# A Platform for Named Entity Processing

Jens Illig, Charles La Rosse, Iliana Simova, Dominikus Wetzel
Supervisors: Dr. Günter Neumann, Bogdan Sacaleanu

Software Project: Named Entity Processing

Department of Computational Linguistics, Saarland University
Deutsches Forschungszentrum für Künstliche Intelligenz (DFKI), Saarbrücken

May 11, 2011

**Abstract**

This report describes in detail the design and implementation of a system for Named Entity Recognition. It uses a voting strategy to combine the results produced by several existing NER systems (OpenNLP, LingPipe and Stanford), aiming at reducing the amount of errors produced by them individually. The system's architecture is based on the framework of OSGi - a Java service platform and module system, which offers flexibility in terms of component management. The project can be run as and accessed via a web service and comes with a graphical web user interface.

# Contents

# 1 Introduction

Various Open Source libraries for natural language processing are publicly available. Among them are several Java implementations for the Named Entity Recognition task. However, even though these libraries all provide implementations of the same task, they lack a common interface for it. Systems that use several libraries for Named Entity Recognition can be useful, since they allow running experiments and applications with different implementations in the same system environment. Especially meta-component approaches like voting or stacking as presented in [3] are interesting applications requiring several implementations of that task in one system. But also preprocessing of Named Entity Recognizer systems like for instance tokenization or sentence detection are offered in various implementations by the libraries. Bringing these together allows new setup possibilities which mix implementations of several libraries in one application. For example, a tokenizer of library A might be used by a named entity recognizer of library B if the components of all libraries can represented by a common interface. Additionally, multiple individual Named Entity Recognizers in a meta recognizer could share common preprocessing steps.

Such systems, that define clear interfaces for particular tasks and subtasks with dependencies among them, can be described as so called Service Oriented Architectures. In such an architecture, tasks are represented as services which are defined and made available via precise interfaces. Service providers are instances of service implementations and service consumers depend on the availability of other services. In Java software development the OSGi framework specification is one example following such an architecture. In addition, OSGi allows to start, stop, add, and remove service implementations during runtime and extends jar meta-information by standard attributes. A very successful and prominent application based on OSGi is the Eclipse development platform.

In this project, Named Entity Recognition and several preprocessing tasks have been unified to common interfaces which are used as OSGi services definitions. Implementations for LingPipe, Stanford and OpenNLP have been exposed as service implementations and a voting based meta named entity recognizer has been added on top of those services. All the services are made accessible via a simple but flexible and fully OSGi compatible web-based user-interface.

# 2 General Architecture

## 2.1 Framework Architecture

The Named Entity Recognizer Platform has been developed as a service-oriented OSGi API with an HTML web frontend. This means the functionality of the application has been split into several jars with special attributes in their `MANIFEST.MF` self-description. These attributes tell the OSGi framework about the content and the dependencies of the jar and thereby make an OSGi bundle jar out of otherwise regular jar files.
The main application i.e. the place where the main-method resides is in the OSGi framework code. OSGi then loads and starts the application bundle jars together with bundles for further services. Most notably, there is a HttpServlet container called *Jetty*. It offers services to basically do, what is usually done in a `web.xml` file of a regular standalone application server like the standalone Jetty or Tomcat. Servlets are registered by method calls on that service in the java code of web frontend bundles. The web frontend bundle in the Named Entity Recognizer Platform project is `webui`. The particular class registering servlets and JSPs is `de.uni_sb.coli.nep.webui.NerWebUi`. This class is itself instantiated as a service component. Service components are instances of classes that implement a service interface. In the platform project, these objects are instantiated, registered, and autowired with one another via an OSGI mechanism called Declarative Services. That mechanism uses a `MANIFEST.MF` property to refer to several declarative service XML files that each instantiate an object consuming and/or exporting a service. For more Information on OSGi and declarative services see [4] and [1].
For the web frontend a simple Model 2 approach has been followed. That means a Servlet reads from the HTTP request, performs necessary API calls and prepares the data for the response which it stores in a java bean in the request scope. It then calls a renderer, which usually is a JSP page rendering the prepared information to an HTML page. Alternatively, for XML file downloads the content is rendered by java code instead of a JSP page.
Configurations listing all the bundles required to run the Named Entity Recognizer Platform as a web application are available as described in chapter 5.

## 2.2 Bundle Separation

Bundles in OSGi are modules containing classes and other resources in the java classpath that can dynamically be added and removed to and from the classpath of several classloaders inside a running OSGi container. Each bundle developed for the Named Entity Recognizer Platform project corresponds to one eclipse project.
It is a common practice to split interfaces from implementations into separate bundles to allow for changes of the implementation while the interface remains the same. This approach has also been followed in the Named Entity Recognizer Platform project. The central API is in a bundle called `nerinterface`. Other bundles refer to this interface without having to be dependent on particular implementations while accessing interface functionality. However, this independence has not been followed to the full extend, since objects are still instantiated directly with the `new` keyword. Because all all other bundles access the newly created object only by interface methods any implementation of that interface is appropriate for instantiation. Because usually the same implementation can be used in different bundles, there is also a bundle project for common implementations called `nerimpl`.
These are the most central bundles other bundles depend on. See figure 1 for an overview of all the bundles that are directly part of the Named Entity Recognizer Platform project[1].

[1] Bundles that do not depend on any classes or interfaces of the Named Entity Recognizer Platform project are not part of figure 1. Most notably the OpenNLP bundle uses other bundles that are not dependent on the platform API. Of course, also all the framework bundles like Jetty belong this category of independent bundles.

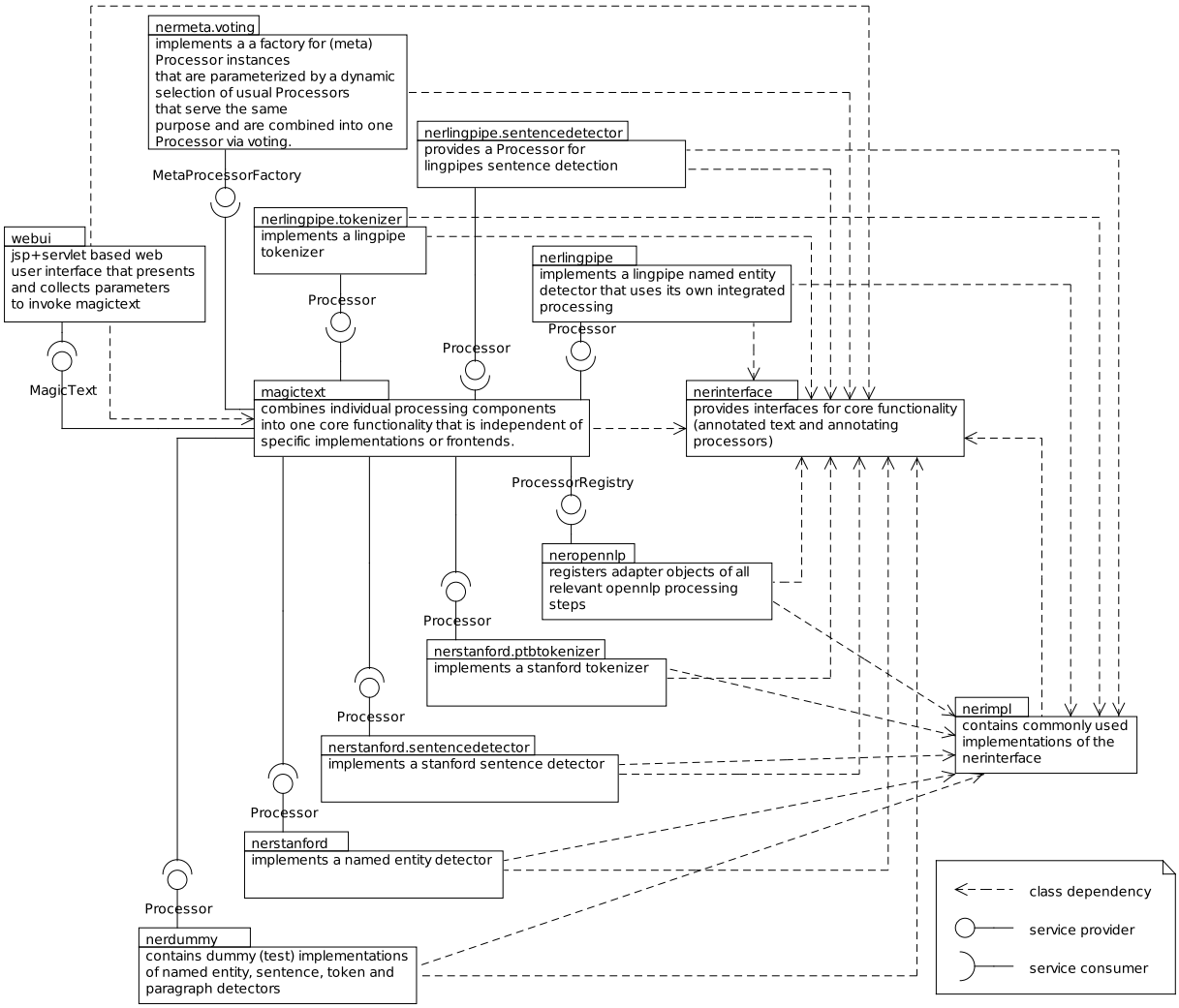


Figure 1: OSGi bundles depending on the Named Entity Recognizer Platform

## 2.3 General Interface Design

The Named Entity Recognizer Platform project models several (pre)processing steps. These are:

- Paragraph detection
- Sentence Detection
- Tokenization
- Named Entity detection

All those tasks are captured with the same interface called `Processor`. This models all the mentioned tasks as a process of splitting a larger piece of text into smaller pieces of text. The main functionality of a Processor is to take an `Iterator` of inputs (potentially intermediate results of previous `Processor` invocations) and transform N input elements into M output elements. This is a more general representation of the splitting task, While splitting always takes one input and produces M output elements, it is also possible to rejoin

them into larger objects by reading N inputs and for example returning only one output element for these N input objects.
It is also possible to implement other processing steps than mentioned with this interface. In order to allow dynamic discovery and presentation of available processing steps, the `Processor` interface is also self-descriptive i.e. it offers methods that define the name of the task that the `Processor` implementation performs. For a more detailed description of the interface and the steps implementations should perform see section 5.4.

## 2.4 Configurable Processing Chains

OSGi allows loading and unloading of different service-implementing bundles in a service oriented architecture.
One advantages one might see in such runtime-exchangeable modular bundles with different implementations of the same interface is the possibility to setup different constellations of chained processing steps. This for example allows to change from a processing chain that solely uses opennlp components to a chain that uses the same components as before but instead of the opennlp sentence splitter uses the stanford implementation of a sentence splitter. This would, by means of pure OSGi, be realized via stopping the bundle that provides the opennlp sentence splitter service while starting the one with the sentence splitter component of the Stanford NLP package. This, while allowing for any configuration of `Processor` component implementations, would on the other hand require that the change of components has to take place by a system administrator with system-wide consequences. The approach followed throughout the named entity system described here is to also allow the enduser to select a configuration from a set of available `Processor` components without affecting the configurations of other users. This is particularly important because server-side applications like http servlet application servers are multi-threaded, multi-user systems. This per-request selection of `Processor` service implementations is achieved by collecting all active services in the service consumer and presenting them to the user-interface (this is done in the MagicText class). This approach combines the benefits of OSGi runtime bundle exchange with flexible per-request processing chain configuration. Activation and deactivation of an OSGi bundle that provides `Processor` services then not results in an immediate system-global component reconfiguration but instead dynamically changes the available components from which the user can select.

## 2.5 Streaming

The system architecture has been designed with the capability of data-streaming in mind. Data streaming is an architecture style that allows to process a theoretically endless amount of data by not storing data for the whole input in memory, but only input data and corresponding internal model representations of a certain context around the current reading position.

A very simple example of a system that is incapable of processing its input as a data-stream is a program that first completely reads a textfile into memory and then outputs all the data it read to `System.out`. Such a system is limited in the size of the file it can process. The limit is defined by the amount heapspace memory assigned to the executing java virtual machine. In contrast, a stream capable implementation sequentially reads chunks of a limited size into memory and outputs them. Since a chunk is – once it has been written – never needed again, the memory for one data-chunk can be deallocated before the next chunk is read. The system then never needs to have more than one data-chunk in memory and is therefore able to efficiently process a potentially unlimited amount of data.

The named entity processing platform interfaces use the standard java interface `Iterator` to support streaming. By returning `Iterator` instead of a `Collection`, not all results (e.g. detected named entities)

must be stored before the first entry is read from the `Iterator`. Instead, the returned `Iterator` implementation can itself be intelligent in terms that a call to the `next` method might trigger the detection of the next entry (a paragraph, a sentence, a token, or a named entity) as such instead of simply iterating through a precalculated `Collection` of results in memory.
Memory efficiency is especially important with respect to the exponential growth of nodes in a tree structure with every layer of the tree. In the case of named entity processing, this structure begins with a root node (a `Reader` on the input) and contains paragraphs, sentences, and tokens or finally named entities. To allow garbage collection of already processed subtrees, a node in the tree (corresponding to an instance of `NerResult`) does not hold a list of its child nodes but instead only has a reference to its parent (`null` in case of the root node). The result `Iterator` of the `Processor` chain gives access to the leaf nodes of the tree. If all leaf nodes of a subtree are traversed and written to e.g. the HTML frontend without storing the leaf nodes, then the subtree can be garbage collected since a reference to the object graph is no longer present.

While the main architecture (`Processor` chains) allows streaming, not all implementations really need to support it. Precalculating results into a `Collection` container and afterwards returning an `Iterator` on that is still possible.

## 2.6 Processing Order

The different processors such as tokenizers, sentence detectors, named entity recognizers, etc. have to be applied in a certain order. In theory this order is not set into stone and can be changed in the code or made available for change on the user side. Furthermore, the order doesn't have to be followed strictly by the individual processors. We have designed a strategy for that which is explained in Section 2.7. In any case, an order needs to be specified and we have made the decision to keep it along the lines of the hierarchical structure from the entire document down to individual tokens. The exact processing order on which we have decided is: paragraph detection (`paragraph`), sentence detection (`sentence`), tokenization (`token`) and then named entity recognition (`name`).

As mentioned above, the order is currently set in the constructor of the backing bean of the web user interface, i.e. in `de.uni_sb.coli.nep.webui.NerBean`. A better way of handling this is described in Section 5.6.

## 2.7 Preempting

In most of the NLP packages, the different processing steps such as tokenization, sentence detection, etc. have to follow a certain processing order which can be quite inflexible. This is usually not a big problem, as the different processors within the package are designed to work together nicely. However, we are faced with the problem that we have several processors of different NLP packages each with their own processing order requirements in addition to specific input and output requirements.

Since the user shouldn't be limited in any way when it comes to selecting and combining the different processors, the normally fixed processing order of one NLP package has to be tackled in a more flexible way. For example a sentence detector of one package might require tokens as input, and thus tokenization has to be performed first. Another sentence detector might ask for plain text as input and thus, tokenization has to be done afterwards.

Since the processing order for our task was defined along the natural hierarchical structure of preprocessing tasks (cf. Section 2.6) the sentence detector of e.g. LingPipe and OpenNLP wouldn't fit in there. Thus we came up with a preemption mechanism, that allows the otherwise passive processing bundles to actively call other processors further down the pipeline. This way the above sentence detectors can actively call the tokenizer and work with tokens.

In order to avoid duplicate work, the succeeding processors of a preempting processor shouldn't perform the same computations again. This means that the tokenizer shouldn't tokenize the input again, after the sentence detector has done its job. For this purpose, we store the results of the preempting call to the tokenizer in a result list of the incoming NerResult. In our example this would be a paragraph. One important step for each processor is to do a check whether it has been preempted or not. This can easily be done by checking if the list which stores the preempted results is empty. If yes, then the processor just performs its task, and if it is not empty, then it simply returns the stored result list.

In our implementation it is only possible to do a first order preemption. This means, that a processor can preempt exactly one other processor's task. This limitation is due to the fact that only one list of results of the preempted processor can be stored within a result, which was sufficient for our platform. However, this could be changed very easily by allowing to store a map of processor type to result list and thus in theory it would be possible for a processor to completely circumvent the given processing order.

There are some pitfalls when it comes to implementing a preemptable processor and a processor which can do preemption. In order to avoid these, they are explained later in the development section about how to add new processors (cf. Section 5.4).

## 2.8 Result Hierarchy

The result hierarchy can be defined in a very flexible way. Furthermore, it isn't defined explicitly anywhere, it is rather a design decision that is made for a specific purpose, which has to be obeyed by the individual processors involved. Again, since this hierarchy isn't defined explicitly, it isn't checked whether the processors follow the design decision. It is up to the person who implements a new processor to provide properly set result objects with respect to the hierarchy, as other processors expect a certain behaviour for the input – just as the new processor will.

For our purpose, i.e. the processing chain of paragraph detection and sentence detection, tokenization and named entity processing, we have made the following hierarchy decisions:

- `document` immediately dominates (*id*) `paragraph`
- `paragraph` *id* `sentence`
- `sentence` *id* `token`
- `sentence` *id* `name` (named entity)

It is vital that all implementations of processors obey this hierarchy. That way consistent behaviour can be ensured.

## 2.9 User Interface

The user interface is divided into three sections: for providing the input text from which NEs are to be extracted, selecting the different preprocessing and NER tasks, and the format in which the output will be presented.

NerWebUI offers two possibilities for specifying the input. It can be provided in the form of a plain text file, or directly entered in a text field. The plain text file is expected to have UTF-8 encoding. The same applies for the text field - any text which is entered there is converted to UTF-8.

There are four different options for displaying or returning the result. The first two of them (see below) involve displaying the original text with some markup denoting the NEs which were recognized on the web page itself. Therefore they are more suitable for smaller texts.

1. text with markup - NEs are highlighted in different colors with the help of XSLT transformations. The actual NE label can be seen when the mouse cursor is over the highlighted span.
2. inline XML - NEs are marked with the inline XML tags <*label*>, which contain the attribute "value" to denote the type of NE this label represents, for instance: <*label value="person"*> *John Smith* </*label*>.
3. XML file - an XML file offered as download, containing the original text represented as described in 2.
4. table with indices - a table containing only the NEs found in the text with their corresponding begin and end indices and the name of the named entity processor which produced this output. The indices are relative to the sentence in which the named entity occurs. The start index is inclusive, the end index is exclusive.

The middle panel of the GUI provides options for choosing processors for the different tasks: paragraph detection, tokenization, sentence detection, named entity recognition, and voting (Meta processor). Typically only one processor can be selected per task. In the case of named entity recognition, one or more options can be selected, but when only one NER is chosen, no voting is performed.

## 2.10 Running Example

This section aims at demonstrating the main workflow of the system with a practical example, namely what happens with a sample input text in the different processing stages.

Example settings:

- Input text: *John studies at Saarland University. Marie lives in Saarbrücken.*
- Processors: paragraph = opennlp, sentence = lingpipe, token = stanford, name = lingpipe, stanford, opennlp_combined_en[2]
- Meta processors: voting

The input gets passed from the user interface to MagicText and wrapped in a NerResultImpl at the level "document" with begin and end indices spanning the whole input. MagicText then calls the different available processors (*paragraph*, *sentence*, *token*, and finally *name*) with it.

The wrapped input is first passed to the three components performing text preprocessing tasks. Paragraph detection, sentence splitting and tokenization processors each generate an intermediate NerResultImpl representation encoding the additional information they provide. All of these results are combined in a hierarchical structure (see 2.8), which gets passed to every next processor.

For this particular example, the intermediate results generated by the preprocessors are displayed in Figure 2.

The first processor which is called, the OpenNLP paragraph detector, returns only one paragraph, due to the small text example. LingPipe sentence detector recognizes the two sentences, whose NerResultImpl *parent* is set to the paragraph. Finally, the tokens found by the Stanford tokenizer are assigned to their corresponding parent sentences in the hierarchy.

Each NerResultImpl in Figure 2 is additionally described by two sets of indices indicating its position in the text. The first set denotes the position with respect to its parent in the hierarchy, and the second - to

[2]Our OpenNLP NER bundle has a slightly different approach of providing access to it's functionality to the user. You have the choice of selectively recognizing entities, or recognizing them all at once as the other NER bundles do by selecting opennlp_combined_en.

the original input. This is visible at the token level, where due to the fact that the first six tokens belong to the first sentence, and the rest to the second, the two sets of indices describing the tokens in the second sentence differ.

After calling all of the preprocessing components, MagicText calls the meta processor of type *name* in nermeta.voting. This processor is responsible for named entity recognition, more specifically, calling the actual named entity recognizers, collecting the results from them and voting on the final result. The input NerResultImpl which it receives from MagicText already contains all the information provided by the paragraph, sentence and token processors. The different levels of the hierarchy can be used by each NER individually, based on the kind of input it accepts.

The result produced by the three NERs and the final result after voting can be seen in Figure 3. The two example sentences illustrate the advantage of using the voting strategy, since in this case it manages to avoid all of their individual errors and produces the desired output.

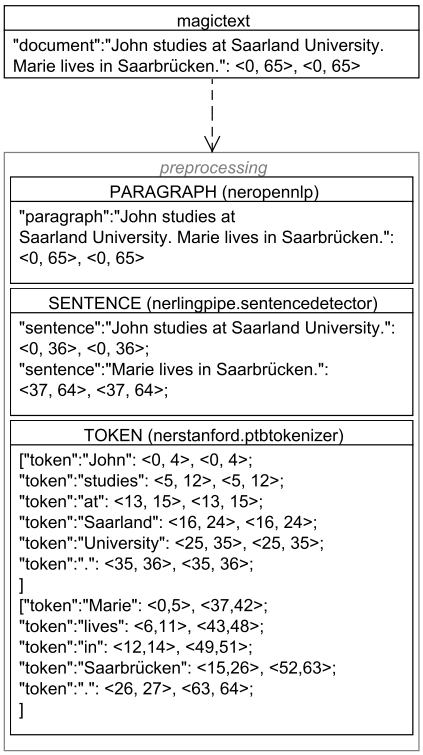


Figure 2: Intermediate results of text preprocessing

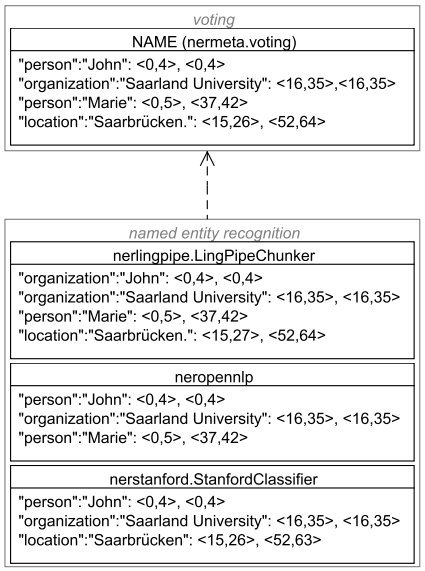


Figure 3: Intermediate results of NER and voting

# 3 Preprocessing and Named Entity Recognition Implementations

In this chapter, we introduce the three different NLP packages LingPipe, OpenNLP and Stanford NLP[3] which we have used to integrate in our platform. It describes the specific classes and models that we have used from those packages as well as give relevant implementation details. Furthermore, the bundle dependencies are mentioned. For more information and a detailed description of bundle implementation you should refer to Chapter 5.

## 3.1 LingPipe

### 3.1.1 Named Entity Recognizer

#### General Notes

There are several named entity recognizers (NERs) provided in LingPipe such as a rule-based recognizer which works with regular expression matching or a dictionary based recognizer. For this project, however, we have used a statistical NER, in particular a Hidden Markov Model based recognizer. LingPipe provides default models for this recognizer and we have decided to use `ne-en-news-muc6.AbstractCharLmRescoringChunker` which handles the named entities `PERSON`, `LOCATION` and `ORGANIZATION` for English texts.

The input of this recognizer is a String which can be any kind of text from the entire document to a single sentence. It could also provide just a few words of a sentence, however providing tokenized text is not possible. This fact has to be noted as any preprocessing such as paragraph or sentence detection and tokenization will be ignored.

The recognizer uses its own tokenizer which is integrated in the model, i.e. it is the same tokenizer with which the model has been trained. Furthermore, the recognizer does neither paragraph nor sentence detection which means that it will work with the obtained tokens directly.

As result, the recognizer returns a first-best list of named entities (NE) in the way we have used it. The elements of the list provide access to the NE label, the start and end indices and a confidence value. The latter is catered for in our platform but at time of writing never used. This means you will have access to the confidence value in the platform result object. Since it is not straightforward to compare confidence values or probabilities from different NLP packages or models a lot of care should be taken when it comes to comparing these scores.

#### Implementation Details

LingPipe NE recognition in our platform is designed to recognize entities given a sentence as input. Even though, preprocessing results are ignored by the actual recognizer, the bundle providing the NER service implementation gets token results as input. In order to perform recognition, the parent of the current result token, i.e. the sentence result, is obtained. This sentence is then given as input the chunking method. When the next available token result is read, a check is performed to avoid processing the same sentence multiple times.

The bundle also implements the interface `LabelSupporter` in order to provide information whether a named entity label is supported or not. This is defined as follows: if the corresponding label weight is not existent or if it is 0.0, then `false` is returned.

The way the processor is implemented is such that preemption is ignored. This means that if a previous processor preempts NE recognition and stores the results in the preempted list, the LingPipe bundle will

[3] http://alias-i.com/lingpipe/, http://incubator.apache.org/opennlp/ and http://nlp.stanford.edu/

perform the same recognition again while completely ignoring the results in the preempted list. However, it is trivial to implement a check to simply return existing results in the preempted list, instead of recognizing the input again.

#### Configuration File

In the root of the bundle, there is a configuration file `config.xml` which is used to declare the mapping between LingPipe specific named entity labels and our generic platform labels, as well as defining weights for each of the supported labels. The entries are organized as key-value pairs.

The label mapping is from LingPipe specific (key) to platform generic (value) labels. Weights have the generic label as prefix followed by a suffix `.weight`, e.g. `person.weight`. If label weights are set to 0.0, then they are not and should not be supported by the LingPipe named entity model.

### 3.1.2 Tokenizer

In order to have a bigger choice of processing components for tokenization, we provide the functionality of the same tokenizer (i.e. `com.aliasi.tokenizer.IndoEuropeanTokenizer`), which is integrated in the default NER model, which we have used. It is important to note, that even though the same tokenizer is provided as a processing step before the NE recognition, the LingPipe NER will still perform tokenization again. The input to the tokenizer is a sentence, the output is a list of tokens as NerResults. Furthermore, it is capable of making use of preemption, by returning an instance of `AbstractPreSplitIterator`.

### 3.1.3 Sentence Detector

The same integration issue with the LingPipe NER for the tokenizer (cf. Section 3.1.2) holds for the sentence detector, i.e. it is provided to have a bigger choice of sentence detectors, but the NER bundle won't make use of the sentence boundaries. The decision which LingPipe sentence detector to provide in the bundle was arbitrary and fell on the `com.aliasi.sentences.MedlineSentenceModel`.

The sentence detector requires a list of tokens, however in our processing chain, tokenization follows sentence detection. Therefore, the implementation of the sentence detector in our platform needs to make use of the preempting strategy, which calls the tokenization on its own initiative and stores the results in a list for later retrieval.

The exact input which is required is a list of tokens and a list of the whitespace characters which were found between each token. Thus, the length of the whitespaces list is one more than the list of tokens. The first element of the whitespace list is set to the empty String. All other whitespaces are reconstructed via the parent of the tokens, i.e. sentences containing the token.

The detector bundle is itself preempting capable by returning an instance of `AbstractPreSplitIterator`.

### 3.1.4 Bundle Dependencies

All of the LingPipe bundles have to import the bundles `de.uni_sb.coli.nep.nerimpl` and `de.uni_sb.coli.nep.nerinterface` via the MANIFEST.MF file. The NER bundle also requires the bundle `org.apache.log4j` with version 1.2. This last requirement however is only for exemplary use in order to demonstrate how to make use of the logging bundle. In fact the NER bundle doesn't really write any important log messages.

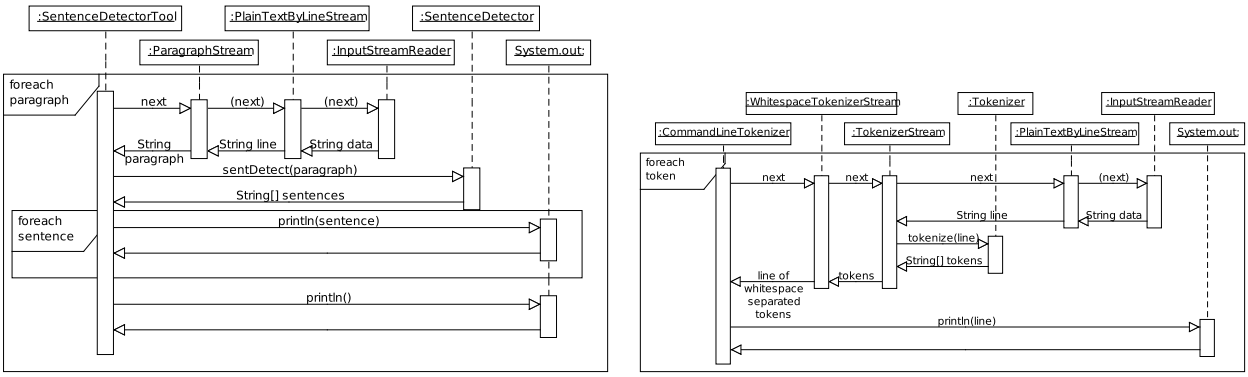


Figure 4: Partial stream architecture of OpenNLP commandline sentence detection (left) and tokenization (right)

## 3.2 OpenNLP

OpenNLP is a java based library for various different natural language processing tasks. For this project paragraph detection, sentence detection, tokenization and named entity recognition are used. For the latter three there are different implementations available in the OpenNLP library. There are regular expression based versions and statistical variants. For the named entity platform, statistical *Maximum Entropy*-based versions for all three tasks used. Paragraph detection is a simple static algorithm looking for more than one newline.

### 3.2.1 Plain OpenNLP Named Entity Detection Process

With an out-of-the-box OpenNLP library, full named entity recognition with all necessary preprocessing steps can be performed via the commandline:

```
bin/opennlp SentenceDetector models/en-sent.bin < text |
bin/opennlp SimpleTokenizer |
bin/opennlp TokenNameFinder models/en-ner-person.bin models/en-ner-location.bin
```

`bin/opennlp` is a shell script that starts the java application `opennlp.tools.cmdline.CLI` which in turn runs the command given in the first argument.

With argument one set to `SentenceDetector`, a combination of paragraph splitting and sentence detection is performed on the data and sentences are put out one per line followed by an empty line at the end of a paragraph. This output is piped to the standard input of another JVM running `SimpleTokenizer` which then outputs whitespace separated words per sentence line line to feed it to a `TokenNameFinder` JVM process. However, Java code that combines all the steps in one method without having to run several JVM operating system processes with disk access is NOT provided by the plain OpenNLP library.

The three commands read from standard input and write to standard output. Their The basic control flows of the processes mentioned above are shown in figures 4 and 5.

It can be seen from the next() method calls in the sequence diagrams that the preprocessing commands mentioned above partially use a streaming architecture. However, a closer look at the available classes reveals that this architecture is not complete in terms of a lacking possibility to set up one single chain of stream components with all necessary detector invocations glued together in object structures. The result is, as already mentioned, that several JVM processes on the operating system level and disk access is necessary.

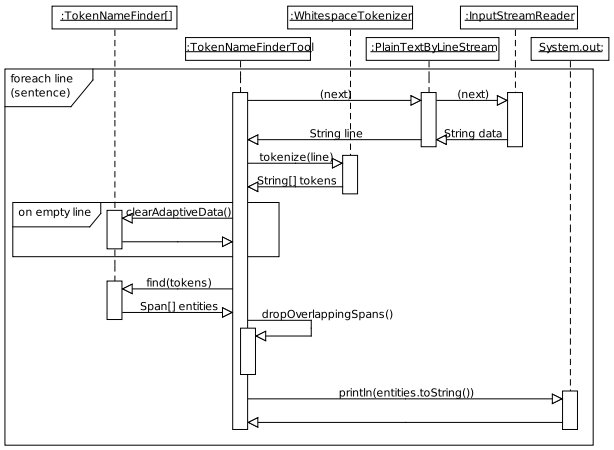


Figure 5: Partial stream architecture of OpenNLP commandline named entity recognition

Besides the efficiency drawback of multiple necessary JVM initializations, an essential requirement of a combined named entity recognition platform is also not met by of-the-shelf OpenNLP named entity processing. This requirement is the definition of detected substring labellings by start and stop indices on character level. Labellings - typically tokens, sentences, paragraphs or a named entity types - are instead defined by an *n-th input item* addressing scheme (for example: named entity N starts with token A and end ends with token B). In order to get back to absolute coordinates on character level one would need beginning end end coordinates of the input items. Plain OpenNLP, however, drops the original input text already in the first JVM process and outputs only normalized lists of result items. From there on, it is for all subsequent processes unclear how positions in an intermediate output correspond to absolute positions in the original input.

Substring range definition on character level is used as a common addressing scheme shared by all processing components registered to the named entity detection platform. Indices like *n'th token* are problematic since the final highlighting of input data requires absolute indices.

### 3.2.2 The OpenNLP Extension Library

An extensions around OpenNLP has been build that adds required value-adding features to OpenNLP. The main features are described in the following paragraphs of this section.

**A model for substring range definitions based on character level** The OpenNLP extension models labeled substring via a range object starting from character index A in a string to character index B. Interfaces of this model are defined in the `opennlp.extapi` bundle.
Model objects are entirely accessed throughout the OpenNLP extension code via interfaces and instantiated via factories to allow a replacement of model implementation classes via object structures without modification of the logic code that operates on the model.

**Independence from the platform and any other project or library** The OpenNLP extensions have been designed as an independent library project that does not have any compile or runtime dependency on the *Named Entity Processing Platform* project. Vice versa, the platform project has no dependency on the OpenNLP extension project. This complete independence is achieved by a glue bundle `neropennlp` that

depends on the APIs of both projects and provides a combined java model which implements the model interfaces of both projects. Via factory-based model instantiation in the OpenNLP extension project, the integrated combined model can transparently be used inside the OpenNLP extensions without any unwanted dependency on the classes of the combined model.

**A complete stream processing chain for named entity detection and required preprocessing steps** Since the individual commandline processes provided by OpenNLP partially use a streaming architecture internally as seen in section 3.2.1, in this respect only the necessary glue chain elements were missing. This streaming architecture has been completed in the OpenNLP extensions library.

**OSGi compatible bundling without redundant class loading** Six OSGi bundles have been created to achieve independence from frameworks and between interfaces and implementations. These are depicted and explain in figure 6. The core extensions are split into an interface and an implementation bundle. The interface bundle has no dependencies at all, and the implementation bundle only depends on a compatible OpenNLP OSGi bundle. this way, extension bundles are kept separate from the bundle containing the out-of-the-box OpenNLP library to allow separate versioning (including updating) as well as the option for other bundles in the OSGi container to access plain OpenNLP classes without loading these again in a separate bundle-classloader.
Since plain OpenNLP is not itself provided as an OSGi bundle, an OSGi wrapper bundle called `opennlp.osgi` has been created around the plain OpenNLP jars. This has been done with the maven-bundle-plugin[4].

**Runtime Extensibility by additional pretrained models** Pretrained Models for OpenNLP sentence, token and various named entity type detection exist for different languages and new models can be trained by OpenNLP users. Such additional models can be added as OSGi bundles during runtime. Model bundles do not contain any additional java class. Instead, they import external classes that are instantiated and exposed as service implementations (OSGi framework dependent classes with the suffix `StreamFactoryOsgi` from the `opennlp.ext.osgi` bundle). The exposed service components load model data from the classpath via OSGi functionality and themselves use OSGi services implementing `StreamFactoryFactory` interfaces in a compatible version to process the binary model data. Note, that even though the only currently existing OpenNLP model-bundle contains models for tokenization, sentence detection and several named entity detectors for the English language in a single bundle, this is not necessary. Instead it might make more sense to provide the models in separate bundles. However, since in the first version only one model for every detection type has been used it was convenient to have all models at one position.
Models are loaded in a lazy manner and are not stored as hard references in order to keep them only in memory if and as long as they are frequently accessed. This can become important if many models are deployed and started in the OSGi container at once. However, since management of `SoftReference` references is a system dependent responsibility of the JVM, it might on certain systems (especially those with high cache pressure due to e.g. low memory) be counterproductive. If the JVM starts garbage collecting frequently used `SoftReference` objects then the model data will have to be loaded frequently as well. If this is the case, then it is worth trying to create a modified version of `opennlp.ext` where the `SoftReference` in the class `LazyModelCachedDetectorProvider` is changed to a usual reference.

**No compile- or runtime-dependency of the extension bundle on OSGi** The core extension bundles `opennlp.ext` and `opennlp.extapi` as well as all their dependencies do not have any dependency on

[4]See section 3.2.3 for a description of how to run maven

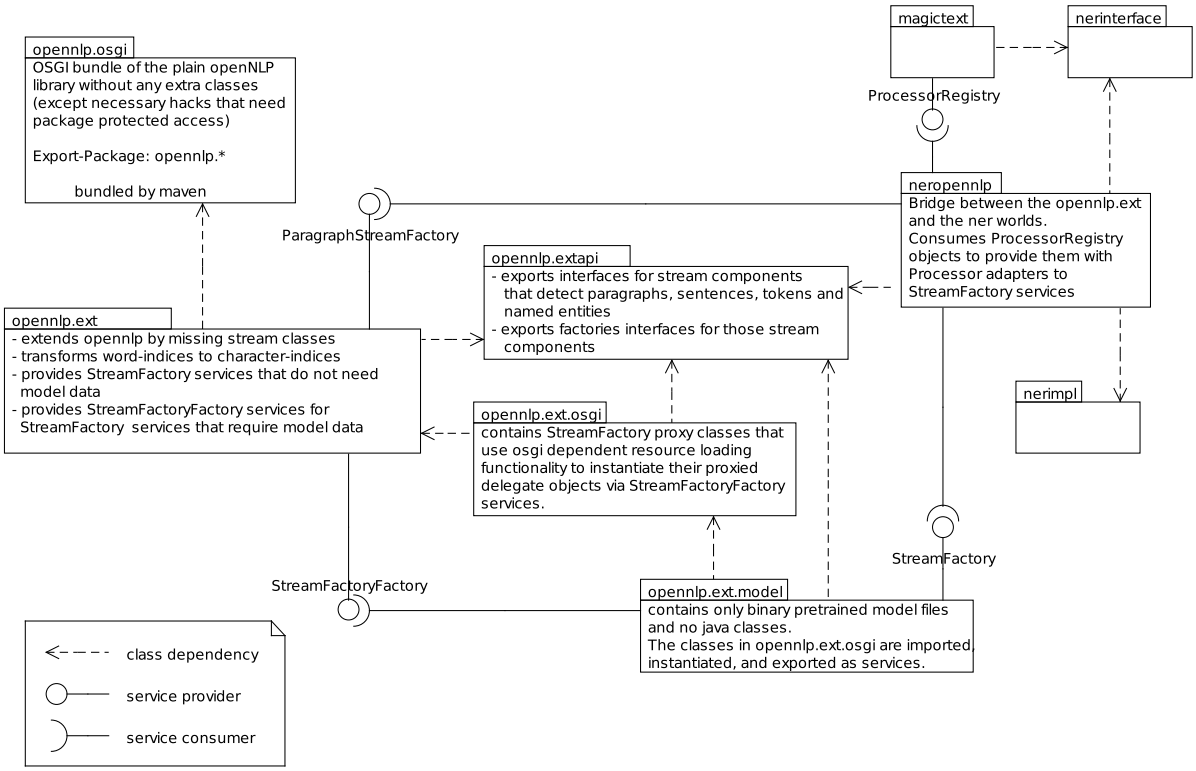


Figure 6: OSGi bundles of the OpenNLP subproject and its connection to the Named Entity Recognizer Platform.

OSGi. Their jar packages can be run regularly in a non-OSGi project. Dependency on OSGi is only required to load the model files from OSGi's bundle classpath at component startup time. In order to keep the core bundles clean, this OSGi dependent code has been factored out to the optional `opennlp.ext.osgi` bundle. Non-OSGi projects could load and instantiate models from another source with the same code-base that the OSGi dependent bundle uses.

**Multithreaded parallel named entity detection** OpenNLP provides named entity detection via distinct models for every named entity type. These detectors although being theoretically independent, cannot be run in parallel (not even in a multiuser parallel-request environment like all state-of-the art servlet containers) due to an unsynchronized shared static instance in the of-the-shelf OpenNLP code. This has been patched for the OpenNLP extensions, but, since the patch requires access to package-protected members, it has to reside in the `opennlp.osgi` bundle that should otherwise not contain any extension classes. With this fix, multithreaded implementations are possible.
In order to run the distinct named entity detectors in parallel also within one user-request, multithreaded access to a shared input stream is required since an `Iterator` can only be read once. This has been implemented behind the service interface `ParallelStreamFactory`. The provided implementation produces

an `Iterator` stream chain-link that iterates over merge-joined results of parallel running internal Iterators. These Iterators are produced from a set of StreamFactory arguments and are automatically wired to an internally created shared input `Iterator` which buffers all entries of the real input `Iterator` until all threads have processed that entry.

### 3.2.3 Bundling OpenNLP with Maven2

Since OpenNLP is not provided as an OSGi bundle, an OSGi wrapper bundle called `opennlp.osgi` has been added. It is build by the maven-bundle-plugin, which automatically generates a jar with all the necessary libraries and an auto-generated MANIFEST.MF file that exports all packages defined in the jars.
To invoke the necessary commands, install maven2 and export the directory containing the executable `mvn` file to your PATH variable. Then run the shell script `init.sh\` via your operating system's commandline in the directory of the project `opennlp.osgi`. In order to also have a correctly running eclipse project, follow the instructions in the output of the script. For non Unix operating systems take a look inside the script and run the command manually.

## 3.3 Stanford

The implementation of the Stanford Named Entity Recognizer[5] is based on linear chain Conditional Random Field (CRF) sequence models. It comes with its feature extractors for named entity recognition and two types of models, trained on CoNLL, MUC6, MUC7 and ACE, and on the CoNLL 2003 English training data. The default classifier of the first type is used for this implementation. It tags named entities and assigns them to the categories *PERSON*, *ORGANIZATION*, and *LOCATION*.

The CRFClassifier offers flexibility in the choice of input and output formats[6]. Possibilities for the input are plain text, XML file, or text file in different formats. The output formats include XML, inlineXML, slashTags and character offsets, the latter being of special interest for this implementation (See 3.3.2).

Among the features used for classification are the words and their contexts (current, previous, or next word, all words in the sentence), orthographic features (case), word prefixes and suffixes, and label sequences. In order to extract these features from raw text, two preprocessing steps are necessary: tokenization and sentence splitting. The architecture of the classifier allows for these two steps to be carried out separately from the named entity recognition itself.

### 3.3.1 Preprocessing

Each of the preprocessing steps required for NER are separated into their own bundles. These include:

- de.uni_sb.coli.nep.nerstanford.ptbtokenizer
- de.uni_sb.coli.nep.nerstanford.sentencedetector

Tokenization is the first preprocessing step in the original pipeline. The tokenizer used for this is edu.stanford.nlp.process.PTBTokenizer, which conforms to the Penn Treebank tokenization conventions. It reads raw text and outputs tokens marked by their begin and end position in the text.

The component used for sentence boundary detection is edu.stanford.nlp.process.WordToSentenceProcessor. A prerequisite is tokenized text. Tokens are grouped into sentences and their begin and end indices marked in a similar way.

[5]The version of the classifier used for this project is ner-eng-ie.crf-3-all2008.ser.gz from stanford-ner-2009-01-16.
[6]See classify* methods in edu.stanford.nlp.ie.AbstractSequenceClassifier for more details.

Since according to the order of preprocessors in our implementation tokenization follows sentence splitting, the sentence detection bundle is responsible for calling the tokenizer itself and adding the sentence and token information to NerResult in the proper hierarchical order.

### 3.3.2 NE Recognition

The method from the original classifier implementation edu.stanford.nlp.ie.AbstractSequenceClassifier.classifySentence(List<? extends HasWord> sentence) was used for the actual classification. It takes as input a tokenized sentence (a List of tokens with their begin and end position with respect to the sentence). This suits our purposes perfectly, since this kind of information is already provided by the tokenizer and sentence splitter in the previous step.

The output of the method is a similar list, where each token contains additional feature annotations including whether it is a named entity and what label it has.

The only disadvantage of this output is that it does not group several consecutive named entities of the same kind which belong together. For instance, the person name *John Smith* would be analyzed as *John/Person Smith/Person* instead of *John Smith/Person*.

To solve this issue, an additional method (partially based on edu.stanford.nlp.ie.AbstractSequenceClassifier.classifyToCharacterOffsets(String sentences)) was implemented to transform this output, taking into account the correct spans of the named entities.

# 4 Voting Strategy and Implementation.

Preprocess sends a copy of the input iterator to the preprocess method of each NER processor, which return an iterator for the resulting NER Result table. After all NER processors have completed, the results are aligned and then an election is held for each named entity to determine which is the most agreed on. The winner of each election is then added to the NER Result table. Finally, an iterator to the result table is returned. The election algorithm is:

1. A start index is maintained to determine which result to compare next. It is initially set to zero.
2. Each processor's result iterator is advanced until the result's start index is greater than or equal to that of start index.
3. The start index is set to the minimum start index of the candidates.
4. If the start index is less than the start index of the candidate, that candidate is replaced with a placeholder (empty) entry.
5. If there are no more entries from a processor, a placeholder entry is created.
6. An election is held to find the winning result.
7. If the winning result is not empty, or if returnEmpties in the voting strategy is set to true, the result is added to the result table.
8. After the election, the start index is either incremented to just past the end of the winning candidate's end index, or in the case that an empty result was chosen to just past the smallest of the non-empty candidate's end index.
9. Steps 2 through 8 are repeated until all results have been consumed.

## 4.1 Election.

The winning candidate is determined by sending a list of candidates to the winner method of an instance of the VotingStrategy class.

**VotingStrategy** has two attributes: emptyHasMeaning, returnEmpties; a list of VotingMethods, and a method, winner. Currently only one instance of VotingStrategy is created. (See To Do below.)
The method winner takes as arguments a list of candidate NERresults and returns the (hopefully) best one. It does this by calling each voting method in turn until the result set has only one element.

**VotingMethod** is an abstract class which has one method, winners, which receives a list of candidates and returns one or more winners. There are four derived classes from VotingMethod.

**VotingRandom** selects one of the candidates at random and returns it as the winner.

**VotingFirst** returns the first candidate in the list as the winner. It isn't currently used, but it could be useful for debugging.

**VotingLongest** returns the candidate(s) with the longest length. Empty candidates are considered to have a length of zero.

**VotingMajority** is the most useful voting method. It groups candidates by label, start index, and end index and computes a tally. The group(s) with the highest counts are returned as winners.

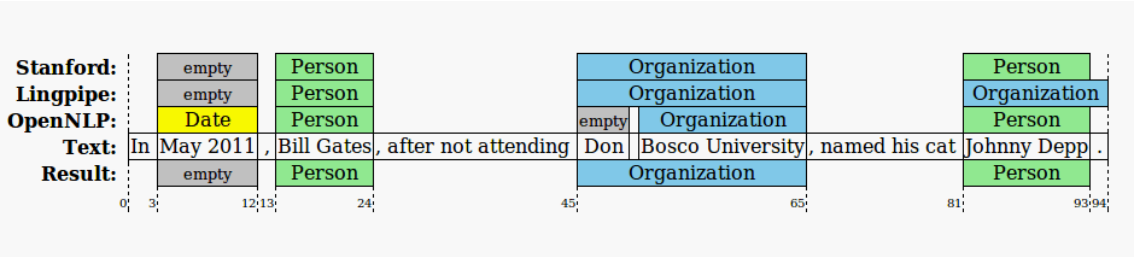


Figure 7: an example

As an example, the sentence:

`In May 2011, Bill Gates, after not attending Don Boscu University, named his cat Johnny Depp.`

After calling the preprocess function for Lingpipe, Stanford, and OpenNLP, the NERresults shown in Figure 7 are returned. Start is initially set to 0. OpenNLP returns a date entity from 3 to 12. Empty entries are created for the other two processors. An election is held and the empty result wins. Because an empty result won, Start is incremented by nine (one plus MinStartIndex, which is the smallest length of the found entities). so the search for the next entity will start at position 12. Three entities for person are found at position 13. No election is needed as they all agree. Start is advanced by 11 (1 + endIndex) to position 24. Next Stanford and Lingpipe both find an organization entity starting at position 45, while OpenNLP doesn't find one until 49 so another empty is created for it. As the non-empty results are the same, they win the election; start is incremented to 66 and "Bosco University" is ignored. The last entities all begin at position 81 and again there is a majority for person. Start is incremented to 94, but as there are no more entities beginning past this index, the loop terminates.

**Other helper classes** include Nerator and MyNerResultTable. Nerator is a custom iterator to simplify accessing the NER results. In addition to the normal iterator functions next() and hasNext() it also has current(), which returns the result from the last call to next(), isNull() which returns true if current() is null, and parent() which returns the parent NERresult of current().

MyNerResultTable is simply a NerResultTable with a public Nerator.

## 4.2 To Do.

- Weights. Each label type for a NER can be assigned an arbitrary weight; however, weighting in voting is only partially implemented. To make weighting functional, line 75 in NerVoting.java needs to be changed to use the label weight instead of the value 1.
- Currently the voting function uses the confidence field in the NERresult class to store the calculated weight and therefore overwrites the confidence value received from from the NER. If this isn't desirable, then another field must be added to the NERResult class for weights.
- Ignoring "empty" results for unsupported labels. The idea here is that should a NER such as OpenNLP return an entity that the other processors don't recognize, then that result should always win against

empty results from the others. This is partially implemented. Line 294 in NerVoting.java needs to be changed to test if the result label is implemented in the parent processor.

- Adding voting bundles. Voting bundles that contain various voting strategies should be made. Then a voting strategy can be selected in the same way as a NER and preprocessor.
- It might be nice to have the NERresult contain a list of NER names instead of just one.
- For performance, the member function getAbsoluteStartIndex should save it's computed value rather than recalculating it over and over... and over... and over.

# 5 Development and Deployment

This chapter provides an overview of the SVN repository used to develop the project and describes how to run the platform as a standalone version. It also informs you what you need and what you need to do when you want to continue development using Eclipse. More importantly, it gives a detailed description of the different ways to extend the functionality of the platform by explaining how to add a processor or meta processor of an existing type, and sketches the steps to be considered when implementing processors or meta processors of new type, such as coreference resolution.

## 5.1 SVN Repository Layout

The SVN repository is accessible at this URL: http://subversion.dfki.de/nep/. All current developments, including the documentation, are committed to the trunk. The repository layout is as follows:

* branches/
* tags/
* trunk/
    * RunStandalone/
    * de.uni_sb.coli.nep.logging/
    * de.uni_sb.coli.nep.magictext/
    * de.uni_sb.coli.nep.nerdummy/
    * de.uni_sb.coli.nep.nerimpl/
    * de.uni_sb.coli.nep.nerinterface/
    * de.uni_sb.coli.nep.nerlingpipe/
    * de.uni_sb.coli.nep.nerlingpipe.sentencedetector/
    * de.uni_sb.coli.nep.nerlingpipe.tokenizer/
    * de.uni_sb.coli.nep.nermeta.voting/
    * de.uni_sb.coli.nep.neropennlp/
    * de.uni_sb.coli.nep.nerstanford/
    * de.uni_sb.coli.nep.nerstanford.ptbtokenizer/
    * de.uni_sb.coli.nep.nerstanford.sentencedetector/
    * de.uni_sb.coli.nep.opennlp.ext/
    * de.uni_sb.coli.nep.opennlp.ext.model/
    * de.uni_sb.coli.nep.opennlp.ext.osgi/
    * de.uni_sb.coli.nep.opennlp.extapi/
    * de.uni_sb.coli.nep.opennlp.osgi/
    * de.uni_sb.coli.nep.webui/
    * documentation/
    * org.eclipse.equinox.jsp.jstl/
    * report/

All folders starting with "de.uni_sb.coli.nep" and "org.eclipse.equinox.jsp.jstl" are Eclipse projects and correspond to an OSGi bundle with the same name. "RunStandalone" contains all necessary dependencies of the project as jar files as well as all project specific bundles which have been packaged as jars. "documentation" and "report" are self-explanatory.

## 5.2 Standalone Version

### Running

The standalone version of our platform contains all the jars required for the project to run, i.e. all the OSGi bundles from our project and their dependencies. Thus, if you check it out from the repository, you should be able to start and run the platform without any further changes.

For this purpose, you have to run the shell script `start.sh` which starts up the entire project and an OSGi console. If you point your favorite browser to `http://localhost:8080/ner/ner`, you will see the web user interface. The only option you might have to change is the port on which Jetty will listen. If you want to set it to a port other than 8080, you have to change the value in the line containing `org.osgi.service.http.port=8080` of the start script.

### Packaging

The easiest and most straight forward way of packaging an Eclipse bundle project into a deployable OSGi bundle is to use the functionality that is provided by Eclipse. Go to `File > Export ... > Deployable plug-ins and fragments` and then select the appropriate projects you want to export. It is advisable to keep the standalone version up-to-date such that whenever you release a working version of your bundle, you put an exported jar file into the `RunStandalone` folder in the repository. You do not have to change any configuration, since the star script will look for jars in the directory and start them. However, you may want to remove the old version if it you do not want several versions to be started.

## 5.3 Development with Eclipse

The Named Entity processing project was developed using heterogeneous Eclipse versions. Among them Helios (3.6) and 3.5.2. All versions with the Plug-in Development Environment (PDE), WTP and the Subclipse plugin. (The latter is optional, however it makes development much more convenient). You have to check out all the folders containing Eclipse bundle projects from the repository as described in Section 5.1, i.e. all folders starting with `de.uni_sb.coli.nep` and `org.eclipse.equinox.jsp.jstl`. In Eclipse you can use `File > New > Other ... > Checkout Projects from SVN` and check out all relevant projects at once.

In the root of `de.uni_sb.coli.nep.webui` there is an Eclipse launch configuration which can be used to start the entire project and an OSGi console within Eclipse. This is especially useful for debugging. In the launch configuration all the required bundles are selected to be included and started. The configuration is set such that Eclipse will automatically include and start any other OSGi bundle (e.g. your own NER bundle) which is in the current Eclipse workspace.

There is a bundle in the project which you can use as reference when you start extending the platform, by e.g. adding new processors or meta processors. The bundle is called `de.uni_sb.coli.nep.nerdummy`. The bundle provides four processors (tokenizer, sentence and paragraph detector and a very stupid named entity recognizer) and one meta processor (named entity voting). Please note, however, that it is not advisable to put several processors and meta processors in the same bundle, as it will make it impossible to individually start and stop them. In addition to this bundle, the following three sections (5.4, 5.5 and 5.6) give you a detailed description for extending the project.

## 5.4 Adding New Processors of Existing Types

### Basic Architectural Requirements

We have designed the platform in such a way that adding new processors of existing type like tokenizers, sentence and paragraph detectors, and named entity recognizers requires only little effort. The first step is to create a new bundle project of type "Plug-in Project" in Eclipse. The basic structure will be created for you. You will usually not need an `Activator` class, so deselect that if it is checked in the wizard.

Then you create a new Java class and put it into the `src` folder. The best practice is to use packages, i.e. don't put it into the default package. The naming convention of most of our processing bundles is such that the package name will be equal to the folder of the SVN repository and the Eclipse project name. The newly created class has to implement the `de.uni_sb.coli.nep.nerinterface.Processor` interface. In the `preprocess` method, you will put your processing code, e.g. a named entity recognizer. Via the arguments of the method, you will have access to the list of results from the previous processing steps (e.g. list of tokens for the named entity recognizer), and access to the map of selected processors. The two other methods you have to implement provide the type of processor (use the constants provided in the `Processor` interface) and a short one-word name which will be used for displaying in the user interface. In combination with the type it should form a unique identifier.

You might have to import other bundles of the project, most notably `de.uni_sb.coli.nep.nerimpl` and `de.uni_sb.coli.nep.nerinterface`. So you simply add them to the MANIFEST.MF file to an existing Import-Package statement or you add a new Import-Package statement.

Furthermore, you will most likely need to have access to libraries and data from your native processor. For the libraries, one strategy[7] is to add a new folder `lib` to the root of your bundle and add the jar files of your native processor there. You have to adjust the MANIFEST.MF to include a statement like `Bundle-ClassPath: ., lib/native-processor-4.0.1.jar`, you have to add the lib folder to build.properties `bin.includes` and you have to add the jars to the Build Path of the Eclipse project. The latter step is only required for Eclipse. For model files, the strategy along the same line is to create a folder `data` in the root of the bundle. You have to add this folder to the attribute `bin.includes` in the build.properties.

Once you have implemented the class with the processor functionality, you have to make it available to MagicText. Thus, you have to create a folder in the root of the project and call it OSGI-INF. In this folder you have to put a new component definition file. If you use Eclipse to create one for you, it will automatically take care of adjusting the MANIFEST.MF and the build.properties file to include the component definition. (In Eclipse, go to `File > New > Other ... > Component definition`.) In any case your component.xml should provide a service for `de.uni_sb.coli.nep.nerinterface.Processor` and specify the above class which implements this interface.

The following two listings show the basic code (cf. Listing 1) and the component definition (cf. Listing 2) as described above.

Listing 1: Java class implementing a processor functionality.

```
public class DummyNer implements Processor {
        @Override
```

[7] This isn't the best strategy, since you might have to include the same libraries and models in several bundles, e.g. a tokenizer and named entity recognizer bundle of your native processor. It would be better to have a separate bundle which just provides the libraries and models, such that other bundles implementing a processor can access them from that bundle. In order to do this you can use maven and its bundle-plugin. For an example how this works see the pom.xml in the opennlp.osgi bundle.

```
public Iterator<NerResult> preprocess(final Iterator<? extends
    NerResult> previousLeafNodes, final Map<String, Processor>
    preprocessorSelection) {
        return new AbstractSplitIterator<NerResult, NerResult>(
            previousLeafNodes) {

                @Override
                protected Iterator<? extends NerResult> split(
                    NerResult nr) {
                        String findName = "Johnny";
                        if (nr.getTextAsString().contains(findName)) {
                                int inNrStartIndex = nr.
                                    getTextAsString().indexOf(findName
                                    );
                                NerResult child = new NerResultImpl(
                                                new NerLabelImpl("
                                                    person"),
                                                nr.getTextAsString().
                                                    substring(
                                                    inNrStartIndex,
                                                    findName.length())
                                                    ,
                                                nr.getStartIndex() +
                                                    inNrStartIndex,
                                                nr.getStartIndex() +
                                                    inNrStartIndex +
                                                    findName.length(),
                                                -1.0);
                                child.setParent(nr.getParent());
                                return Arrays.asList(child).iterator()
                                    ;
                        }
                        return Collections.<NerResult>emptyList().
                            iterator();
                }
        };
}

@Override
public String getImplName() {
        return "dummy";
}

@Override
public String getType() {
        return NAME;
```

```
        }
}
```

Listing 2: Component definition to make the processor available to the platform.

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0" name="de.
    uni_sb.coli.nep.nerdummy">
   <implementation class="de.uni_sb.coli.nep.nerlingpipe.DummyNer"/>
   <service>
      <provide interface="de.uni_sb.coli.nep.nerinterface.Processor"/>
   </service>
</scr:component>
```

### Setting the Parent

It is important to set the parent of the newly generated results. E.g. if a tokenizer creates the results, their parents have to be set to the containing sentence. The `NerResult` (the class that holds a specific result (e.g. a token)) is provided with a method `setParent` for that purpose. If the parent isn't set, then reconstruction of the text - especially when it comes to generating the XML output - will fail and you will most likely encounter something like a `NullPointerException` or an `IndexOutOfBoundsException`. The only case where the parent of a result may and should be null, is the object that represents the entire input, i.e. at document level. In order to find out the appropriate parent of a particular result, please refer to Section 2.8.

### Different Processing Order and Preemption

Preempting is explained in Section 2.7. It is a way to tackle different requirements of processor inputs. A sentence detector for example could take a paragraph as input and return the sentences as result. Concerning the processing hierarchy this would seem to be the natural order. However, some implementations of sentence detectors require tokens as input rather than a complete paragraph.

There is a little pitfall, when preemption is used. So when for example the sentence detector calls the selected tokenizer to generate tokens of the current paragraph, the tokenizer will return NerResult objects representing tokens. You then do the sentence detection, create NerResult objects for sentences and set the generated token NerResults to the preemption list with the method `setPreemptedList`. If you stop here, you will have a wrong hierarchy of results, since the tokenizer worked with a whole paragraph, it will also set the token's parent to this paragraph. However, when you use the preempted list after sentence detection, then you will just end up with the same tokens which have a paragraph as parent, whereas you would want to have the proper sentence as parent.

Thus, you have to make sure, that when you set the preempted list, you change the parent of the elements to the proper parent. Furthermore, you have to make sure to update the start and end indices since they are always relative to the result's parent.

## 5.5 Adding New Meta Processors of Existing Types

Meta processors accumulate at least two processors of the same type and trigger their execution. The results returned by individual processors have to be merged according to a certain strategy.

The first step which has to be performed is creating a class, that implements the merging strategy. Furthermore, it has to implement the interface `Processor`. It must keep track of all the processors which should be involved in the merge. The references to these processors get set by a factory for the meta processor.

The second step is to create the factory for the meta processor by implementing the `MetaProcessorFactory` interface. It contains a method which has to return a new instance of the above voting class and which also sets all the processors whose result should be merged.

The factory is the class of the service object which has to be provided to MagicText. This is accomplished via declarative services by defining a component definition file. The file should be at the usual place in the OSGI-INF folder and has to be mentioned in the MANIFEST.MF. The service it provides is `MetaProcessorFactory` and the implementing class is the meta processing factory.

The following three listings show basic code for the class implementing the merging strategy (cf. Listing 3), for implementing the factory (cf. Listing 4) and for the component definition (cf. Listing 5).

Listing 3: Java class implementing the merging strategy.

```
public class DummyNerVoting implements Processor {
        private List<Processor> pl;
        public DummyNerVoting(List<Processor> selectedNers) {
                pl = new ArrayList<Processor>();
                pl.addAll(selectedNers);
        }

        @Override
        public Iterator<? extends NerResult> preprocess(final Iterator<?
            extends NerResult> previousLeafNodes, final Map<String, Processor>
             preprocessorSelection) {
                // TODO return iterator of merged NerResults
        }
        // TODO implement other missing methods
}
```

Listing 4: Factory for creating the above class.

```
public class DummyNerVotingFactory implements MetaProcessorFactory {
        @Override
        public Processor produce(List<Processor> selectedNers) {
                return new DummyNerVoting(selectedNers);
        }
        // TODO implement other missing methods
}
```

Listing 5: Component definition file for a Meta Processor.

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0" name="de.
    uni_sb.coli.nep.nerlingpipe">
   <implementation class="de.uni_sb.coli.nep.nerlingpipe.LingPipeChunker"/>
   <service>
      <provide interface="de.uni_sb.coli.nep.nerinterface.PreprocessingFactory
          "/>
   </service>
</scr:component>
```

## 5.6 Adding New (Meta) Processors of New Types

The above two sections described the process of extending the platform by existing types of processors. In this section, we will explain the steps you have to consider when integrating a processor of a new type.

The first and most easy thing to do is to define a constant which holds a representative string describing your processor type. Of course this is not strictly necessary, but it will make maintenance and development much easier. The type constants are defined in the processor interface `Processor`. Just add your new type constant there, e.g. "coref" for co-reference resolution.

The next and more important point you have to consider is how you will integrate the results of your processor in the result hierarchy (cf. Section 2.8). Decisions on that issue also have to be based on expected input and output of the processor. In the example of co-reference resolution, you might have tokens as input and an entirely new hierarchy of results as output. Furthermore, you would want to make use of recognized named entities.

In this example you could have a processing order of `"paragraph, sentence, token, coref, name"`. When you come to coreference resolution, you will do a preemption of named entity recognition and thus have both tokens and the named entities available as input to coreference resolution. Since the latter task most likely generates hierarchical chains of markables that all link to the preceding markable of the same linguistic referent, you could make use of the `parent` instance field of NerResults. Thus you would return an iterator of NerResults, where each next element represents a coreference chain and via the method `getParent` you would be able to move backwards in each chain and thus retrieve the chain e.g. for output generation.

Of course this is a simplified approach to coreference resolution, since it assumes that all chain members have exactly one antecedent which is not always the case (e.g. if you consider references of type `split_antecedent` – cf. [2]). Handling this more appropriate model within our platform is left as an open development issue and won't be of concern in this rather sketchy description.

Once you made the input and result hierarchy design decisions, you have to start implementing the processor of the new class. The best and easiest would be if your new processor implements the `Processor` interface. That way registering the new processor on the platform is as easy as adding a new processor of existing type (cf. Section 5.4). Even if you use a new type constant, the platform is designed in such a way, that no further code modifications are necessary. Modifications in the user interface aren't necessary either, as the processor lists get generated dynamically.

The only change you have to make to the code with respect to the web user interface is the list where the processing order is defined. The class where the order is defined is the backing bean `NerBean`. Currently, it is defined in its constructor. Please note that this is not the best place since it induces the necessity to change a class for a purpose which could be better handled in a configuration file. An even better approach would be to give the user a predefined list of possible processing orders via the user interface. However, this is left for future developments.

The above sketched approach is most conservative in the sense that it requires minimal changes to the

existing platform. It will not always be possible to extend the functionality of the platform in this way so possible alternative considerations would be

a) changing or extending the NerResult such that it accommodates the new needs, but at the same time doesn't affect the functionality of the existing processors

b) using a completely different data structure to hold the results of a processor

c) if the decision is made for b), then this also means that the Processor interface can't be used anymore. A new infrastructure would have to be programmed into the platform along the lines of the processors and meta processors

d) you keep the implementation of the NerResult, but define a new interface for different kinds of processors

As long as you go with the most conservative approach described above and maybe option a), adding a meta processor of a new type doesn't require any further changes. You should be able to implement it by following the steps as described for meta processors of an existing type (cf. Section 5.5).

# 6 Conclusion

In this report we have presented a description of the design and implementation of a meta system for named entity recognition as a combination of several available state-of-the-art systems for this task - OpenNLP, Stanford, and LingPipe. In order to combine them, a voting strategy, aiming at achieving higher overall accuracy, was used over their individual results.

The goal of designing a flexible and easily-expandable framework was reached with the help of OSGi, a service platform and model system for Java. It nicely suits our purposes, since it allows for the dynamic management of the components available in the system. Furthermore, it enables the use of self-contained modules for the different NLP tasks - text preprocessing and named entity recognition. We have also presented a detailed description of the necessary steps for adding new instances of existing modules and sketched possibilities to extend the platform's functionality for new NLP tasks.

The final result of our work is a easy-to-use and easy-to-expand system, which can be accessed via a web-based user interface and run as a web service.

# References

[1] The OSGi Alliance. Declarative Services Specification, August 2009.

[2] Karin Naumann. Manual for the Annotation of in-document Referential Relations. Technical report, Department of Linguistics, University of Tübingen, 2007. http://www.sfs.uni-tuebingen.de/tuebadz.shtml and http://www.sfs.uni-tuebingen.de/resources/tuebadz-coreference-manual-2007.pdf.

[3] Georgios Sigletos, Georgios Paliouras, Constantine D. Spyropoulos, and Michael Hatzopoulos. Combining information extraction systems using voting and stacked generalization. *Journal of Machine Learning Research*, 6:1751–1782, 2005.

[4] Lars Vogel. OSGi with Eclipse Equinox - Tutorial, 2010. `http://www.vogella.de/articles/OSGi/article.html` @ 2010-12-14.