- Excluding some files:
  - Pattern separate: PARAM_LOCATION: "files/texts", PARAM_PATTERNS: String[] {"*.txt", "[-]broken*.txt"}
- Read from a ZIP archive:
  - PARAM_LOCATION: "jar:file:archive.zip!texts/**/*.txt"
- Read from the classpath:

    - PARAM_LOCATION: "classpath*:texts/*.txt"

*Table 5.2. File writer parameters*

| Parameter | Description | Default |
|---|---|---|
| PARAM_TARGET_LOCATION | Location to write to. | mandatory |
| PARAM_COMPRESSION | Compression algorithm to use when writing output. File suffix automatically added depending on algorithm. Supported are: NONE, GZIP, BZIP2, and XZ (see class `CompressionMethod`). | NONE |
| PARAM_STRIP_EXTENSION | Whether to remove the original file extension when writing. E.g. with the XmiWriter without extension stripping, an input file `MyText.txt` would be written as `MyText.txt.xmi` - with stripping it would be `MyText.xmi`. | false |
| PARAM_USE_DOCUMENT_ID | Use the document ID as the file name, even if an original file name is present in the document URI. | false |
| PARAM_ESCAPE_DOCUMENT_ID | Escape the document ID in case it contains characters that are not valid in a filename. | false |

Most file writers write multiple files, so PARAM_TARGET_LOCATION is treated as a directory name. A few only write a single file (e.g. NegraExportWriter), in which case the parameter is treated as the file name. Instead of writing to a directory, it is possible to write to a ZIP archive:

- Write to a ZIP archive
    - PARAM_TARGET_LOCATION: "jar:file:archive.zip"
    - PARAM_TARGET_LOCATION: "jar:file:archive.zip!folder/within/zip"

## 5.2. ACL anthology

*Table 5.3. Components*

| Component | Comments |
|---|---|
| AclAnthologyReader | |

Known corpora in this format:

- ACL Anthology Reference Corpus (ACL ARC)[1]

[1] http://acl-arc.comp.nus.edu.sg

## 5.3. Binary CAS

The CAS is the native data model used by UIMA. There are various ways of saving CAS data, using XMI, XCAS, or binary formats. This module supports the binary formats.

*Table 5.4. Components*

| Component | Comments |
|---|---|
| BinaryCasReader (RR) | |
| BinaryCasWriter (FW) | |
| SerializedCasReader (RR) | |
| SerializedCasWriter (FW) | |

See also:

- Compressed Binary CASes[2]

## 5.4. Bliki (Wikipedia)

Access the online Wikipedia and extract its contents using the Bliki engine.

*Table 5.5. Components*

| Component | Comments |
|---|---|
| BlikiWikipediaReader | |

See also:

- Java Wikipedia API (Bliki engine)[3]

## 5.5. British National Corpus

*Table 5.6. Components*

| Component | Comments |
|---|---|
| BncReader (RR) | BNC XML format |

Known corpora in this format:

- British National Corpus[4]

## 5.6. CoNLL Shared Task Data Formats

The CoNLL shared tasks use different tabular, tab-separated formats. Almost every year a CoNLL shared task was held, a new format was defined. Due to their simplicity, the CoNLL formats are

[2] http://uima.apache.org/d/uimaj-2.4.2/references.html#ugr.ref.compress
[3] http://code.google.com/p/gwtwiki/
[4] http://www.natcorp.ox.ac.uk

quite popular. Many corpora are provided in the CoNLL-X (2006) format and many tools can natively read or write this format.

*Table 5.7. Components*

| **Component** | **Comments** |
|---|---|
| Conll2000Reader (RR) | Chunking task |
| Conll2000Writer | Chunking task |
| Conll2006Reader (RR) | Dependency parsing task |
| Conll2006Writer | Dependency parsing task |
| Conll2009Reader (RR) | Semantic dependencies |
| Conll2009Writer | Semantic dependencies |
| Conll2012Reader | Syntactic parsing & coreference |
| Conll2012Writer | Syntactic parsing & coreference |

Known corpora in this format:

- CoNLL 2000 Chunking Corpus[5] - English (CoNLL 2000 format)
- CoNLL-X Shared Task free data[6] - Danish, Dutch, Portuguese, and Swedish (CoNLL 2006 format)
- Copenhagen Dependency Treebanks[7] - Danish (CoNLL 2006 format)
- FinnTreeBank[8] - Finnish (CoNLL 2006 format, in recent versions with additional pseudo-XML metadata)
- Floresta Sintá(c)tica (Bosque-CoNLL)[9] - Portuguese (CoNLL 2006 format)
- SETimes.HR corpus and dependency treebank of Croatian[10] - Croatian (CONLL 2006 format)
- Sk#adnica zale#no#ciowa[11] - Polish (CoNLL 2006 format)
- Slovene Dependency Treebank[12] - Slovene (CoNLL 2006 format)
- Swedish Treebank[13] - Swedish (CoNLL 2006 format)
- Talbanken05[14] - Swedish (CoNLL 2006 format)
- Uppsala Persian Dependency Treebank[15] - Persian (Farsi) (CoNLL 2006 format)

[5] http://nltk.org/nltk_data/
[6] http://ilk.uvt.nl/conll/free_data.html
[7] https://code.google.com/p/copenhagen-dependency-treebank/
[8] http://www.ling.helsinki.fi/kieliteknologia/tutkimus/treebank/
[9] http://www.linguateca.pt/floresta/CoNLL-X
[10] http://nlp.ffzg.hr/resources/corpora/setimes-hr/
[11] http://zil.ipipan.waw.pl/Sk%C5%82adnica
[12] http://nl.ijs.si/sdt/
[13] http://stp.lingfil.uu.se/%7Enivre/swedish_treebank/
[14] http://stp.lingfil.uu.se/%7Enivre/research/Talbanken05.html
[15] http://stp.lingfil.uu.se/%7Emojgan/UPDT.html

## 5.7. CLARIN WebLicht TCF

*Table 5.8. Components*

| Component | Comments |
|---|---|
| TcfReader (RR) | |
| TcfWriter | |

## 5.8. HTML

*Table 5.9. Components*

| Component | Comments |
|---|---|
| HtmlReader | |

## 5.9. IMS Open Corpus Workbench

The IMS Open Corpus Workbench is a linguistic search engine. It uses a tab-separated format with limited markup (e.g. for sentences, documents, but not recursive structures like parse-trees). If a local installion of the corpus workbench is available, it can be used by this module to immediately generate the corpus workbench index format. Search is not supported by this module.

*Table 5.10. Components*

| Component | Comments |
|---|---|
| ImsCwbReader (RR) | |
| ImsCwbWriter | |

See also:

- IMS Open Corpus Workbench[16]

Known corpora in this format:

- WaCky - The Web-As-Corpus Kool Yinitiative[17] - corpora crawled from the world wide web in several different languages (DeWaC, UkWaC, ItWaC, etc.)

## 5.10. JDBC Database

Access a JDBC database using an SQL query.

*Table 5.11. Components*

| Component | Comments |
|---|---|
| JdbcReader | |

[16] http://cwb.sourceforge.net
[17] http://wacky.sslmit.unibo.it

## 5.11. JWPL (Wikipedia)

Access an offline Wikipedia database dump created using JWPL.

*Table 5.12. Components*

| Component | Comments |
|---|---|
| WikipediaArticleInfoReader | |
| WikipediaArticleReader | |
| WikipediaDiscussionReader | |
| WikipediaLinkReader | |
| WikipediaPageReader | |
| WikipediaQueryReader | |
| WikipediaRevisionPairReader | |
| WikipediaRevisionReader | |
| WikipediaTemplateFilteredArticleReader | |

See also: JWPL and the Wikipedia Revision Toolkit[18]

## 5.12. NEGRA Export Format

*Table 5.13. Components*

| Component | Comments |
|---|---|
| NegraExportReader | Supports version 3 and 4 |

See also:

- Thorsten Brants, 1997, NeGra Export Format for Annotated Corpora (Version 3)[19]

Known corpora in this format:

- Floresta Sintá(c)tica (Bosque)[20] - Portuguese
- NeGra[21] - German
- TIGER[22] (until version 2.1) - German
- TüBa D/Z[23] - German

[18] http://code.google.com/p/jwpl/
[19] http://www.coli.uni-saarland.de/%7Ethorsten/publications/Brants-CLAUS98.pdf
[20] http://www.linguateca.pt/floresta/corpus.html
[21] http://www.coli.uni-saarland.de/projects/sfb378/negra-corpus/
[22] http://www.ims.uni-stuttgart.de/forschung/ressourcen/korpora/tiger.html
[23] http://www.sfs.uni-tuebingen.de/de/ascl/ressourcen/corpora/tueba-dz.html

## 5.13. PDF

*Table 5.14. Components*

| Component | Comments |
|---|---|
| PdfReader (RR) | |

## 5.14. Penn Treebank

*Table 5.15. Components*

| Component | Comments |
|---|---|
| PennTreebankChunkedReader | |
| PennTreebankCombinedReader | |
| PennTreebankCombinedWriter | |

Known corpora in this format:

- Floresta Sintá(c)tica (Bosque)[24] - Portuguese

## 5.15. TCF

The TCF (Text Corpus Format) was created in the context of the CLARIN project. It is mainly used to exchange data between the different web-services that are part of the WebLicht platform.

*Table 5.16. Components*

| Component | Comments |
|---|---|
| TcfReader (RR) | |
| TcfWriter (FW) | |

Known corpora in this format:

- None

## 5.16. TEI

*Table 5.17. Components*

| Component | Comments |
|---|---|
| TeiReader (RR) | |

[24] http://www.linguateca.pt/floresta/corpus.html

Known corpora in this format:

- Brown Corpus (TEI XML Version)[25]
- Digitale Bibliothek bei TextGrid[26]

## 5.17. Text

This module supports just simple plain text.

*Table 5.18. Components*

| Component | Comments |
|---|---|
| StringReader | Read text from a Java string |
| TextReader (RR) | Read text from files |
| TextWriter (FW) | Write text to files |

## 5.18. TIGER XML

The TIGER XML format[27] was created for encoding syntactic constituency structures in the German TIGER corpus. It has since been used for many other corpora as well. TIGERSearch[28] is a linguistic search engine specifically targetting this format. The format has later been extended[29] to also support semantic frame annotations.

*Table 5.19. Components*

| Component | Comments |
|---|---|
| TigerXmlReader (RR) | Read TIGER XML format from files |
| TigerXmlWriter (FW) | Write TIGER XML format to files |

Known corpora in this format:

- Floresta Sintá(c)tica (Bosque)[30] - Portuguese
- Semeval-2 Task 10[31] - (extended format)
- Sk#adnica frazowa[32] - Polish
- Swedish Treebank[33] - Swedish

  Talbanken05[34] - Swedish

[25] http://nltk.org/nltk_data/
[26] http://www.textgrid.de/Digitale-Bibliothek
[27] http://www.ims.uni-stuttgart.de/forschung/ressourcen/werkzeuge/TIGERSearch/doc/html/TigerXML.html
[28] http://www.ims.uni-stuttgart.de/forschung/ressourcen/werkzeuge/tigersearch.html
[29] http://www.lrec-conf.org/proceedings/lrec2004/pdf/202.pdf
[30] http://www.linguateca.pt/floresta/corpus.html
[31] http://semeval2.fbk.eu/semeval2.php
[32] http://zil.ipipan.waw.pl/Sk%C5%82adnica
[33] http://stp.lingfil.uu.se/%7Enivre/swedish_treebank/
[34] http://stp.lingfil.uu.se/%7Enivre/research/Talbanken05.html

- TIGER[35] - German

## 5.19. TüPP-D/Z

TüPP D/Z is a collection of articles from the German newspaper taz (die tageszeitung) annotated and encoded in a XML format.

*Table 5.20. Components*

| Component | Comments |
|---|---|
| TueppReader (RR) | Read TüPP-D/Z XML format from files |

Known corpora in this format:

- TüPP-D/Z[36] - German

## 5.20. Web1T

The Web1T n-gram corpus is a huge collection of n-grams collected from the internet. The jweb1t library allows to access this corpus efficiently. This module provides support for the file format used by the Web1T n-gram corpus and allows to conveniently created jweb1t indexes.

*Table 5.21. Components*

| Component | Comments |
|---|---|
| Web1TFormatWriter | |

See also:

- jweb1t - Java API for text n-gram frequency data in Web1T format[37]
- Web1T n-gram corpus[38]

## 5.21. XMI

*Table 5.22. Components*

| Component | Comments |
|---|---|
| XmiReader (RR) | |
| XmiWriter (FW) | |

## 5.22. XML

[35] http://www.ims.uni-stuttgart.de/forschung/ressourcen/korpora/tiger.html
[36] http://www.sfs.uni-tuebingen.de/de/ascl/ressourcen/corpora/tuepp-dz.html
[37] http://code.google.com/p/jweb1t/
[38] http://catalog.ldc.upenn.edu/LDC2006T13

*Table 5.23. Components*

| **Component** | **Comments** |
|---|---|
| XmlReader | |
| XmlReaderText | |
| XmlReaderXPath | |
| XmlWriterInline | |

## 5.23. TGrep

TGrep and TGrep2 are a tools to search over syntactic parse trees represented as bracketed structures. This module supports in particular TGrep2 and allows to conveniently generate TGrep2 indexes which can then be searched. Search is not supported by this module.

*Table 5.24. Components*

| **Component** | **Comments** |
|---|---|
| TGrepWriter | Supports tgrep2 (uses native binary) |

See also:

- TGrep2[39]

[39] http://tedlab.mit.edu/%7Edr/Tgrep2/

# Chapter 6. Testing

The testing module offers a convenient way to create unit tests for UIMA components.

The AssertAnnotations class offers various static methods to test if a component has properly created annotations of a certain kind. There are methods to test almost every kind of annotation supported by DKPro Core, e.g.:

- assertToken
- assertSentence
- assertPOS
- assertLemma
- assertMorpheme
- assertStem
- assertNamedEntity
- assertConstituents
- assertChunks
- assertDependencies
- assertPennTree
- assertSemanticPredicates
- assertSemanticField
- assertCoreference
- assertTagset
- assertTagsetMapping

A typical unit test class has consists of two parts

1. the test cases
2. a `runTest` method - which sets up the pipeline required by the test and then calls `TestRunner.runTest()`.

In the following example, mind that the text must be provided with spaces separating the tokens (thus there must be a space before the full stop at the end of the sentence) and with newline characters (\n) separating the sentences:

```
@Test
public void testEnglish() throws Exception
{
  JCas jcas = runTest("en", null, "This is a test .");

  String[] constituentMapped =
    { "NP 0,4", "NP 8,14", "ROOT 0,16", "S 0,16", "VP 5,14" };

  String[] constituentOriginal =
    { "NP 0,4", "NP 8,14", "ROOT 0,16", "S 0,16","VP 5,14" };

  String pennTree = "(ROOT (S (NP (DT This)) (VP (VBZ is) " +
    "(NP (DT a) (NN test))) (. .)))";

  String[] constituentTags =
    { "ADJP", "ADV", "ADVP", "AUX", "CONJP", "FRAG", "INTJ",
      "LST", "NAC", "NEG", "NP", "NX", "O", "PP", "PRN", "PRT",
      "QP", "S", "SBAR", "SQ", "TYPO", "UCP", "UH", "VP",
      "WHADJP", "WHADVP", "WHNP", "WHPP", "X" };

  String[] unmappedConst =
    { "ADV", "AUX", "NEG", "O", "TYPO", "UH" };

  AssertAnnotations.assertPennTree(
    pennTree, selectSingle(jcas, PennTree.class));
  AssertAnnotations.assertConstituents(
    constituentMapped, constituentOriginal,
    select(jcas, Constituent.class));
  AssertAnnotations.assertTagset(
    Constituent.class, "ptb", constituentTags, jcas);
  AssertAnnotations.assertTagsetMapping(
    Constituent.class, "ptb", unmappedConst, jcas);
}

private JCas runTest(String aLanguage, String aVariant, String aDocument)
  throws Exception
{
  AnalysisEngineDescription parser = createEngineDescription(
    OpenNlpParser.class,
    OpenNlpParser.PARAM_VARIANT, aVariant,
    OpenNlpParser.PARAM_PRINT_TAGSET, true,
    OpenNlpParser.PARAM_WRITE_PENN_TREE, true);
  return TestRunner.runTest(parser, aLanguage, aDocument);
}
```

Test cases for segmenter components should not make use of the `TestRunner` class, which already performs tokenization and sentence splitting internally.