Technical Report

LIPN UIMA Platform 0.1.*
User/developer guide

Erwan Moreau
`Erwan.Moreau@lipn.univ-paris13.fr`

October 2010
last update: May 2011

Contents

1 Introduction

This document describes the "*LIPN UIMA Platform*" software, developed in 2010 by the author at LIPN[1]. This software mainly consists in a UIMA-based evolutive platform devoted to corpora annotation. It is divided into two modules (*lipn-uima-core* and *lipn-nlptools-utils*) and is distributed as an archive containing a environment providing a few tools (scripts, CPE descriptors, examples etc.). Currently the software is located[2] at:

`http://www-lipn.univ-paris13.fr/~moreau/uima/lipn-uima-core.tgz`

Remark. Two modules are provided: UIMA-dependent packages (annotators etc.) belong to *lipn-uima-core*, whereas all other packages belong to *lipn-nlptools-utils*[3]. The latter is contained in the former archive as a JAR package, but can also be found as a standalone archive at the same address.

1.1 Preamble: what this guide is (and is not)

The *LIPN UIMA Platform* components are quite complex. This document is intended to help the user understand what are their goals, how they can be used, and the reasons of various design/implementation choices. It is mainly thought as a "user guide", not as an exhaustive documentation: for precise documentation on some class/method/parameter, **please refer to the Javadoc API**. This document is **neither a UIMA tutorial**: to obtain information about this framework please see the UIMA official site (see 2.1). It is rather intended to propose a global point of view on the *LIPN UIMA Platform* components, with explanations about how these components work together. It also details the conventions/guidelines which have been used, and that the user should preferably also follow when coding new components using/relating/belonging to *LIPN UIMA Platform*. Thus this guide is intended to different kind of readers/users:

- If you[4] only want to run the components from a Java program or from a command line, see 2.6.4.
- If you only want to use the UIMA independent packages, see 6 and API.
- If you have at least basic knowledge about UIMA (main audience) and you want to be able to use the components in an unconstrained UIMA environment, you should preferably have a look at all sections, but you can esily skip details/implementation parts (in particular section 6). Do not forget to read details about the annotators parameters in the descriptors files.
- If you plan to develop new UIMA components using *LIPN UIMA Platform* ones the same applies, but you should probably skip less parts.
- If you want to correct/improve the components that this document describes, I humbly suggest you read the whole guide (and I will probably hear about you sooner or later!)

The structure of this document is perhaps a bit confused, because there are many ways to tackle UIMA software in general and this one in particular. This user guide should hopefully make clear most aspects of the components, but feel free to contact the author if you think something is wrong/missing/should be improved[5].

1.2 Main objectives and content

These software components were created in the following goals:

[1]This work was funded by OSEO the Quaero project. The preliminary steps of this work were implemented by Sondes Bannour. The author also thanks Fabien Poulard, PhD student at LINA, who helped him a lot to understand UIMA mechanics during this work.
[2]If the link does not exist anymore, please contact the author.
[3]Actually the latter still depends on the *uima-core* package, but only because the UIMA `Logger` class is used (this technical point is not significative).
[4]In these sentences "you want to ..." can be replaced with "your PhD advisor asked you to ..." if applicable.
[5]The author also apologizes for the very poor English in this document, sorry!

- Update/improve existing software tools used at LIPN, mainly to make them more flexible (in the way to use them) and more robust. These tools are (more or less) those previously used in the *Ogmios/Alvis* platform (TreeTagger and YaTeA essentially). In this goal, tools are provided to use the corresponding components as "black boxes", allowing the user not to care about all the UIMA stuff.
- Build the core of a UIMA platform devoted to LIPN research tasks (semantic annotation), in order to initiate a long-term approach towards a better organization of software developped in the lab. This is the main reason why UIMA has been chosen. In particular, we hope that future pieces of code will be more uniform, easier to combine with each other and also easier to maintain. Thus choices are made to make this platform as evolutive as possible.
- Propose a new approach in the design of an NLP platform, with two main ideas:
  - Using a very **generic *Type System*** (see 4) in order to promote the modularity of components in a processing pipe. Following UIMA's principles about making data transmission easier between different components not knowing about each other (or not much).
  - Making possible and as convenient as possible the use of ***concurrent annotations*** (see 4.2). This point requires that components are designed in a way which makes it possible, and also suitable tools for a user.

As a side effect, this library also fulfill several other needs:

- It provides tools for calling an *external program* (see 6.1) from Java in a quite robust, safe and convenient way. The library contains both a UIMA independent component and a UIMA generic annotator (see 3).
- It provides tools for re-aligning and/or converting structured data (mainly intended for NLP standard annotations format) in a uniform and modular way (see 6.2). A UIMA independent package is provided, which can also easily be used inside UIMA.

Most of the UIMA annotators provided in this library are **wrappers** for previously existing tools. This choice is done for time and cost reasons: it is clearly more reasonable to use these existing annotation tools, among which some are LIPN expertise, rather than re-coding them from scratch in a "pure" UIMA environment. That is why these first components will actually be wrappers for what we call "external programs", in the sense that they run as a black box (in the UIMA platform viewpoint). This is clearly not the ideal situation: UIMA provides a very complete, convenient and safe environment for connecting annotators together in some complex system (error handling, logging, resources management, deployment, etc.), so calling external programs introduces a lot of possible flaws (portability problems, I/O errors, uncontrolled use of resources, concurrency errors, etc.). This a critical point because a complex task can involve quite a lot of different components, so it can be very vulnerable to any problem in the chain: clearly one does not want that the complete system fails because a single document makes a single component crash due for example to some minor charset encoding problem.

Thus a large part of this work has been devoted to deal with these drawbacks with care. This is actually the reason why the code has been divided into two distinct modules: *lipn-uima-core* contains the real UIMA components, while *lipn-nlptools-utils* is devoted to this kind of problems and consists in several utility packages which are (almost) UIMA independent (see 1).

One of the most important consequences about this recycling strategy is that the LIPN components are **not portable** (at least they can not be considered portable). Due to the software used as external programs itself and/or the constraints to use it from inside the platform, all wrapper components are Linux (or Unix-like) only! The following (main) components are provided (most of them handle only English and/or French as language):

- The **TagEN** named entities recognizer (for French and English), created a few years ago at LIPN by Thierry Poibeau and Jean-François Berroyer. An updated version has been integrated to the platform. Warning: resources used by this tagger are a bit old.

- The **TreeTagger** Part-of-speech tagger[6], by Helmut Schmid, including tokenizers, lemmatizers and chunkers for a lot of languages (English, French, German, Spanish, Italian, Dutch, Greek, Bulgarian).
- The **"LIA tools"** are a set of taggers for French and English under GPL (General Public Licence) by Frederic Bechet. It includes a tokenizer, a POS tagger, a lemmatizer and a NE recognizer (French only).
- The **YaTeA** term extractor for French and English, by Thierry Hamon and Sophie Aubin.

Some more packages provide tools for the common tasks of LIPN UIMA annotators. These are not only provided for convenience, but also to ensure that all annotators follow some general guidelines, in order to strengthen consistency and compatibility among components.

1.3 History and current state

This library has been written from June to December 2010, using UIMA 2.3.0 and Sun/Oracle Java 6 SDK. Since Apache UIMA is a dynamic fast-growing project[7], it is possible that future UIMA versions make some parts of this work obsolete. Future maintainers are encouraged to follow UIMA developments and make improvements/corrections appropriately!

The current version 0.1.* proposes the basic bricks of this UIMA platform. A lot of design/implementation choices have been made: most of them (or at least the major ones) have been carefully thought before adoption. In general, these choices are meant to provide an approach that future component developers can simply follow, unless there is an important reason not do so.

Nevertheless, as of version 0.1.* the *LIPN UIMA Platform* library should be considered as a beta version: there are probably still bugs, and maybe even some crucial parts will have to be modified in the future. In particular, it is possible that future versions do not satisfy backward compatibility (even if compatibility should always be prefered if possible). Indeed there is not enough retrospect to consider this library as an achieved product. This is one of the reasons why this document will try to detail design and implementation choices whenever possible, in order to make future evolution easier.

About future work, see also 8.

2 Getting started

2.1 Getting help with UIMA

This document does not address UIMA general documentation. The reader who wants to learn about UIMA can refer to the UIMA official site[8], which contains documentation and tutorial(s). There are also mailing lists to which users can ask their questions: the official one of course[9], but also the UIMA-FR one. UIMA-FR[10] also provides some good tutorials and explanations for beginners (in French).

2.2 The *LIPN UIMA Platform* environment

Basically UIMA is only a set of Java packages, which can simply be used by adding the corresponding JAR archives in the `CLASSPATH` variable. However "installing" UIMA (uncompressing the archive and possibly setting Eclipse

[6]There is also a UIMA TreeTagger wrapper proposed by the LINA: `http://www.lina.univ-nantes.fr/-Composants-UIMA-.html`
[7]It became an Apache *Top-Level Project* in May 2010.
[8]`http://uima.apache.org`
[9]See `http://uima.apache.org/mail-lists.html`
[10]`http://uima-fr.org`

plugins) makes life easier because the `apache-uima` directory contains a lot of tools, examples, descriptors (and using UIMA Eclipse plugins is more convenient). The *LIPN UIMA Platform* has been packaged in the same way: the essential packages are contained in two JARs (corresponding to modules *lipn-uima-core* and *lipn-nlptools-utils*), but a "user friendly" environment is provided: it contains API documentation, scripts, examples, descriptors. Additionally this environment is an "src" release, that is to say it contains the Java sources and the way to compile them easily (see 2.7.3). For all these reasons we recommend using this environment[11].

A module structure contains the following directories:

- `src` contains the Java sources
- `lib` contains the required libraries (generated after compiling with Maven)
- `bin` contains the binary classes after compilation (generated after compiling)
- `target` contains the final JAR archive of the module (generated after compiling with Maven)
- `doc` contains the Javadoc generated API.

Additionaly the following directories are provided in *lipn-uima-core*:

- `desc` contains the descriptor files
- `resources` contains resources files (needed in the `CLASSPATH` and included in the final JAR)
- `conf` contains configuration files (like the logging parameters file)
- `install-scripts` contains scripts to install external programs (see 2.3)
- `tests/data` contains example data (see 2.5)
- `tests/CPEs` contains example CPEs (see 2.5)
- `output` contains the resulting XMI files after running a CPE contained in `tests/CPEs`

We tried to make this environment as easy to use as possible. In particular relative path are used, since an absolute path is more likely to be wrong in general (see also 7.0.3). There is no need to set an environment variable indicating the path to project root (contrary to the UIMA SDK environment), but consequently some path problems may happen, in particular when using the provided examples in the `tests` directory from another location: whenever possible, call a test from the root project. If this is not possible/convenient, be sure to include the right parameters concerning path and/or `CLASSPATH` (the `lipn-run-cpe.sh` script handles some options to do so). Anyway, be prepared to face such problems using UIMA!

2.3 Installing the external programs

2.3.1 Installing TreeTagger, Unitex, TagEN, LIA Tagg and LIA NE

As explained in the introduction, the current library needs some external software to run since most components are wrappers. Depending on the user needs, it is not always necessary to install all programs:

- TreeTagger[12] (no version number in TreeTagger: use a recent version, since the authors sometimes improve the soft/correct bugs)
- Unitex[13] version $\geq$ 2.0 (needed for TagEN). Tested with 2.0 and 2.1beta

[11] And you can of course release a program using *LIPN UIMA Platform* by simply providing the JAR files (or, better, specifying the dependencies in a Maven project), as usual.
[12] `http://www.ims.uni-stuttgart.de/projekte/corplex/TreeTagger`
[13] `http://igm.univ-mlv.fr/~unitex`

- TagEN[14] version ≥ 2.0
- LIA Tagg[15] version ≥ 1.1 (needed for LIA NE). Tested with 1.1.
- LIA NE[16] version ≥ 2.2. Tested with 2.2.
- YaTeA[17] version 0.5.

A few scripts are provided to help installing these tools, but be careful: they are not robust at all! They should not destroy anything (hopefully!), but they will likely fail if the version changes or for any unexpected configuration. These scripts actually only download the program, execute any needed compilation step as indicated in each program documentation, and finally run a simple test. In other words, each script executes the normal procedure you would have to do by hand. If it does not work run the normal manual installation, and if you experience problems you can have a look inside the corresponding script to understand what is wrong. A few more details are available by running any of these scripts with the `-h` option. The scripts are located in the `install-scripts` directory of the *lipn-uima-core* environment. You can run `install-scripts/install_all.sh` to try to install all programs **except YaTeA**. If you plan to install YaTeA (see below), it is more convenient to install it before running `install_all.sh`, because it will try to create links to the executable/config files needed by the library.

2.3.2 Installing YaTeA

There is no script for installing YaTeA, you must install it manually because it uses the Perl modules install mechanisms. The more convenient way to install YaTeA is to use the CPAN command line tool:

- run it as root if you want to install YaTeA in the standard `/usr/...` location.
- on the contrary, if you plan to install it locally (say in `~/my_perl_libs`) set `PERL5LIB` environment variable so that it includes your local Perl modules directory, e.g.:
  `export PERL5LIB=$PERL5LIB:~/my_perl_libs/share/perl5:~/my_perl_libs/lib/perl5`
- run `cpan` (if this is the first use, cpan asks you about the configuration: you can accept the automatic procedure)
- if you plan to install YaTeA locally, before installing YaTeA type the following commands to tell cpan where is the target dir (inside cpan):
  - `o conf makepl_arg PREFIX=~/my_perl_libs`
  - `o conf make_install_arg PREFIX=~/my_perl_libs`
  - `o conf commit` (to remember changes)
- inside cpan, type `install Lingua::YaTeA` and allow cpan to look for dependencies. Normally that is enough, however you will likely experiment errors: the compilation (usually) works fine but the process fails when trying to copy executable and configuration files to the right directories. You wil probably have to copy these files manually[18]:
  - `cd ~/.cpan/build/Lingua-YaTeA-0.5-<xxxxxx>`
  - `cp -r etc/yatea /usr/local/etc` (root) or `cp -r etc/yatea ~/my_perl_libs/etc` (local)
  - `cp -r share/* /usr/local/share` (root) or `cp -r share/* ~/my_perl_libs/share` (local)
  - `cp -r bin /usr/local/bin` (root) or `cp -r bin ~/my_perl_libs/bin` (local)

If the `cpan` procedure fails, you will have to install YaTeA manually:

[14]Currently available at `http://http://www.lipn.univ-paris13.fr/~moreau/tagen-2.0.1.tgz` (temporary location)
[15]`http://pageperso.lif.univ-mrs.fr/~frederic.bechet/download_fred.htm`
[16]`http://pageperso.lif.univ-mrs.fr/~frederic.bechet/download_fred.htm`
[17]`http://search.cpan.org/~thhamon/Lingua-YaTeA-0.5`
[18]The directory structure may vary depending on your system.

```
tar xvfz Lingua-YaTeA-0.5.tar.gz
cd Lingua-YaTeA-0.5
perl Makefile.PL [PREFIX=~my_perl_libs]
make
make test
make install
```

You will probably face missing dependencies (then the `make` step will fail indicating their names), and have to install each such module manually (using the same procedure).

IMPORTANT: There are several bugs in Yatea, some of them require manual correction. Make sure to check part 7.0.5 for instructions.

2.3.3 The recommended way

By default all the components are parameterized to run the suitable program located in a directory named `tools`: for example, the TreeTagger wrapper will look for the program `tools/TreeTagger/bin/tree-tagger`. Of course the executable name can always be provided as a parameter, but to run these components in the most convenient way we recommend to create a directory called *tools* containing all needed external software: to be able to run all components, `tools` must contain the following directories (with the exactly same name):

- `tagen2` for TagEN, containing a directory (possibly a link) called `unitex` containing (resp. refering to) Unitex ≥ 2.0;
- `TreeTagger` for TreeTagger ;
- `lia_tagg` and `lia_ne` for LIA tools ;
- The YaTeA case is a bit more complex because of the way YaTeA is installed by default[19]:
  - a symbolic link `YaTeA` pointing to the directory `my-path-to-perl-modules/share/YaTeA` (which contains `config` and `locale` subdirectories) ;
  - a directory `bin` containing a symbolic link to the `yatea` executable.

Warning: most of these tools can not be installed in a path containing whitespaces.

Notice that you can use symbolic links everywhere in this file structure, e.g. `tools` and/or `TreeTagger` may be a link to the real directory, and/or any of the other directories.

2.4 Installing the main software

The software can be found at: `http://www-lipn.univ-paris13.fr/~moreau/uima/lipn-uima-core.tgz`

Since *LIPN UIMA Platform* is a set of Java packages, strictly speaking the installation procedure is very simple (see below). However you will probably want to learn more about the "environment", especially if you experiment problems: details are provided in 2.2. Additionally dependencies between modules are described in 2.7.1, and you may also want to compile sources (or understand how it works): see 2.7.

the *LIPN UIMA Platform* JAR packages can be directly (simply by adding their location to your `CLASSPATH` or calling java with the `-cp` option). However *LIPN UIMA Platform* is provided with an "environment" intended to make it more convenient to use: in this section we will consider using this environment. You only need to decompress the `.tgz` archive wherever you want:

[19] This point could be improved in the future

```
tar xvfz lipn-uima-core.tgz
```

The *lipn-nlptools-utils* package is included in the environment. Moreover there is no real need to install UIMA to use *LIPN UIMA Platform*, it is only required to have all the required JAR packages (including the UIMA ones). However for any "not beginner" use the UIMA environment is recommended (see UIMA official documentation), because it contains useful tools (see also 2.2).

2.5 Quick testing

In the *LIPN UIMA Platform* environment several examples are provided in the `tests` directory. To test the installation you can simply call the `./lipn-run-cpe.sh` script with one of the CPEs as parameter (from the environment root), for example:

```
./lipn-run-cpe.sh tests/CPEs/lia-tagg-chain+lia-ne-fr-small-iso.xml
./lipn-run-cpe.sh tests/CPEs/tagen-eng-small-utf8.xml
./lipn-run-cpe.sh tests/CPEs/tt-yatea-fr-small-iso.xml
```

For each of them, the expected output is a `.xmi` file which will be written in the `output` directory (notice that the first and the last example use the same input so the output filename is the same). If an error occur, refer to 7.

Warnings:

- Some of the CPEs may not work, because the data they refer to is not included as part of the environment.
- Some of the CPEs process big-sized files, therefore need a lot of memory to run (see the `lipn-run-cpe.sh` script options, Java documentation and 7.0.6).

2.6 Using *LIPN UIMA Platform*

LIPN UIMA Platform is a set of UIMA components, and as wrappers components they need that the underlying software be installed. Basically that is all that experienced UIMA users need to know; nevertheless we provide below a short survey about the different ways to use UIMA components, together with some more specific notes.

2.6.1 Creating/editing a CPE descriptor

A CPE descriptor contains all the information needed to run a process: the components themselves (collection reader, AEs and possibly CAS consumers) and for each of them values for its parameters. In particular it is worth noticing that the input data is also specified in the CPE, since it is usually a parameter of the collection reader[20].

Of course The CPE descriptor can be created by hand or using any XML editor. For convenience a standard graphical editor is provided in UIMA. Even if this editor has some limitations (and sometimes even bugs), it makes the task easier, in particular by making visible the parameters which can be set for each component (though the description of a parameter or a component is not visible). The CPE GUI can be called in three different ways:

- by running class `org.apache.uima.tools.cpm.CpmFrame`
- by using the UIMA script `$UIMA_HOME/bin/cpeGui.sh`
- in Eclipse, by running the UIMA example application *UIMA CPE GUI*

[20]Because even if files are frequently used as input, there is no reason to limit a CPE to this kind of input: one can imagine a collection reader grabbing its input on the internet, for example.

One of the most important limitations of the CPE GUI is about “imports by name”. In UIMA descriptors (not only for CPEs) one can use either “import by location” or “import by name” to include an external element (in this case a component). The former requires the location of the imported descriptor file in the filesystem, whereas the latter requires the name of the descriptor in the (current) Java `CLASSPATH`. Using “imports by name” is far more convenient: this way descriptors are read in the `CLASSPATH`, thus one has only to include the right JAR files in it (the *lipn-uima-core* JAR for example), so there is no path problem, and no need for the user to change the descriptor each time she wants to run it from a different location.

The CPE GUI can not:

- write any import by name (only by location),
- and read a CPE containing “import by name” for the Collection Reader and/or the CAS consumer (strangely it seems to work for AEs).

As explained above, this is a real problem in order to make the descriptor independent from the place it is used. This is probably something that will be corrected sooner or later in IUMA. Currently we recommend to:

1. create and configure a CPE using the CPE GUI, possibly test it, and save it
2. for future use (i.e. for a CPE intended to be run several times possibly in different locations), open the descriptor using any XML editor (or any editor!) and modify it by hand: you only have a few lines to change to replace the “imports by locations” with “imports by name”[21] (see UIMA doc).

This is the way the provided example descriptors have been created (see 2.5). As aforementioned, the drawback is that such descriptors can not be open using the CPE GUI. In the future, an alternative CPE graphical editor should be released from the LINA[22].

2.6.2 Running a CPE

There are several ways to run a CPE:

- using the UIMA “run CPE” tool:
  - running class `org.apache.uima.examples.cpe.SimpleRunCPE`
  - using the UIMA script `$UIMA_HOME/bin/runCPE.sh`
  - in Eclipse, using the UIMA example application *UIMA Run CPE*
- using the `lipn-run-cpe.sh` script (see below)
- using the UIMA CPE GUI tool:
  - running class `org.apache.uima.tools.cpm.CpmFrame`
  - using the UIMA script `$UIMA_HOME/bin/cpeGui.sh`
  - in Eclipse, using the UIMA example application *UIMA CPE GUI*
- using the UIMA document analyser, which can be used to run a CPE and/or visualize annotations:
  - running class `org.apache.uima.tools.docanalyzer.DocumentAnalyzer`
  - using the UIMA script `$UIMA_HOME/bin/documentAnalyzer.sh`
  - in Eclipse, using the UIMA example application *UIMA Document Analyzer*
- and finally from a Java application (see UIMA doc)

[21]Frequent error: don’t forget to remove the `.xml` filename extension!

[22]The tool is called *Dunamis* and is already available for testing, contact Jérôme Rocheteau for more details.

The script `lipn-run-cpe.sh` is intended to make a bit more convenient the configuration needed to start a CPE in the context of the LIPN environment. It will use either the UIMA standard way to run a CPE (`UIMA_HOME` must be set), or use the `CLASSPATH` variable to read UIMA libraries. This script should preferably be called from the project root, especially when using it with the test data provided with *lipn-uima-core* (but this is not mandatory). Run `./lipn-run-cpe.sh -h` for details.

2.6.3 Visualizing results

The standard way to store the annotated documents in UIMA is the XML XMI standard format (see UIMA doc). Such an output is produced by the UIMA standard CAS consumer `org.apache.uima.tools.components.XmiWriterCasConsume` (descriptor available in `$UIMA_HOME/examples/descriptors/cas_consumer/XmiWriterCasConsumer.xml`). Therefore there are UIMA tools to visualize the artifact and annotations contained in such files:

- The UIMA document analyser:
  - running class `org.apache.uima.tools.docanalyzer.DocumentAnalyzer`
  - using the UIMA script `$UIMA_HOME/bin/documentAnalyzer.sh`
  - in Eclipse, using the UIMA example application *UIMA Document Analyzer*
- The UIMA visualizer:
  - running class `org.apache.uima.tools.AnnotationViewerMain`
  - using the UIMA script `$UIMA_HOME/bin/annotationViewer.sh`
  - in Eclipse, using UIMA example application *UIMA Annotation Viewer*

Warning: a very frequent error consists in passing a wrong TS as a parameter to the visualizer. Usually there is no error message, but any annotation type which is not in the TS will not be shown in the result: thus not seeing all annotations (sometimes no annotation at all), the user may erroneously think that the process failed, although they are simply not visible.

This tool has also some limitations. The *Dunamis* tool planed by the LINA should be more complete (see above).

2.6.4 Running as a black box

Unfortunately there was not enough time to implement this behaviour. Hopefully that will be achieved later.

2.7 Compiling sources

This part explains how to compile sources, this is not necessary for most users (who can simply use the JAR archives). We propose two methods, using Eclipse and Maven. Of course it is also possible to compile these modules without these tools, but be careful: do not assume that basic dependencies between Java classes are enough, because UIMA components do not have a `main` method. That means that the project does not have a tree structure concerning compiling dependencies. Moreover do not forget to copy the content of the `resources` and `desc` directories into `bin`.

LIPN UIMA Platform was developed using UIMA 2.3.0 and Sun/Oracle Java 6 SDK. If you encounter problems when trying to compile/run the software, check that you use the Sun JRE[23].

[23] In Eclipse: go to Window/Preferences/Java, then in *Installed JREs* you can configure the JRE to use, and in *Compiler* you can set the Java version.

2.7.1 Modules dependencies

The following dependencies must be satisfied for the components to run. If you experiment `Class not found` errors when testing any component, check that your `CLASSPATH` contains the right modules.

- The *lipn-nlptools-utils* requires the following modules: `commons-cli`[24] and `uimaj-core` (used only for the `Logger` interface).
- The *lipn-uima-core* requires the following modules: `uimaj-core`, `uima-cpe.jar`, `uima-document-annotation.jar`, `uima-tools.jar` and `commons-cli`. It also requires *lipn-nlptools-utils*.

In both modules these packages are included in the `lib` directory. Be careful however: there may exist newer versions at the time you use it (possibly including bug corrections). Notice that you can simply use the *lipn-nlptools-utils* JAR to compile the *lipn-uima-core* (instead of compiling both).

2.7.2 With Eclipse

To open the *lipn-nlptools-utils* using Eclipse, create a new Java project and choose *Create project / Java / From existing sources*. Normally the needed JAR files will automatically be loaded from the `lib` directory; if not, add them manually to the *Build path*.

To open the *lipn-uima-core* using Eclipse, create a new Java project and choose *Create project / Java / From existing sources*. Normally the needed JAR files will automatically be loaded from the `lib` directory; if not, add them manually to the *Build path* (see 2.7.1). Then add the `desc` and `resources` directories as *source folders*. Binaries will be written to the `bin` directory.

If you want to use the version of*lipn-nlptools-utils* you opened as an Eclipse project, you should remove the corresponding JAR from the *Build path* (or remove it from the `lib` directory before creating the project), and add the *lipn-nlptools-utils* to the *Build Path* as a project.

Note: if you work with `svn` and do not want to use the `svn` Eclipse plugin, you can exclude `.svn` folders from the buiding process. This is useful in particular for resources folders, because Eclipse will complain about duplicate resources otherwise (warnings). Simply go to Window -¿ Preferences. . . , Java -¿ Compiler -¿ Building. Under "Output folder" add ", .svn/" to "Filtered Resources" (so that you get "*.launch, .svn/")[25].

2.7.3 With Maven (command line)

Apache Maven[26] is a software project management and comprehension tool. This tools is intended to make easier the compiling/deploying process, in particular in dealing with dependencies between projects. As a very rough comparison, its role is similar to tools like `make` or `ant`. This tool is probably already a standard one in Java development, and is frequently used in the UIMA community (because sharing UIMA components often requires to deal with dependencies).

That is why Maven is used in the *LIPN UIMA Platform* modules. Nevertheless we did not use the standard Maven project structure, in order that the modules could esily be used without Maven[27]. The main part of the POM (Maven `pom.xml` config file) is:

```
<build>
```

[24]Used to parse command line options.
[25]From `http://blog.projectnibble.org/2009/08/11/repost-make-eclipse-ignore-svn-directories`
[26]`http://maven.apache.org`
[27]We have followed a tutorial by Fabien Poulard: `http://www.uima-fr.org/planet/#article10`

```
    ...
    <sourceDirectory>src</sourceDirectory>
    <outputDirectory>bin</outputDirectory>
    <resources>
      <resource>
        <directory>desc</directory>
      </resource>
      <resource>
        <directory>resources</directory>
      </resource>
    ...
   </resources>
  </build>
```

Compiling using Maven[28] is very simple: run `mvn install` (at the root of the project)[29] for *lipn-nlptools-utils* firstly, then for *lipn-uima-core*, because the latter depends on the former[30]. If all is fine a JAR archive has been created in the `target` directory.

Remark for Maven beginners: the `install` phase is almost the highest in the compiling process. It actually implies (if needed) several other phases among which:

- `mvn compile` which only compiles sources (thus updates the `bin` directory)
- `mvn package` which creates a JAR archive stored in the `target` directory
- `mvn install` which installs the module in the Maven local repository[31], so that it can be used by other modules needing this one as a dependency.

2.7.4 SVN repository structure, stable and unstable versions

The current `svn` repository contains the development version(s) in the `dev` directory, whereas stable releases are put in the `pub` directory. Developers writing to the repository are encouraged to stick to such a policy (or a better one) in the future. The following Maven convention is also used: development versions are named with suffix `-SNAPSHOT`. As usual, using such a version is unsafe but possibly corrects bugs compared to the latest stable release.

3 Wrapper annotators: principle, approach and main issues

3.1 Definition and issues

The first LIPN components (and actually most components in version 0.1.*) which have been created are wrappers for previously existing programs. An *external program wrapper* is a UIMA annotator which delegates its work to some external program. Thus such a component consists in

1. Initializing any parameter that the external program needs (possibly resources etc.)
2. Reading the CAS and converting the data (text and/or annotations) in the format expected by the external program

[28]Installing Maven is also simple: generally you just need to run `sudo apt-get install maven2`.
[29]Maven will probably download a lot of dependencies the first time it is used. This happens only once.
[30]Currently you can not compile only *lipn-uima-core* without, because the *lipn-nlptools-utils* must be in the local Maven repository in order to satisfy the dependency.
[31]By default it is `$HOME/.m2/repository`.

3. Calling the external program, and
    - providing it with the input (obtained from the CAS and converted)
    - receiving its output
4. Reading the external program output and converting it back to the CAS (finding the annotations and adding them)
5. Cleaning any temporary element

As one can see, the main task of such an annotator consists in converting the input and output. But there are serious issues to address:

Portability/Safety Using a system call is very unsafe, especially in Java because this language is intended to be portable. Of course, a system call is almost never portable (as of version 0.1.**LIPN UIMA Platform* components are only Linux/Unix compatible). Moreover it relies on the existence and behaviour of a component which is out of Java control. A lot of general safety mechanisms become ineffective (exceptions, shutdown hooks, concurrency, etc.), and this is the reason why using such an approach requires a lot of care to minimize the risk of errors/problems.

Safety/UIMA process The UIMA framework is supposed to have complete control over the process during a run. This is how safety is guaranted in a complex process where there can be concurrency issues, network issues, etc. UIMA deals in a subtle and robust way with errors (suitable exceptions), with resources management, with logging possibilities. Thus the general UIMA processing flow is broken in duch a context.

Ease of development/use In general, it will be more difficult to use a wrapper component than a standard one: of course the user must install the external program, and has sometimes to accept unexpected constraints. Moreover the component must (try to) handle all errors that could happen inside this program, which is harder in this (more or less) black box case. It must also make these errors as clear as possible for the user, in order that he does not have to look at the external program level to understand what is wrong. Flaws in these development requirements would clearly imply very complex problems for the user.

Input/Output transmission:

- Conversions entail potential flaws in the data viewpoint. Format errors may occur because sometimes it is hard to take all possible cases into account. Additionnaly the external program format is not always[32] fully specified (forbidden/special characters, program expecting or returning "normalized" sequences, etc.)
- I/O physical transmission can be achieved in different ways: file stored in the disk is the most simple one, but some programs can/must read/write their input/output as the standard input/output stream (`stdin/stdout`). Both methods have drawbacks and advantages (see 3.3).
- Whatever the I/O transmission method, once again there are important safety issues since any I/O operation can fail or block for various reasons.

Efficiency Calling an external program in a UIMA process necessarily implies time and space waste: data conversion and transmission are clearly an additional step in the process. Furthermore, there can be serious memory overload problems because the memory still contains all the UIMA stuff in memory (including the CAS) during the call. As a rough approximation, that means that the document and its annotations will be stored twice during the execution of the external program (see 3.3 for more details).

The *lipn-nlptools-utils* module has been created to deal with these issues (see 6). But as a general rule these issues imply that a strict and careful programming policy is required. Indeed, there are a lot of possible flaws and the developer should not forget that the UIMA components could be used in various contexts and with various kind of data. If these components are used as a part of a big UIMA system processing large documents, the user should be able to trust the basic annotators and should not have to face any hazardous behaviour. Above all he should not

[32] Almost never actually!

have to dig into the annotator code to undertand what happened if an error occured. The following guidelines are intended to prevent such problems.

Remark: during the period where *LIPN UIMA Platform* was implemented, another wrapper generic annotator has been created by Nicolas Hernandez at LINA, see:
`http://enicolashernandez.blogspot.com/2010/10/reutiliser-des-outils-externes-via-des.html` [fr]

3.2 General programming good practices

To sum up, the context is: (1) the coded components are intended to be used in various not already known contexts, (2) the coded components should be able to deal with huge documents, (3) it uses black boxes sub-programs, and (4) a lot of possible technical flaws are expected... That may sound obvious, but programming in such a context requires more care than when you have to code some simple script that will be used only once or twice. Thus a few classical guidelines are in order:

- Documenting the components outside: preparing clear Javadoc API, possibly enriching it with some other documentation (user guide)
- Documenting the components inside: comments for future developers/maintainers;
- Logging, i.e. writing any intermediate information that could be useful for debugging to the log file. Notice that UIMA comes with a built-in logging mechanism;
- Testing components as much as possible;
- Searching and using previously existing objects/methods (which have been widely tested and used) rather than writing possibly buggy code to do the same.
- Searching and using any standard/convention about representing data of any kind (e.g. using XML rather than a component-specific tagging format, read/write charset encoding names rather than assuming always some particular charset, indexing String in the standard Java way, etc.)
- Avoiding ambiguities or possible coding flaws: for example by removing any warning at compile time[33]: even if the developer knows what he does, someone else could modify the code in a way which makes it buggy.

It is impossible to guarantee that a code is error-free; however it is possible to make the code simpler to correct/debug/maintain. Clearly that takes time, but this is the price to pay for "long-term programming", which is the objective to have in mind wuth UIMA.

3.2.1 Genericity, re-usability

The Object Oriented Programming paradigm offers convenient "tools" to develop in a generic way, that is to say a way which permits to re-use as much as possible existing code, by factorizing common methods/objets and specializing them whenever necessary. This is a time consuming effort: indeed, it is usually simpler to analyze each particular problem/task one after the other and not bother about the previous or next one. Nevertheless it is once again not a good solution in the long term: planning future uses of the implementation makes it more scalable and easier to maintain; additionally, it is useful to guarantee some local form of standardized behaviour between similar tools. As an example, `Reader` and `Writer` objects[34], which are classical Java objects, are used a lot in *LIPN UIMA Platform*: this permits to process the same way (for example) a text file, a `String` object or a text stream (like `stdin/stdout`).

In most complex NLP components, there are multiple small choices which are mainly conventions, in the sense that any option is acceptable; however using always the same option (which is easy when relying on generic objects

[33]This is rather easy with an IDE like Eclipse (no, not by using `@SuppressWarnings` ;-).
[34]To avoid confusion, please recall that a `Reader` (resp. `Writer`) is not an object *which* reads (resp. writes), but an object *in which* it is possible to read (resp. to write).

since the same method is called) improves consistency between different components, and finally makes a whole system more robust. Needless to say, it is also a huge advantage when debugging: (1) when trying to track a bug, it is easier when all similar code is located in the same place; (2) when correcting a bug/problem, the modification applies to all components using this generic object, contrary to the case where it is necessary to make the same correction in the different components (risking to forget it somewhere). Of course, such a generic implementation may seem more complex (and actually *is* generally more complex).

3.2.2 Versions

Remark For future developers of LIPN UIMA Platform*:* This system is intended to be scalable/progressive, so it is susceptible to become more and more complex when growing. This is why it is important to have a clear versioning policy, at least making the difference between stable and unstable versions and keeping a copy of old stable releases. Currently we only initiated a very simple versioning approach; a clear policy should be adopted in the future.

One of the most important points in following a rigorous version policy (like in most recent implementations of any software) is *not* to consider that a software is *achieved*: systems are too complex to be considered achieved because

- each component rely on a lot of other components (ranging from the libraries used to the programming languuage and even the operating system), which can change themselves.
- at least in NLP, any component which can seem simple is actually not: there are always different choices, formats, features which can need to be modified in the future.
- Finally a developer who pretends to provide a totally bug free code should be asked to prove it!

As a consequence software components evolve, and it is necessary to organize this evolution in a rigorous way. software versioning tools are now widely used and there are a lot of guides/documentation about using them wisely.

3.3 Input/output issues

When calling an external programs from a UIMA process, the UIMA annotator task consists in extracting data from the CAS, formatting it, sending it to the program as input, and then to receive the program output, reading it and adding new annotations to the CAS. For the sake of efficiency the data is transmitted "on the fly" whenever possible: indeed, a lot of tools read their input as a stream (*stdin*) and write their output as another one (*stdout*). Transmitting data on the fly avoids storing it as a file or in memory: firstly this saves space, and (memory) space may sometimes be a problem with very large documents. Secondly this is a gain of time, since both in input and output the receiver does not have to wait for the sender to transmit the whole data (the gain is even more important compared to storing the data as a file, since disk access are very time consuming).

A comparison between the different ways to transmit data from the CAS to the external program and back is given in table 1. In this table

- `p` means "process" (time for the external program to process the data)
- `[rw][mds]` means Reading|Writing Memory|Disk|Stream
- `//` means "simultaneously", but not necessarily in different real threads/processes: for example the reading and writing operations in reading from a file to memory are called "simultaneous" here. It is worth noticing that `x//y` is lower or equal than `x+y`: maybe nearly equal if in the same thread, but lower if in different threads.

Moreover two cases (A) and (B) are considered:

(A) The external program needs to store the complete data in memory

(B) The external program does not store the whole data in memory, and processes data on the fly (for example it receives a sentence, processes it and writes it before processing the next one)

	Space used	Time used
Temporary file (A)	2× RAM + disk	2× `rm//wd` + 2× `rd//wm` + `p`
Temporary file (B)	1× RAM + disk	`rm//wd` + `rd//p//wd` + `rd//wm`
Stream using a `String`[35](A)	3× RAM	2× `rm//wm` + 2× `rm//ws//rs//wm` + `p`
Stream using a `String` (B)	2× RAM	2× `rm//wm` + `rm//ws//rs//p//ws//rs//wm`
Stream using a `Reader` (A)	2× RAM	2× `rm//ws//rs//wm` + `p`
Stream using a `Reader` (B)	1× RAM	`rm//ws//rs//p//ws//rs//wm`

Table 1: The different ways to transmit data.

Table 1 clearly shows that transmitting data on the fly is a lot more efficient: using a temporary file requires disk accesses which are very time consuming, and using a `String` object requires a lot more mremory space. Of course these differences are negligible when processing small documents, but it is important to remember that the component may be used in the future in cases that had not been planed. Furthermore, the risk of failure for a component should be minimized as much as possible, because it can have serious consequences in a complex system. This is particularly true for memory overload, which will eventually make the other components fail if they use the same memory space.

See also 6.1 and 6.1.2, in particular about charset encoding issues.

3.4 (Threads) concurrency issues

Threading data transmission when calling an external program requires quite careful programming and testing, but it is not very hard (see 6.1). On the contrary, using concurrency when accessing the CAS is quite challenging: actually UIMA is designed to deal with different threads running several (instances of) annotators in the same time, but it is not intended to handle concurrency *inside* a single annotator. Thus all CAS accesses have to be protected (synchronized), which is in fact quite tedious. LIPN had then to develop an approach and tools to embed most thread-related code, in order to make things easier for the annotator developer. These tools are now achieved, and have been rather thoroughly tested (since concurrency bugs are a real pain to track). See 5.1.2 for the general principle, and the API for more details.

3.5 Dealing with errors

The developer must keep in mind that the component he builds might be used in a wide range of different contexts. As a consequence, the way errors are handled is crucial, because it affects all components which would be run after it, and any data which would be computed based on its own output. Additionally, it is important to remember that the component is likely to be used with huge data in a very time-consuming process in which any fatal error can be very costly. Therefore it is important:

- for a component to try as much as possible to solve the errors which belong to its scope (i.e. that it is able to solve itself), in order not to
  - keep a minor error propagate and possibly alter the posterior data computed after it;
  - make the process stop in an unrecoverable way in some posterior stage of the process.

[35]We suppose using a `String` object both for input and output: this means that the data must be copie from the CAS to a `String` before sending it, and conversely is received in a `String` before being copied in the CAS (this is why there are 2 `rm//wm`).

- but a component must not extrapolate error corrections to cases where
  - it does not have the needed “knowledge” to fix it (e.g. with some ambiguous data, unknown character, assumption about what the user intention, etc.);
  - there can be different interpretations of the problem: something which is invalid for some component may be valid for another one, thus no final decision should be made which would alter the behaviour of the posterior components.

  In those cases, most of the time the component should indicate the error/problem to its environment:
  - throwing a Java/UIMA exception, possibly caught at some higher level of the process
  - writing/storing the pieces of information using the UIMA logging mechanism if it is a minor problem.

Explaining in the component description/documentation/API the errors cases and especially the ambiguous cases (where the problem can be interpreted both as an error or not) is very important. In general, up to now a “stop as soon as possible” policy has been applied (in the cases where the component can not internally solve the problem):

- whenever a component encouters invalid data or parameter, it throws an exception. It does not have to try to infer the value (or use a default value) unless it is clearly its role to do so (for example, if an invalid character is read in the data and the user had requested the component to ignore such characters)
- this behaviour should be applied as soon as possible: for example if a user defined parameter is invalid, it is a good idea to stop the process (throwing an exception) in the initialization stage of the component, and not to wait for the processing stage. This behaviour saves time, and if the process is very long it is preferable to stop it early in order that it does run uselessly before failing.
- as a consequence, it is preferable to check for errors even if they do not have an impact at the current stage. For example, most wrappers annotators have to parse the external program output in order to extract the data needed to add some annotations; suppose the output is supposed to contain one token by line (that was provided as input) followed by some tags (which is the actual output): the annotator could easily ignore the token since they are already stored in the CAS, but by checking that each token received indeed corresponds to the input token the annotator ensures that there is no gap which could possibly cause an error later in the proess[36]. Moreover it is a way to reinforce the robustness and consistency of the whole process.

4 The *LIPN UIMA Platform* Type System

4.1 A generic Type System

One of the most important points in designing a complex UIMA platform is the *Type System* (TS), i.e. the typology of the annotations. Generally speaking, there are two main approaches in designing a TS: either it is intended to be very precise and exhaustive, or on the contrary very general and abstract. The former approach (for which [**?**] is an example) describes the data in straightforward and complete way. But its fixed nature is a serious issue: it is then very hard (sometimes impossible) to create new kinds of annotations, or to slightly modify the existing ones (e.g. to add a feature), because the components using this TS are intended to em a priori know the whole set of possible annotations. Moreover, it is by definition less flexible, and therefore less suited for concurrent annotations (see below). Additionally, this kind of TS is often very complex and may be difficult to handle at first (this issue being known as the “Monster TS” problem).

On the opposite, the latter approach consists in providing the components with the widest possible range of freedom

[36]It is however advised to keep an option (usually a component parameter) which disables this strict behaviour: there are cases where it is either useless, too time-consuming, or even counter-productive (e.g. one knows there are errors but wants voluntarily to try it anyway). It should also be mentioned that this behaviour has some drawbacks: sometimes the strict checking causes more problems that it solves, if the external program on which it relies has some hardly predictable behaviour.

about extending the TS[37]. In that case the TS is very small, and it is intended to be extended and/or used with the least possible constraints. Nevertheless it must be noticed that the problem of representing annotations is not solved this way, it is only partly delegated to the future components designers. Thus such a TS is a lot more flexible, but often more difficult to use "in a clean way", because it is not constraining. Therefore maintaining consistency between components depends both on the original desing of the TS (including use guidelines) and on the components designers/developers.

The main reasons why LIPN choses the latter "abstract TS" approach are the following:

- It is better suited for the sake of flexibility and modularity. Our goal being to permit to integrate any kind of annotations, it is preferable to postpone the concrete creation of a relevant TS as late as possible, in order to have the best possible information when taking decisions.
- It is also more suited to the actual context where it takes place: the goal is not (at least currently) to achieve a complete self-contained system dedicated to some particular task; LIPN intends to progressively build tools which will hang on this platform, without knowing exactly which ones, when, and how.
- The first components that will be created are mainly existing tools which were not intended to be integrated in such a framework and not always easily adaptable; that is also why it is important not to impose strong constraints.
- Finally we also think that this approach is more suitable for dealing with concurrent annotations (see 4.2).

The generic TS that we opted for is represented in figure 1. All types derive from the standard UIMA supertype for text annotations, namely `uima.tcas.Annotation` (which makes a lot of types-related operations easier). All types used in LIPN annotators will actually derive from `GenericAnnotation`, which inherits from `uima.tcas.Annotation`. This root type (and consequently any inheriting type) includes the following features:

- a confidence score `confidence`;
- a type identifier `typeId`, that annotators can use as an alternative to adding new "real" specialized types in the TS;
- a component identifier `componentId`, in order to identify the annotator responsible for this annotation, which is especially important in the case of concurrent annotations (see 4.2);
- a user defined identifier `runId` (see 4.2).

Moreover LIPN types must inherit from one of the three following types:

- `Segment`: if the type simply defines an area (e.g. tokens, sentences, chunks);
- `AnnotationTag`: if the type adds some information about the data (e.g. a POS tag, a NE category);
- `Relation`: for "meta-annotations", that is to say annotations related to some other annotations (e.g. syntactic relation based on POS annotations);
- `TaggedRelation` for a `Relation` labeled with some information (e.g. kind of syntactic relation).

As a generic/abstract TS, this Type System is planed to be locally specialized by the AEs (generally groups of related AEs). However these extensions should remain consistent with the genericity-oriented framework that the original TS aims at building. Thus the following conventions should be followed:

- The specialized/local part of the TS should also be as generic as possible. In particular it should avoid using a high number of different types (precise information should preferably be stored inside annotation

[37]Remark: we actually want to use at least a minimal TS structure, but there are also ways not to make any assumption about the TS: about such "TS agnostic" approaches, see `http://lexgarden.blogspot.com/2010/07/4-strategies-for-building-type-system.html`.

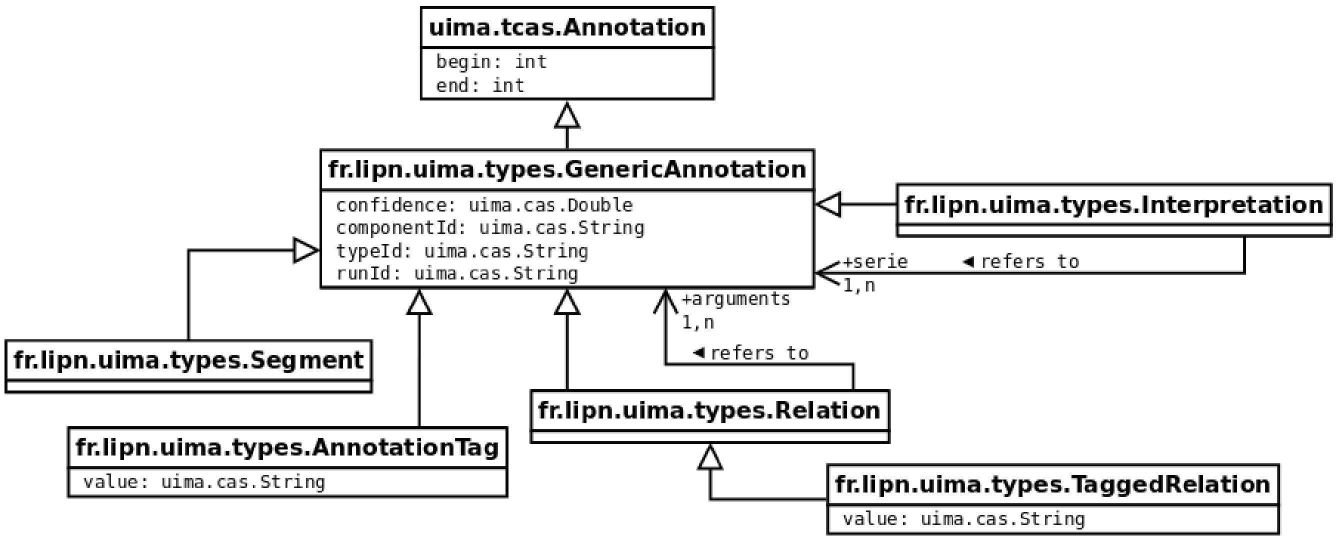
uima.tcas.Annotation
begin: int
end: int
fr.lipn.uima.types.GenericAnnotation
confidence: uima.cas.Double
componentId: uima.cas.String
typeId: uima.cas.String
runId: uima.cas.String
fr.lipn.uima.types.Interpretation
+serie
1,n
◄ refers to
+arguments
1,n
◄ refers to
fr.lipn.uima.types.Segment
fr.lipn.uima.types.Relation
fr.lipn.uima.types.AnnotationTag
value: uima.cas.String
fr.lipn.uima.types.TaggedRelation
value: uima.cas.String

Figure 1: The LIPN Type System

features); similarly it should not make the inheritance hierarchy too complex/specialized, nor adding complex relationships between types (especially mandatory ones). This is important for the reasons explained above, in order not to impose constraints about the way the other components have to deal with these annotations.

- All annotations should remain as atomic as possible; in other words, the types are supposed to contain a minimum number of features. This way (1) a component can work on the exact part of the annotations it has to, without having to manage with some extra features (possibly it does not know how to interpret them and/or what value to specify for them); (2) it is more convenient to handle concurrent annotations (see below), since there can be different possibilities for a given feature and only one for the other.
- The components should always prefer using the generic methods provided in `fr.lipn.nlptools.uima.common` to handle annotations (see the API for details). If there is no suitable method for some particular (non-specialized) task, then the *LIPN UIMA Platform* maintainer should be noticed about this missing feature.

In order to remain consistent with the principle of a generic "simple" TS, the extensions of this TS should avoid using complex types (with a lot of features, or with complex relations linking types together, etc.) and prefer small, simple and independant types. Thus concise types (typically based on `AnnotationTag` extensions) should be used whenever possible, even if that multiplies the number of different types. For example we have chosen to represent Part-of-Speech annotations and Lemmas annotations in two distinct types (see 4.3):

- this way a component can create Part-Of-Speech annotations whithout defining Lemmas and without leaving an undefined value for the lemma[38]; moreover it makes it possible to have one Part-of-Speech together with several possible Lemmas (see also 4.2);
- if a components has some more precise information to write (adding features), it can use a specific extended (inherited) type so that a component which does not need/know this extension can ignore it (reading it as the original type);
- the drawback is that matching related annotations together is harder: usually one wants a Lemma to be connected to its corresponding Part-of-Speech annotation. In the simple case where there is no ambiguity, it is sufficient to have both annotations exist together superimposed: it is not very hard to match them with using their indexes in the text (see also 4.3 about priority lists); in the most complex cases, a specific mechanism should be used, typically using an `Interpretation` annotation (see 4.2).

[38]Undefined values should be avoided as much as possible, because they can be a source of errors: a component could suppose that the value exists for all annotations, or it could interpret the undefined value differently (e.g. "not initialized", "unable to find a value", "an error occured", ...)

4.2 Concurrent annotations

4.2.1 Definition

Intuitively, concurrent annotations are distinct annotation sets which relate to the same text, in other words wich "superimpose" together. Such a definition is too loose, in particular because this concept is not needed in some classical cases like superimposing a POS tag and a lemma tag (and possibly some others like NE, term etc.). However this concept is useful mainly in the two following cases:

Alternative annotations , where annotations from the two sets (or at least a subset of them) play a similar role, in other words describle the same kind of information. Example: two series of tags obtained by two distinct tools which yield the same kind of annotations.

Conflicting annotations , where annotations from the two sets are not compatible. Example: two series of annotations, one being POS tags and the other being NE tags, in the case where the tools which have yielded them do not tokenize words in the same way; such a case may cause overlapping or nested tokens, which are a source of compatibility problems.

From these two cases, we propose the following definition[39]:

Definition. Two sets of annotations A and B are called *concurrent series of annotations* if they satisfy the four following conditions:

1. they relate to the same text (complete document or portion of text);
2. the annotations contained in any of the two sets (considered independently) are compatible together;
3. the sets A and B are independent, i.e. there is no reference between an annotation from A to an annotation from B and conversely;
4. the set formed by the union of A and B contains incompatibilities.

About the third condition, a dependency relation exists between two annotations in different cases:

- if there is an explicit reference, e.g. in the case of a relation (see above);
- but also if there is an implicit reference, e.g. a pair of POS/lemma tags which are jointly computed.

These cases are excluded in order to maintain consistency: two series of annotations can not be both dependent together and incompatibles[40].

Example. Tokenization is the most simple example of concurrent annotations: for example, if a tokenizer X considers the word *aujourd'hui* as one single token whereas another tokenizer Y breaks it into three different tokens *aujourd* - *'* - *hui*, then a tokenizing process using X *and* Y on a text containing this word yields two series of concurrent annotations.

4.2.2 Motivation

There are numerous reasons why concurrent annotations are useful:

[39]In which there is no explicit reference to the case of "alternative annotations, because this is useless: two alternative series must contain at least one incompatibility, otherwise they would not differ and therefore would not really be "alternatives".

[40]Incompatibility among the annotations inside one serie being excluded by the second condition, such a case would mean either that the annotations distribution between the two sets is wrong, or that the system which yielded the set containing the incompatibility is faulty.

- Comparing different series of annotations in order to evaluate (quantitatively ot qualitatively) different tools is one of the first applications: being able to represent simultaneously different series of annotations coming from different analysis components:
  - permits to observe clearly the differences between these components;
  - makes easier a figured comparison (assuming an homogeneous format among these annotations).
- Merging the yield of different comparable components: using some aproach which allows to select the best label among the different annotations coming from the components (e.g. the one obtained by most components), it is possible to build a meta-component which may perform better than any individual component.
- Improving the quality of annotations *a posteriori* by using annotations which are generally computed at an earlier stage, so unavailable to the component. For example, to obtain an enhanced words tokenization, it is worth computing it in two stages: the first classical one, and the second one after running some other components, using syntactic information, NE and/or terms tags etc. This second step should increase the likelihood of the whole tokenization.
- Considering the example with tokenizers X and Y (see above), suppose that we want to build a processing pipeline including some components X' and Y' which require tokenized text; suppose also that we know that X' gives better results when using the tokenization performed by X, whereas on the contrary Y' prefers the Y tokenization. Thus it is preferable to run both tokenizers, then to select the right serie of tokens annotations before starting each component X' and Y'.
- Representing ambiguities: in some rather simple cases such as assigning different labels to the same word (typically the different POS tags together with their associated probability); but also in some complex cases, for example in syntactic analysis, where it is necessary to be able to represent different superimposed alternatives, and even possibly to nest such alternatives at several levels. Any posterior process (including human observation) will be easier if a suitable representation is used.

Thus the ability to deal with concurrent annotations in a complex (modular and parameterizable) analysis system is important in general: without it, the system can only be used in the classical sequential way, which is rather static and constraining.

4.2.3 Representing concurrent annotations in UIMA

Dealing with concurrent annotations in a generic framework is not trivial, because the cases where concurrency will be used and the way it will be used are not known a priori. This requires to design some general mechanism so that this feature is taken into account at several levels:

- At the lowest level (probably in a package or class which embeds all technical details about concurrent annotation access, providing methods to be called by the actual annotators), the question is: how to represent two concurrent series of annotations in the CAS? If possible, this representation should be efficient, and convenient access to this level should be provided.
- At the Annotator Engine level, there are cases where the annotator is aware that it works with concurrent annotations (e.g. to select the most relevant annotation among different series); but it must also be possible to run a "standard" component (not intended to deal with concurrent annotations) on some particular serie of annotations among several concurrent series (ignoring the other ones). That means that it should be possible for a component to access to concurrent annotations "transparently", i.e. in the same way than usual annotations. That implies that it is necessary to determine when and how an annotator access to the "concurrent side" of the annotations.
- At the highest user[41] level, it is necessary to provide a way to control the series of concurrent annotations: if the CAS contains different series of annotations, it must be possible to tell a "standard" annotator (which

[41]It is worth noticing that in the UIMA framework it is sometimes hard to precisely define the *user* level, because this user may be either someone who only runs the component on some document, or someone who develops a new component which uses its output, or someone who includes this component in a pipe process: these different "users" do not have exactly the same viewpoint and knowledge about the component, and the developer must take all these into account.

reads a single set of annotations and write some new ones) which serie it should read, and possibly if the annotations it writes should be considered as a concurrent serie among some other ones.

We have studied three different approaches, among which the first two have been implemented in the current version:

- **A specific feature.** This what the `runId` feature (see fig. 1) is intended to (and also the `componentId` feature in a minor way). The value set for this feature can be used to distinguish between different series of annotations. This is simple and sufficiant for most cases, as soon as (1) this feature can be parameterized by the user (to select the serie to read/write bu a component) and (2) the component takes this parameter into account[42]. Nevertheless this solution can not handle complex cases, because the user must be able to control the number and the identifiers of the concurrent series (which excludes cases where a component internally handles different series).
- **A meta-type** used to encapsulate the annotations contained in a concurrent serie. UIMA permits to define a feature as a reference to another type or to an array of other types. This ability is used in the `Interpretation` (see fig. 1), which is intended to point to the different annotations contained in a serie. This method is a bit more complex in a technical viewpoint, but it is more complete and allows a component to (internally) handle different series of annotations (e.g. in the case of dealing with syntactic ambiguities). Additionally it is worth noticing that it is possible to nest as many levels as needed using such annotations.
- **Using UIMA's concept of view**. Originally designed to represent the same document in different ways (e.g. with/without HTML tags, or the same content in different languages), a view can also be used as a container for a concurrent serie of annotations (even for a small text portion). It simply requires that a special type be defined to indicate the concurrency, in such a way that its instances indicate both the position of the text and the identifier of the view in which the concurrent serie is written (in other words views are used as generic pointers to the annotations series). As in the case of of the `Interpretation` type above, it is possible to deal with complex cases and in particular to nest different levels of concurrency. However there are two drawbacks: the access time/space to the different views may be costly, and it is more difficult in that case to come back from an annotation to the text that it relates to (and/or to its position, depending on the way it is implemented).

See also 5.1.

4.3 The extended LIPN TS

As explained above in 4, the TS we use is a generic one, which implies that it is can be extended. This part describes how it has been extended for the requirements of the components which have been implemented so far. In the same time, these explantations can be seen as an illustration of how our generic TS can be used.

4.3.1 Modularity

The generic part of the TS is intended to be shared by all components; but the extensions should not be common in general, because this is precisely one of the advantages of a generic TS not to be complex (and extending it for all components together would basically dismiss this advantage). That is why extensions are supposed to be local: of course some types are very commonly used and have to be shared; additionally, "local" does not mean local for a single component: on the contrary, it is important that a group of components which are intended to work together (or simply belong to the same domain) use the same extended TS, in order to facilitate communication among this group. That is why we recommend not to define the TS in the component descriptor, but rather in an independant TS descriptor[43] (which is then imported by the components which need it).

[42]In our opinion this point is implicitly induced by the fact that the component uses this TS, in the sense that it is therefore supposed to be aware of the role played by each type/feature.

[43]Even if there is only one component using this particular TS, because one should always think that future components may want to use it.

Therefore we propose that the basic TS be extended in a modular way, which is rather easy in the UIMA framework, thanks to the "import" mechanism: any TS can import one or several other TS defined somewhere else. This is what was done for the first components, and the current TS is divided into three parts:

1. `fr.lipn.nlptools.uima.common.lipn-base-TS` is the generic basis (see 4.1);
2. `fr.lipn.nlptools.uima.common.lipn-standard-TS` is the common part among the existing components: it contains some very common types, which are all defined in a very simple way (in order to avoid types with multiple features/complex relations:
   - inherited from `Segment`: `Token` `Sentence` and `DocumentDivision`;
   - inherited from `AnnotationTag`:
     - `PartOfSpeech` → expected `value` is the POS label[44];
     - `Lemma` → expected `value` is the lemma;
     - `NamedEntity` → expected `value` is the kind of entity (category, e.g. "person", "location", "organization",...);
     - `TermOccurrence` → expected `value` is the term lemma;
   - inherited from `Relation`: `Term` which, if used, should contain the list of all occurrences for this term[45];
   - inherited from `Interpretation`: `POSLemmaPairInterpretation`, which is intended to link together a `PartOfSpeech` and its corresponding `Lemma` annotation.
3. `fr.lipn.nlptools.uima.common.lipn-extended-TS` is the whole TS containing not only the standard one but also the local specific TSs. This TS should not be imported in another TS or in a component, it is provided only for convenience (it can be useful in order to visulalize an annotated document when one does not know what annotations ot contains). Currently this TS consists in the import of[46]:
   - `fr.lipn.nlptools.uima.common.lipn-standard-TS`;
   - `fr.lipn.nlptools.uima.common.lia.lia-TS`, which defines a type `LiaCleaned`[47] which is specific to the LIA tools (LIA Tagg and LIA NE);
   - `fr.lipn.nlptools.uima.common.yatea.Yatea-TS`, which defines two specific types extending `Term` and `TermOccurrence` with some additional features.

Some built-in iterators are provided for a component to easily deal with these types. But it can be useful to undertand the priority lists mechanism that we use: [TODO à finir]

5 Components: tools and annotators

[TODO not platform independent!]

The first point is that an Annotator Engine (AE) developed in the *LIPN UIMA Platform* framework should always be defined as a subclass of `fr.lipn.nlptools.uima.common.LipnExternalProgramGenericAnnotator`. This is how it can have access to various common methods to deal with the particular features provided in the framework.

[44]Be careful, no normalization is required: a component can use any set of labels.

[45]Normally these occurrences should all have the same lemma, but that is not mandatory.

[46]It is worth noticing once again that new TS descriptors are created even to add only one type for the sake of modularity (see above).

[47]This type is used to store the "cleaned version" of a token that a LIA component has modified. Actually there still are some issues with this type, because the data it contains should be used as a replacement for the covered text: there is no way in UIMA for a given annotation to return "its own version" of the covered text (as that would be done using an interface with standard Java objects), therefore any component which is not aware that such annotations exist can not transparently obtain the correct value. The only solution would be to consider this case as a general case within *LIPN UIMA Platform* framework so that a component always has to check for such annotations, but we think this case is too specific to be handled this way.

5.1 Handling annotations

In order to maintain as much as possible a consistent representation/usage of annotations in the context of the *LIPN UIMA Platform* components, we propose some guidelines together with some built-in methods that any component can call to deal with annotations in the "LIPN intended way".

5.1.1 Adding annotations: `setGenericAttributes`

Every annotation contains the following features: `confidence`, `typeId`, `componentId` and `runId` (see 4). The confidence is an optional information that a component writing an annotation can provide if it is able to provide a degree of confidence for this annotation (the value is supposed to be in [0,1]; other values are permitted but then that should be clearly stated in the documentation of the component); the three latter are intended to permit a component to filter the annotations it wants to read. Therefore it is important to set these values carefully, even if you do not intend to use them, since some future components might expect them. In order to make the process of setting up new annotations easier, the following methods are provided in
`fr.lipn.nlptools.uima.common.LipnExternalProgramGenericAnnotator`:

```
public static void setGenericAttributes(GenericAnnotation a, int start, int end,
                                        String componentId, String typeId, String runId,
                                        double confidence);
public static void setGenericAttributes(GenericAnnotation a, String componentId, String typeId,
                                        String runId, double confidence);
public void setGenericAttributes(GenericAnnotation a, String componentId, String typeId,
                                 double confidence);
public void setGenericAttributes(GenericAnnotation a, int start, int end, String componentId,
                                 String typeId, double confidence);
public void setGenericAttributes(GenericAnnotation a, double confidence);
public void setGenericAttributes(GenericAnnotation a, int start, int end, double confidence);
public void setGenericAttributes(GenericAnnotation a, int start, int end);
public void setGenericAttributes(GenericAnnotation a);
```

All these methods are variants of the first one: they simply assign the provided values to suitable features in annotation `a` (first parameter). The first obvious advantage for a component tu use one of these methods is to synthetize all these assignments in one line, which makes the code more readable (and is faster to write!). The variants with one or more "missing" parameters always assign a standard default value to the "missing" parameter(s), which are:

- for `confidence`, `DEFAULT_CONFIDENCE`, which is defined as `Float.NaN`;
- for `typeId`, the annotation object class name (obtained via `a.getClass().getSimpleName()`);
- for `componentId`, `DEFAULT_COMPONENT_ID`, defined as `this.getClass().getCanonicalName()` (in other words the Java class name of the AE; notice that this is the actual annotator name, not `fr.lipn.nlptools.uima.common.LipnExternalProgramGenericAnnotator`);
- for `runId`, the value is supposed to be obtained dynamically as a component parameter that the user is able to define. Thus **this mechanism works only if**
  - the component descriptor declares a `RunIdValue` parameter (defined as optional and of type String), and
  - the common initialization method is called by the component: either
    `void initCommonParameters(UimaContext, String, String, String, boolean)`
    or `initCommonParameters(UimaContext, String, String, String)`;
    alternatively the component can manage by itself to read the parameter value and assign it to the `runIdValue` variable.

See also 4 and 5.1.3.

The `Begin/End` features must always be set; the variants without them are intended for components which need to assign values to them at a different time than the other features.

5.1.2 Reading annotations: iterators and filters

ThreadSafeFSIterator

All components in the LIPN framework are expected to read annotations only through the generic iterators provided. This is intended to make sure the components behave consistently with respect to concurrent annotations and annotations filters. Indeed, the mechanism used to deal with concurrent annotations works in the following way (see part 4.2):

- a component writes a concurrent serie of annotations in two cases:
  - it itself writes several series of annotations: it must then provide a way for further components to distinguish between different series;
  - it writes a serie of annotations which is in fact added to some previous concurent serie of annotations. In that case the component is "not aware" (in general) of concurrency: at the higher level (CPE), a parameter has been provided in order to make the component write annotations with a distinctive label.
- similarly, a component reads a concurrent serie of annotations in two cases:
  - it itself "knows" that there are different series and how to interpret these;
  - it reads a serie wich is actually concurrent to some other series. Once again, the component is in general "not aware" of concurrency: a parameter has been provided to make it read the right serie.

The first case in either reading or writing annotations is rather simple in terms of management of annotations: the component is responsible for its behaviour. However in the second case (which is in fact the more interesting and useful one) almost the whole task must be dealt with at a different level

5.1.3 Filters management

5.2 Other features/conventions

5.2.1 Language

Language[48] is represented using the international conventions (see 3.2): the two lowercase letters code for language (following ISO 639-1), possibly followed by a two or three letters code for country (ISO 3166), for example *en-US* for American English. The second code can be omitted (up to now no *lipn-uima-core* component differentiate between country codes).

In UIMA language is (can be) stored as a feature in the special instance of `DocumentAnnotation` associated with the CAS (cf. UIMA JCas API). Language can be set in (at least) three different ways:

- The *Collection Reader* component can set the language in the CAS depending on a parameter that the user gives (as in the UIMA standard `FileSystemCollectionReader`), or depending on some other information (for example read in the metadata if processing some known format of document).

[48]General localisation issues are currently not addressed in *lipn-uima-core* because the existing AEs make no difference, but the same kind of convention should be followed in this case.

- Each AE can have a language parameter which is also set by the user (normally this parameter is not written in the CAS)
- An AE responsible for detecting the language can be used, which should set the language in the CAS.

The prefered way to deal with language is to use the (standard) `DocumentAnnotation` in the CAS, either set by the *Collection Reader* or by a specific language detector AE. This option is simpler for the user, since she does not have to provide language as a parameter to all different AEs. Moreover it more flexible to use the CAS `DocumentAnnotation`, in particular by permitting that different documents (thus different CASes) can be in different languages.

That is why we consider that the use of a parameter specific to the AE should be interpreted as a strong constraint: if the user choses to set it for an AE in particular, then this choice overrides any other option. Thus the priority strategy of the *lipn-uima-core* AEs ti determine which language to use for a given component is the following:

1. if the AE defines a language parameter and the user sets a value for it, then consider this is the language
2. otherwise, look in the CAS `DocumentAnnotation` and consider this is the actual language if defined.

Then language is checked against the set of the possible languages of the AE.

We apply our "stop as soon as possible approach" (cf 3.5): the language (if defined as a parameter of the AE) is verified in the `initialize` method (and an exception may be raised here if it is defined but not valid), then we check it again in the `process` method in case it was not defined as a parameter but it is in the CAS (once again an exception is thrown if it is not valid). The first test in the `initialize` method is not necessary since it will be done a second time in `process`, but it should be emphasized that it is useful: supppose a CPE containing a chain of AEs in which A is at the end; if the language parameter for A is wrong and is not checked in the `initialize` method (which is called in the initialization step of the whole CPE) then all preceding AEs will be run in vain, and eventually the process will stop with an error when it reaches A.

5.2.2 Charset encoding

Normally the input charset encoding is a parameter of the Collection Reader (component responsible for reading the input and writing it to the CAS, see UIMA documentation). Then the textual data is stored in the CAS as UTF16 (as usual in Java) by the UIMA engine. Thus it is important to notice that the encoding issues concerning the wrapper annotators are totally masked to the user: he only has to provide the correct encoding as input, and not to care about the encoding used by the wrapped programs.

Thus a fair amount of work has been done to prevent charset encoding issues, so that the end-user should not have to bother with those *which are due to the external programs used*[49]. That means that each *lipn-uima-core* component must try to convert the input in the encoding expected by the external program (and convert it back, but this is usually easier) without restrictions on the original data encoding. This is not possible in general, nevertheless it is possible to make this process painless for the user, at least in the most frequent cases and for the most used encodings (ASCII, ISO8859-1 and UTF-8, UTF-16).

Of course problems can still happen, thus it is important to either correct them smoothly if required and possible, or to report them in a clear way (typically by providing the position where the error happens, see 6.2). In *lipn-uima-core*, in order to ignore invalid characters (see also 6.1.2), it is recommended to use the `replaceCharacterCodingErrorReplacemen` option, which replaces any invalid character with a custom defined one[50].

[49]The user still has to deal with the encoding of the input data which is provided: clearly problems will occur if the input is not valid w.r.t the given encoding.

[50]This choice is due to the fact that chosing the "ignore" option causes errors when re-aligning the annotated text w.r.t the reference text, because the former does not contain the wrong characters anymore but the latter still contains them. This problem is solved by replacing them with a specific value (generally chosen because it is a very unlikely one) which can be taken into account in the re-aligning process (see wildcards parameters in the related classes API).

5.3 Example: the TreeTagger wrapper AE

[TODO ...]

5.4 Components specific features

5.4.1 Yatea AE

The Yatea AE (more precisely the `YateaXMLOutputParser` class) deals with Yatea bug about wrong positions (see 7.0.5) in the following way:

- if both `CheckOccurrenceForm` and `CorrectYateaPosition` parameters are `true` (default): when reading a new term from Yatea XML output, `YateaXMLOutputParser` will check (for every occurrence) that it is indeed this term form which appears at the expected position (given by Yatea). In case there is no match, it will then try to recover the right position in the sentence, by comparing every token in the term with with every token in the source sentence. In case there are several occurrences for a given token, the one closest to the expected position is selected. *Warning:* this is a heuristics, and as such does not guarantee to provide the exact result[51].
- if only `CheckOccurrenceForm` is set, then there can be no error recovering: in case such an error happens, the AE raises an exception. The advantage is that if there is a result (i.e. no exception occured) it does not contain any error (which might be caused by the heuristics).
- if only `CorrectYateaPosition` is set, then the behaviour is the same as if `CheckOccurrenceForm` was also set (because it does not make sense to correct something if it is not checked before).
- if none is set, then the position told by Yatea is always trusted: there is no check so if the corresponding position is found the mapping it is annotated with it, no matter what the covered text is. However if Yatea is wrong it usually does not provide a valid position which exists in the mapping, therefore an exception is raised if this happens.

6 UIMA independent packages: module *lipn-nlptools-utils*

The very first part of implementation has consisted in designing, coding and testing a few packages which are responsible for interfacing with any external program in an "as robust and safe as possible" way. Here we only describe the two main packages in *lipn-nlptools-utils*, because the three others are simply not important. The first one, namely `fr.lipn.nlptools.utils.externprog`, is devoted to the environment in which the external program is called (see 6.1); the second one, namely `fr.lipn.nlptools.utils.align`, handles the output format issues (see 6.2). It is important to notice that both modules do not depend on UIMA[52], and can be used from any other Java program (see example classes `fr.lipn.nlptools.utils.testing.ExternalProgramTest` and `fr.lipn.nlptools.utils.align.TaggedTextAligner`).

6.1 Package `fr.lipn.nlptools.utils.externprog`

This package is intended to control the process of calling an external program, dealing with the possible problems (interruption, I/O error, possibly time out to avoid forever loop), raising suitable exceptions if needed. It is also

[51]Though I did not find such a case during tests.

[52]Actually the latter still depends on the *uima-core* package, but only because the UIMA `Logger` class is used (this technical point is not significative).

implemented in an efficient way concerning data transmission: whenever possible, instead of simply copying the data from memory to a file then copying back at the end of the process, the data is transmitted on the fly (using several threads), thus minimizing in the same time the time and space needed. The implementation was originally inspired by class `ProcessLauncher`, written by Fabio Marazzeto and Yann D'Isanto.

6.1.1 `ExternalProgram`: principle

The most important class in `fr.lipn.nlptools.utils.externprog` is `ExternalProgam`. It works in the following way:

1. Initialize an `ExternalProgam` object: the only mandatory parameter is the command line[53]. Additionaly several options are proposed: time out/working directory/charset encoding can be set using the detailed constructor, but several other options are available through different methods after initializing the object (e.g. setting environment variables, setting the behaviour in case of encoding error, etc.).
2. Call the `run` method: it will cause the process to start and will not return until one of the following condition happens:
   - the external process has finished;
   - running for too much time (if time out was set)
   - an error occurred (an `ExternalProgramException` will be thrown)

By default the external program is supposed not to expect any input stream and its output/error streams are simply stored in some String objects. These strings can be recovered after the `run` method has finished by calling `((StringReaderConsumer) getLastStd[out|err]Consumer()).getString()`. But it is often necessary (and better) to parameterize an `ExternalProgram` with some custom input/output objects, as explained below.

6.1.2 Providing input / recovering output of an external program

The `fr.lipn.nlptools.utils.externprog` package is particularly suited for programs using the `stdin/stout` streams for textual input/output. This feature is actually the main contribution compared to the Java standard `ProcessBuilder` class (`ExternalProgram` can of course still be used in the more simple case). There are two main issues in dealing with such I/O.

The first issue is only a technical one and the user does not need to care about it. It consists in taking a lot of care to send/receive the data, mainly because particular I/O errors can happen during the process. As stated in the Java API (see `java.lang.Process`): *"Because some native platforms only provide limited buffer size for standard input and output streams, failure to promptly write the input stream or read the output stream of the subprocess may cause the subprocess to block, and even deadlock."*. In other words, there is a deadlock problem if the external program writes some output to `stdout` and waits for it to be read before continuing, while the Java program which called this program waits for it to finish before reading the output. The case actually happens quite frequently when processing large amount of text as streams. This is the reason why it is necessary to run each part of the process in its own thread. Thus there is for example a thread responsible for providing the input stream to the program, and another one for reading the output stream that the program writes. This way no deadlock can happen, since input/output is processed "on the fly": for example the output stream reader thread will continue its task if the program stops, thus freeing the buffer and allowing the program to continue. The second advantage with this approach is that processing data on the fly is more efficient (see 3.3).

The second issue is due to the fact that transmitting textual data on the fly (using text streams) is a bit more complex than simply transmitting String objects references. Thus it requires some mechanism to interface between

[53]Be careful, the command line must be provided as an array of String objects representing the *tokens* of the actual command line. For example `{"myprog", "-a", "myfile1", "-b", "myfile2"}` is correct. It is not possible to use any shell specific feature like *pipe* or redirection, you must create an external script file and call it to do so. See Java API about `java.lang.ProcessBuilder` for more details.

the calling program and the external program. Since the package is intended for text files, `Reader` and `Writer` objects are used. In order to transmit such objects and to start reading/writing inside the corresponding thread (see above), the two following interfaces are provided, which are widely used in the package and in the *lipn-uima-core* module:

- An object which implements the `fr.lipn.nlptools.utils.externprog.ReaderConsumer` interface is intended to *consume* a `Reader` object. That means that it provides a method (`consumeReader(Reader)`) which will read on the `Reader` object until the whole data is read. It is typically called inside the output stream reader thread, thus permitting:
  - to customize the behaviour depending on the task planed with the output,
  - and not to wait for the end of the process to start processing the data (saving time and memory).
- In the same way, an object which implements `fr.lipn.nlptools.utils.externprog.WriterFeeder` interface is intended to *feed* a writer object. That means that it provides a method (`feedWriter(Writer)`) which will write on the `Writer` object all the data which has to be written. It is typically called inside the input stream writer thread, with similar advantages than in the `ReaderConsumer` case.

Several useful classes which implement these interfaces are provided: among others, `StringWriterFeeder` and `StringReaderConsumer` can be used to transmit/receive text as a simple String object (not recommended for large texts); `FromReaderWriterFeeder` and `ToWriterReaderConsumer` can be used to write the content read in a `Reader` to a `Writer` and conversely; the `BroadcastReaderConsumer` is a special `ReaderConsumer` which multiples the stream and sends it to several other `Readerconsumer` objects: in other words it acts like the unix standard `tee` program, and it is useful when one one needs both to store an intermediate result and process it normally (such an option is provided with most *lipn-uima-core* annotators).

Finally the features about charset encoding issues deserve a special explanation: firstly even if the default is to use the same encoding for the three streams (`stdin`, `stdout` and `stderr`), it is possible to specify a different one for each stream through accessors methods `[g|s]etStd[in|out|err]Encoding(...)`. Moreover it is possible to define the behaviour in case a charset encoding I/O error happens, using the Java objects `CodingErrorAction`: *"instances of this class are used to specify how malformed-input and unmappable-character errors are to be handled by charset decoders and encoders"* (from Java 1.6 API). This is useful: by default an exception is raised if (1) a invalid character is read w.r.t the given input encoding charset or (2) a character can not be mapped into the given output encoding charset when writing. But this is a typical frequent problem to process a dirty corpus containing a few invalid characters, and sometimes one simply wants to ignore them rather to stop the whole process (see also 5.2.2).

6.2 Package `fr.lipn.nlptools.utils.align`

Since each external program expects its input and provides its output in a particular format, the transmission of the data is also a critical point. The output direction (from the external program to the main process) is more complex because it is necessary to re-align the data with respect to the original input while taking new annotations into account. In the UIMA framework, annotations are stored outside the document data itself by referencing them with start/end indexes; but some tools use (XML) tags, some other ones use a tabular format (token followed by a set of annotations), etc. That is why a complete re-aligning solution has been implemented, in a way that offers as much flexibility as possible: a first objet is responsible for reading the annotated data, a second one reads the original data and matches the portions of text, and finally the third one has to store this information in the right way. This modular approach permits to easily combine each kind of object; for example, it is possible to read any kind of tagged text with the "tagged text reader" without knowing the content of the annotations, since interpreting this information is the task of the third object. Of course this simple approach consists only in good programming practices (modularity to avoid having the same code in different places), but it should be emphasized that such an approach is crucial concerning conversion issues: it is highly important to avoid the multiplication of small independents converters, which are frequent sources of problems and are a lot harder to maintain (of course a bug-free implementation can never be guaranted, but it can make the correction more or less easy - see 3.2).

6.2.1 Principle

Let consider the reference[54] text:

```
0    5    10   15   20   25
Peter (really) loves Nutella. (1)
```

Suppose a NE tagger is applied on this text and returns

```
0    5    10   15   20   25   30   35   40   45   50   55   60
<person>Peter</person> ( really ) loves <org>Nutella</org> . (2)
```

Notice that the tool applied did not only add some tags, it also added spaces around punctuation marks (it could also have added or removed line breaks, etc.). The goal of the `fr.lipn.nlptools.utils.align` package is to *re-align* (2) with respect to (1). It can be done in a lot of different ways, and have a lot of different output forms. For example the TagEN tools uses this package to re-align the output produced by Unitex (which changes whitespaces and/or line breaks), in order to provide a text output which is exactly the input text where only NE tags have been added.

In the framework of UIMA we will usually be interested in recovering the positions of the annotated text: in the above example we want to add two UIMA annotations, covering tokens *Peter* and *Nutella* in the original text. Thus we need to obtain positions 0-5 and 21-28[55]. This can not be done directly from the output text, since positions are altered both by the added tags and the fact that whitespaces have been added.

As shown in figure 2, the general principle in this set of tools provided in the `fr.lipn.nlptools.utils.align` package is the following:

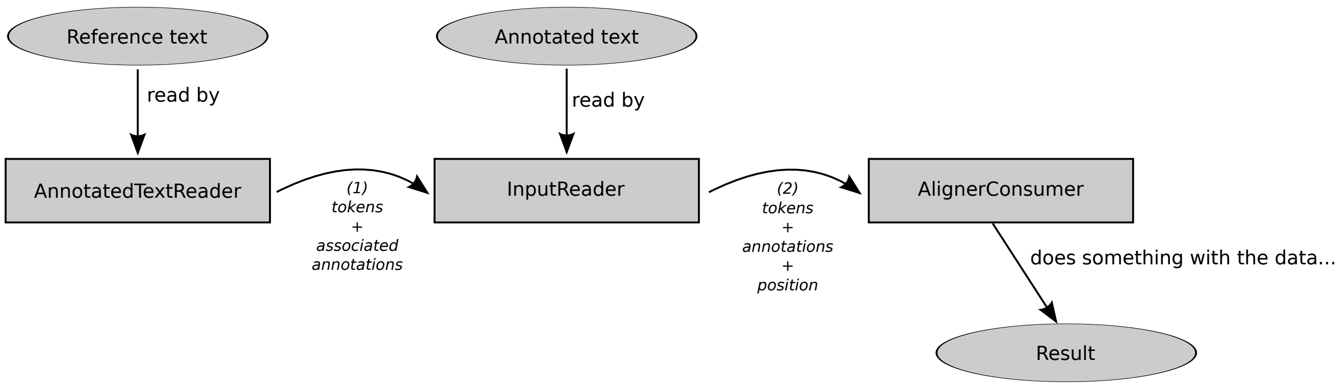
Reference text
read by
Annotated text
read by
AnnotatedTextReader
(1) tokens + associated annotations
InputReader
(2) tokens + annotations + position
AlignerConsumer
does something with the data...
Result

Figure 2: The fr.lipn.nlptools.utils.align package: general principle

1. An `AnnotatedTextReader` object is responsible for reading the annotated text and tokenizing it incrementally. It transmits each token and possibly the annotation(s) related to this token to an `InputReader` object.
2. An `InputReader` object is responsible for reading the reference (source) text and tokenizing it incrementally (or managing to recover positions of tokens contained in the text in any way). When receiving data from the `AnnotatedTextReader` object, it has to find the corresponding token in the original text and transmit the whole information to an `AlignerConsumer` object.
3. Finally an `AlignerConsumer` object receives the data and "consumes" it in some way: standard behaviours include writing the data into a file or in a String object, or writing it into the CAS in the UIMA case. There are different options about the form of the data it receives, depending on the need for all tokens or only annotated ones, positions of tokens, etc.

[54]Also named *source* or *original* in the implementation.

[55]The usual convention in String indexing about returning the last+1 character position as the last index is used (in such a way that end-start=length).

6.2.2 Usage

These three elements are provided as interfaces, except `AnnotatedTextReader` which is an abstract class because it "controls" the whole process: it contains the main method `align` which (normally) does not need to be overriden. This method simply calls the specific overriden methods in the other classes (see javadoc API for details). Several useful implemented classes are provided, among which:

- `SimpleTaggedTextReader` and `SimpleTokenByLineReader` are two `AnnotatedTextReader` objects. The former reads a text annotated with tags `<mytag>...</mytag>` (like in the above example). The latter reads a text annotated in the form of one token by line, followed by annotations, i.e.
```
Peter  person
(      -
really -
)      -
loves  -
Nutella       org
.      -
```
- `SimpleInputReader` and `TokenIteratorInputReader` are two `InputReader` objects. The former reads the source text from a `Reader` object (file, String or stream), and the latter expects an already tokenized text provided using an `Iterator` object.
- `TagAlignerWriterConsumer` and `TagPositionConsumer` are `AlignerConsumer` objects. The former writes the text together with its annotations to a `Writer` object, while the latter is intended to deal with positions (this one has to be instanciated with a `TagPositionConsumer.AnnotationReceiver` which is resposible to "do something" with the annotations and positions). Notice that contrary to the two previous classes the `AlignerConsumer` object is generally totally overriden, because its behaviour is very specific.

This modular approach permits to deal with different formats at any of the three levels in a similar way: for example it is possible to use any of the two `InputReader` implemented classes with any of the two `AnnotatedTextReader` implemented classes, and the same holds (more or less[56]) for `AlignerConsumer` objects. This way a lot duplicate similar code is avoided, and the behaviour of all components using these classes is consistent (see 3.2). Finally components using these objects only have to deal with their specific format issues, and do neither add duplicate code for the common parts of the process.

7 Troubleshooting

[TODO 1) regarder msg d'erreur complet (stack trace), 2) regarder log file 3) éventuellement changer le type de log et relancer ; distinguer erreurs à l'init et erreurs au process]

Here are listed several problems which can appear quite frequently with UIMA and/or the *lipn-uima-core* module, together with their solution. Unfortunately it is not possible to propose an exhaustive list!

7.0.3 UIMA general path problems

Path problems are very very frequent using UIMA, because of the descriptors system: where a classical Java program would simply include calls to some other packages/classes (and therefore missing classes would make the compilation fail with a clear statement of what is missing), with a UIMA program it is impossible to know whether something is missing at compilation time. Additionally:

[56]Of course meaningless combinations do not work: reading "token by line" annotations from the annotated text as "tags" annotations in the consumer will surely fail.

- there is no way to check that an XML descriptor is valid w.r.t the elements it refers to (other descriptors, Java classes);
- the modular policy implies that there are a lot of such references;
- the *"by location"* references become wrong as soon as something changes in the filesystem (file or directory);
- there can be confusions between references *"by name"* and *"by location"*.

Therefore errors occur quite frequently, but the following recommendations may help:

- Whenever possible, prefer **import by name** rather than **import by location**. *import by name* means that the UIMA engine will look for the descriptor in the `CLASSPATH`[57]: this is a lot safer, because this way all the dependencies are contained in the same place (usually in a JAR archive) that you specify in your CLASSPATH.
  - common error with *import by name*: do not write the `.xml` extension for the descriptors. UIMA engine will automatically look for the name plus its xml extension (and then will not find some `mydesc.xml.xml` file!).
  - common error causing a descriptor missing in the `CLASSPATH`: forgetting to copy the data directories (in particular `desc` containing the descriptors) in the Java output directory (usually `bin`). See 2.7 for how to do so depending on the compilation method.
  - when using the UIMA CPE GUI tool (provided with UIMA) to create CPE descriptors, the imports are always written as *by location* absolute pathes. This is not convenient at all and causes frequent errors once you change anything in the filesystem. Either do no use this tool at all, or edit the descriptor by hand after creating it (see also 2.6.1).
- If ever using imports by location, do not use absolute pathes in locations but relative ones, otherwise the system is broken as soon as the project is moved elsewhere.
- Have a clear strategy about where descriptors should be located, and what kind of import (reference) is used. When you perform changes in the `CLASSPATH` folders structure, do not forget that it can impact descriptors.

7.0.4 Wrapped programs problems

- **path problems**. if a CPE using *lipn-uima-core* complains at initialization step about not finding some file/directory/program, check:
  1. that the external program called is installed on your system;
  2. that it can be accessed by the way specified through the descriptor: the default structure is detailed in 2.3. If the default does not meet your needs, you have to specify explicitly the location(s) in the descriptor (check the parameters descriptions to understand exactly what should be provided).

7.0.5 Yatea bugs

There are several bugs in Yatea 0.5[58] (official current version). Below is a non exhaustive list together with some workarounds. Some of them are errors which are recovered by the Yatea AE provided in this package, but others are internal bugs: it is **strongly recommended** to apply the changes for the latter[59].

- Symptom: in some cases the following fatal error stops the Yatea process:

[57]or datapath (as far as I know this is the same in standard use). If you do not understand how Java works with `CLASSPATH` (this is not specific to UIMA at all), please refer to some Java official documentation about that. In particular let me recall that the `CLASSPATH` does not only contain Java classes, but can also contain any file, e.g. XML descriptors.

[58]Yatea does not seem to be maintained anymore: I reported these bugs to Thierry Hamon (the author) and did not receive any answer. As far as I know the official CPAN module has not been modified for a few years.

[59]The script `install-scripts/patch_yatea.sh` can perform these changes for you.

```
Can't locate object method "getFather" via package "Lingua::YaTeA::RootNode"
                  at [your-perl-modules-location]/Lingua/YaTeA/Node.pm line 1496.
```

Cause: missing parentheses in two places (lines 1496 and 1528, same condition) in `Node.pm` in the following condition (correction in red):

```
    if(
       (isa($node,'Lingua::YaTeA::InternalNode'))
&&
       ($node->getFather->getEdgeStatus($position) eq "HEAD")
&&

(


       (
        ($position eq "LEFT")
&&
        ($node->getRightEdge->searchLeftMostLeaf->getIndex < $to_insert)
        )
       ||
       (
        ($position eq "RIGHT")
&&
        ($node->getFather->getRightEdge->searchLeftMostLeaf->getIndex < $to_insert)
        )

)

       )
```

This bug is corrected by `install-scripts/patch_yatea.sh` (you have to provide the path).

- Symptom: in some cases the following fatal error stops the Yatea process:

  ```
  Can't call method "getIndex" on an undefined value at
                  [your-perl-modules-location]/Lingua/YaTeA/Node.pm line 2099
  ```

  Cause: missing test in the following condition line 2097 in `Node.pm` (correction in red):

  `if(defined($new_next) && ($new_next->getIndex > $index))`

  This bug is also corrected by `install-scripts/patch_yatea.sh` (you have to provide the path).

- Symptom: when no term at all has been found, Yatea crashes with the following error:

  ```
  Illegal division by zero at [your-perl-modules-location]/Lingua/YaTeA/Corpus.pm line 1299.
  ```

  Cause: no test for zero in division line 1299 in `Corpus.pm` (correction in red):

  `$mean_occ=(scalar(keys %$term_candidates_h)>0)?($total_occ/scalar keys %$term_candidates_h):'NaN';`

  This bug is also corrected by `install-scripts/patch_yatea.sh` (you have to provide the path).

- Positioning strategy (more a particular feature than a bug): In Yatea output tokens are indexed according to their sentence number and position in this sentence. However the position index is also incremented by 1 at each token encountered, as if tokens were separatated by one space[60]. The Yatea AE (more precisely the `YateaXMLOutputParser` class) deals with this indexing and recovers the original position in the text.

[60]Yatea is not provided with the actual corpus, its input is a TreeTagger file with one token by line.

- Errors in positioning: in some cases involving complex terms containing shorter terms, yatea will assign wrong position indexes. This happens where there are difference occurrences of the same term with different words orders. For instance, in a corpus containing both *"Pneumopathie lobaire inférieure droite infectieuse"* and *"Pneumopathie infectieuse lobaire inférieure droite"* (and maybe other variants), Yatea is able to recognize that they both refer to the same term. But the Yatea output format does not permit different forms for a single term, so they are both assigned the same form (the first one): so far this is not an error (maybe a questionable design at worst); however it becomes bad when shorter terms are contained in these terms : probably because their position is computed by Yatea using the longer term from which they are extracted *possibly using the wrong form*, they can be assigned shifted position indexes. In the case of the long term, this is not too bad since it is still inside the terms bounds, but positions of the term itself (if it appears outside the context of the long term) are also wrong. The Yatea AE tries to recover the right position (and usually succeeds): see part 5.4.1 for details.

7.0.6 Miscellaneous

- **Charset encoding**. Normally the AEs deal with most charset encoding issues, but errors can still happen in the following cases:
  - the input encoding is wrong or (more frequently) the input text actually contains invalid characters (w.r.t this encoding). Theoretically in this case an I/O exception should be raised by the Collection Reader (i.e. at the beginning of the process). But for some unknown reason (?) Java seems not to report all errors, so the process continues until another conversion is needed and this is often when *writing* the data into another encoding that the error is reported. Such cases were encoutered but we do not have real solutions for that; nevertheless being aware of that behaviour may help the user fixing problems.
  - the encoding expected by the program does not permit to represent some characters: this is an unavoidable restriction by the external program. If these characters can be ignored without serious consequences, the workaround consists in using the `replaceCharacterCodingErrorReplacementValue` parameter (which is provided with all wrapper AEs in *lipn-uima-core*).

  See also 6.1.2.
- the **UIMA Annotation visualizer**: if you do not see all annotations in the resulting files (or even no annotation at all), do not forget to check that you specify the right Type System descriptor to the visualizer (any type which is not defined in this TS will not be visible).
- **Java memory**: if the process stops with an exception related to lack of memory space, you need to specify more memory for the Java Virtual Machine, using the -Xms ans -Xmx options (see Java documentation). Be careful, when using a 32 bits architecture the JVM is usually limited to 1.5G to 3G.

[TODO pbm possible avec lia : -m 32, manque de mémoire]

8 Future work

LINA components: [TODO `http://www.lina.univ-nantes.fr/-Composants-UIMA-.html`]

- Continue to test the platform in various cases, possibly correct/improve code.
- Develop new / integrate existing annotators. There are different ideas that LIPN may study:
  - A term tagger, because it is a very useful component for LIPN,
  - A syntaxic parser, which would be useful of course but would also have another interest: it is indeed interesting to create/test concurrent annotations in a context where there are possibly different nested structures (due to syntactic ambiguities),
  - A customizable tokenizer, based on applying some rules defined by the user.

8.0.7 LIA `bin/lia_nomb2alpha` Numbers converter

Among LIA programs there is number converter named `lia_nomb2alpha`, which converts (digits) numbers into their expanded (letters) form. This program is supposed to be called after tokenizing the input text (as tokens and sentences), and before calling the POS tagger `bin/lia_tagg`. In particular it is used in the LIA NE named entities recognizer, and probably this NER performs better if this preprocessing step has been applied. It seems to work only for French, because there is no English corresponding resource. Here is an example:

```
echo "Parmi les 2397 personnes présentes le 25 avril 2008, 12 possèdent plus de 41,53 $." |
 $LIA_TAGG/bin/lia_tokenize $LIA_TAGG/data/lex80k.fr.tab |
 $LIA_TAGG/bin/lia_sentence $LIA_TAGG/data/list_chif_virgule.fr.tab |
 $LIA_TAGG/bin/lia_nett_capital $LIA_TAGG/data/lex80k.fr.tab |
 $LIA_TAGG/bin/lia_nomb2alpha $LIA_TAGG/data/list_chif_virgule.fr.tab
```

This command output is:

```
<s>
parmi les deux mille trois cent quatre vingt dix sept personnes
présentes le vingt cinq avril deux mille huit , douze possèdent
plus de quarante et un virgule cinquante trois dollars . </s>
```

Notice that this text is tokenized as follows (obtained by running the same command without the last line):

```
<s>
parmi les 2397 personnes présentes le 25 avril 2008 , 12 possèdent
plus de 41 , 53 $ .
</s>
```

Thus as one can see this tools has the particularity not only to *transform* the input text (contrary to simply *add* some information), but also to transform a given token into several tokens. In the next steps (e.g. POS tagging) these tokens will be considered exactly as original tokens. There are two issues with this behaviour:

- The first issue is: how is it possible to represent such a transformation in the CAS ? This question is not trivial, but there are approaches to solve it. In particular we could use our *concurrent annotation* mechanism (see 4.2: for example it is possible to use the `Interpretation` type to build an annotation containing the annotations concerning each such special element (typically it would contain a list of `Token` types, and probably `PartOfSpeech` types etc. later). Moreover that would be consistent with the general idea of concurrent annotation: seeing numbers in their expanded form is another point of view on these tokens, and it is a reasonably useful information to keep.
- The second issue is a simple technical problem: it is necessary for each UIMA wrapper annotator engine to be able to interpret the external program output, in particular to be able to locate exactly any information it receives. This is usually done using the alignment tools (see 6.2) which compare the source input text to the annotated text, and for each annotated token return its position in the source text to know where the added information should go. Considering several tokens at once is not a problem, as soon as it is always possible to know where the annotation starts and ends. Unfortunately this is where things go wrong here, because there can be any number of consecutive "old" tokens transformed in any number of "new" tokens. Therefore there is no way to know where the first token ends and the second begins, like in the example: without knowing exactly how the program works, it is impossible to tell which new tokens concern "`41`", which ones concern "`,`", etc.

One can notice that the problem would be solved if the program output had follow the original tokenization in any way, e.g. by providing the set of (new) tokens corresponding to a given (old) token on the same line. This is not the case, even if the tokens are provided one per line (to check call `$LIA_TAGG/bin/unmotparligne` before `bin/lia_nomb2alpha` in the previous example).

This is why it is (at least currently) impossible to integrate LIA `bin/lia_nomb2alpha` to the *LIPN UIMA Platform* platform.

9 Glossary

For UIMA terms and acronyms, please refer to UIMA official documentation. In particular, a convenient glossary is proposed in the "Overview and Setup" part[61].

- **Alignment** In *LIPN UIMA Platform*, refers to the re-alignment process needed when trying to integrate the annotations added by some external program into the source text. It is then necessary to find the exact corresponding part between the annotated text and the original one. The `fr.lipn.nlptools.utils.align` package is provided to do this task, see 6.2
- **CAS (Common Analysis Structure)** The main UIMA data structure containing the document data, the Type System and the annotations. See UIMA documentation.
- `CLASSPATH` The standard environment variable which contains the locations where Java looks for classes (and any other files). Since this mechanism is very important to understand how descriptors and classes work together in UIMA, refer to the Java official documentation in case you are not confortable with it.
- **Concurrency** Can refer to two very different points that should not be confused: the usual *"time" concurrency* refers to using different threads to process different tasks at the same time (see Threads/Concurrency). Whereas *annotations concurrency* (concurrent annotations) is about different sets of annotations existing in the same environment (see Concurrent annotations).
- **Concurrent annotations** Refers to the case where different sets (or series) of annotations which are independent are used in the same environment, typically the same CAS. In general these series have more or less the same "role" (e.g. different tokenizations of a text), this is why it may be an issue for an annotator not to confuse between the different series (see 4.2).
- **CPE** Collection Processing Engine. From "UIMA Overview and Setup": *Performs Collection Processing through the combination of a Collection Reader, zero or more Analysis Engines, and zero or more CAS Consumers. The Collection Processing Manager (CPM) manages the execution of the engine. The CPE also refers to the XML specification of the Collection Processing engine. The CPM reads a CPE specification and instantiates a CPE instance from it, and runs it.*
- **External program** A non-Java program called by a wrapper annotator using a system call (an equivalent to `Runtime.getRuntime().exec(String)`). A package is devoted to running such a program, see 6.1.
- `ReaderConsumer` **(interface)** An object implementing this interface is able to read the content of a `Reader` object when its `consumeReader` method is called. Used to read an external program output on the fly, see 6.1.2.
- **Threads/Concurrency** Running different threads to process different tasks at the same time. This is an issue if they share the same CAS (see 3.4)
- **Wrapper** A component whose main task is actually proceded by some sub-component which was not intended to be used in this context. Thus this sub-component has a "black box" behaviour, since there can be no communication between it and the main process. See 3.
- `WriterFeeder` **(interface)** An object implementing this interface is able to write some data to a `Writer` object when its `feedWriter` method is called. Used to write the external program input on the fly, see 6.1.2.

[61] `http://uima.apache.org/downloads/releaseDocs/2.3.0-incubating/docs/html/overview_and_setup/overview_and_setup.html#ugr.glossary`