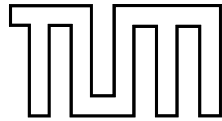
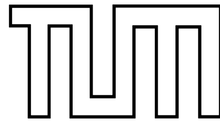


Department of Informatics
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master’s Thesis in Information Systems

A web based Workbench for Interactive Semantic Text Analysis: Design and Prototypical Implementation

Tobias Waltl



Department of Informatics
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Information Systems

A web based Workbench for Interactive Semantic Text Analysis: Design and Prototypical Implementation

Ein webbasierter Arbeitsplatz zur interaktiven semantischen Textanalyse: Entwurf und prototypische Implementierung

Student:	Tobias Waltl
Supervisor:	Prof. Dr. Florian Matthes
Advisor:	Bernhard Waltl, M.Sc.
Submission date:	15.10.2015

I confirm that this master's thesis is my own work and I have documented all sources and materials used.

Garching b. München, 15.10.2015 ______________________

Place, Date Signature

Acknowledgments

First and foremost, I would like to thank my advisor Bernhard Waltl for his great support and fruitful input for this thesis. Furthermore, I would like to thank Prof. Dr. Florian Matthes for his ideas and for making it possible to write this thesis at the chair Software Engineering for Business Information Systems held by him. Last but not least, I want to thank my interviewees Konrad Heßler, Christopher Splinter, and Johannes Riederer who gave valuable suggestions for possible features of the web application.

Abstract

Due to the huge and continuously increasing amount of unstructured information, i.e. text, the analysis of legal literature has become a complex and time consuming task. Texts may refer to and are referred by other texts which can lead to complex dependencies. For example, a condition specified in a text maybe excepted in another one. Furthermore, there may be exceptions of exceptions. On the other side, computer science provides means that can support information processing. Natural language processing (NLP) technologies, for instance, evolved rapidly during the last years and several architectures for NLP systems have been proposed. Most of these architectures are general purpose architectures that are not tailored to any specific domain. One exemplary adaption for a domain can be found in bioinformatics where already some NLP applications exist that are capable of recognizing proteins, for instance.

This work presents the prototypical implementation of a web based NLP application tailored to the legal domain. For this, it is analyzed what requirements such an application should fulfill. These include functional requirements that apply for an NLP application in general as well as functional requirements that are specific for the legal domain. Furthermore, nonfunctional requirements, which mainly refer to the architecture of such an application, are presented. The aforementioned existing architectures are explored and assessed against those requirements. Therefore, it is analyzed what the key concepts of the respective architectures are and what consequences they imply regarding the conformance with these requirements.

Based on this assessment UIMA has been considered as the architecture that meets the requirements best and therefore, is the fundament of the implementation which is presented in chapter *6 Implementation of the workbench*. It is explicated what features the application provides and how NLP technologies have been integrated to realize these features. Moreover, it is pointed out what the main challenges have been and what the limitations of the application are.

Finally, a critical review is provided where it is discussed which of the requirements the current implementation already fulfills.

Table of Contents

Acknowledgments i

Abstract iii

List of Figures vii

List of Listings ix

List of Tables xi

List of Abbreviations xiii

1 Introduction 1

1.1 Motivation 1

1.2 Basic knowledge 3

1.3 Research questions 6

2 Related work 7

2.1 GATE Developer 7

2.2 BRAT 8

2.3 U-Compare 9

2.4 Argo 10

2.5 WebLicht 11

2.6 WebAnno 12

2.7 AnnoMarket 13

3 Research method 15

4 Requirements 17

4.1 Functional requirements 19

4.2 Nonfunctional requirements 24

4.3 Delimitation of the requirements 27

5 Assessment of existing architectures for NLP systems 29

5.1 TIPSTER 29

5.2 Ellogon 32

5.3 LIMA 35

5.4 Whiteboard architecture 37

5.5 TALISMAN 41

5.6 TalLab 43

5.7 Heart of Gold 45

5.8 GATE 48

5.9 UIMA 52

5.10 Summary and conclusion 60

6 Implementation of the workbench 63

6.1 Type system and component collection 63

6.2 Architecture of the web application 63

6.3 Implementation of an own UIMA component 65

6.4 Developing rule-based components with UIMA Ruta 67

6.5. From raw documents to annotated documents 68

6.6 Reflection of fulfillment of the requirements 77

7 Conclusion and outlook 83

Appendix 85

Interviews 85

Figures 102

List of Figures

Figure 1: Screenshot of an annotated document within GATE Developer 8

Figure 2: Screenshot of an annotated document within BRAT 9

Figure 3: Screenshot of U-Compare... 10

Figure 4: Screenshot of an annotated document in Argo 11

Figure 5: Screenshot showing the creation of a workflow in WebLicht................................. 12

Figure 6: Screenshot showing the manual annotation in WebAnno 13

Figure 7: Screenshot of an annotated document in AnnoMarket.. 14

Figure 8: Illustration of the Ellogon architecture 32

Figure 9: Illustration of the Whiteboard architecture.. 38

Figure 10: Illustration of the Heart of Gold architecture 45

Figure 11: Representation of two conceptually different annotations having the same value for their type declaration 51

Figure 12: Architecture of the DKPro Core component collection 59

Figure 13: Overview of the DKPro Core type system 60

Figure 14: Overview of the application's architecture 64

Figure 15: The architecture of the text mining engine 65

Figure 16: Screenshot of the view allowing users to develop their own Ruta components..... 68

Figure 17: Screenshot showing the list of imported documents 69

Figure 18: Screenshot denoting the selection of the analysis components 70

Figure 19: Illustration of the pipeline model. Each component fills the JCas object with its annotations. 72

Figure 20: Users can select which annotation types are available when viewing the annotated document 73

Figure 21: Screenshot of the view presenting the annotated document 74

Figure 22: Screenshot of a previous project used as an interactive mockup in interviews 102

Figure 23: The type system of U-Compare 103

List of Listings

Listing 1: An example for inline markup of annotations embedded into the original text 4

Listing 2: An example for stand-off markup referring to the text by begin and end attributes . 5

Listing 3: An example annotator annotating exceptions of legal norms................................. 66

Listing 4: The current state of the type system 67

Listing 5: Example of a basic pipeline including components for tokenizing and sentence splitting, POS tagging, legal exceptions and the Ruta scripts selected by the user 71

Listing 6: Excerpt of an exemplary annotation type structure created during the annotation process 73

Listing 7: Conversion from plaintext position to HTML position 76

List of Tables

Table 1: Overview of requirements and their priority for the prototypical implementation.... 18

Table 2: Summary of the architectures and the requirements they fulfill 61

Table 3: Summary of the requirements and their implementation state 82

List of Abbreviations

AAE	Aggregated Analysis Engine
AE	Analysis Engine
API	Application Programming Interface
AS	Asynchronous Scaleout
ASCII	American Standard Code for Information Interchange
BGB	Bürgerliches Gesetzbuch
BRAT	brat rapid annotation tool
CAS	Common Analysis Structure
CPSL	Common Pattern Specification Language
CREOLE	Collection of REusable Objects for Language Engineering
DTD	Document Type Definition
GATE	General Architecture for Text Engineering
GDM	GATE Document Manager
GGI	GATE Graphical Interface
GUI	Graphical User Interface
HTML	HyperText Markup Language
IDE	Integrated Development Environment
IE	Information Extraction
JAPE	Java Annotation Patterns Engine
JAR	Java ARchive
JCoRe	JULIE Component Repository
JSON	JavaScript Object Notation
MAS	Multi-Agent System
NER	Named Entity Recognition
NLP	Natural Language Processing
OS	Operating System
PEAR	Processing Engine ARchive

PDF	Portable Document Format
POS	Part of Speech
ProdHaftG	Produkthaftungsgesetz
REST	Representational State Transfer
RMRS	Robust Minimal Recursion Semantics
RPC	Remote Procedure Call
Ruta	Rule-based Text Annotation
sebis	Software Engineering for Business Information Systems
SGML	Standard Generalized Markup Language
SMTP	Simple Mail Transfer Protocol
Sofa	Subject of Analysis
UI	User Interface
UIMA	Unstructured Information Management Architecture
WebLicht	Web-Based Linguistic Chaining Tool
XMI	XML Metadata Interchange
XML	eXtensible Markup Language
XSL	eXtensible Stylesheet Language
XSLT	eXtensible Stylesheet Language Transformation

1 Introduction

1.1 Motivation

With rising amounts of legal literature it has become nearly impossible to know all relevant literature for a specific case. For example, within in the legislative period from 2009 to 2013 553 laws have been passed in Germany. Within the legislative period from 2005 to 2009 even 616 laws have been passed (Deutscher Bundestag, 2014). In addition to that, a lot of decisions and legal commentaries are published every year (cf. (Bundesgerichtshof, 2015; Walter, 2010, p. 81)).

Even though the German jurisdiction is independent from previous judgements, in practice they are still taken into consideration when a decision has to be made in the court. Therefore, lawyers should be able to know all literature relevant for their specific case. Besides applicable laws this also includes prior judgements and legal commentaries, for instance. As a consequence, dealing with all that literature, which does not only include searching and reading but also interpreting it, consumes much time and requires a lot of knowledge.

On the other side, technologies regarding the semantic analysis of texts are evolving and thus providing new possibilities to deal with this literature. Natural language processing (NLP) is a research discipline that aims at using the capabilities of computer science to process unstructured information formulated in natural language. One direction of NLP is information extraction (IE), which is targeted at processing unstructured textual information and deriving structured information from it (Basili, Di Nanni, & Pazienza, 1999; Gaizauskas & Wilks, 1998; Grishman, 2010). While it is very time consuming and error-prone to enter this structured information manually, the usage of tools that apply these technologies can support their users with providing this structured information in a context-aware way (Bontcheva, Cunningham, Roberts, & Tablan, 2010; Yimam, Eckart de Castilho, Gurevych, & Biemann, 2014). Thus, they have the potential to significantly reduce the users' expenditure of time when working with such literature.

While already several tools exist providing generic linguistic information like part of speech (POS) tags or syntactic dependencies of words within their sentence, there are only few tools dedicated for a specific domain. Domain specific tools have to cover more specific requirements like annotating named entities that are crucial for that domain, for instance. These existing tools

mainly concentrate on applications in the medical and bioinformatics area. A subset of them will be presented in chapter *2 Related work*. An example for named entity recognition (NER) specific for these areas is the recognition of proteins as it is performed by ABNER, for instance (Settles, 2005).

In this work it is analyzed how NLP technologies can be embedded into a collaborative web application that is tailored to the legal domain. Therefore, it is examined what existing projects, which mainly reside in the bioinformatics domain, exist. Afterwards, the results of a requirements analysis for such a legal domain specific web application are explicated. General requirements have been elicited via a literature analysis while requirements specific for the legal domain have been obtained with expert interviews. In chapter *5 Assessment of existing architectures* it is elaborated on existing architectures for NLP applications and it is expounded how well they fit the requirements and how they address them. It is disclosed that the Unstructured Information Management Architecture (UIMA) meets the requirements best and thus, is selected as basis for the implementation.

Subsequently, it is demonstrated how the prototypical implementation of the web application has been developed, what the major challenges have been, and how they have been addressed. By means of the example of an exemplary process it is shown how the current implementation embeds NLP algorithms as well as self-developed components into a collaborative web application. Moreover, it is presented how the Apache UIMA Ruta Engine, that was formerly known as TextMarker and provides a rule language that uses regular expression-like patterns for matching annotations, has been integrated so that it is not only possible to integrate Ruta scripts as components into a workflow but also to develop own scripts directly from within the web application (Kluegl, Atzmueller, & Puppe, 2009; Kluegl, Toepfer, Beck, Fette, & Puppe, 2015).

After that, a critical review is conducted where for each requirement it is analyzed how well they are satisfied by the current implementation. Finally, the work is concluded and an outlook for future work is provided.

1.2 Basic knowledge

In this section some fundamental terms are explained. This knowledge forms the basis for the rest of this work and therefore is required for the following chapters.

1.2.1 Structured information vs. unstructured information

Information can be decomposed into structured and unstructured information. Structured information refers to information having a predefined data model which allows computers to process this information as its interpretation is possible without ambiguities (Ferrucci & Lally, 2004; Ferrucci, Lally, Verspoor, & Nyberg, 2009). Examples for representations of structured information are the eXtensible Markup Language (XML) format or objects in object-oriented programming languages.

In contrast to this, unstructured information does not have a predefined data model and therefore cannot be processed by computers as the interpretation may be ambiguous (Ferrucci & Lally, 2004; Qureshi, Memon, & Wiil, 2011). Examples for unstructured information are arbitrary texts or pictures. Unstructured information is the lion's share and fastest growing kind of the available information and thus, may contain lots of useful information, e.g. for companies (Ferrucci et al., 2009).

1.2.2 Natural Language Processing (NLP)

Natural language processing is a research discipline focusing on analyzing unstructured information provided in natural language by the utilization of computer systems. Therefore, this unstructured information is enriched with meta-information provided in a structured form. Natural language refers to human languages like English or German, for example. It may be provided in different forms like written text or spoken text (Allen, 2003; Bird, Klein, & Loper, 2009; Liddy, 2001). However, this work only considers written text as input. NLP is a very high-level discipline that can be decomposed into several sub-disciplines like information extraction, machine translation or language generation, for instance. In this work the focus is set on the former one.

NLP has received lots of attention during the last years due to the constantly improving performance of computer systems and increasing amounts of publicly available language resources like dictionaries and thesauri (Jackson & Moulinier, 2007). This evolution has been entailing new possibilities that have not been imaginable in the past. Distributed and concurrent

processing is such a possibility that has been enabled by machines having several processors instead of only one. This has made it feasible to apply complex algorithms to large text collections, for instance.

1.2.3 Annotation

The results computed by NLP components are typically represented as annotations. The representation of annotations can basically be subdivided into two major approaches (Cunningham, 2002; Cunningham, Humphreys, Gaizauskas, & Wilks, 1997; Wilcock, 2009). The first one is to embed the annotations directly into the text they refer to. This is known as *inline markup* or *embedded markup* and typically implemented with an XML syntax. The main advantage of this strategy is that changes of the annotated text are propagated to the annotations. I.e. if a word is changed and, as a consequence, the indices of the following words are shifted, the indices of the annotations associated with them are shifted as well. The main drawback of this approach is that it is not possible to represent overlapping annotations in a proper way (Nelson, 1997). Apart from that, all components of a workflow except the first one have to be reconfigured so that they treat the markup inserted by the first one properly and do not consider it as content of the original document that shall be annotated, for example (Kontonasios, Korkontzelos, Kolluru, & Ananiadou, 2012). Another disadvantage is that the original document is changed. Listing 1 provides an example of inline markup of the text “I love text analysis.” as it may be produced by a POS tagger using the Penn Treebank Tagset (Marcus, Marcinkiewicz, & Santorini, 1993).

```
<sentence>
    <pronoun tag="PRB">I</pronoun>
    <verb tag="VBP">love</verb>
    <noun tag="NN">text</noun>
    <noun tag="NN">analysis</noun>
    <punctuation tag=".">.</punctuation>
</sentence>
```

Listing 1: An example for inline markup of annotations embedded into the original text
Source: Own illustration

The other approach is to store the annotations separately from the original document and is called *stand-off annotations*. This is the one which is widely adopted and recommended as it has some advantages over the inline markup (Nazarenko et al., 2006). One of them is that the original document is not changed and therefore, can still be used for other purposes after the annotation process. Another benefit is that overlapping annotations can be represented without having to deal with XML structures that are not well formed. Indeed, for this approach XML is

often used as well but its elements are not embedded into the original document but rather refer to that original document by using start and end positions for the respective annotation. An example of a stand-off representation of the same annotations represented inline in the previous example is shown in Listing 2.

```
<sentence begin="0" end="21">
    <pronoun tag="PRB" begin="0" end="1"/>
    <verb tag="VBP" begin="2" end="6"/>
    <noun tag="NN" begin="7" end="11"/>
    <noun tag="NN" begin="12" end="20"/>
    <punctuation tag="." begin="20" end="21"/>
</sentence>
```

Listing 2: An example for stand-off markup referring to the text by begin and end attributes
Source: Own illustration

Annotations can also include further information related to this annotation. These are typically called *features (Ferrucci, Lally, Verspoor, & Nyberg, 2009; Hahn, Buyko, Tomanek, Piao, McNaught et al., 2007; Wilcock, 2009)*. In the examples above each annotation has an attribute that represents the feature “tag” denoting the POS tag according to the Penn Treebank Tagset. The stand-off annotations have the two additional features “begin” and “end” which are used for referring to the associated positions within the original text.

1.2.4 Type system

Annotations typically have a description of their type which aims at distinguishing different categories of annotations. The examples presented in Listing 1 and Listing 2 contain annotations marking a span as sentence, pronoun, verb, noun, or punctuation. Therefore, the element types of these XML representations refer to the annotations’ types. In chapter *5 Assessment of existing architectures* different architectures for NLP systems are presented. While some of them use only a textual representation of the annotations’ type (e.g. GATE (Cunningham et al., 1997)), others require the annotations to be strictly typed (e.g. UIMA (Ferrucci & Lally, 2004)). In the latter case annotation types may inherit and form a hierarchy. The set of all annotation types and their relations to each other forms a schema and is referred to as a *type system* (Ferrucci et al., 2009, pp. 11f).

1.3 Research questions

Within in this thesis mainly four research questions have been investigated which are shortly explicated in the remainder of this subsection.

What are requirements for a software architecture to support semantic analysis of legal literature?

First of all, a requirements analysis has been performed to ensure that the workbench and its architecture hits the target to allow for solving the problem mentioned above. These requirements have been elicited with a literature review and expert interviews with possible end users of the workbench.

What are common software architectures based on these requirements that support semantic analysis in web applications?

Based on those requirements, I explored existing software architectures, analyzed them, and decided which one fits best the requirements for a web based workbench allowing for interactive semantic text analysis in the legal domain.

How can semantic entities (in terms of annotations) and their dependencies be represented?

This questions deals with how to represent the semantic information extracted by the workbench. As mentioned above, approaches found in the literature can be mainly subdivided into two strategies: Inline markup and stand-off annotations (Wilcock, 2009). Both approaches are exemplified in this thesis. They are compared to each other and it is illustrated which approach is used for the workbench and what the reasons for the decision are.

How does a prototypical integration of an architecture enabling semantic analysis on legal literature look like?

Answering this question it is shown how the workbench is implemented using the selected architecture. Furthermore, it is presented how to extend this workbench with additional analysis components.

2 Related work

In this chapter several annotation tools, which are somehow related to the web application presented in this work and therefore serve as sources of requirements, are shortly examined. The tools are mainly described from an end user's view neglecting details regarding their implementation and rather elaborating on the main use case of the respective tool.

2.1 GATE Developer

The General Architecture for Text Engineering (GATE) is a project providing a powerful infrastructure for natural language processing. It comes with GATE Developer, an integrated development environment (IDE) for developing GATE pipelines (Cunningham, Tablan, Roberts, & Bontcheva, 2013). These pipelines are built by the combination of analysis components and can be saved. Such a saved pipeline cannot only be used again with GATE Developer but also within *GATE embedded*, which allows the integration of GATE functionality into an own application. In this subsection it is shortly shown how to work with annotations in GATE Developer. More details about GATE's architecture and its framework *GATE embedded* are provided in subsection *5.8 GATE*.

It is possible to execute pipelines directly within GATE Developer and visualize the results afterwards. An example of a short text annotated with GATE Developer can be found in Figure 1. On the left hand side the loaded components are listed. In the middle of the screenshot the annotated document can be found. The highlighted spans within the text refer to the annotation types that are displayed on the right hand side and have the associated color. Below the document a table of all annotations is provided. For each annotation its start and end positions are shown. Furthermore, a set of key-value pairs denotes the features of the corresponding annotation. Also, it is possible to edit annotations manually. For this, a highlighted span has to be clicked. If there are more than one annotation referring to this span, one has to select the annotation that shall be edited. It is possible to change the type or a feature of the annotation as well as its start and end position.

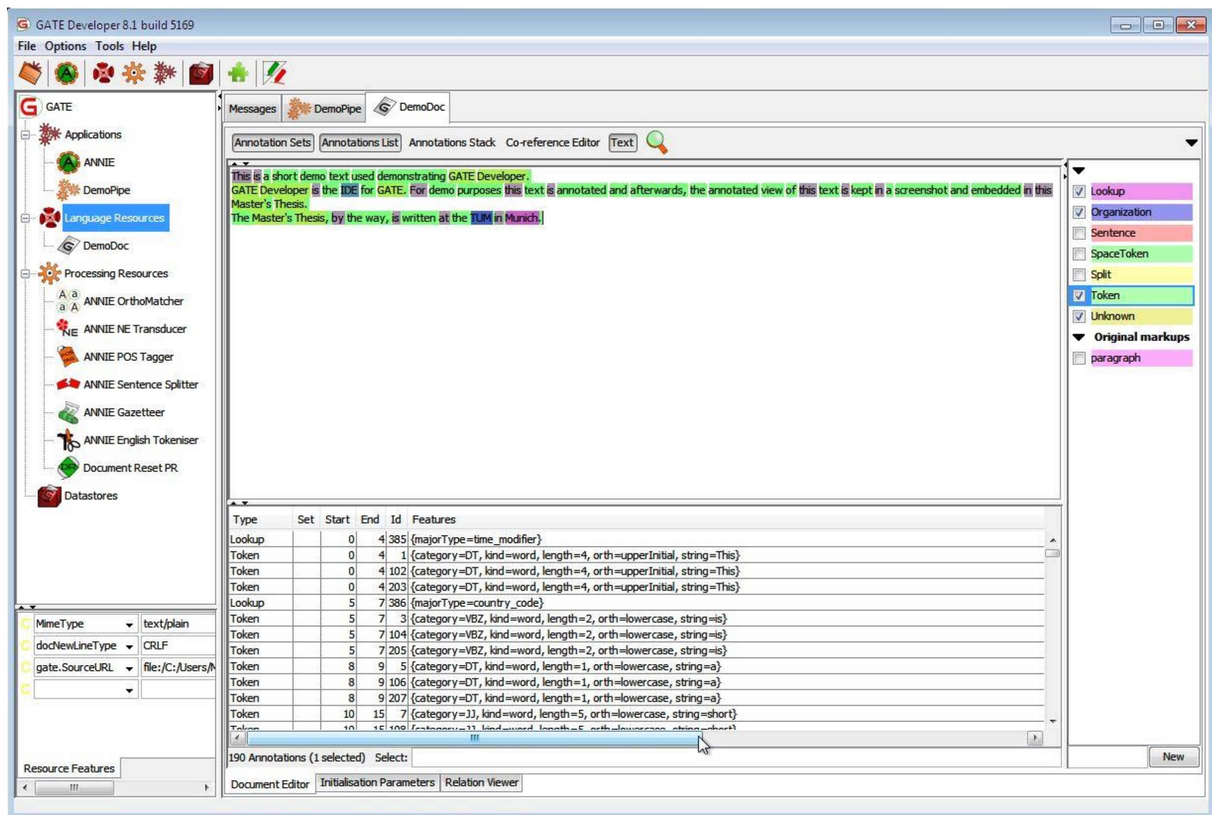


Figure 1: Screenshot of an annotated document within GATE Developer
Source: Screenshot taken from GATE Developer 8.1

2.2 BRAT

The brat rapid annotation tool (BRAT) is a web based tool, which has been developed in order to facilitate the manual annotation process (Stenetorp et al., 2012). BRAT is also collaborative, i.e. users can curate the same document and will see their changes in real time. New annotations can be added with selecting the span of text associated with this annotation. Annotations that refer to other annotations can be easily created with dragging from one annotation to the other. The result will be an arc between these annotations. BRAT also supports n-ary associations between annotations. Furthermore, BRAT provides a validation mechanism that checks whether an annotation fulfills all constraints of its type's schema. Additionally, an extensive search interface is provided that allows searching for text as well as for annotations. Annotations are represented in a BRAT-specific stand-off format but can be converted from and to common annotation formats. Moreover, the client part of BRAT can be integrated into an NLP application using it for the visualization of annotations computed by this very application. A screenshot of an annotated document within BRAT is given in Figure 2.

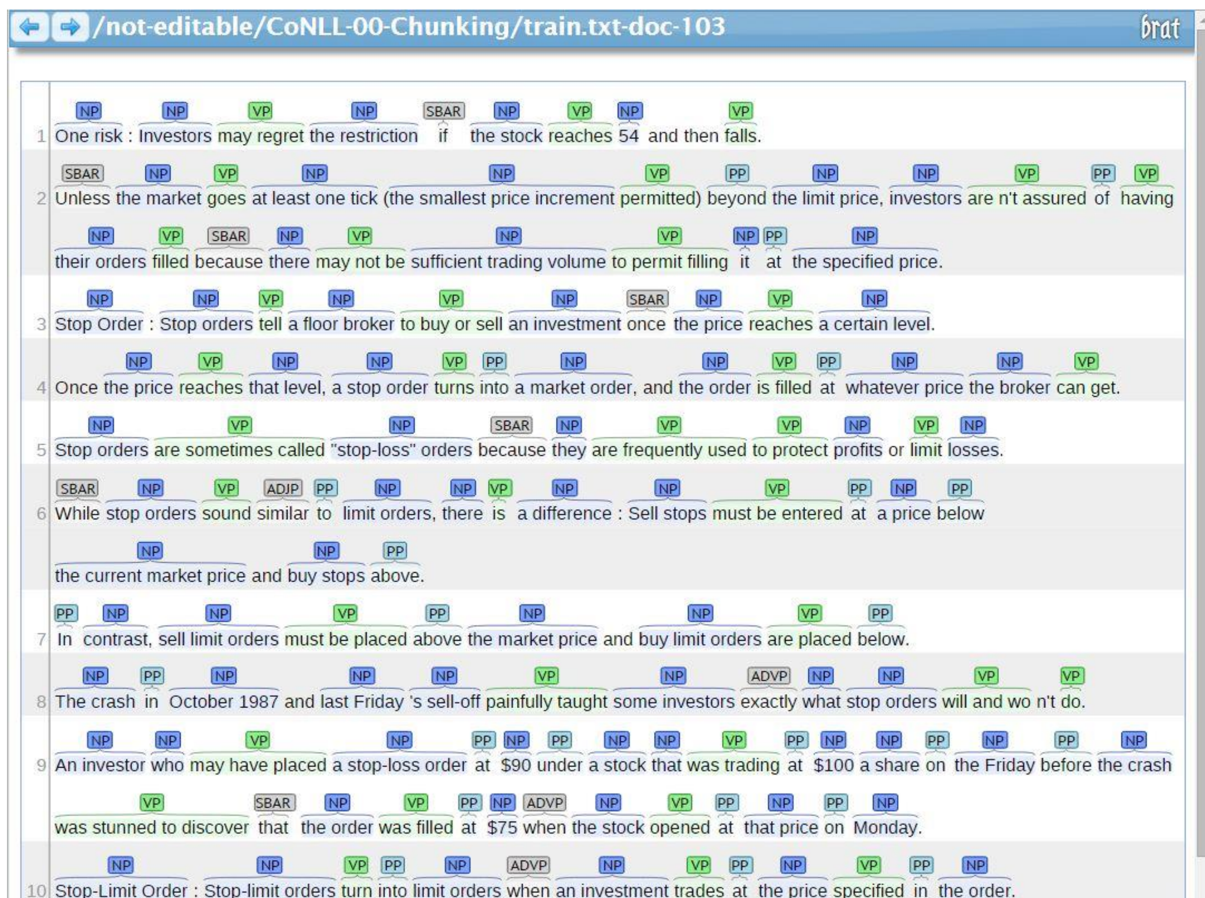


Figure 2: Screenshot of an annotated document within BRAT
Source: Screenshot taken from the live demo for BRAT [1]

2.3 U-Compare

U-Compare is a text mining system based on UIMA (cf. subsection *5.9 UIMA* for more details about UIMA) and mainly aims at providing a feature for comparing the results computed by analysis components (Kano, Baumgartner et al., 2009; Kano, Dorado, McCrohon, Ananiadou, & Tsujii, 2010). Users do not have to install U-Compare. Instead, an executable Java archive (JAR) is provided. A screenshot of U-Compare is provided in Figure 3. Workflows can be built by dragging a component from the available components listed on the right side and dropping it on the left side. These components are not included in the JAR file but rather accessed as web services. When a component is selected its expected inputs and computed outputs are denoted in the bottom right corner. In the foreground a visualization of an annotated document is shown. Manual editing of annotations is not supported by U-Compare.

[1] http://weaver.nlplab.org/~brat/demo/latest/#/not-editable/CoNLL-00-Chunking/train.txt-doc-103 (last access on 07.10.2015)

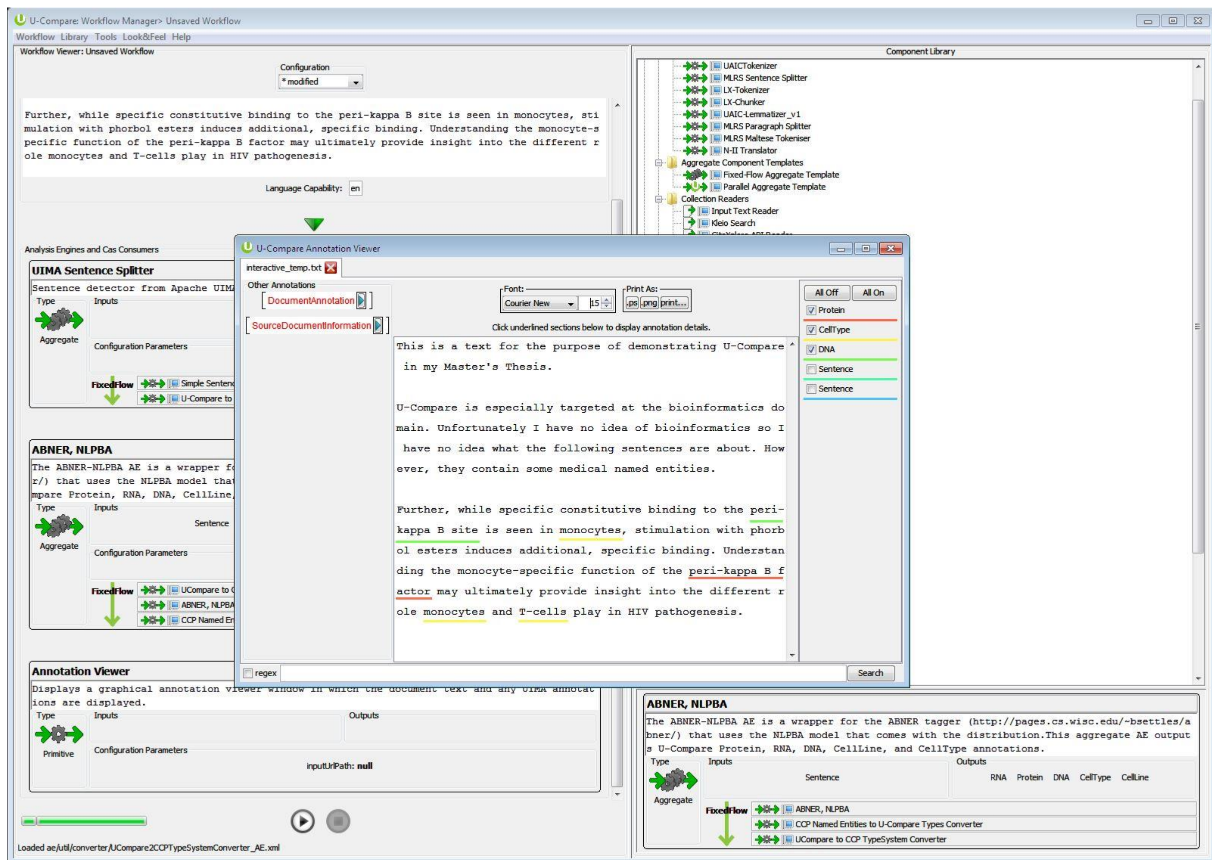


Figure 3: Screenshot of U-Compare
Source: Screenshot taken from U-Compare 1.1.8

2.4 Argo

Argo is a web application allowing the individual combination of components into workflows (Rak, Batista-Navarro, Rowley, Carter, & Ananiadou, 2013; Rak, Carter, Rowley, Batista-Navarro, & Ananiadou, 2014; Rak, Rowley, Black, & Ananiadou, 2012; Rak, Rowley, Carter, & Ananiadou, 2013). It is inspired by U-Compare and is also based on UIMA. It does not only allow for automatic annotation but also for manual annotation. Workflows can be created by selecting components and linking them. The result is a directed graph of components representing the workflow. These workflows can be saved and, like documents, be shared with other users. Figure 4 provides a screenshot of a document after it has been annotated. On the left side the documents are listed. In the middle the annotated document can be seen. On the right side a list of annotations is provided. After expanding such an annotation its features can be edited. Within the tab *Labels* it can be selected which annotation types shall be highlighted within the document. To create a new annotation one has to click on the green *Create* button at the top. A list of annotation types will pop up and after selecting one the new annotation will be displayed on the right side and its features can be edited.

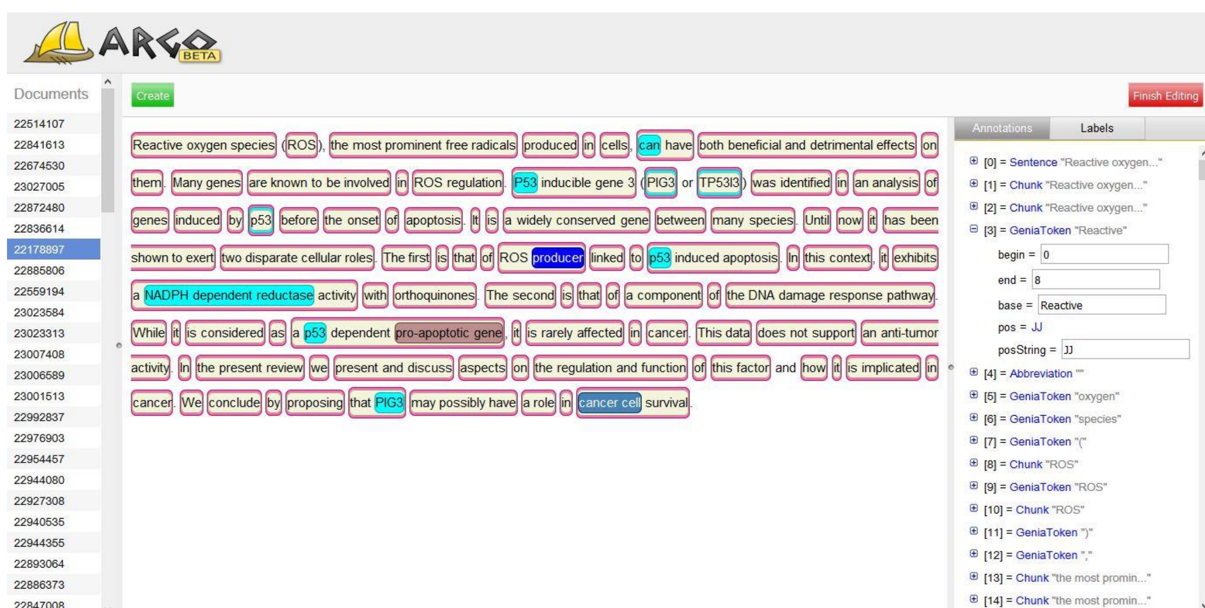


Figure 4: Screenshot of an annotated document in Argo
Source: Screenshot taken from Argo [2]

2.5 WebLicht

The Web-Based Linguistic Chaining Tool (WebLicht) is another web application that allows to build custom workflows (Hinrichs, Hinrichs, & Zastrow, 2010). When building a workflow WebLicht automatically determines which components can be appended based on whether the input requirements of these components are already met by the workflow. If these requirements are not fulfilled for a component, this very component is not offered. Figure 5 shows a screenshot of this step. The upper half contains all components whose input requirements are met and thus, can be appended to the workflow shown in the lower half of the screenshot. As this workflow has already been processed there are icons below each component that allow for downloading and viewing their results. For the visualization of the results WebLicht provides a table view as well as a graphical view if the annotations refer to each other with arcs. For the latter case BRAT (cf. subsection *2.2 BRAT*) has been integrated.

In addition to that, WebLicht is also a service oriented architecture (SOA) that provides its components as web services that can be accessed programmatically. In order to achieve compatibility between the components of WebLicht, annotations have to be represented in the stand-off Text Corpus Format (TCF) (Heid, Schmid, Eckart, & Hinrichs, 2010).

[2] http://argo.nactem.ac.uk/app/ (last access on 08.10.2015)

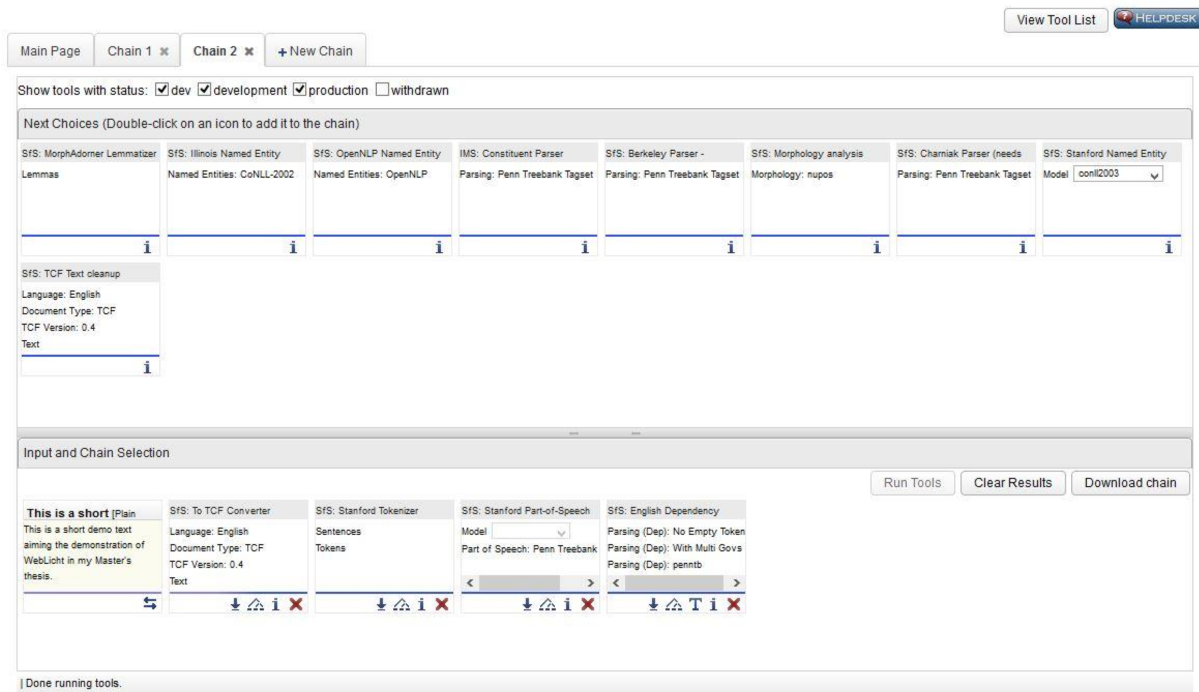


Figure 5: Screenshot showing the creation of a workflow in WebLicht
Source: Screenshot taken from WebLicht [3]

2.6 WebAnno

WebAnno is a web based annotation tool supporting project management and different user roles like project manager, annotator, or curator (Yimam, Eckart de Castilho, Gurevych, & Biemann, 2014; Yimam, Gurevych, Eckart de Castilho, & Biemann, 2013). Depending on the project type annotations can be created automatically or manually. In the former case the project manager has to upload training data for the annotation layers to be created. In the latter case one or several annotators manually annotate the document. Figure 6 shows a screenshot of this manual annotation process. As it can be seen in the background for the visualization of the annotations BRAT (cf. subsection *2.2 BRAT*) is used. The available annotation types and their features are defined by the project manager. In the foreground of the screenshot a pop up for editing an annotation can be seen. In this case the word "screeshot" has been marked as a *Typo* which is an annotation type I had introduced before in the project management view. The annotation contains a feature *Correction* having the value "screenshot" in this example. When the annotator has completed annotating the document, he clicks on the *Done* link in the upper right corner. Afterwards, a curator or the project manager can view these annotations. In case of multiple annotators annotating the same document the curator sees the annotations of all

[3] https://weblicht.sfs.uni-tuebingen.de/weblicht/ (last access on 08.10.2015)

annotators and can merge these version into one resulting document. Documents can also be exported to and imported from several standardized representation formats like, the above mentioned TCF.

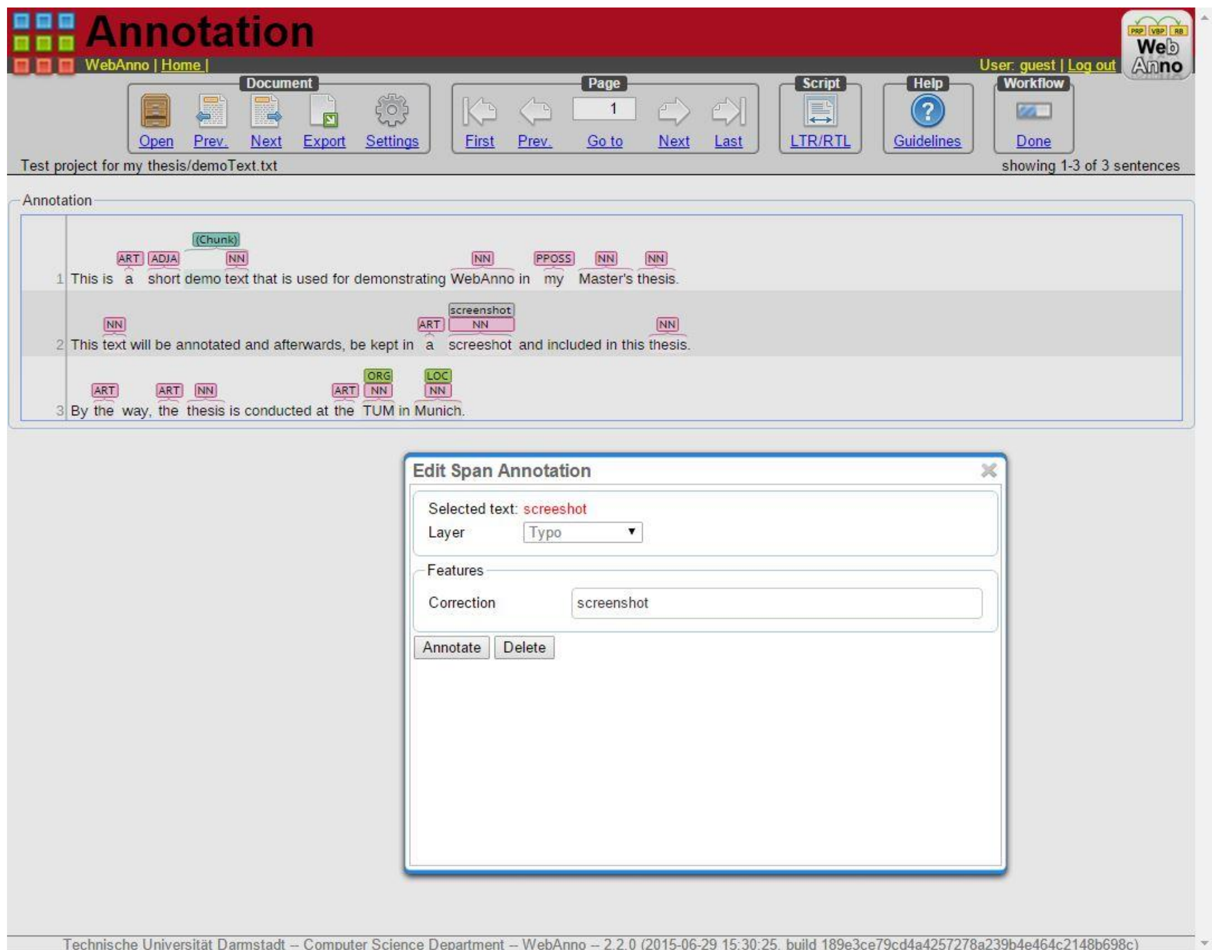


Figure 6: Screenshot showing the manual annotation in WebAnno
Source: Screenshot taken from WebAnno 2.2.0 [4]

2.7 AnnoMarket

AnnoMarket is an open cloud based platform where GATE components and resources can be bought and sold on a pay-per-use basis (cf. subsection *5.8 GATE* for more details on GATE). It is based on GATECloud.net and runs its components in the Amazon Cloud (Dimitrov et al., 2014; Tablan, Bontcheva, Roberts, Cunningham, & Dimitrov, 2013; Tablan, Roberts, Cunningham, & Bontcheva, 2011, 2012). The service can be accessed via its graphical user interface (GUI) with a browser as well as via its REST[5] application programming interface

[4] https://maggie.lt.informatik.tu-darmstadt.de/webanno/ (last access on 08.10.2015)
[5] Representational State Transfer

(API) for programmatic access. Figure 7 shows a screenshot of an annotated document in AnnoMarket.

Figure 7: Screenshot of an annotated document in AnnoMarket
Source: (Tablan, 2013)

3 Research method

Regarding the research method I roughly oriented to the design science approach described in (Hevner, March, Park, & Ram, 2004). While behavioral science is characterized by the development and verification of “theories that explain or predict human or organizational behavior” (Hevner et al., 2004, p. 75) the design science approach suggests to develop new artifacts.

Before starting to develop the prototypical implementation of the web application, which represents the artifact according to the design science approach, a comprehensive literature review has been conducted. Desired outputs of this literature review have been

- an overview of related work,
- a set of requirements for such a web application, and
- a set of existing architectures for NLP applications.

For an early evaluation a web application that is capable of annotating basic linguistic information like POS tags has been demonstrated in expert interviews (see Figure 22 in the appendix for a screenshot). Using this web application interviewees were provided a more detailed look & feel of such an application than it would have been possible with static mockups. The interviewees were asked to imagine that the software would provide information that is relevant for the legal domain in the same way as it did with the linguistic information. This web application had been developed at a previous project.

In order to gather requirements specific for the legal domain the interviewees have been asked what information they would appreciate from such a tool and what additional features could alleviate their work.

Based on these results the prototypical implementation of the web application has been developed.

4 Requirements

The first step of developing a workbench that embeds NLP technologies in order to enrich legal texts with useful meta-information is to conduct a requirements analysis. As described in the previous chapter, the requirements have been elicited via a literature review as well as interviews with experts from the legal domain. In this chapter the essential requirements of the workbench are presented and it is explained why the workbench should fulfill these requirements and what consequences they implicate. For this, the requirements are divided into functional and nonfunctional requirements. The former ones are further subdivided into requirements that apply for such a workbench in general and requirements that are specific for a workbench that is tailored to the legal domain. The following table gives an overview and indicates the priority on a scale of one to three for each requirement for the prototypical implementation where one is the highest and three is the lowest priority. As the implementation is a prototype incorporating a sustainable architecture and being the fundament for extending the workbench, the main focus is on architectural requirements. This is the reason why these architectural requirements have in general been higher prioritized that the functional ones.

Functional / Nonfunctional	**Requirement**	**Priority**
Functional	The workbench should support adding, removing, and editing of annotations	1
Functional	The workbench should support the persistence of annotations	1
Functional	It shall be possible to fold sections of the displayed text	3
Functional	It shall be possible to leave own comments in the documents	3
Functional	It shall be possible to set bookmarks in the documents	3
Functional	It shall be possible to edit the texts	1
Functional	It shall be possible that multiple users work on the same document and track their changes	3

Functional	The workbench shall feature a comparison of documents and their different versions	3
Functional	The workbench shall be able to import documents with different formats	1
Functional	The workbench shall allow for exporting documents in different formats	3
Functional	The workbench shall provide information about incoming and outgoing references	2
Functional	The workbench shall annotate legal definitions	2
Functional	The workbench shall annotate exceptions of legal norms	2
Functional	The workbench shall annotate legal consequences	2
Functional	The workbench shall provide linguistic information	1
Nonfunctional	The workbench should be a web application	1
Nonfunctional	The system's architecture should foster reuse of components	1
Nonfunctional	The system's text mining engine should incorporate a common type system for the created annotations	1
Nonfunctional	The system's text mining engine should comply with a standardized data format for data exchange between its components	1
Nonfunctional	It should be easy to integrate and to interchange foreign components	1
Nonfunctional	The system's text mining engine should support parallel processing of NLP tasks	2

Table 1: Overview of requirements and their priority for the prototypical implementation

4.1 Functional requirements

4.1.1 General requirements

These requirements apply in general for such a workbench and do not focus explicitly on the legal domain.

The workbench should support adding, removing, and editing of annotations

The workbench shall not only allow for annotating text with NLP components but also for managing those annotations manually by the user (Yimam et al., 2013). I.e. the user shall be able to add own annotations. This includes not only assigning existing types of annotations to a range within the text but also defining new, custom types of annotations. Additionally, it shall be possible to drop annotations, set manually or calculated by the system, via the user interface (UI). Finally, the user has to be able to edit annotations. Editing annotations means to change the type of an annotation as well as to adjust the borders of the annotation's range. This requirement is mainly important because natural language processing is an error-prone discipline by its very nature and thus, the user should be able to correct the calculated results. An important consequence of this requirement is that annotations should be kept separate from the original document to be able to efficiently access these annotations and change them without affecting the other annotations.

Besides viewing the annotations within the text it shall also be possible to query annotations and to retrieve them as a list. For example, a user shall be able to get a list of all annotations having the type "incoming reference".

The workbench should support the persistence of annotations

As experiences of previous projects have shown, the processing of NLP tasks is complex and hence time-consuming. Therefore, it would be distracting for the user if all annotations would have to be calculated every time afresh when a document is opened. Also, annotations which have been manually added or edited would be lost. Due to these reasons the annotations should be persisted and loaded when the respective document is opened. The user therefore has to be provided a menu where it can be selected which annotations are marked in the text and which ones shall be turned off so that only the information is displayed which is currently relevant for the user.

It shall be possible to fold sections of the displayed text

As legal literature like laws, legal commentaries, judgements, etc. are mainly long texts, it should be possible to collapse and expand parts of these documents. In laws, for example, the user shall be able to fold articles that are irrelevant for their current work and expand articles which are not (Splinter, 2015). This feature will increase the lucidity and therefore decrease the expenditure of time while working with those documents.

Another idea in this context is to indicate the user's current position within the text. One possibility to achieve that is to generate a table of contents based on the document's headlines and display it besides the text. The headline of the current position could be highlighted (Riederer, 2015). Another possibility is to provide breadcrumbs. Both approaches will prevent the user from getting lost in the document. The former approach would also support the navigation within the document if the headlines in the table of contents would link to their corresponding position.

It shall be possible to leave own comments in the documents

To further increase the efficiency while working with legal literature it is necessary to provide a feature that allows for leaving own comments associated with a specific span of text (Riederer, 2015). Even though this is not only applicable for legal literature, commenting is something that is already extensively used when working with legal literature. While this is done on paper at the moment, digital comments offer plenty of new possibilities: They can easily be changed or deleted and they can be found much easier and faster using a search feature, for example.

It shall be possible to set bookmarks in the documents

Especially in long texts it is time consuming to find a specific passage in the text. A feature allowing for setting bookmarks within the documents leads to a fast access of frequently used text passages and thus, avoids this unnecessary expenditure of time (Riederer, 2015).

It shall be possible to edit the texts

The workbench shall allow for semantic analysis and annotation of legal literature like laws, judgements, or legal commentaries. This kind of documents is authored by judges and the legislative organ rather than by attorneys. However, judges are also possible users. Using the workbench they can formulate such literature being supported by semantic information. For example, they could get a hint if they use a term which is not defined in a legal definition.

Furthermore, lawyers shall be supported in writing their arguments. For these reasons the workbench shall not only allow to work with static texts but also for editing those texts.

This is specified as a dedicated requirement because it comes with some challenges regarding the annotation of the documents. For example, if a word is changed from a noun to a verb the previously calculated POS tag of this word has become invalid. This issue also applies for more complex patterns. Moreover, the offsets of the annotations change when the document is changed.

It shall be possible that multiple users work on the same document and track their changes

At chambers it is common that different people work on the same document. For instance, one employee performs an initial analysis, another bridges the gaps, and a third reuses the document for further work. It should also be possible to track the changes to see who is responsible for which changes. In addition to that, users should be able to comment their changes like it is done in version control systems. It shall also be possible to use hyperlinks for linking to other articles or relevant commentary. This enables the users to justify their changes and prove them with the relevant literature (Riederer, 2015).

The workbench shall feature a comparison of documents and their different versions

A consequence of the previous two requirements is that users will have different versions of a document. To be able to compare those versions users should be provided a view showing the difference of the texts, similar as it is known from version control tools.

Additionally, it shall be possible to compare the semantic data in the form of annotations of the documents. Therefore, users shall have a dashboard where they can compare meta-information of annotations for selected versions. An example for such meta-information is the frequency of an annotation type like “incoming reference” within a document. Another example is meta-information not directly related to annotations but to the document as a whole like its readability. Therefore, several readability indices like the *Flesch Reading Ease* or the *Wiener Sachtextformel* exist (Amstad, 1978; Bamberger & Vanecek, 1984; Flesch, 1948). Such readability indices may be particularly interesting for the legislator as this information aids in writing laws that can be easily understood by the public. Whereas the text difference between two completely different documents does not provide any value for the user such metrics like

the Flesch Reading Ease or the length of the text may not only be compared within different version of the same document but also between completely different documents.

The workbench shall be able to import documents with different formats

Before the semantic analysis of the literature can be performed, first of all the literature has to be imported into the workbench. As the literature is available in different data formats, the application has to be capable of importing files with several data formats (Bontcheva, Cunningham, Roberts, & Tablan, 2010; Cunningham, 2002). German laws, for example, exist as XML files whereas judgements exist as PDF[6] files. Therefore, it is important that the workbench can deal with several file formats which includes parsing, preprocessing, and the extraction of the raw text which is then passed to the text mining engine for semantic analysis.

Furthermore, the importer part shall be developed in a generic way so that additional importers can easily be integrated. Such an additional importer may deal with another file format but also with a new kind of literature. As the kind of literature implies its structure, there have to be different importers for different sorts of literature. For example, a law is structured hierarchically in paragraphs, sections, and sentences. In contrast to that, judgements can rather be divided into the decision and the reasons upon which the decision is based. However, even though the workbench shall fulfill this requirement, it is not the focus of this work. In fact this topic is explored in (Graß, 2015b).

The workbench shall allow for exporting documents in different formats

To use the documents after working with them with the workbench it is required that those documents can be exported. Most chambers - and most companies in general – use Microsoft Word for writing text documents. As a consequence, the workbench should enable users to export their documents into Microsoft Word (Splinter, 2015).

4.1.2 Requirements specific for the legal domain

The following requirements are especially relevant for a workbench that shall be tailored to the legal domain and cover mainly important annotation types for this domain.

[6] Portable Document Format

The workbench shall provide information about incoming and outgoing references

Authors of legal literature make extensive use of references. For example, when examining whether an article is applicable for a specific issue, mostly a set of other articles also have to be examined because they may be referred to as prerequisites or as legal consequences. Especially because these referenced articles are widely spread, it would save a huge amount of time when those outgoing references would be linked to the corresponding text passages (Splinter, 2015).

In addition to that, it would also be helpful to get a list of incoming references. An incoming reference is an outgoing reference vice versa, i.e. an article *A* has an incoming reference from another article *B* if article *B* references article *A*. Having a list of incoming references provides a quick overview over all usages of this article (Heßler, 2015).

The workbench shall annotate legal definitions

Legal definitions define a term and imply what is meant when this term is used in the respective document. This information is especially valuable as it often happens that the same term is used in different contexts and therefore have different meanings. Hence, the workbench should be capable of detecting such terms and provide their corresponding definitions. This will also save time since it makes it obsolete to look these definitions up (Splinter, 2015). One example for such a legal definition is § 2 ProdHaftG, which defines the term „product“ for the rest of the law: “Produkt im Sinne dieses Gesetzes ist jede bewegliche Sache, auch wenn sie einen Teil einer anderen beweglichen Sache oder einer unbeweglichen Sache bildet, sowie Elektrizität” (ProdHaftG, § 2). However, this detection of legal defined terms is an error-prone process due to the fact that the term might still be used with a meaning that is different from the respective legal definition (Heßler, 2015).

The workbench shall annotate exceptions of legal norms

As already mentioned above, it is necessary to check several preconditions when examining whether a certain article is applicable for a case. Furthermore, it is required to check for applicable exceptions. Many articles contain some conditions which do not apply for the case that some other conditions are fulfilled (Heßler, 2015). When these exceptions are marked, they will catch the reader’s eye so that the risk of overlooking them can be reduced. This is particularly worthy when those exception do not follow the condition immediately but are rather contained in a different article (Splinter, 2015). An example formulation for an exception within the same article can be found in § 13 ProdHaftG Abs. 1: “Der Anspruch nach § 1 erlischt zehn

Jahre nach dem Zeitpunkt, in dem der Hersteller das Produkt, das den Schaden verursacht hat, in den Verkehr gebracht hat. Dies gilt nicht, wenn über den Anspruch ein Rechtsstreit oder ein Mahnverfahren anhängig ist" (ProdHaftG, § 13 Abs. 1). Here, the first sentence defines when a claim expires whereas the second sentence formulates an exception.

The workbench shall annotate legal consequences

Articles in German laws can be divided into articles which contain a legal consequence and articles which do not (Riederer, 2015). An example for a legal consequence is formulated in § 1 ProdHaftG: "Wird durch den Fehler eines Produkts jemand getötet, sein Körper oder seine Gesundheit verletzt oder eine Sache beschädigt, so ist der Hersteller des Produkts verpflichtet, dem Geschädigten den daraus entstehenden Schaden zu ersetzen". The first part of the sentence specifies the precondition of the article while the second part defines the consequences. To detect at first glance to which group an article belongs it would be helpful if such legal consequences will also be labeled (Heßler, 2015).

The workbench shall provide linguistic information

This requirement is targeted at users that use the workbench for writing their own texts. Linguistic information aids in writing good texts. Therefore, spelling mistakes and grammar errors shall be highlighted. Moreover, texts tend to be too long (Splinter, 2015). To avoid this the user shall see how many words the text contains. Additionally, it is important to keep a text as readable as possible. For gauging this several readability indices like the already mentioned Flesch-Reading-Ease or the Wiener Sachtextformel exist (Amstad, 1978; Bamberger & Vanecek, 1984; Flesch, 1948). The workbench should provide the result of such an index for the text so that the author knows whether his text is easy to understand or too complex.

4.2 Nonfunctional requirements

These requirements mainly refer to the software architecture of the system.

The workbench should be a web application

One reason for this requirement is the increased usability as there is no installation required (Eckart de Castilho & Gurevych, 2014; Hinrichs, Hinrichs, & Zastrow, 2010; Rak, Rowley, Black, & Ananiadou, 2012). Another advantage is the independence of a specific platform compared to desktop software, which is constrained to a specific operating system (OS) or at least to a virtual machine like the Java Virtual Machine. Moreover, the user's data is stored centrally on a webserver which allows for usage with multiple devices without having to

concern about synchronization. In addition to that, NLP tasks can be very complex and time consuming (Kano et al., 2010). When those tasks are processed remotely, which might also include cloud computing, it is still possible to use the workbench on simple computers with weak processors (Yimam et al., 2013). One important consequence of this requirement is that the workbench will be used by multiple users simultaneously and therefore the application has to be thread safe.

The system's architecture should foster reuse of components

To meet the goal of developing a workbench with a sustainable architecture this very architecture should be designed in a way that fosters reuse of its components for several tasks (Eckart de Castilho, 2014). This leads to less redundancy and a better maintainability compared to the approach of developing very large components, which handle a whole use case but cannot be used for other use cases (Hahn et al., 2008; Wolinski, Vichot, & Gremont, 1998). This requirement is highly related to the next two requirements because a common type system and a standardized data format for data exchange between the components are prerequisites for combining components of different developers with different use cases in their minds. Being able to reuse components and combining them with other components enables building complete new workflows for information extraction. For example, some tasks like tokenization and segmentation are recurring tasks that are the first step for most NLP workflows. A basic component performing such a task can then be reused within several workflows of different NLP tasks. Another design principle is to configure components with parameters. For instance, such a parameter could be the language of the document. Passing such a parameter to a component avoids using different components for the same task with only minor differences in the setting (Bank & Schierle, 2012).

The system's text mining engine should incorporate a common type system for the created annotations

In the course of time much effort of different groups has been conducted in researching NLP technologies and in developing software components applying those technologies to perform dedicated tasks (Hahn et al., 2008). As a consequence of being developed by different groups, components are often incompatible with each other. And worse, these components often lack of a well-specified API defining the expected input and output which makes it tedious to integrate them into an application (Gambäck & Olsson, 2000). Most components need a preprocessing of the language resource which has to be performed by another component. Thus,

a component has to work with the results of prior components. For example, a sentence splitter which is a component that divides a given text into its sentences needs the result of a component that extracts all tokens of the text. Therefore, the tokenizer will analyze the text and create annotations for each token. To be able to work with those results the sentence splitter needs to know the types of the annotations produced by the tokenizer. If a period at the end of a sentence is marked as an annotation of the type "period" but the sentence splitter uses a different tagset and thus looks for annotations of the type "point", for instance, these two components would be incompatible. Requiring a common types system (cf. subsection *1.2.4 Type system*) to be incorporated by the used components avoids this incompatibility (Eckart de Castilho & Gurevych, 2014; Hahn et al., 2008; Kano, Baumgartner et al., 2009; Thompson et al., 2011).

The system's text mining engine should comply with a standardized data format for data exchange between its components

Another issue is the way of passing data between components. Even though two components use the same types of annotations, they are still incompatible if the first component emits its results in an XML format but the second one expects a Java object holding the annotations (Thompson et al., 2011). Hence, it is important to not only comply with common annotation types but also a standardized format of the data exchanged by different components to avoid such misunderstandings.

It should be easy to integrate and to interchange foreign components

To increase the maintainability of the system it is important that the components have a high cohesion and are loosely coupled. High cohesion means that the components form logical units each performing a specific task. A high cohesion of components often implies a loose coupling, which means that components only loosely depend on each other (Gomaa, 1984; Gui & Scott, 2006; Yourdon & Constantine, 1979). This is usually achieved with data encapsulation and information hiding combined with well-defined interfaces (cf. (Cohen, 1984; Parnas, 1972)). Following this approach helps to easily interchange components. As long as a new component fulfills the specification of the interface there will not be any side effects. In addition to that, the architecture should allow for the integration of components that may also be written in a different language to broaden the spectrum of components that can be used for the workbench (Comeau et al., 2013; Wolinski, Vichot, & Gremont, 1998).

Furthermore, an easy integration of new components also comes with the advantage that the architecture allows for processing documents in several languages. If the workbench shall be extended to support a new language, everything that has to be done is the integration of the according components.

The system's text mining engine should support parallel processing of NLP tasks
Due to their complex nature NLP components have a high consumption of computing resources particularly processor capabilities (Kano, Dorado, McCrohon, Ananiadou, & Tsujii, 2010; Petasis, 2010). Especially as the workbench is a web application and thus has potentially a lot of users, the application shall support parallel processing of NLP tasks so that a reasonable performance can be achieved (Bank & Schierle, 2012; Cunningham, Bontcheva, Tablan, & Wilks, 2000; Petasis, 2010). On the one hand this means parallel processing of several documents by different components. I.e. for a document of one user POS tagging is performed while at the same time for another user's document some kind of named entity recognition is executed, for instance. On the other hand parallel processing also includes the parallel execution of several NLP components for the same document (Wolinski et al., 1998). The latter part of this requirement, of course, has to ensure that a component *A* still has to be executed before component *B* if the results of component *A* are required as input of component *B*. For example, if both, a component performing a named entity recognition and a sentence splitter, only need the results of a tokenizer as input, the NER component and the sentence splitter could be executed in parallel after the tokenizer has finished its work.

4.3 Delimitation of the requirements

In the following chapter *5 Assessment of existing architectures* several software architectures for NLP applications are presented. As there is also research regarding information extraction from audio and picture data, it is important to mention that this thesis and the workbench is only targeted at dealing with textual data even though some of the presented architectures are not limited to this kind of documents.

5 Assessment of existing architectures for NLP systems

In this section a description and comparison of existing software architectures for NLP systems is provided. First it is described what the key concepts of the respective architectures are. The focus for that will be how annotations are represented, whether components can easily be interchanged, how they can be combined to workflows, and whether it is possible to perform NLP tasks in parallel. After that, for a subset of the requirements that where examined in the previous section it is analyzed how and how well the architecture fulfills them. This subset consists of those requirements that have an impact on the text mining engine's architecture:

- The workbench should be a web application
- The system's architecture should foster reuse of components
- The system's text mining engine should incorporate a common type system for the created annotations (cf. subsection *1.2.4 Type system*)
- The system's text mining engine should comply with a standardized data format for data exchange between its components
- It should be easy to integrate and to interchange foreign components
- The system's text mining engine should support parallel processing of NLP tasks

Finally, the findings are summed up in Table 2 and it is explained for which architecture it has been decided and what the reasons have been.

5.1 TIPSTER

5.1.1 Description

The TIPSTER program has been a research program with the goal to provide a system for document detection and information extraction especially for governments (Grishman, 1996). The architecture should scale to large text collections and also provide support for multilingual applications. Additionally, it has been a requirement to allow the integration of different modules from different developers into the system.

The most central object of the TIPSTER architecture is the document object which holds all information about its text using attributes and annotations. Information about the whole document is stored in its attributes whereas information targeting at parts of the document is stored in annotations. Each document belongs to at least one collection object that contains several documents.

Representation of annotations

Annotations are stand-off annotations and stored in an object holding all annotations for a document and providing efficient access to them. Annotations are typed and their types have to be declared. Although developers can specify almost any annotation type, there is a minimal set of annotation types already predefined. This set consists of very basic annotation types that are fundamental for almost every NLP task. Tokens, for instance, will be annotated with an annotation having the type *token*. Requiring compliance with such a basic set of annotations aims at easier integration of new components because developers can rely on the fact that whatever tokenizer will perform the tokenization process it will annotate the tokens as *token*. TIPSTER also provides an external representation of annotations using the Standard Generalized Markup Language (SGML). It offers means to convert annotations into SGML as well as to read from SGML and convert it into the internal representation.

Modularity and interchangeability

There are no detailed specifications for processing resources (Bank & Schierle, 2012). The only effort made to foster the interchangeability and an easy integration of components is to require a minimal common set of basic annotation types and the possibility to write and read annotations in an SGML representation. Also, the architecture does not cover the management of resources and it does not specify how to build workflows that combine several components.

5.1.2 Assessment of requirements

The workbench should be a web application

As there is no specification regarding the components and thus, one cannot rely on thread safety of components, this requirement is missed.

The system’s architecture should foster reuse of components

As there do not exist any detailed specifications of how to combine components into workflows and there is no configuration management on the part of the architecture, this requirement is not met.

The system’s text mining engine should incorporate a common type system for the created annotations

The architecture requires components to share a minimal set of annotation types for the very basic annotation types. It is possible to extend this type system and create own annotations arbitrarily. Therefore, the TIPSTER architecture fulfills this requirement thoroughly.

The system’s text mining engine should comply with a standardized data format for data exchange between its components

Components can access the document’s annotations via its *AnnotationSet* efficiently. Moreover, if a task shall be outsourced to an external component that is not integrated into the application it is possible to write the annotations in an SGML representation. If the external component is capable of processing SGML and returns its output in SGML as well, this output can be converted back into the internal representation again and is available for further processing. For these reasons this requirement is also fulfilled exhaustively.

It should be easy to integrate and to interchange foreign components

Due to the lack of workflow management it is not clear how to combine and exchange components. As long as two components consume the same annotation types as input and produce annotations of the same types it should be possible to exchange them without any side effects. This requirement is only partially fulfilled.

The system’s text mining engine should support parallel processing of NLP tasks

There is no specification of how components have to be implemented. Therefore, one cannot rely on their thread safety when integrating them. As a consequence, parallel processing is not possible.

5.2 Ellogon

5.2.1 Description

Ellogon is an architecture that is based on the TIPSTER architecture. Its main goals are to be multilingual by using Unicode, to support multiple platforms, being modular and to ensure an efficient consumption of hardware resources. It comes with a framework, some already included components, a GUI, and an IDE. Its core is implemented in C and can be accessed via an object oriented C++ API which itself is exposed to languages like C++, Java, Tcl, Perl, and Python (Petasis, Karkaletsis, Paliouras, Androutsopoulos, & Spyropoulos, 2002; Petasis, Karkaletsis, Paliouras, & Spyropoulos, 2003). An illustration of the architecture can be seen in Figure 8.

The data model is the same as in the TIPSTER architecture and comprises collections that contain documents holding a text and linguistic information about this text in form of attributes and annotations.

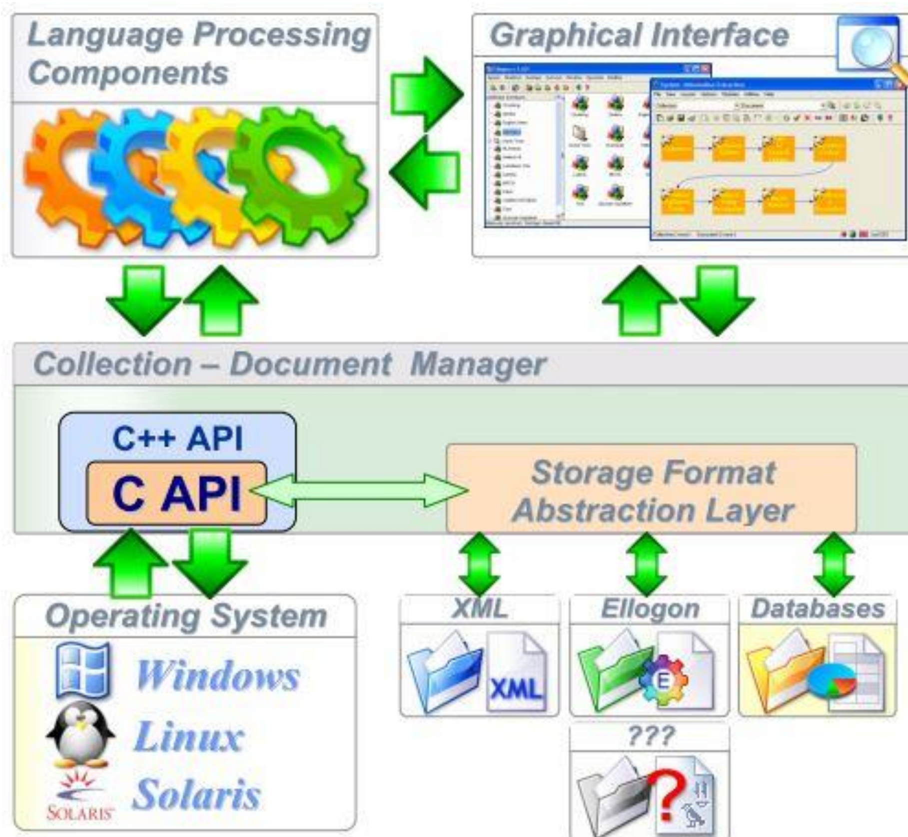


Figure 8: Illustration of the Ellogon architecture
Source: (Petasis, Karkaletsis, Paliouras, & Spyropoulos, 2003, p. 2)

For rule-based information extraction Ellogon provides the Ellogon Pattern Engine (Petasis, 2014). It is loosely based on the Common Pattern Specification Language (CPSL) developed during the TIPSTER program and provides an engine for matching annotations and their features with context-free grammars (Appelt & Onyshkevych, 1998). For each match actions like the creation of a new annotation, for instance, can be defined.

Representation of annotations

Annotations in Ellogon are stand-off. Annotations consist of an identifier, a type, a set of spans having a start and end position within the document's text, and a set of attributes. It is important to mention here that the type of the annotation is only a textual description and that therefore the annotations are not typed in a way allowing for type validation. I.e. if two components produce an Annotation having the type "Token", for instance, this does not necessarily mean that they incorporate the same data structure. For external representation of annotations they can be exported to SGML and XML.

Modularity and interchangeability

Ellogon is a modular, decomposable architecture that allows the integration of external components as well as the integration of Ellogon components into other applications. A guide for developers that describes how to develop a component for Ellogon as well as how to import external components into Ellogon makes it easy to create and interchange components within the Ellogon architecture (Afantenos, Petasis, & Karkaletsis, 2002).

Workflow management

Components can be combined into a workflow called *System* in Ellogon (Petasis et al., 2002, p. 3). A component has to define a set of pre-conditions and a set of post-conditions. The former specifies the expected input of linguistic information while the latter denotes the output of the component being the information that will be added to the document after the component has run successfully. Configuration and parametrization is possible with a configuration file. Components and resources like dictionaries, for instance, can be loaded at runtime.

Parallelization & distribution

Aiming at a reduction of memory usage Ellogon makes use of caching and reusing objects. In fact, memory usage could be drastically removed and therefore the number of documents being processed increased. The other side of the coin is thread safety. Even though the core

component of Ellogon is already thread safe, the caching mechanism will not prevent the cached objects from being accessed by different components running on different threads (Petasis, 2010). As this has not been solved yet, thread safety does not seem to be feasible without a major reimplementation of the architecture.

5.2.2 Assessment of requirements

The workbench should be a web application

As a web application implies usage by multiple users at the same time by its very nature thread safety is a prerequisite for this requirement. Due to the fact that the current implementation of Ellogon does not meet this prerequisite this requirement is not fulfilled.

The system's architecture should foster reuse of components

The architecture provides a pluggable component system. These components can be easily combined into several workflows. Configuration of components is possible with configuration files and parameters. For these reasons it can be said that the Ellogon architecture fosters reuse of components.

The system's text mining engine should incorporate a common type system for the created annotations

The type of an annotation can be chosen arbitrarily and is only a String containing a textual description of the type for categorization. Hence, there are no means for type validation. Thus, this requirement is not fulfilled.

The system's text mining engine should comply with a standardized data format for data exchange between its components

Components can directly access collections containing the documents that hold the annotations. In addition to that, it is possible import and export documents to SGML or XML. Accordingly, this requirement is satisfied.

It should be easy to integrate and to interchange foreign components

The architecture is modular. Moreover, there is a guide aiding in developing and embedding components. Due to the fact that processing results can also be imported from and exported to SGML or XML it also possible to use external components without having to embed them into the Ellogon application. Hence, this requirement is also considered as met.

The system's text mining engine should support parallel processing of NLP tasks

Since Ellogon applications are not thread safe, it is not possible to process NLP tasks in parallel. Consequently, this requirement is missed.

5.3 LIMA

5.3.1 Description

LIMA is also a TIPSTER based architecture. The key requirements that LIMA had to fulfill have been to be capable of handling multiple languages, to support the integration of new functionalities, and to process large corpora in an efficient manner. Besides the architecture it also provides tools and resources as well as an IDE for developing NLP-based applications. The implementation is done in C++ (Besançon et al., 2010).

Representation of annotations

Annotations are stand-off and stored in graphs. There is an analysis graph having tokens as nodes that link to their corresponding annotations. Its edges can either be sequential links to adjacent tokens or be syntactic edges as they are produced by a dependency parser, for instance. Furthermore, there is another graph which contains the actual annotations. The edges and nodes of the latter one can be associated to the edges and nodes of the former one resulting in a mesh. These graphs can be accessed from each component of the workflow. Annotations can be arbitrary data structures.

Modularity and interchangeability

LIMA is a modular architecture that provides means for configuration via configuration files. An exchange of a component can be done with a simple change in such a configuration file. LIMA itself can also be integrated into other applications by acting as a CORBA server or via its SOAP[7] interface.

Workflow management

Components as well as linguistic resources are loaded at runtime with the needed configuration incorporating the factory pattern. However, configuration has still to be provided in a static way using configuration files. Consequently, components cannot be configured based on the results gathered by previous components within the pipeline. This issue has been left open for future

[7] Originally an acronym for Simple Object Acces Protocol

work. Linguistic resources are managed independently from the processing resources, i.e. the components, so that they can be shared among them. This avoids loading the same resource for every component afresh.

Parallelization & distribution

Parallel processing has been addressed for future work in (Besançon et al., 2010). However, since then there has been only one publication about LIMA afterwards that focused on how they adapted LIMA for making it publicly available (Chalendar, 2014). Therefore, the question regarding how to exploit parallelism with LIMA seems still to be subject of further research.

5.3.2 Assessment of requirements

The workbench should be a web application

As parallel execution of components is not supported, LIMA does not fit the needs for usage in web applications that have to be capable of handling requests from multiple users simultaneously.

The system's architecture should foster reuse of components

Components can be configured and reused in several workflows. For instance, it is possible to use the same component for different languages using the corresponding configuration and language resources. However, as this configuration can only be done statically this requirement is only partially fulfilled.

The system's text mining engine should incorporate a common type system for the created annotations

Annotations can be represented by arbitrary data structures. Hence, there are no specifications regarding annotation types. Consequently, this requirement is not addressed by LIMA.

The system's text mining engine should comply with a standardized data format for data exchange between its components

Components can access the annotation graphs directly. Apart from that, there is no common external representation of annotations and such functionality has to be developed on one's own. However, as the shared annotation graphs fit the needs for internal components, this requirement is considered as fulfilled.

It should be easy to integrate and to interchange foreign components

Components can easily be interchanged with updating a configuration file. The integration, however, seems to be quite difficult. As there is no common type system, developers cannot rely on a specific input. Thus, a component depends strongly on other components and therefore implies a tight coupling between them. Consequently, this requirement is not met.

The system's text mining engine should support parallel processing of NLP tasks

As there has not been any publication addressing this issue since it was mentioned as future work in (Besançon et al., 2010), this seems to be a difficult task that has not been solved yet. Thus, this requirement is not fulfilled.

5.4 Whiteboard architecture

5.4.1 Description

The whiteboard architecture has especially been designed for providing means to integrate heterogeneous components into an NLP system. Heterogeneous in this context does not only mean the use of different data structures but also different programming languages and even the execution on different machines. This is made possible by the introduction of a central *coordinator*. Component processes are encapsulated in *managers* which perform the communication with the coordinator with the usage of a mailbox system. Each manager has a mailbox for the input and another one for the output. An illustration of this can be seen in Figure 9. The communication is done by the utilization of the simple mail transfer protocol (SMTP) (Zajac, Casper, & Sharples, 1997). This approach allows for reuse of partial results from other components running in parallel. As components do neither know other components nor the coordinator, the manager is also in charge of converting input and output data between the component's internal representation and its external representation (Boitet & Seligman, 1994). The architecture has originally been developed for speech translation systems which is also the use case for its prototypical implementation, called *Kasuga*, but claims to be applicable for all NLP systems where the integration of heterogeneous components has a high priority (Boitet & Seligman, 1994, pp. 429f).

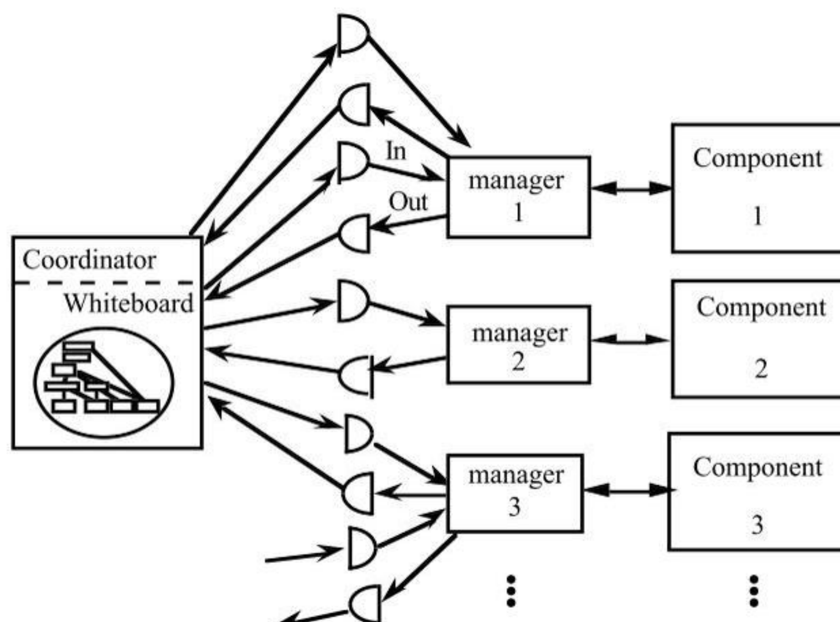


Figure 9: Illustration of the Whiteboard architecture
Source: (Boitet & Seligman, 1994, p. 427)

Representation of annotations

There are no constraints regarding the internal representation of a component's results. Those results, however, have to be converted by the component's manager which sends them to the coordinator. The coordinator holds a whiteboard where all already computed results are stored. Components have no direct access to this whiteboard but have to retrieve their input data via their manager. There is no specification of a type system as "it remains open what data structures the whiteboard itself should use" (Boitet & Seligman, 1994, p. 427).

Modularity and interchangeability

Components can be written in different languages and may run on different machines. It is required to "(…) completely declare the appearance of the partial results of the components on the whiteboard" (Boitet & Seligman, 1994, p. 426). I.e. developers of components have to exactly specify the expected inputs and outputs of their components. Therefore, the architecture is indeed very modular but interchangeability is still hampered due to the fact that the whiteboard's data structure is not standardized. The good thing about that is, that the architecture is not limited to a specific domain. The other side of the coin is that different developers will develop components that might be incompatible. The consequence is that not the developer of a component is responsible for developing a manager for the communication with the coordinator but rather the one who is integrating the component into his NLP system

with his individual whiteboard. Thus, although the main goal of the architecture is to provide a way to integrate heterogeneous components it implies a lot of work for doing so.

Workflow management

As there is not any specification of how to combine the components into workflows, this part remains open to the developer.

Parallelization and distribution

The whiteboard architecture is a distributed one allowing for components running on different machines. Furthermore, it is not only possible to execute them in parallel but components can even reuse partial results of other components which avoids unnecessary redundant work.

5.4.2 Assessment of requirements

The workbench should be a web application

The architecture allows for parallel execution of components which is one prerequisite for being implemented in a web application. However, there still has a mechanism to be implemented that ensures that the same component can be executed multiple times in parallel without interfering with its parallel executions. Especially as components may run on different machines, this will introduce a serious complexity issue as it implies all of these components on their different machines having to be thread safe. Firstly, this is an error-prone concept as this mechanism has to be implemented on every machine. Secondly, it introduces lots of extra work. Therefore, the whiteboard architecture does not meet this requirement sufficiently.

The system's architecture should foster reuse of components

The architecture is modular and it is possible to reuse components for several workflows. However, as it there is no specification of how to parametrize the components for configuring them at runtime, it might be necessary to have an own component for each configuration. Additionally, due to missing standards regarding the data exchange developers might tend to develop components having a low cohesion. This leads to less work on the developer's behalf because less components imply that less managers have to be developed. But the other side of the coin is that such a component is less likely to be reused than a component dedicated at one specific task and thus, having a high cohesion. Another consequence is that there is always extra work if an external component shall be integrated because the manager has to be written by the

integrator, who designs the data structure of the whiteboard, and not by the component's developer. For these reasons the architecture only partially fulfills this requirement.

The system's text mining engine should incorporate a common type system for the created annotations

The whiteboard architecture does not specify any kind of data structure within its whiteboard. Instead, developers have to encapsulate their components into managers that convert the component's internal representation of the data into the external representation used within the whiteboard and vice versa. Therefore, this requirement has not been addressed by the architecture.

The system's text mining engine should comply with a standardized data format for data exchange between its components

Data exchange between components is only possible through their managers communicating with the whiteboard holding the information. External components that may read and write XML structures, for instance, cannot be directly embedded but need a manager as well. Since managers have to translate between the individual data structure within the whiteboard and the component's internal one, the whiteboard architecture cannot be said of meeting this requirement.

It should be easy to integrate and to interchange foreign components

Again the concept of encapsulating the components in managers hampers the satisfaction of this requirement. As explicated above, the required extra work for writing managers for all components that shall be integrated does not make it easy to integrate new or interchange existing components. On the other side, this approach allows for the integration of components written in different languages. Hence, even though it remains unclear how the conversion of the data structures can be achieved by the managers, this requirement is considered as partially met.

The system's text mining engine should support parallel processing of NLP tasks

The whiteboard architecture allows for parallel execution of components, which may also be executed on different machines. Moreover, it is possible to use partial results already computed by other components. Thus, the architecture fulfills this requirement.

5.5 TALISMAN

5.5.1 Description

TALISMAN is another distributed approach whose main goal is to achieve a higher accuracy than architectures in which components are executed sequentially. For this, TALISMAN incorporates a multi-agent system (MAS) approach as described by (Guessoum & Briot, 1999) and formerly introduced by (Genesereth & Ketchpel, 1994). Each component in TALISMAN is an agent according to the MAS approach. To improve accuracy of the computed results the agents communicate directly with each other. Therefore, there is no representation of the computed results that is globally coherent for all components. Instead, every component has its own local representation that might differ from the representation within other components. The direct communications between the components allows for reuse of partial results computed by other components. It is possible that different components compute different results for the same input. For this case, the resulting conflicts can be resolved via *laws* which define rules that agents have to obey (Pallotta & Ballim, 2001; Stefanini & Demazeau, 1995; Stefanini & Warren, 1996).

Representation of annotations

There is no specification about how the components' results are represented. There is no common data structure but rather an individual representation within each component. Consequently, there is no standardized type system, either.

Modularity and interchangeability

The architecture is modular by having different agents for different tasks. Interchangeability is not given with this architecture as agents directly communicate with each other. The fact that there is no common type system hampers exchanging components further. This approach leads to a tight coupling of components by its very nature as components have to be concerted (Leidner, 2003).

Workflow management

In TALISMAN there is not *the* predefined workflow of components. They rather run concurrently and communicate directly with each other. The communication and the behavior of components is governed by rules called *laws*.

Parallelization and distribution

Components are executed in parallel and reuse partial results from other components. For this, they communicate directly with each other. The knowledge of already computed results differs from component to component dependent on their communication behavior.

5.5.2 Assessment of requirements

The workbench should be a web application

By now, there is no literature reporting usage of TALISMAN for a web application. Issues like thread safety would have to be addressed by the developer. Therefore, each component would have to be thread safe so that it will not interfere with a parallel execution that might have been triggered by another user of the web application. As there seems to be no support by the architecture for this requirement, it is not fulfilled by TALISMAN.

The system's architecture should foster reuse of components

Due to missing standards regarding the data structure of the results exchanged by the components it is nearly impossible to reuse existing components. For different workflows[8] different components may be required. Therefore, in the worst case each component needs a dedicated interface for each other component within the workflow. Therefore, TALISMAN does not meet this requirement.

The system's text mining engine should incorporate a common type system for the created annotations

As there is no standard regarding the data structure of exchanged results, there is not any common type system as well.

The system's text mining engine should comply with a standardized data format for data exchange between its components

The direct communication implicates having several interfaces for each workflow. Compliance with a standardized data format could only be achieved if each component provides interfaces that deal with such a data format. However, as this would be the developer's task and as there is no specification about this, this requirement is not met.

[8] Workflow in this context does not need to be a sequential workflow but a high-level task that is split into low-level tasks performed by different components / agents.

It should be easy to integrate and to interchange foreign components

Due to the component's tight coupling introduced by the direct communication without specifying a standardized format for this it will be difficult to interchange components (Petasis et al., 2002). For a foreign component several interfaces would have to be developed before it could be integrated. If there would be a central repository holding all the already computed results in a specified data structure, this overhead would not be necessary. As a result, this requirement is not addressed by TALISMAN.

The system's text mining engine should support parallel processing of NLP tasks

Components run concurrently and reuse partial results from other components. They retrieve them via direct communication. Therefore, this requirement is fulfilled.

5.6 TalLab

5.6.1 Description

TalLab is another architecture incorporating the MAS approach aiming to "(…) ease the work of software engineers producing, deploying and monitoring the applications" (Wolinski et al., 1998, p. 1). Apart from the MAS approach the extensive reuse of the OS and refusing to specify a standard regarding the implementation of components are salient traits of TalLab. The latter does not only refer to the data structure of exchanged knowledge but also for programming languages and even the OS (Wolinski et al., 1998). However, it remains unclear how the architecture can make intensive reuse of the OS without imposing the OS that shall be used.

Representation of annotations

As no single standard for representing linguistic data is imposed, there is no common data structure holding the results computed by the various components. Instead, components communicate directly via their wrappers that transform the data between different data formats.

Modularity and interchangeability

Like TALISMAN the architecture is modular by having different agents for different tasks but as already expounded above, interchangeability is hampered due to missing standards regarding the communication between components.

Workflow management

Components are arranged in circuits directly communicating with each other. However, it remains unclear how a workflow can be built of these components. Moreover, it is not clear whether components can be configured and how this would be done.

Parallelization and distribution

Components can be executed in parallel and on different machines. Thus, parallelism is exploited to get a task completed in a shorter amount of time.

5.6.2 Assessment of requirements

The workbench should be a web application

Each instance of a component is represented by a dedicated agent. As long as these components avoid usage of static fields the application appears to be thread safe. Therefore, this requirement is considered to be met.

The system's architecture should foster reuse of components

Like in TALISMAN missing standards regarding the data structure of the results exchanged by the components implicate that it is not easy to reuse existing components. Hence, this requirement is not fulfilled.

The system's text mining engine should incorporate a common type system for the created annotations

As there is no standard regarding the data structure of exchanged results, there is not any common type system as well.

The system's text mining engine should comply with a standardized data format for data exchange between its components

TalLab does not impose any standards regarding the data format that is used for the communication between components. Thus, this requirement is not met.

It should be easy to integrate and to interchange foreign components

Due to missing standards components have to be wrapped in agents providing an adapter that converts the data structure of the component's input and output. (Wolinski et al., 1998) claim that it is easy to write such adapters and that the development of such an adapter for a search engine has taken only a few days. However, developers have to write such adapters for each

component. Finally, an effort of several days for an adapter does not stand for an easy interchange of components. Therefore, TalLab does not satisfy this requirement.

The system's text mining engine should support parallel processing of NLP tasks
Components run concurrently and reuse partial results from other components. They retrieve them via direct communication. Consequently, this requirement is fulfilled.

5.7 Heart of Gold

5.7.1 Description

Heart of Gold is an XML-based middleware architecture that is positioned between the application and external components. Application in this context means the part of the software containing the business logic and needing the results of external components for its processes (Schäfer, Ulrich, 2006, 2007, 2008). A sketch of the architecture is given in Figure 10.

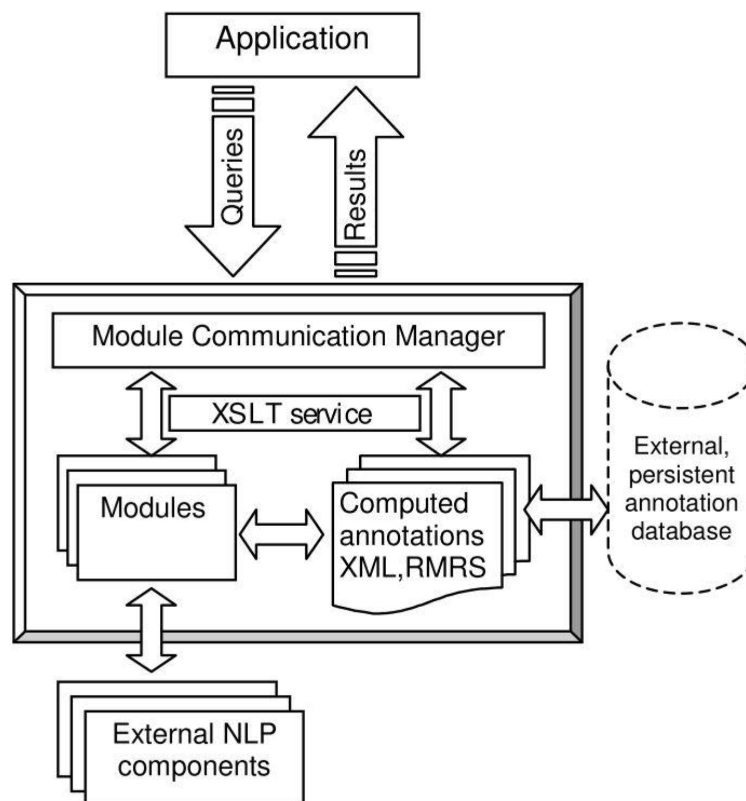


Figure 10: Illustration of the Heart of Gold architecture
Source: (Schäfer, Ulrich, 2006, p. 81)

Representation of annotations

Each component has to provide its computed annotations in a stand-off XML structure. The only requirement is that this XML structure is well formed, i.e. there is no specific document type definition (DTD) or schema specified by Heart of Gold. However, the already integrated components Heart of Gold comes with use an RMRS (Robust Minimal Recursion Semantics) format (Copestake, Flickinger, Pollard, & Sag, 2005). If a component relies on the results of a component using a different format, an adapter, called *module*, has to be implemented that uses extensible stylesheet language transformation (XSLT) to transform those results between the different XML formats. If a component shall be integrated but it does not provide its results in a stand-off XML representation, the corresponding module has also be to capable of generating this XML representation based upon the structure of the component's results.

Modularity and interchangeability

Components can be integrated by providing *modules* that act as interfaces to other components. *Modules* can be Java-based or called via XML-RPC (XML Remote Procedure Call). XML-RPC is a protocol that is language-independent. Thus, using XML-RPC allows for the integration of NLP components implemented in different programming languages. Configuration of components can be done with parameters.

Workflow management

Workflows can be defined using the System Description Language (SDL) that allows for sequential, parallel, and iterative processing (Krieger, 2003). Such an SDL description is compiled into a Java program that executes the specified workflow.

Parallelization and distribution

Components can be executed in parallel as this is supported by SDL. Due to the support of XML-RPC Heart of Gold can also integrate components distributed on different machines.

5.7.2 Assessment of requirements

The workbench should be a web application

The *Module Communication Manager* can manage several sessions. Each session represents an instance of the Heart of Gold architecture. Since for each user of the web application an own session could be created and components can be executed in parallel running in different threads, the prerequisites for this requirement are fulfilled.

The system's architecture should foster reuse of components

Using XML-RPC it is possible to integrate components written in different languages and maybe residing on different machines. It does not prescribe a specific type system but it already comes with some components that use an RMRS format. If a component does not emit its results in this RMRS format – or even not in XML –, it is necessary to implement an adapter that uses XSLT for transforming the format. Furthermore, components can be configured with parameters. However, this configuration is static, i.e. a component cannot be configured based on the results of previous components in the workflow. Consequently, this requirement is only partially fulfilled.

The system's text mining engine should incorporate a common type system for the created annotations

Heart of Gold does not restrict components to comply with a specific type system. It does, however, suggest an RMRS format. Therefore, this requirement is partially fulfilled.

The system's text mining engine should comply with a standardized data format for data exchange between its components

Each component has to produce its results in an XML structure. Components that do not meet this prerequisite need an adapter that converts the component's data format into an XML representation. Thus, this requirement is satisfied.

It should be easy to integrate and to interchange foreign components

Foreign components can be integrated with adapters. As for every component an extensible stylesheet language (XSL) file is needed for further transformation, this requirement is only partially fulfilled.

The system's text mining engine should support parallel processing of NLP tasks

Components can run in parallel in separate threads and also on different machines. After the execution of the workflow the results of these components are aggregated. Consequently, this requirement is met.

5.8 GATE

5.8.1 Description

GATE, a General Architecture for Text Engineering, is a widely used and mature TIPSTER-based architecture. Its development hast started over 15 years ago with the goal to provide an infrastructure for NLP components. Its current version is 8.1 and it comprises an IDE, many integrated compatible components, and a framework, which is called *GATE embedded* and allows the integration of these GATE components into a Java application utilizing them (Cunningham, 2000, 2002; Cunningham, Humphreys, Gaizauskas, & Wilks, 1997; Cunningham, Tablan, Roberts, & Bontcheva, 2013). For more details regarding GATE's IDE cf. subsection *2.1 GATE Developer*.

The architecture consists of three major elements: The GATE Document Manager (GDM), a Collection of REusable Objects for Language Engineering (CREOLE), and the GATE Graphical Interface (GGI), which will be neglected here. The data model of the GDM is based on the TIPSTER model containing a collection of documents consisting of text and annotations. The GDM is therefore the central information repository and the component's common interface for reading and writing linguistic information of the currently processed document. I.e. components do not communicate directly with each other, which reduces coupling between them. The CREOLE is the collection of NLP components, called *processing resources* in GATE.

For rule-based information extraction GATE provides an implementation of the CPSL. It is called Java Annotation Patterns Engine (JAPE) and it makes it possible to match patterns over annotations and their features in a way similar to regular expressions (Cunningham, Maynard, & Tablan, 2000). For the matches actions can be defined so that for each match the corresponding actions are performed.

Representation of Annotations

Annotations in GATE are stand-off. They have attributes specifying the start and end position of the text span they relate to, a type declaration, and further attributes representing linguistic information. Like in Ellogon, the type of the annotation represents only a textual description and is not appropriate for type validation.

Modularity and interchangeability

GATE is a modular architecture where every component, i.e. every processing resource, is plugged into the application. Components have to specify pre- and post-conditions, i.e. the expected input and the produced output. As long as a component has the same pre- and post-conditions as another one they can be easily exchanged. If the conditions differ, it depends on the other components within the workflow. I.e. if a component requires a different input than the other, it is still possible to exchange them if the previous components produce this different input anyway. If the output differs, it depends on the subsequent components of the workflow: If the change causes a specific annotation type to be dropped out, this change can still be done if this annotation type is not required as input by one of the following components within the corresponding workflow.

Workflow management

GATE workflows are linear. Processing resources can be created with factories and can be added to a controller which represents the workflow. The configuration of a processing resource is given in a configuration file, the creole.xml. Dynamic configuration via parameters is not possible in GATE.

Parallelization and distribution

GATE does not support parallel processing of workflows. However, if parallel execution is required, GATECloud.net[9] might be worth a deeper look (Tablan, Roberts, Cunningham, & Bontcheva, 2011, 2012). It is a platform that allows users to execute their workflows on the Amazon cloud.

5.8.2 Assessment of requirements

The workbench should be a web application

As already mentioned above, for web applications multithreading is crucial. GATE Embedded, the framework that is provided for integrating GATE into an own application, is thread safe in principal. However, according to (Cunningham et al., 2014) there are some restrictions:

[9] https://gatecloud.net/ (last access on 08.10.2015)

- The developer has to take care of initializing GATE exactly once,
- it is not possible to make calls that change the global state of GATE in several threads simultaneously, and
- processing resources are not thread safe by design.

While the first issue is easy to achieve, the other restrictions implicate some drawbacks. The second issue implies that loading and unloading of components cannot be done concurrently. The suggested solution is to load all components that may be needed later already at the startup phase of the application. Especially for large applications utilizing a lot of different components, however, this will lead to a long startup phase. Another solution might be to define one generic method that takes the component as a parameter and loads it. If this method is synchronized (e.g. with the keyword "synchronized" available in Java), it can be guaranteed that it is never executed by more than one thread at the same time. However, the drawback that loading or unloading components cannot be done in parallel remains. The third restriction can be avoided by the developer of the component. Particular attention has to be paid on the usage of temporary files or other resources like dictionaries within a component. As a developer of an application integrating several components one has either to be sure that these issues have been addressed by the component's developer or the component, and thus the whole workflow, can only be executed within a synchronized block. In the former case it is possible to create a dedicated instance of the processing resource for each thread. The latter, of course, subverts the idea of multithreading. For these reasons this requirement is only partially fulfilled.

The system's architecture should foster reuse of components

The plugin mechanism of GATE allows to combine components into workflows easily. Components can be configured via parameters and it is distinguished in static configuration parameters and in runtime parameters. Especially if all components are loaded at the application's startup phase, it is very important that the component's developer has taken care of this separation and marked each non-static configuration parameter as a runtime parameter. Otherwise, an instance of the already loaded component cannot be reused for different workflows if they need the component in different configurations. However, as there is no common type system, the reuse of components is still hampered. Therefore, GATE fulfills this requirement only partially.

The system's text mining engine should incorporate a common type system for the created annotations

GATE does not prescribe a specific type system. In combination with the fact that annotations are not strongly typed but have only textual descriptions of their type and may have arbitrary data structures it may even lead to unexpected results. For example, if a component produces annotations having a specific type and another expects annotations whose textual description of its type is the same but its data structure differs, it looks like these components were compatible even though they are not. This is especially problematic if the description of the type is ambiguous and different components produce completely different annotations that share the same textual descriptions of their type. This is illustrated in Figure 11. On the left side a case is meant to be a legal case having further information like the corresponding prosecutor. On the right side a case is represented as a bag that has a manufacturer and a price, for instance. If a component expects *Case* annotations as input, it may expect them to have a feature *prosecutor*. If another component produced *Case* annotations for each bag, these annotations do not have the feature *prosecutor* and the consuming component's behavior may cause surprises. Thus, this requirement is not fulfilled by GATE.

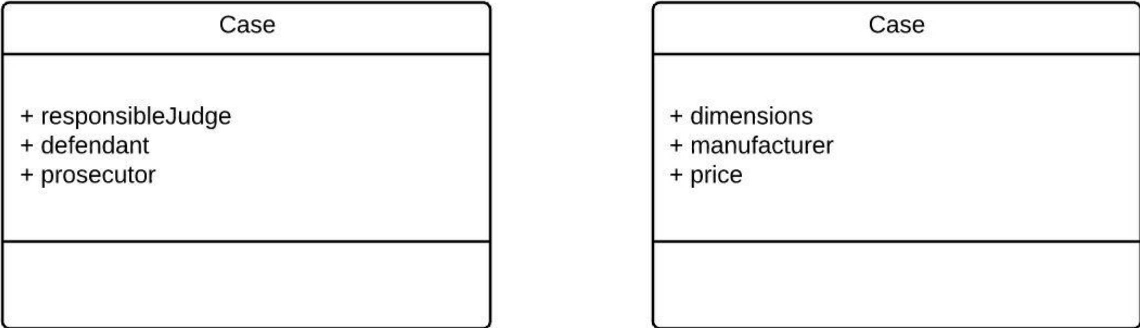


Figure 11: Representation of two conceptually different annotations having the same value for their type declaration
Source: Own illustration

The system's text mining engine should comply with a standardized data format for data exchange between its components

Components in GATE access the linguistic information about the documents they process via the GDM which holds the documents containing their respective annotations. Hence, this requirement is met.

It should be easy to integrate and to interchange foreign components

Foreign components have to be wrapped so that they can be integrated. The wrapper is responsible for the access to the GDM API. This way it is also possible to integrate components written in a different language like Prolog, for example (Cunningham, 2002). As there is no common type system in GATE, one has to make sure on one's own that the data structure of the expected input and the produced output is the same for two components if they shall be exchanged.

An interesting feature is the compatibility with Apache UIMA which is presented in the following subsection. An existing UIMA component can be wrapped into a GATE processing resource. For this, a mapping has to be defined that maps UIMA annotations to GATE annotations and vice versa. This is also possible the other way round: With a user defined mapping it is possible to embed a GATE component into a UIMA workflow (Cunningham et al., 2014). This compatibility allows for access to a huge set of components developed for one of the two most widely used NLP architectures. All in all, this requirement is fulfilled.

The system's text mining engine should support parallel processing of NLP tasks

GATE does not support parallel processing. Instead, there is GATECloud.net where users can upload their workflows or choose from a set of predefined ones and download the results afterwards. However, at the time of writing there is no programmatic access to this cloud service. Thus, it is not possible include such a cloud service into a web application. Consequently, this requirement is not satisfied.

5.9 UIMA

5.9.1 Description

The Unstructured Information Management Architecture (UIMA) has been developed by IBM with the main goal to provide means for reusing components. In 2006, IBM donated it to the Apache Foundation and in 2010 UIMA was promoted to a top-level Apache project. In 2009, the design of UIMA was published as an OASIS standard. It comes with several Eclipse plug-ins and an implementation of its framework in Java and in C++ (Bank & Schierle, 2012; Ferrucci et al., 2006; Ferrucci & Lally, 2004; Ferrucci, Lally, Verspoor, & Nyberg, 2009; Götz & Suhre, 2004).

The most central concept of UIMA is the Common Analysis Structure (CAS), which is the central information repository holding all the linguistic information about a document. It is shared between all components of a workflow, i.e. components do not communicate directly with each other but retrieve their expected input from and store their computed output in the CAS. For the Java framework an interface to the CAS has been developed. It is a JCas object providing means for efficient access and iteration methods for the CAS (Götz & Suhre, 2004; Schor, 2003).

For rule-based information extraction UIMA provides a Rule-based Text Annotation (Ruta) engine. Ruta provides an imperative rule language where a rule combines a pattern of annotations and further conditions with a set of actions that are executed if the pattern matches (Kluegl et al., 2015). For the development of Ruta components the UIMA Ruta Workbench is available (Kluegl, Toepfer, Beck, Fette, & Puppe, 2014). It provides editing support with syntax highlighting and further features like rule explanation, rule validation, and the automatic creation of the descriptors based on the script containing the rules. Further information about descriptors and UIMA components in general is provided in the subsection *Modularity and interchangeability*.

Representation of annotations

Annotations in UIMA are stand-off: The unstructured content of the document, also called Subject of Analysis (Sofa) in UIMA, is stored in a separate object than all the annotations. Annotations refer to the Sofa by having attributes denoting their start and end position in the Sofa. The CAS comprises the Sofa as well as the corresponding annotations and represents therefore the document with all its information.

Annotations are typed and UIMA requires the CAS to comply with a type system. Types are classes that define the structure of the annotation and can be inherited. A type system is the collection of these type definitions (cf. subsection 1.2.4 Type system). UIMA does not prescribe a specific type system but rather provides a very basic type system including the primitive types defined by Ecore (Steinberg, Budinsky, Merks, & Paternostro, 2008). The developer has to extend this basic type system with type definitions specific for the application.

Modularity and interchangeability

UIMA is a component-based architecture. Components, also called Analysis Engines (AE) in UIMA, have XML descriptors, which specify the expected input and the produced output. Furthermore, these descriptors define parameters, their default values and whether they are required. Hence, such a descriptor is tightly coupled with the component it describes and a change of the component typically implies having to change the descriptor as well. To avoid this, a library called Apache uimaFIT, formerly known as UUTUC, has been developed (Ogren & Bethard, 2009). It provides Java annotations for describing a component. Thus, with uimaFIT developers can describe a component directly in its Java class and therefore, do not have to specify the XML descriptor anymore.

Interchangeability of two components is given if they consume the same input types and produce the same output types. As annotations in UIMA are strongly typed, i.e. their type is defined in a class and not only as a textual representation, a developer can rely on these definitions and hence, does not have to fear conflicts of different annotations having the same name as it is the case with GATE.

Workflow management

Analysis Engines can be combined into an Aggregated Analysis Engine (AAE) which represents a workflow. These AE can be instantiated and configured at runtime using the parameters specified in their descriptors or via the Java annotations provided by uimaFIT. For the instantiation of AE factories are provided. In addition to that, developers can add a custom Flow Controller to an AAE. Such a Flow Controller allows the developer to further specify the execution order of an AE within the AAE. For example, the developer can specify that several AE shall be executed logically in parallel (Apache UIMA Development Community, 2015b).

Parallelization and distribution

AE can be executed logically in parallel. Physically, an AAE is running single-threaded, i.e. AE that shall be executed in parallel run in an arbitrary order. For real multithreading environments UIMA Asynchronous Scaleout (AS) should be used (Apache UIMA Development Community, 2014). UIMA AS provides capabilities supported in UIMA that allow for further scaling like, besides multithreading, exploiting a cluster of machines. Standard UIMA components can run in UIMA AS without any code modifications. Apart from the usage

of static fields, UIMA components do not have to be thread safe as UIMA AS creates multiple instances of the components guaranteeing that each component is accessed by only one thread.

5.9.2 Assessment of requirements

The workbench should be a web application

With UIMA AS, UIMA provides means for multithreading and therefore, meets the main prerequisite for this requirement. Hence, UIMA can be integrated in web applications.

The system's architecture should foster reuse of components

Components can be configured at runtime and hence, be reused for various situations if they differ in only minor details that can be mapped by such configuration parameters. Requiring the CAS to adhere to a specific type system UIMA further facilitates the reuse of components.

The system's text mining engine should incorporate a common type system for the created annotations

UIMA does not prescribe a specific type system. Instead, it provides a very basic type system which has to be extended with types required for the specific application. However, it requires a CAS to comply with a specific type system. Thus, even though the already provided type system is quite sparse, this requirement is fulfilled as a common type system has to be specified by the developer.

The system's text mining engine should comply with a standardized data format for data exchange between its components

Each component accesses the CAS to gather its expected input and to store its computed output. Furthermore, a CAS can be converted to and from the XML Metadata Interchange (XMI) format and the Ecore model specified by the Eclipse Modeling Framework (Steinberg et al., 2008). Thus, this requirement is met.

It should be easy to integrate and to interchange foreign components

Requiring a common type system makes it possible to see at first glance whether two components are compatible or can be replaced with each other. However, as UIMA prescribes only a very basic type system interoperability could still be increased. For example, for the types of annotations denoting the POS tag of a word a common tagset like the Penn Treebank could be required (Marcus et al., 1993). If a foreign component adhering to another type system

shall be integrated, the developer has to develop and interpose a component that maps the concepts between the different type systems. All in all, this requirement is satisfied.

The system's text mining engine should support parallel processing of NLP tasks

Components can run in parallel in separate threads and also on different machines. After their execution the results of these components are aggregated. Consequently, this requirement is met.

5.9.3 Existing type systems and component collections

UIMA requires a CAS object to adhere to a type system but does not prescribe any type system. As a consequence, several component collections and their corresponding type systems have emerged. The remainder of this subsection shortly examines the three major UIMA component collections and their corresponding type systems.

JCoRe

The JULIE Component Repository (JCoRe) is a collection of components for reading documents and basic NLP tasks like tokenization or POS tagging (Hahn et al., 2008). Additionally, it provides components for persisting a CAS. However, it is a relatively small collection and even though there is a component for each important basic NLP task, there is no choice of many components for this very task but rather only a very limited set to choose from. Hence, the selection of components that fit the needs for a specific task is limited and one might be forced to use a component that is not appropriated for this task in a specific scenario. JCoRe does not come as a single versioned package. Instead, each component is available as a PEAR[10] package on its own.

The type system that JCoRe components adhere to can be decomposed into five layers (Buyko & Hahn, 2008; Hahn, Buyko, Tomanek, Piao, McNaught et al., 2007; Hahn, Buyko, Tomanek, Piao, Tsuruoka et al., 2007):

1. Annotations regarding bibliographic information about the document fall into the *Document Meta layer*.

[10] The Processing Engine ARchive (PEAR) is the UIMA standard packaging format for UIMA components

2. The *Document Structure & Style layer* contains information about the document structure like paragraphs and sentences. It was especially designed for scientific publications about life sciences.

3. The *Morpho-Syntax layer* holds annotations produced by morpho-syntactic components like tokenizers and POS taggers.

4. The *Syntax layer* contains annotations produced by phrase chunkers and dependency parsers.

5. Named entities reside in the *Semantics layer*.

U-Compare

U-Compare is an integrated text mining system that features comparison of the analysis results computed by different components (Kano, McCrohon, Ananiadou, & Tsujii, 2009). Therefore, it comprises a large set of components and an own type system. For more information about the comparison of analysis results with U-Compare, please refer to subsection *2.3 U-Compare*. For the integration of external components that adhere to another type system U-Compare makes use of special components converting these type systems. The combination of such a type system converter with an external component into an AAE makes this component compatible (Kano et al., 2008).

U-Compare also introduces an own type system, which "(…) is intended to be a shared type system capable of mapping types originally defined as part of independent type systems" (Kano, McCrohon et al., 2009, p. 25). It can be decomposed into three major parts:

1. The *Syntactic Level* comprises annotations holding syntactic information like tokens, syntactic dependencies, and constituents.

2. The *Semantic Level* spans annotations like named entities and co-references. For named entities a more fine granular distinction of types is provided that is especially targeted at the bioinformatics domain.

3. The *Document Level* encompasses annotations regarding the document's structure like paragraphs or its title.

An overview providing more details of the U-Compare type system can be found in the appendix in Figure 23.

DKPro Core

DKPro Core is a broad-coverage collection of interoperable analysis components (Eckart de Castilho, 2014; Eckart de Castilho & Gurevych, 2014). It comprises a large set of components and also provides several components for a specific task like POS tagging so that it is possible to select from many POS taggers the one that is most appropriate for the developer's purpose. Examples of POS tagging components featured by the current version 1.7.0 of DKPro Core include the Stanford POS Tagger and the TreeTagger, for instance (Schmid, 1994, 1999; Toutanova, Klein, Manning, & Singer, 2003; Toutanova & Manning, 2000)[11]. Furthermore, lots of resources like models for these components have been integrated for different languages. This allows reusing a component for different scenarios by specifying the resource via a configuration parameter. In order to ease the exchange of components naming conventions for the components' parameters have been established. Thus, the parameter specifying the language, for example, is the same for all components. Consequently, developers do not have to learn a new parameter set when exchanging a component.

DKPro Core components are able to determine on their own which resources they need. This and the provision of default values for configuration parameters reduces configuration efforts to a minimum. Moreover, many DKPro Core components are not only able to determine the resources they need based on the processing data, but also to download them at runtime on their own. This is enabled by translating the values of the corresponding configuration parameters – given by the developer or determined automatically – into Maven coordinates that are resolved to the correct resource of the associated Maven repository. The primary focus of DKPro Core is to integrate third-party components like the Stanford CoreNLP components, for instance (Manning et al., 2014). However, there are also some self-developed components, especially for reading and writing different data formats.

Figure 12 depicts the architecture of the DKPro Core component collection. It relies on uimaFIT (cf. *Modularity and interchangeability*, p. 54), which allows the developer to conveniently combine components to aggregated analysis engines with only few lines of code. The API

[11] The complete list of components can be retrieved from https://dkpro.github.io/dkpro-core/releases/1.7.0/components.html (last access on 10.10.2015)

modules provide the type system and common functionality like the detection of which resources have to be loaded. The input / output modules are components that read and write external data formats. The analysis modules contain the components that perform the actual NLP task while the resource modules represent resources like models needed by analysis components.

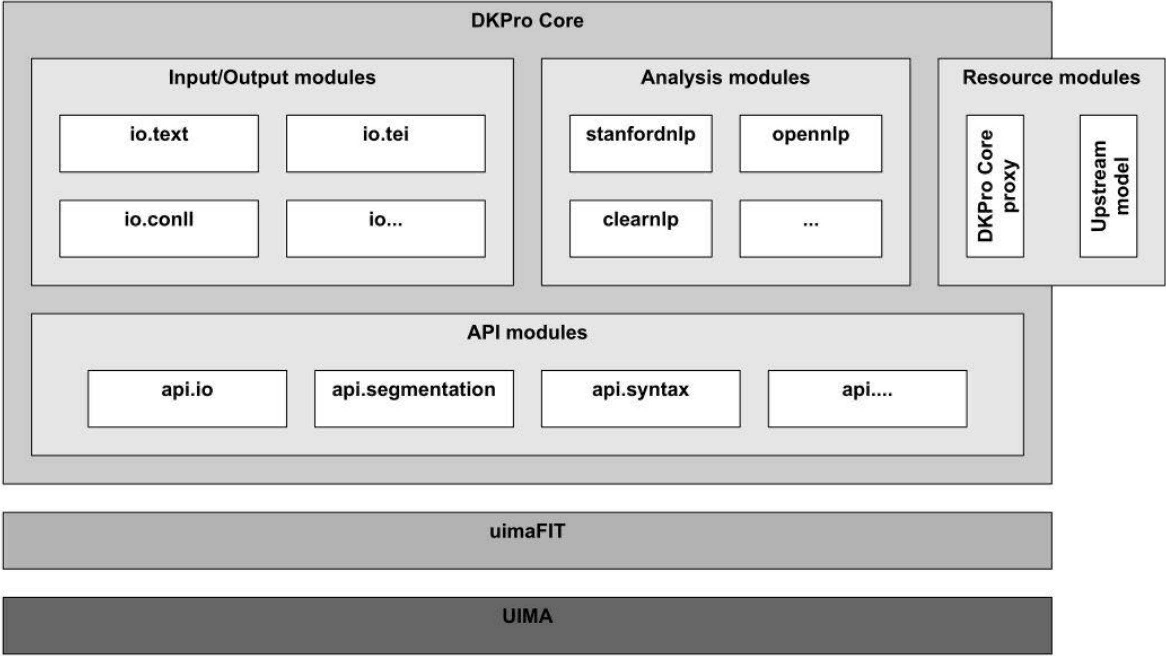


Figure 12: Architecture of the DKPro Core component collection
Source: (Eckart de Castilho, 2014, p. 134)

Figure 13 illustrates an overview of the type system provided by DKPro Core. Each annotation type has attributes representing its features as it is exemplified for the annotation type *DocumentMetaData*. Feature can be primitive features having primitive data types as it is the case for all the exemplified features of *DocumentMetaData*. But a feature may also have a more complex nature when being a feature structure itself.

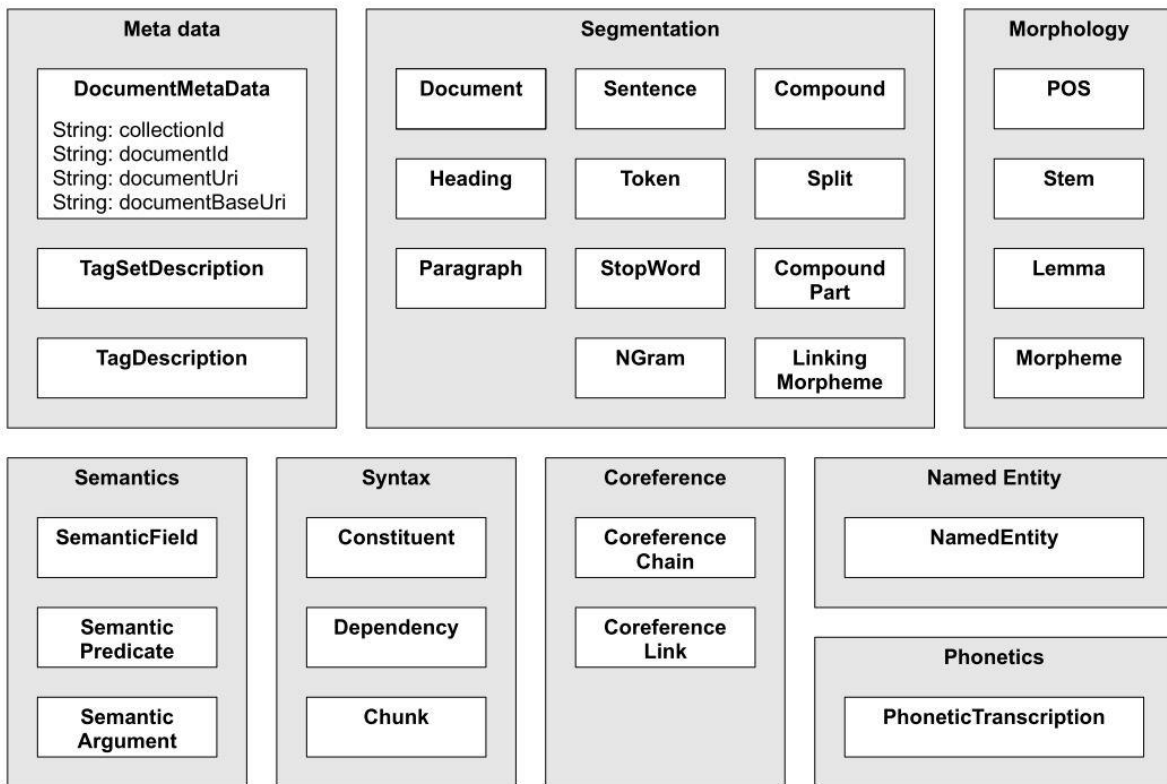


Figure 13: Overview of the DKPro Core type system
Source: (Eckart de Castilho, 2014, p. 135)

5.10 Summary and conclusion

Table 2 summarizes the findings explicated above and gives an overview of how well the presented architectures meet the requirements and whether a programming framework exists that implements the architecture. The scale is – *0* + whereas – indicates that the requirement is not fulfilled, *0* indicates that it is partially fulfilled, and + indicates that the requirement is fully satisfied. The consequence of this assessment is that Apache UIMA is suited best for a web based NLP application that is tailored to a specific domain, which is the legal domain in this case. Furthermore, for the preprocessing NLP tasks DKPro Core has been selected as it provides a large set of compatible components, a comprehensive type system, and further facilitations like the naming convention for parameters.

Requirement \ Architecture	TIPSTER	Ellogon	LIMA	Whiteboard Architecture	TALISMAN	TalLab	Heart of Gold	GATE	UIMA
Web application	-	-	-	-	-	+	+	0	+
Reuse of components	-	+	0	0	-	-	0	0	+
Type system	+	-	-	-	-	-	0	-	+
Common data format	+	+	+	-	-	-	+	+	+
Integration and interchangeability	0	+	-	0	-	-	0	+	+
Parallel processing	-	-	-	+	+	+	+	-	+
Framework	-	+	+	-	-	-	+	+	+

Table 2: Summary of the architectures and the requirements they fulfill

6 Implementation of the workbench

This chapter demonstrates how a prototypical implementation of such a web application that features interactive semantic text analysis for texts from the legal domain can look like. At first, it is elaborated on the type system (cf. subsection *1.2.4 Type system*) of the application and on how it relates to the type system of DKPro Core. In the following subsection 6.2, the architecture of the prototype is described. Then, it is presented how to integrate individual UIMA components into the application. Subsequently, it is shown how the web application allows for the development of rule-based components using UIMA Ruta. Following this, the whole process from a raw document to an annotated one is exemplified. Finally, it is critically reflected which of the requirements presented in chapter *4 Requirements* are already satisfied by the current prototypical implementation.

6.1 Type system and component collection

As already elucidated above, Apache UIMA requires a CAS to comply with a user-defined type system which all components processing this CAS have to adhere to. As components are incompatible if they incorporate different type systems, several component collections and type systems associated with them have emerged. In order to provide an architecture that allows for convenient interchangeability and to have access to a set of components for basic NLP tasks like POS tagging, for instance, it would not have been a good practice to reinvent the wheel and develop another component collection and a corresponding type system. Instead, it appeared to make sense to reuse and extend one of the existing type systems and utilize the existing components adhering to this type system (Griss, Martin, L., 1997; Standish, 1984). For this, DKPro Core (cf. subsection *5.9.3 Existing type systems and component collections*, p. 58ff.) has been selected. For types specific for the legal domain an own type system, which imports the one provided by DKPro Core, has been introduced.

6.2 Architecture of the web application

Figure 14 illustrates a high level description of the prototype's architecture. The application contains importers that are capable of importing legal documents of various sources and data formats. For the persistence SocioCortex, an internal system of the chair for Software Engineering for Business Information Systems[12] (sebis) is used. However, the separation of

[12] https://wwwmatthes.in.tum.de (last access on 07.10.2015)

business logic and persistence achieved with the data access layer allows the usage of other databases as well. It is planned to provide interfaces for exporting the information produced by the application to other sources but these have not been implemented or specified yet.

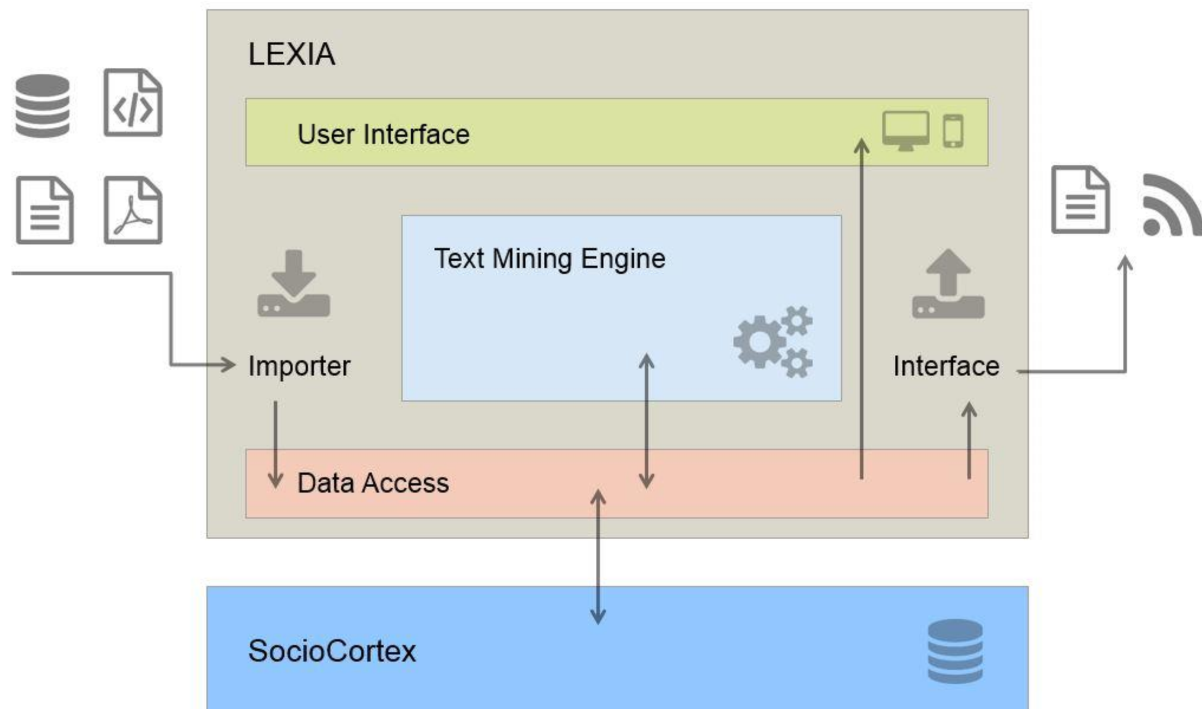


Figure 14: Overview of the application's architecture
Source: (Graß, 2015a)

The focus of this thesis is the block called "Text Mining Engine", which is disclosed in Figure 15. Based on the requirements and comparison above, Apache UIMA was chosen to be the architectural core of the text mining engine. On top of that there are on the one side the components and the type system provided by DKPro Core. In addition to that, the application also contains some self-developed components that rely on uimaFIT as it DKPro Core does. On the other side the application makes also use of self-developed Ruta components (cf. subsection *5.9.1 Description*, p. 53) for rule-based pattern matching over annotations. This part is twofold: It contains components that are provided by the application innately but the application also allows users to write their own Ruta scripts. The required XML descriptors will be generated automatically. For this, the functionality of the UIMA Ruta Workbench has been embedded (Kluegl et al., 2014). A more detailed description of how to develop own Ruta scripts with this web application is provided in the subsection *6.4 Developing rule-based components with UIMA Ruta*. On top of that, pipelines can be built combining these components and metrics like readability indices can be calculated.

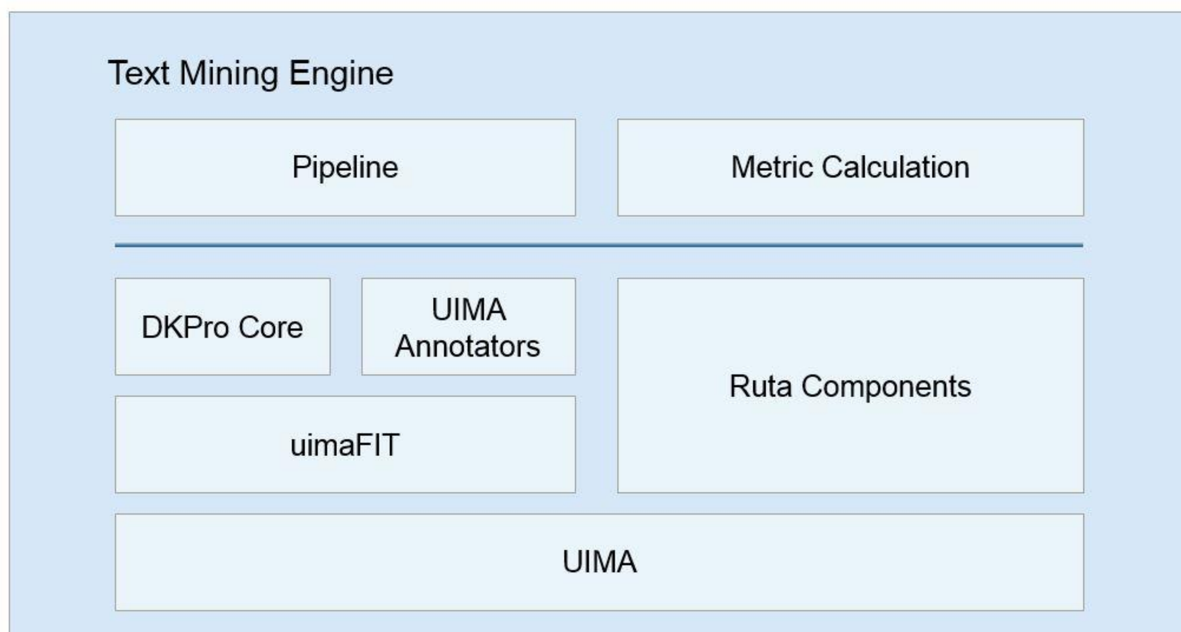


Figure 15: The architecture of the text mining engine
Source: Own illustration

6.3 Implementation of an own UIMA component

As mentioned in the previous subchapter, beside the components provided by DKPro Core also self-developed components utilize uimaFIT. One of the main advantages of uimaFIT is that the engine description of the components can be done with Java annotations directly in the class responsible for the execution of the component. Thus, with uimaFIT for this description no separate XML descriptors are required and therefore, maintainability of components is facilitated as a change affects less files.

Listing 3 demonstrates an example of such an annotator marking exceptions of legal norms. First of all, the Java annotation *@TypeCapability* provided by uimaFIT is used to describe the expected inputs and the computed outputs of the component. In this case the only expected input are *Sentence* annotations and the computed output are annotations having the type *LegalException*. The component defines a very simple pattern that matches sentences indicating exceptions of legal norms. For each sentence it is evaluated whether this sentence defines such an exception. If that is the case, a new annotation having the type *LegalException* that spans the whole sentence is created and added to the JCas object representing the document that is analyzed.

```java
@TypeCapability(
  inputs =
    {"de.tudarmstadt.ukp.dkpro.core.api.segmentation.type.Sentence"},
  outputs = {"informationExtraction.lexiaTypes.LegalException"}
)
public class ExceptionAnnotator extends JCasAnnotator_ImplBase {

  private String exceptionPattern = "(dies|das)\\s(gilt nicht, wenn).*";

  @Override
  public void process(JCas aJCas) throws AnalysisEngineProcessException {
    for (Sentence s : select(aJCas, Sentence.class)) {
      String sentenceText = s.getCoveredText();
      Matcher exceptionMatcher = Pattern.
        compile(exceptionPattern, Pattern.CASE_INSENSITIVE).
        matcher(sentenceText);
      if (exceptionMatcher.matches()) {
        LegalException e = new LegalException(aJCas, s.getBegin(),
            s.getEnd());
          e.addToIndexes(aJCas);
      }
    }
  }
}
```

Listing 3: An example annotator annotating exceptions of legal norms
Source: Own illustration

If such an analysis component introduces a new annotation type this very type has to be defined in the type system descriptor before. Listing 4 shows the current state of the type system introduced for types specific for the legal domain. It imports the type system provided by DKPro Core and describes own types. Currently, it contains only the aforementioned type *LegalException,* which is used by the component denoted in Listing 3. A description of a type includes the type's name, a short textual description, a super type and a description for its features. The type *LegalException* does not have any own features. The features indicating the start and end positions are inherited from its super type *Annotation* and thus, are not described in this subtype. *Annotation* is a very basic type only having these two features for the position. It is provided by the UIMA type system and represents the root within the annotation type hierarchy.

For each type two Java classes are required before this very type can be used. One class is the Java representation of the type. I.e. it provides constructors for instantiating this type and has member variables for its features. This class has the same name as the type. The other class provides the functionality for registering the type at the UIMA framework. Its name is the concatenation of the type's name and the suffix *_Type*. Luckily, a developer usually does not have to create these classes on his own. Instead, UIMA provides a code generator called

JCasGen for this task. Within the UIMA workbench, an IDE based on eclipse, this tool executed with a single click on a button. (Apache UIMA Development Community, 2015a, 2015b). When using another IDE, this method has to be explicitly called. UIMA provides batch and shell scripts for this but also allows for calling the code generator directly in Java. For generating the classes for the legal domain specific types the latter approach has been chosen in order to employ a common way of this generation that is independent of the developer's OS. For this, a simple main method calling the code generator with the correct parameters for the location of the type system descriptor and the output path for the generated Java classes has been implemented. Thus, the execution of this main method is everything a developer has to do when the type system descriptor has changed.

```
<?xml version="1.0" encoding="UTF-8"?>
<typeSystemDescription xmlns="http://uima.apache.org/resourceSpecifier">
<name>LEXIA Typesystem</name>
    <description/>
    <version>0.1</version>
    <vendor>LEXIA</vendor>
    <imports>
        <import location="GeneratedDKProCoreTypes.xml"/>
    </imports>
    <types>
        <typeDescription>
            <name>informationExtraction.lexiaTypes.LegalException</name>
            <description>Exception of a legal norm</description>
            <supertypeName>uima.tcas.Annotation</supertypeName>
            <features/>
        </typeDescription>
    </types>
</typeSystemDescription>
```

Listing 4: The current state of the type system
Source: Own illustration

6.4 Developing rule-based components with UIMA Ruta

The web application allows the user to develop own Ruta components. For this, the user is provided a simple TinyMCE[13] editor in which the Ruta script can be developed. After clicking the *Save* button the application will automatically create the corresponding XML descriptors describing the component and the type system for the types declared within the script. In contrast to UIMA components implemented in Java, it is not necessary to generate the Java classes for the corresponding types as long as they are not instantiated in the Java code. A

[13] http://www.tinymce.com/ last access on (07.10.2015)

screenshot of this feature is provided in Figure 16. For this feature the corresponding functionality of the UIMA Ruta Workbench has been adapted.

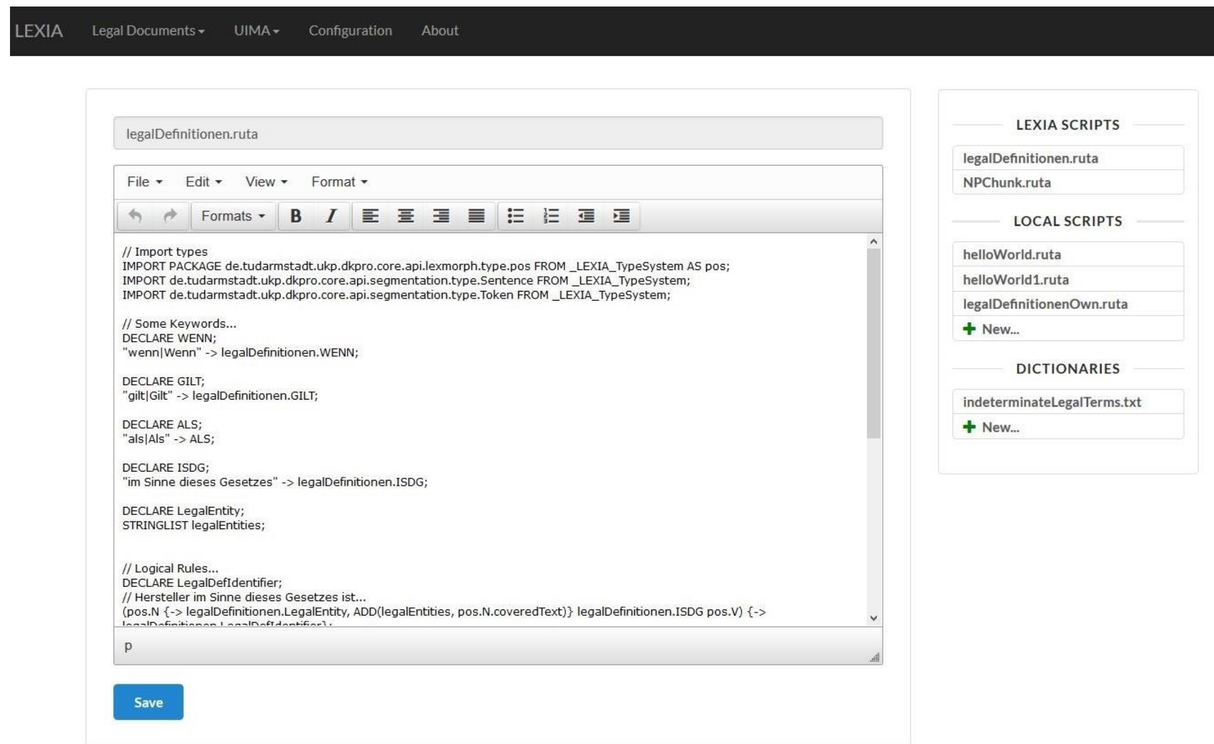


Figure 16: Screenshot of the view allowing users to develop their own Ruta components
Source: Own illustration

6.5. From raw documents to annotated documents

Before a document can be annotated it has to be imported into the application. As this was the focus of (Graß, 2015b), for the sake of brevity the details of the import process are neglected here. Basically, importing a document means to create a representation that is appropriate for persisting it in SocioCortex and for further processing. The latter especially includes to extract the text of the document which may be contained in a PDF or XML file, for instance. After importing a document it appears in a view that lists all imported documents and serves as a starting point for further actions related to a document. This is illustrated in Figure 17.

LEXIA Legal Documents UIMA Configuration About

Legal Documents

7 Legal Documents imported yet. You can import laws here.

Name	Promulgation Date	Creation Date	View	Process	Delete
Telekommunikationsgesetz	21.05.2014	04.10.2015 20:11:31			
Arbeitszeitgesetz	31.07.2013	04.10.2015 20:11:31			
Allgemeines Gleichbehandlungsgesetz	11.04.2013	04.10.2015 20:11:31			
Bundesdatenschutzgesetz	05.04.2013	04.10.2015 20:11:31			
Gesetz über die Haftung für fehlerhafte Produkte	05.04.2013	10.10.2015 00:07:35			
Telemediengesetz	05.04.2013	04.10.2015 20:11:31			
Aktiengesetz	01.01.1989	10.10.2015 00:00:09			
Bulk Operation					

Figure 17: Screenshot showing the list of imported documents
Source: Own illustration

For each document the options *View*, *Process*, and *Delete* are available. For annotating a document the user would click on the *Process* icon that directs him to the view presented in Figure 18.

In the process view, the user can choose the Ruta components that shall be executed via dragging a script from the left container holding the available scripts and dropping it in the right container representing the selected scripts. The available Ruta scripts include scripts provided innately by the application as well as scripts developed by the user. The current implementation is not capable of automatically determining dependencies between Ruta scripts yet. Therefore, if one script consumes annotations produced by another script, the user has to add the scripts producing these annotations in the correct order. The order of the selected scripts can also be changed after the scripts have been selected. For this, the same drag'n'drop mechanism as for the selection is used. The application will apply the order as it is within the container for the selected scripts. In addition to that, the user can choose a pipeline of UIMA components from a dropdown menu.

LEXIA Legal Documents▾ UIMA▾ Configuration About

Gesetz über die Haftung für fehlerhafte Produkte

ANNOTATE

AVAILABLE RUTA SCRIPTS

legalDefinitionen.ruta

helloWorld.ruta

NPChunk.ruta

SELECTED RUTA SCRIPTS

Default Pipe

Submit

Figure 18: Screenshot denoting the selection of the analysis components
Source: Own illustration

After submitting the selection a workflow is created. Within this workflow firstly, the components as they are defined in the selected pipeline and subsequently, the Ruta scripts in the provided order are executed. Listing 5 demonstrates an example of such a workflow.

An imported law is represented as a collection of article containers which, in turn, contain articles (Graß, 2015b). Therefore, this pipeline has to be executed for each article of a law. At first, the method checks whether the content to be annotated is already plaintext and converts it into plaintext if this is not the case. Afterwards, the AE descriptions for the components are generated and aggregated into an AAE. Then, the JCas object that acts as access object to the CAS within a Java environment is created and the pipeline is executed. The execution is done sequentially in the order in which the component descriptions are arranged within the AAE. An illustration of the pipeline model for the components of the workflow exemplified in Listing 5 is given in Figure 19.

```java
public static Map<String, Object> defaultPipeline (String rawContent,
    String[] rutaScripts, JSONObject annotationStructures) {
  Map<String, Object> result = new HashMap<String, Object>();
  boolean isRawContentInHtml = HtmlUtil.isHtml(rawContent);
  String plainText = isRawContentInHtml ?
      HtmlUtil.convertToPlaintext(rawContent) : rawContent;

  try {
    AnalysisEngineDescription tokenizerDesc =
        createEngineDescription(LegalSentenceSegmenter.class);

    AnalysisEngineDescription taggerDesc =
        createEngineDescription(OpenNlpPosTagger.class);

    AnalysisEngineDescription exceptionDesc =
        createEngineDescription(ExceptionAnnotator.class);

    // aggregate AE for basic pipe
    AnalysisEngineDescription basicPipeDesc =
        createEngineDescription(tokenizerDesc, taggerDesc, exceptionDesc);

    // create AEs for Ruta components
    AnalysisEngineDescription[] rutaDescriptions =
        UimaUtil.createRutaDescriptions(rutaScripts);

    // aggregate AE for Ruta components
    AnalysisEngineDescription rutaPipeDesc =
        createEngineDescription(rutaDescriptions);

    // aggregate AE for final pipe
    AnalysisEngineDescription pipeDesc =
        createEngineDescription(basicPipeDesc, rutaPipeDesc);
    AnalysisEngine pipe = createEngine(pipeDesc);

    JCas jCas = UimaUtil.produceJCas(rutaScripts);
    jCas.setDocumentText(plainText);
    jCas.setDocumentLanguage("de");

    try {
      pipe.process(jCas);
      result = createStandoffXmlAndAnnotationStructure(jCas,
          isRawContentInHtml, rawContent, annotationStructures);
    } catch (AnalysisEngineProcessException aepe) {
      aepe.printStackTrace();
    }

    // clean up
    jCas.reset();
    pipe.destroy();
  } catch (Exception e) {
    e.printStackTrace();
  }

  return result;
}
```

Listing 5: Example of a basic pipeline including components for tokenizing and sentence splitting, POS tagging, legal exceptions and the Ruta scripts selected by the user
Source: Own illustration

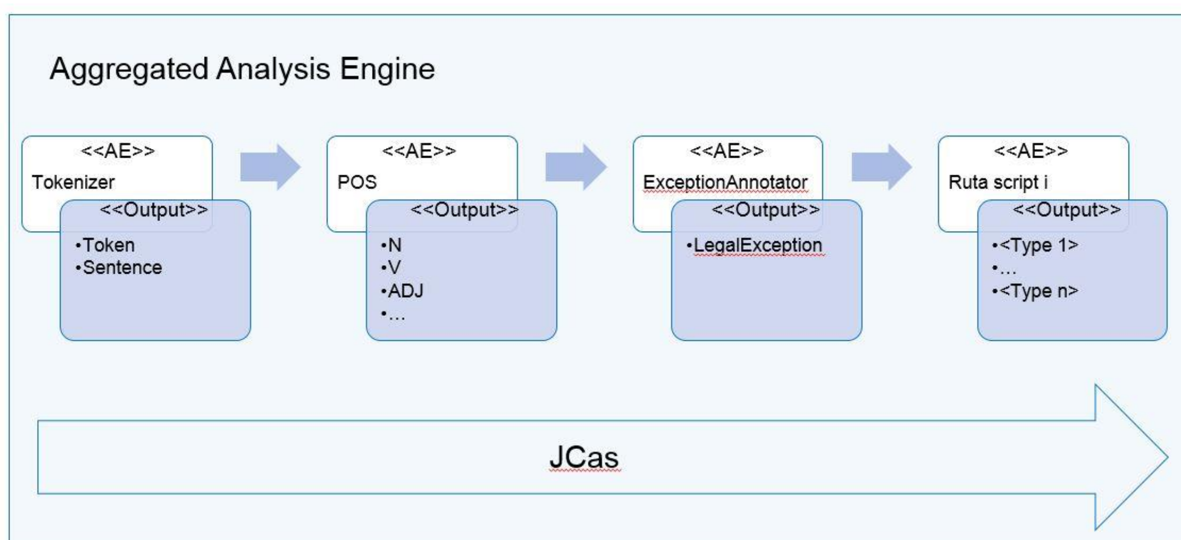


Figure 19: Illustration of the pipeline model. Each component fills the JCas object with its annotations.
Source: Own illustration

After the execution of the pipeline the method *createStandoffXmlAndAnnotationStructure* is called where the JCas object containing the results of all the executed components is used for further processing. This method creates an XML structure as it is shown in Listing 2 representing all annotations contained in the JCas object. This XML structure will later be used to produce a view of the document which highlights the annotations within the text (see Figure 21). Furthermore, this method creates a structure containing a representation of all annotation types that occurred in the document. For this, the JavaScript Object Notation (JSON) is used. This JSON representation is aggregated for the whole law which is the reason for the object holding this structure being passed as a parameter to the pipeline shown in Listing 5. An excerpt of an exemplary annotation type structure is shown in Listing 6.

This JSON structure is then used for providing the user a selection of annotations and their features that shall be available in the annotated view of the document. A screenshot disclosing this selection can be found in Figure 20. This selection is persisted and thus, will still be available when the user comes back later in order to change the selection.

```
{
    de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.NN: {
        isSelected: true,
        package: "de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos",
        shortName: "NN",
        features: {
            begin: 11,
            end: 17,
            posValue: "NN"
        }
    },
    de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.V: {
        isSelected: true,
        package: "de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos",
        shortName: "V",
        features: {
            begin: 15,
            end: 22,
            posValue: "VAFIN"
        }
    },
    ...
}
```

Listing 6: Excerpt of an exemplary annotation type structure created during the annotation process
Source: Own illustration

Figure 20: Users can select which annotation types are available when viewing the annotated document
Source: Own illustration

Now, the user can view the document and mark the annotation associated with it. A screenshot of this view is presented in Figure 21. On the left hand side the annotation types selected before are provided and users can check and uncheck them. When an annotation type is checked, all spans in the document that are associated with an annotation of this type are highlighted with a distinct color. Moreover, on the right hand side for all annotations the text they span is displayed. When hovering such an annotation, the features that have been selected for this annotation in the previous view are presented. In the example demonstrated in Figure 21 the only feature selected for the type of the hovered annotation is the POS tag. In addition to that, at the upper right corner linguistic metrics like various readability indices for the document are provided.

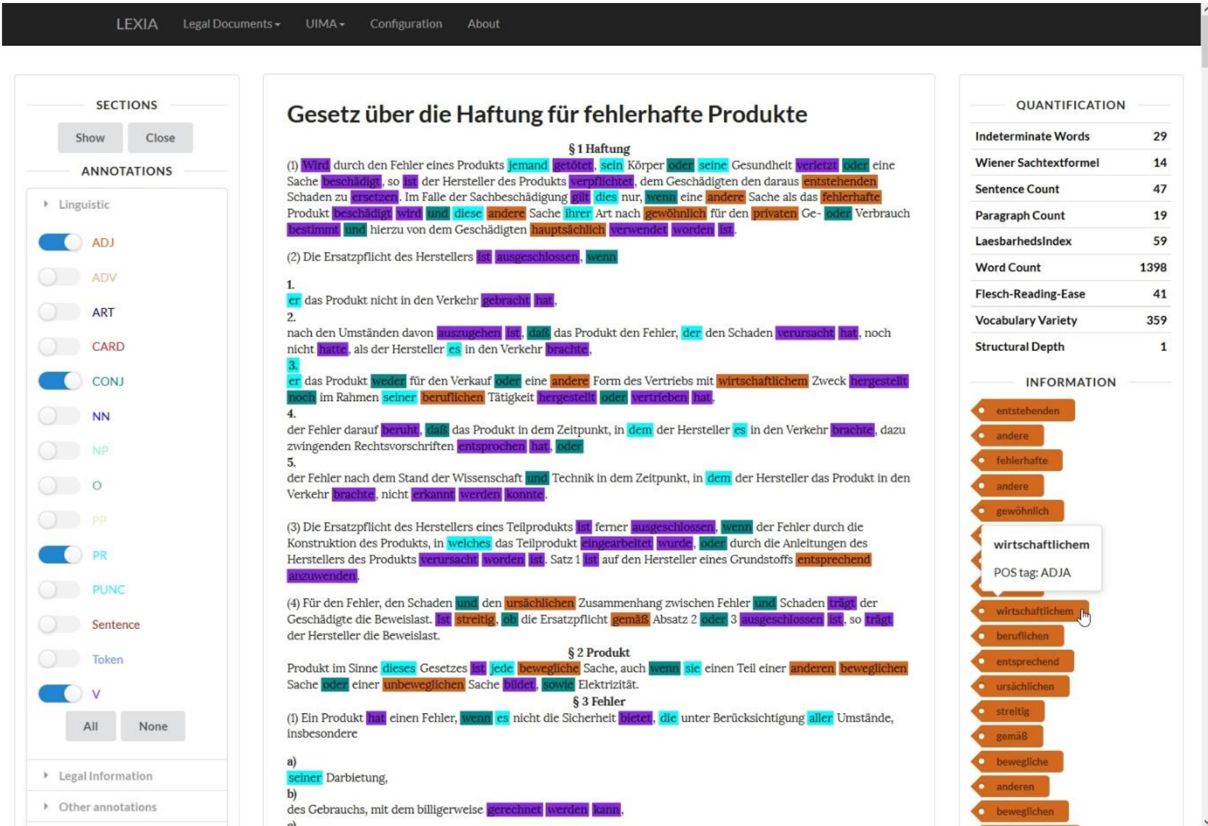


Figure 21: Screenshot of the view presenting the annotated document
Source: Own illustration

One particular challenge for highlighting the text spans that correspond to a specific annotation has been the mapping of the annotation's positions. As the analysis components perform their work on plaintext, this plaintext has to be extracted from the document's raw presentation that might be described with the HyperText Markup Language (HTML), for example. In order to preserve the formatting of the document's content its raw representation is used for the

presentation. As a consequence, the computed positions that refer to the plaintext representation used by the components have to be transformed into the correct positions within the corresponding raw presentation. Listing 7 shows the method performing this conversion. At first, line separators are subtracted from the given plaintext position as they are not included in the HTML representation. Afterwards, the main idea is to virtually restore the HTML representation of the plaintext. For this, the method iterates over the characters of the HTML representation until the correct position is reached. For each HTML tag the length of the tag inclusive its attributes is added to HTML position as well as to the plaintext position. Special characters encoded with a reference to their position in the ASCII[14] character set have also to be considered. While the HTML position is increased by the length of the corresponding HTML code, for the plaintext position this length has to be decreased by one again because there is already one character representing this special character. For plaintext characters only the HTML position has to be increased by one. When both positions are the same and either the current character is a plaintext character or the end of the HTML representation is reached, the correct position has been found.

[14] American Standard Code for Information Interchange

```java
public static int convertPlainPosToHtmlPos(String html, String plainText,
    int plainPos) {
  int htmlPos = 0;
  int tagLength, specialCharLength;

  // Eliminate line separators
  int toSubtract = 0;
  int i = 0;
  String subString = plainText.substring(0, plainPos);
  while (i > -1) {
    i = subString.indexOf("\r\n");
    if (i > -1) {
      toSubtract += 2;
      subString = subString.substring(i + 2);
    }
  }

  plainPos -= toSubtract;
  while (htmlPos <= plainPos) {
    if (htmlPos == plainPos && htmlPos == html.length())
      // We are at the correct position which is the end of the given html
      return htmlPos;

    if (html.charAt(htmlPos) == '<') {
      // We are at the beginning of an HTML tag
      tagLength = html.indexOf(">", htmlPos + 1) - htmlPos + 1;
      htmlPos += tagLength;
      plainPos += tagLength;
    } else if (html.charAt(htmlPos) == '&' && html.substring(htmlPos + 1,
        htmlPos + 7).contains(";")) {
      // This is most probably a special character encoded in its HTML code
      specialCharLength = html.indexOf(";", htmlPos + 1) + 1 - htmlPos;
      htmlPos += specialCharLength;
      // One char is consumed for the special char
      plainPos += specialCharLength - 1;
    } else {
      // We are in plain text
      if (htmlPos == plainPos)
        // we have reached the correct position within the given HTML
        return htmlPos;
      else htmlPos++;
    }
  }

  return -1;
}
```

Listing 7: Conversion from plaintext position to HTML position
Source: Own illustration

6.6 Reflection of fulfillment of the requirements

In this subsection it is reflected which of the requirements presented in chapter *4 Requirements* are already fulfilled by the actual prototype. Afterwards, Table 3 gives a summary.

6.6.1 Functional requirements

The workbench should support adding, removing, and editing of annotations

The current implementation primarily supports the automatic creation of annotations based on the pipeline selected by the user. Adding, editing, or deleting an annotation is currently only possible by editing the XML representation of the annotations directly. While this has been convenient for debugging, for the end user, however, this is not a user-friendly way for working with annotations. Therefore, this requirement is only partially fulfilled and an intuitive UI for this task remains for future work.

The workbench should support the persistence of annotations

Annotations are persisted in the internal sebis system, SocioCortex. The user can select which annotation types and which features shall be available when viewing the annotated document. This selection is done per document and persisted as well. When viewing the annotated document the user can – based on the selection of available annotation types – select which annotation types shall be highlighted. Due to separating the business code from the persistence layer it is also possible to use another database. Consequently, this requirement is considered as met.

It shall be possible to fold sections of the displayed text

Legal documents are internally represented as collections of containers which may contain parts of the content and further containers recursively. In the document view it is possible to collapse and to expand such subsections of the document with a single click. Thus, this requirement is fulfilled.

It shall be possible to leave own comments in the documents

A feature enabling users to leave comments associated with a specific part of the document is currently under development but has not been finished yet. Hence, the current version does not meet this requirement.

It shall be possible to set bookmarks in the documents

This requirement has not been pursued for the first version of the prototype and, as a consequence, is not fulfilled.

It shall be possible to edit the texts

For the current version of the prototype this requirement has not been focused on but left unresolved for future work.

It shall be possible that multiple users work on the same document and track their changes

As the previous requirement has not been addressed yet but is a prerequisite for this requirement, this feature is not supported by the current implementation.

The workbench shall feature a comparison of documents and their different versions

At the time of writing, the prototype does not support this feature.

The workbench shall be able to import documents with different formats

Currently, it is possible to import laws represented as XML or PDF files. The importers are designed in such a generic way that allows to easily include further importers for different file formats or document types and have been developed by (Graß, 2015b). Therefore, this requirement is already partially fulfilled.

The workbench shall allow for exporting documents in different formats

It is planned to provide interfaces for exporting the processed documents and their computed information to other file formats. However, this part has not been implemented yet.

The workbench shall provide information about incoming and outgoing references

UIMA annotators for this are already under development but not finished yet. Hence, the current implementation does not provide this feature.

The workbench shall annotate legal definitions

For annotating legal definitions and subjects defined in such a legal definition a Ruta component has been developed that matches typical patterns indicating a legal definition. Thus, this requirement is satisfied.

The workbench shall annotate exceptions of legal norms

For this, the UIMA component *ExceptionAnnotator* has been developed (cf. Listing 3). It needs *Sentence* annotations as input and checks for each sentence whether this very sentence specifies an exception of a norm using patterns. Each sentence matching one of these patterns is annotated as *LegalException*. Therefore, this requirement is met.

The workbench shall annotate legal consequences

This feature is not supported by the current implementation. However, the application is structured in such a sustainable way that it is easy to implement an analysis component for this.

The workbench shall provide linguistic information

When viewing a document an area providing linguistic information is placed at the upper right corner. Currently, the user is given a set of various readability indices and some further information like the amount of indeterminate words, for instance. Furthermore, a powerful component has been integrated that annotates spelling and grammatical mistakes. This component will especially be useful as soon as editable texts are supported by the application. Consequently, this requirement is met.

6.6.2 Nonfunctional requirements

The workbench should be a web application

The software is a web application and therefore useable without any installation efforts.

The system's architecture should foster reuse of components

With the selection of the Apache UIMA as fundament for the application the first step for this requirement has been done. UIMA incorporates a pipeline model where pipelines can be built by the combination of individual components. A consequence of this approach is that components can be reused in several pipelines. The second step is the selection of a comprehensive component collection for basic NLP tasks that provide results that are required for further processing. DKPro Core provides a large set of compatible components and has been selected for this purpose. These components can be configured at runtime and therefore, can be used for different pipelines addressing different scenarios. The third step is to ensure that self-developed components are reusable as well. Therefore, these components should have a high cohesion and should not depend on the rest of the application. Finally, it can be said that the architecture is very modular and fosters the reuse of components.

The system's text mining engine should incorporate a common type system for the created annotations

When selecting a component collection for the basic NLP tasks one important criterion was that these components incorporate a common type system. DKPro Core provides such a type system (see Figure 13). Additionally, it is paid attention on using a consistent type system for own types. Consequently, this requirement is considered as fulfilled.

The system's text mining engine should comply with a standardized data format for data exchange between its components

All components of the web application, regardless whether they are provided by DKPro Core, whether they are self-developed, or whether they are developed by the user using Apache Ruta, retrieve their expected input from the CAS and also put there their analysis results. Thus, this requirement is fulfilled.

It should be easy to integrate and to interchange foreign components

The component collection provided by DKPro Core provides a variety of components for each task. As these components are compatible with each other, one can easily exchange a POS tagger with another one, for instance. This is further facilitated by the fact that all components employ a common naming convention regarding their parameters. Thus, when exchanging a component it is not required to learn a complete new set of parameters as they will have the same name if they are responsible for the same things. Self-developed components can also easily be integrated and interchanged. It is important to pay attention on adopting the naming convention for self-developed components' parameters in order to keep this advantage. The application's architecture clearly states where to put a new component. If it is an external one, it can be included with an entry in the application's *build.sbt*[15] file. For these reasons this requirement is satisfied.

The system's text mining engine should support parallel processing of NLP tasks

It possible to execute pipelines in parallel without any side effects. Important for this to be preserved in the future is to avoid static fields in analysis components. This is necessary to keep the execution of these components thread safe. As long as all components comply with this convention this requirement is fulfilled. For the current version that is the case.

[15] The application uses the Scala Build Tool where dependencies are listed in the build.sbt file

6.6.3 Summary

Functional / Nonfunctional	Requirement	Fulfilled?
Functional	The workbench should support adding, removing, and editing of annotations	partially
Functional	The workbench should support the persistence of annotations	yes
Functional	It shall be possible to fold sections of the displayed text	yes
Functional	It shall be possible to leave own comments in the documents	no
Functional	It shall be possible to set bookmarks in the documents	no
Functional	It shall be possible to edit the texts	no
Functional	It shall be possible that multiple users work on the same document and track their changes	no
Functional	The workbench shall feature a comparison of documents and their different versions	no
Functional	The workbench shall be able to import documents with different formats	partially
Functional	The workbench shall allow for exporting documents in different formats	no
Functional	The workbench shall provide information about incoming and outgoing references	no
Functional	The workbench shall annotate legal definitions	yes
Functional	The workbench shall annotate exceptions of legal norms	yes
Functional	The workbench shall annotate legal consequences	no

Functional	The workbench shall provide linguistic information	yes
Nonfunctional	The workbench should be a web application	yes
Nonfunctional	The system's architecture should foster reuse of components	yes
Nonfunctional	The system's text mining engine should incorporate a common type system for the created annotations	yes
Nonfunctional	The system's text mining engine should comply with a standardized data format for data exchange between its components	yes
Nonfunctional	It should be easy to integrate and to interchange foreign components	yes
Nonfunctional	The system's text mining engine should support parallel processing of NLP tasks	yes

Table 3: Summary of the requirements and their implementation state

7 Conclusion and outlook

This thesis presented the design and prototypical implementation of a web application for interactive semantic text analysis in the legal domain. For this, an extensive requirements elicitation has been conducted. Architectural requirements have been gained from a literature review of related work and requirements specific for the legal domain have been elicited with expert interviews. It has been analyzed what common architectures for NLP applications exist and how well they fit the needs according to the requirements. Apache UIMA (cf. subsection *5.9 UIMA*) is a project defining a comprehensive architecture and coming with a mature framework that is widely used and forms the fundament of the presented web application. Chapter *6 Implementation of the workbench* explicates the architecture of the web application. Furthermore, it presents how to extend the application with individual components and demonstrates a typical workflow through the application accompanied with code snippets where applicable. Finally, a critical reflection regarding the requirements and how well they are already fulfilled by the current implementation is provided. The result is that all nonfunctional requirements, which have been the main focus of this work, are already met by the current implementation. Moreover, also some functional requirements are already fulfilled.

Future work includes the implementation of the other functional requirements. Especially, allowing the texts to be editable will be a challenge as these changes may imply that other annotations become invalid. Additionally, the positions denoting the begin and the end of an annotation within the text have to be updated. Another challenge will be the connection between different annotations maybe referring to different documents. For instance, an entity defined in a legal definition can already be detected if the annotation corresponding to the legal definition is in the same text. It appears, however, that a legal definition does not only apply for the law it is contained in but also for other laws. Therefore, these annotations should be connected. This is complicate as the term “Sache”, for example, is frequently used with various meanings within legal literature. I.e. the legal definition of this term in § 90 BGB does not imply that each usage in a legal documents refers to the term as it is defined there (BGB, § 90).

Appendix

Interviews

Heßler, Konrad – 03.07.2015

TW:
Guten Tag. Vielen Dank, dass du dir die Zeit für das Interview nimmst.

KH:
Gern.

TW:
Fangen wir gleich mit der ersten Frage an: Wie viel Zeit verbringst du ungefähr mit der Recherche von Rechtsliteratur (Gesetze, Urteile, Kommentare, etc.)?

KH:
Das kommt natürlich ganz stark auf die Tätigkeit an. Als Student verwendet man ja hauptsächlich Ausbildungsliteratur, die eigentlich schon so kompakt aufgearbeitet ist, dass man sich keine zusätzliche Literatur mehr anschaffen muss. Trotzdem muss man auch im Studium bestimmte Urteile analysieren. Bei anderen Tätigkeiten ist das natürlich anders. Bei mir am Lehrstuhl bildet der Hauptteil die Recherche in Datenbanken. Das macht schon einen sehr großen Teil aus.

TW:
Kann man das auch grob nach der Art der Literatur, also z.B. Gesetz oder Urteil, aufteilen? Gibt es eine Art, die in Bezug auf den Zeitaufwand heraussticht?

KH:
Bei Gesetzen hält sich der Aufwand in Grenzen. Dafür hat man ja normalerweise schon Gesetzbücher da, d.h. die muss man sich nicht extra aus der Bibliothek oder dem Internet beschaffen. Anders ist das aber bei etwas spezielleren Gesetzen oder z.B. bei europäischen Richtlinien. Die hat man natürlich nicht sofort parat. Aber auch diese Richtlinien findet man schnell im Internet, d.h. der Bereich „Gesetze“ ist bzgl. Zeitaufwand eher untergeordnet. Der Hauptanteil wird gebildet aus Aufsätzen aus wissenschaftlichen Zeitschriften, natürlich Kommentare und Monographien.

TW:
Wie würdest du denn die Relevanz der unterschiedlichen Literaturarten einschätzen?

KH:
Die verschiedenen Arten erfüllen unterschiedliche Funktionen. Kommentare eignen sich z.B. besonders für den Einstieg und beantworten eher Ja-/Nein-Fragen und Fragen zur Herangehensweise. Monographien liefern dagegen vor allem die Begründung und die Überlegungen dahinter. In der Praxis geht es vor allem darum, die Lösung für ein Problem zu finden; die Begründung der Lösung steht da eher im Hintergrund. In der Argumentation verweisen Anwälte dann eher auf die entsprechende Fundstelle im Kommentar. In der Wissenschaft dagegen sind vor allem die Begründungen und damit die Monographien relevant.

TW:
Gibt es außer den bereits genannten Arten von Rechtsliteratur noch weitere, die in diesem Zusammenhang relevant sind?

KH:
Ich denke, dass auch das sehr stark davon abhängt, welches konkrete Problem gelöst werden soll. Bevor Gesetze verabschiedet werden, werden zunächst immer einige Entwürfe erstellt. Zu diesen Entwürfen gibt es dann diverse Stellungnahmen, die dann auch zusammen mit den Entwürfen veröffentlicht werden. Wenn man nun z.B. die Entstehung eines Gesetzes nachvollziehen möchte, beschäftigt man sich vor allem mit dieser Literatur. Vergleichbar hierzu sind die Motive des BGB. Die Formulierung des BGB hat sich über ca. 20 Jahre erstreckt und gleichzeitig mit dessen Veröffentlichung wurden auch Bände herausgegeben, in denen die Überlegungen zu den einzelnen Formulierungen dargelegt werden. Als letztes fallen mir noch Vortragssammlungen ein, wobei das jetzt keine Art von Literatur ist, die speziell in Rechtswissenschaften vorkommt.

TW:
Insbesondere aus der Perspektive eines Anwalts: Welche Meta-Informationen (z.B. ein- und ausgehende Referenzen) wären wertvoll bei der Lektüre eines Paragraphen / Urteil / Kommentar?

KH:
Also ein- und ausgehende Referenzen wären auf jeden Fall sehr nützlich. Gerade wenn man

zeigen könnte, wo und wie eine Norm verwendet wird. In diesem Zusammenhang wäre es auch interessant, wo es möglicherweise Ausnahmen von der Norm gibt. Ein Problem, das ich bei diesen Meta-Informationen sehe, ist, dass diese zwangsläufig immer ein unvollständiges Bild zeichnen, d.h. einen Blick in den Kommentar werden diese Meta-Informationen sicher nicht ersetzen können. Diese Kommentare sind ja selbst schon nach Paragraphenebenen sortiert, d.h. ein Kommentar ist sozusagen ein Abdruck des Gesetzes angereichert mit Informationen. So gesehen ist ein Kommentar eine Extremform eines mit Meta-Informationen angereicherten Gesetzes.

TW:
Fändest du es hilfreich, wenn auch Hilfsnormen gekennzeichnet werden, sodass z.B. die Definition des Begriffs „Sache“ sofort verfügbar ist?

KH:
Ja, das halte ich für sinnvoll. Allerdings muss man hier aufpassen, da in der Rechtsliteratur Begriffe nicht alle einheitlich verwendet werden. D.h. das Wort „Sache“ kann in einem Paragraph etwas völlig anderes bedeuten als im Paragraph danach. Daher reicht das reine Erkennen des Worts „Sache“ noch nicht aus, um zu bestimmen, dass es um eine Sache geht, wie sie an einer bestimmten Stelle definiert ist. Ansonsten wäre so eine Einblendung der Definition z.B. per MouseOver extrem nützlich.

TW:
Welche Anwendungsfälle könntest du dir für ein solches System vorstellen?

KH:
Die reine Markierung von Substantiven oder Verben löst zunächst noch keine Probleme, das erkennt der Leser ja auch selbst. Die Identifikation von Legaldefinitionen würde ich aber für sehr nützlich halten. Außerdem wäre eine Markierung von Rechtsfolgen oder deren Fehlen sehr hilfreich. Im Strafgesetzbuch ist z.B. festgelegt, was die Folge eines Tatbestands ist. Es gibt aber auch Gesetze, in denen nur das Gesetz festgelegt ist, nicht aber die Folge des Nichteinhaltens. Solche Informationen findet man dann aber im jeweiligen Kommentar.

Was mir deshalb noch nicht ganz klar ist: Mit Meta-Informationen angereicherte Gesetze gibt es ja schon. Das ist ja gerade die Aufgabe des Kommentars. Geht es nun darum den Kommentar über die Software interaktiver zu gestalten?

TW:

Es geht durchaus in diese Richtung. Aber es soll noch darüber hinausgehen. Ein Beispiel wäre, dass nicht nur explizite Verweise, wie z.B. „§ 434 gilt entsprechend“, sondern auch implizite Verweise erkannt werden. So könnte z.B. erkannt werden, dass es in einem Paragraphen um eine Kapitalgesellschaft geht und dementsprechend Paragraphen angezeigt werden, die hier mit berücksichtigt werden.

KH:

Ich glaube nicht, dass das möglich ist. Man kann sicher einige wenige solcher Verweise erkennen aber generell sind die Formulierungen hierfür zu unstrukturiert und außerdem gibt es noch die vorher genannte Problematik, dass die Begriffe nicht einheitlich verwendet werden. Was ich mir aber vorstellen könnte, ist die Visualisierung dieser Informationen, wenn diese vorher manuell in das System eingepflegt wurden.

TW:

Aus der Sicht einer Person, die einen Text verfassen muss, z.B.: ein Anwalt, der eine Begründung formuliert: Wäre es hilfreich, den Text auf unbestimmte Rechtsbegriffe wie z.B. „angemessen“ oder „adäquat“, etc. zu prüfen und diese dem Nutzer anzuzeigen?

KH:

Ich glaube, es wäre sehr sinnvoll, diese Formulierungen zu kennzeichnen.

TW:

Wie hilfreich wären denn linguistische Informationen, wie z.B. die Gesamtlänge des Texts oder dessen Lesbarkeit für den Verfasser?

KH:

Im Alltag eines durchschnittlichen Rechtsanwenders wären diese Informationen eher weniger relevant. Im juristischen Verlagswesen, für Herausgeber von juristischen Datenbanken oder für den Gesetzgeber hingegen, könnte ich mir durchaus vorstellen, dass diese Informationen von größerer Bedeutung sind.

TW:

Ich würde dir jetzt gern eine Webanwendung zeigen, die das Ergebnis eines früheren Projekts ist. Diese Anwendung ist in der Lage, linguistische Informationen aus dem Text zu extrahieren und darzustellen. Stell dir vorher, dieses System wäre auf die Rechtsdomäne zugeschnitten und

würde nicht nur grammatische Einheiten wie z.B. Substantive markieren sondern auch die Referenzen oder Hilfsnormen, über die wir vorhin gesprochen haben. Hältst du einen solchen Ansatz für sinnvoll?

KH:
Grundsätzlich ja, insbesondere dort wo viel mit Definitionen gearbeitet wird. Das Problem an dieser Stelle ist aber, dass es oft mehrere Definitionen gibt. Während sich der Anwalt einfach auf das stützen kann, was sich in der Praxis durchgesetzt hat, müssen im wissenschaftlichen Bereich auch die anderen Definitionen berücksichtigt werden. Vielleicht ist das auch die Abgrenzung der Funktionalität eines solchen Systems.

Ansonsten gibt es in der Rechtswissenschaft noch einige weitere Gruppen von Begriffen deren Markierung wirklich hilfreich wäre. Über eine farbliche Markierung oder wie auch immer das dann aussieht könnte man sehr viele Informationen für den Leser transportieren. Da fallen mir auch wieder Studenten ein, die sich für die Examensvorbereitung ja auch die Gesetzestexte markieren und so versuchen, diese mit Informationen anzureichern.

TW:
Könntest du für diese Gruppen noch ein paar Beispiele nennen?

KH:
Ja, jetzt mal ganz ungeordnet und nicht abschließend, fallen mir da ein: Rechtsinstitute, Rechtsakteure, Gestaltungshandlungen und Willenserklärungen. Das Problem, das ich an dieser Stelle sehe, ist, dass auch hierfür wieder rechtswissenschaftliche Kenntnisse erforderlich sind, da eine Identifikation dieser Begriffe nicht anhand textueller Merkmale möglich ist. D.h. ich glaube nicht, dass man auf den Knopf drückt und dann alle Markierungen automatisch berechnet werden, sondern dass das auch eher auf die manuelle Eingabe dieser Information hinausläuft.

TW:
Wofür würdest du so ein System außerdem verwenden wollen?

KH:
Ich denke, im Bereich Lehre / Rechtsstudium kann das sehr sinnvoll sein. Außerdem wäre es hilfreich, wenn das System ein geeignetes Prüfungsschema, d.h. eine geeignete Reihenfolge

von Fragen, die zur Lösung eines Problems gestellt werden müssen, vorschlägt. Dies wäre nicht nur für Studenten sondern auch in der Praxis hilfreich und würde einige Vorteile mit sich bringen. Ansonsten wäre das System auch für wissenschaftliche Szenarien relevant.

TW:
Fallen dir noch weitere Funktionalitäten ein, die die Nützlichkeit eines solchen Systems steigern könnten?

KH:
Was ich mir noch vorstellen könnte, wäre die Gliederung von Text in Sinneinheiten. Teilweise muss man einen Paragraphen wirklich drei oder viermal lesen, bis man einen guten Zugang dazu hat, weil einfach die Sprache nicht immer ganz trivial ist. Außerdem kommt es häufiger vor, dass ein Satz definiert ist und danach eine Ausnahme, z.B. mit der Formulierung „Dies gilt nicht, wenn…“ folgt. Dies zu kennzeichnen wäre ebenfalls sehr hilfreich. Was mir außerdem noch einfällt, sind Formulierungen wie z.B. „Alle Menschen sind verpflichtet, eine bestimmte Abgabe zu zahlen“. Hier stellt sich die Frage, wer mit „alle Menschen“ gemeint ist. Beispielsweise könnte so eine Formulierung eigentlich alle deutschen Staatsbürger, die das 18. Lebensjahr vollendet haben, betreffen. Wenn solche Formulierungen entsprechend gekennzeichnet deren tatsächliche Bedeutung eingeblendet werden, wäre das ebenfalls unglaublich nützlich. Das ist auf jeden Fall etwas, was sich in der Praxis sehr viele Menschen wünschen werden. So etwas müsste aber definitiv manuell eingepflegt werden, da das natürlich kein System der Welt erkennen / wissen kann.

TW:
Nochmal zurück zur Anwendung, die wir uns vorher angesehen haben. Würdest du an der Art und Weise, wie diese Informationen dargestellt und hervorgehoben werden etwas ändern, um die Benutzerfreundlichkeit zu verbessern?

KH:
Verbessern könnte man noch die Darstellung in Hinblick auf fettgedruckte oder kursive Schrift. Was mir gerade noch einfällt sind Rechtsfolgeverweisungen und Rechtsgrundverweisungen. Letztere bedeutet, dass eine zu prüfende Norm nochmal komplett geprüft wird, während erstere nur auf die Rechtsfolgen verweist. Wenn man nun als Rechtsanwender auf einen solchen Verweis stößt, kommt die Frage auf, zu welcher dieser Gruppen dieser Verweis gehört. D.h. muss der Paragraph, auf den verwiesen wird, nochmal komplett geprüft werden oder müssen

nur dessen Rechtsfolgen berücksichtigt werden? Solche Sachen farblich zu markieren, wäre auch sehr hilfreich.

TW:
Die letzte Frage wäre noch, was ein solches System können müsste, um es für das Verfassen eigener Texte, z.B. als Anwalt, zu verwenden.

KH:
Also gibt ja auch schon bestimmte Word-Plugins für Anwälte und Richter, die entsprechend unterstützen. Deren hauptsächliche Funktionalität besteht in der Unterstützung bei der Formulierung von Textbausteinen. D.h. Anwälte haben viele wiederkehrende Formulierungen, die über solche Textbausteine abgebildet werden können. Ansonsten sind auch Formatierungshilfen, wie z.B. automatische Einrückungen oder Fettdrucke bei bestimmten Formulierungen, sehr wichtig.

Ansonsten, wenn ein Anwalt z.B. eine Klageschrift verfasst und den Standpunkt vertritt, dass sein Mandant einen bestimmten Anspruch hat. Dann muss dies anhand eines Prüfungsschemas begründet werden. Hilfreich wäre in diesem Zusammenhang, eine Checkliste oder eine Prüfung, ob das Prüfungsschema vollständig abgearbeitet wurde. Auch hier wäre ein Textbaustein bzw. eine Vorlage, die bereits alle für Prüfungsschema relevanten Fragestellungen in einer sinnvollen Reihenfolge enthält hilfreich. Der Anwalt würde dann nur noch die Argumentationen zu den einzelnen Punkten einfügen müssen und vermeidet das Risiko, einen relevanten Punkt zu übersehen oder zu vergessen. Außerdem könnte so sichergestellt werden, dass die Formulierung eine geeignete Struktur hat – einige Anwälte schreiben da etwas durcheinander.

TW:
Vielen Dank für das Interview.

KH:
Sehr gern.

Splinter, Christopher – 10.07.2015

TW:

Guten Tag. Vielen Dank, dass du dir die Zeit für das Interview nimmst. Fangen wir gleich mit der ersten Frage an: Wie viel Zeit verbringst du ungefähr mit der Recherche von Rechtsliteratur (Gesetze, Urteile, Kommentare, etc.)?

CS:

Das ist ganz unterschiedlich. Wenn ich ein Thema sehr detailliert bearbeite, verbringe ich sehr viel Zeit damit. Bei oberflächlicheren Themen verbringe ich eher weniger Zeit damit. Im Schnitt sind es aber schon ca. 1,5 Stunden pro Tag.

TW:

Welche Arten von Rechtsliteratur gibt es denn außer Gesetzen, Urteilen und Kommentaren?

CS:

Es gibt vor allem noch Aufsatzliteratur. Ansonsten gibt es noch Bücher / Monographien – hauptsächlich Dissertationen und Habilitationen –, wobei diese in der praktischen Arbeit im Gegensatz zu Aufsätzen kaum verwendet werden. Des Weiteren gibt es noch Gesetzesmaterialien, die die Erwägungen des Gesetzgebers beinhalten.

TW:

Kann man grob sagen, wie sich der Aufwand auf diese Arten aufteilt?

CS:

Das hängt natürlich davon ab, was man macht. Aber in der wissenschaftlichen machen Kommentare, Urteile und Aufsätze sicher ungefähr 90 % aus.

TW:

Und in Kanzleien?

CS:

Da man hier ja immer auch versucht, darzulegen, dass man die allgemeine Rechtsauffassung / Meinung vertritt, spielen auch hier neben den Gesetzen die Kommentare, Urteile und Aufsatzliteratur die gleiche Rolle, um die eigene Argumentation zu stützen. D.h. dass sich hier

die Quellenlage nicht wesentlich von der in der wissenschaftlichen Arbeit unterscheidet – höchsten ein etwas kleinerer Fokus auf Monographien und Gesetzesmaterialien.

TW:
Aus der Perspektive eines Anwalts: Welche Metainformationen fändest du hilfreich, wenn du mit solcher Literatur arbeitest?

CS:
Hilfreich wären relevante Normen. Ansonsten, wobei das wohl sehr aufwändig wäre, wäre es sehr hilfreich, zu einem Rechtsproblem alle relevanten Aufsätze so sortiert aufgelistet zu bekommen, dass man daraus sofort die Aufsätze auswählen kann, die die eigene Meinung stützen. Da viele Urteile recht lang sind, wäre es hilfreich, eine automatische Zusammenfassung / ein automatisches Inhaltsverzeichnis anzeigen zu können. Auch würde es die Tätigkeit sehr erleichtern, wenn der Gesetzestext direkt mit den damit verbundenen Auffassungen verlinkt angezeigt werden kann. Bzgl. des Betriebs eines Kraftfahrzeugs gibt es beispielsweise sowohl die Auffassung, dass der Motor des Fahrzeugs laufen muss, als auch die Auffassung, dass es bereits ausreicht, wenn das Fahrzeug auf der Straße geparkt wurde. Wenn also der Betrieb eines Kraftfahrzeugs für einen Sachverhalt relevant ist, wäre es hilfreich zu sehen, wer welche Auffassung hierzu vertritt.

TW:
Da waren schon einige interessante Sachen dabei. Ich hätte auch noch ein paar Vorschläge. Du hast ja schon relevante Normen genannt. Fändest du auch verwendete Hilfsnormen hilfreich?

CS:
Ja, solche Legaldefinitionen sind aber auch immer recht leicht zu finden, da die Definition des Begriffs dann immer in Klammern steht.

TW:
Gemeint wären damit auch Definitionen wie z.B. § 2 im Produkthaftungsgesetz. Dort steht ja genau, was im Rahmen dieses Gesetzes mit dem Begriff „Produkt“ gemeint ist.

CS:
Ja, das wäre auch sehr sinnvoll. In diesem Zusammenhang wäre auch ein Hinweis auf mögliche Ausnahmen sinnvoll. Es gibt mitunter z.B. auch die Formulierung „Dies gilt nicht, wenn (…)“.

Letztendlich muss für einen Paragraphen immer geprüft werden, ob die Voraussetzungen für dessen Anwendbarkeit gegeben sind. Logisch gesehen sind solche Ausnahmen nichts anderes als negative Voraussetzungen, die allerdings häufig an einer anderen Stelle, z.B. einem anderen Paragraphen, stehen. Daher wäre es sinnvoll, wenn solche Ausnahmen für einen Paragraphen erkannt werden, sodass der Leser weiß, dass neben den dort formulierten Voraussetzungen noch weitere Bedingungen geprüft werden müssen. Damit könnte eine Art „Vervollständigung" der Normen erreicht werden, was schon sehr hilfreich wäre.

TW:
Du hast ja schon im Kontext mit unterschiedlichen Auslegungen ein verlinktes Gesetz vorgeschlagen. Fändest du es auch hilfreich ein- und ausgehende Referenzen zu anderen Paragraphen direkt zu verlinken?

CS:
Da, die referenzierten Paragraphen zum Teil sehr stark verteilt sind, wäre das enorm hilfreich. Das Prinzip wäre ja sehr ähnlich wie bei den Legaldefinitionen und würde sehr viel Zeit sparen.

TW:
Aus der Sicht eines Verfassers von Texten wie z.B.: Gesetze, Argumentationen oder Verträge: Fändest du es hilfreich, wenn auch unbestimmte Rechtsbegriffe wie „angemessen", „adäquat", etc. gekennzeichnet werden, sodass diese vermieden werden können?

CS:
Hmm, eher weniger. Oft will man diese ja gar nicht vermeiden; wenn man z.B.: „angemessen" verwendet, macht man das durchaus mit Absicht, um genau diesen Auslegungsspielraum zu schaffen. Z.B. ist eine „angemessene Nachbesserungsfrist" bei einem bestellten Flugzeug eine ganz andere als etwa bei einer Packung Windeln.

TW:
Und nochmal aus der Sicht des Verfassers von Texten: Wie wichtig fändest du linguistische Informationen, wie z.B. den Flesch-Reading-Ease-Index, der die Lesbarkeit des Textes angibt?

CS:
Sehr wichtig. Häufig sind die Texte z.B. zu lang. Oft sind sie auch zu umständlich formuliert. Es kommt z.B. vor, dass nach einem Satz eine Ausnahme folgt und danach nochmal eine

Ausnahme von der Ausnahme. Je länger diese Kette wird, desto schwieriger ist dieser Text zu lesen.

TW:
Ich würde dir jetzt gern eine Webanwendung zeigen, die in einem Projekt im letzten Semester entstanden habe. Sie ist noch nicht auf die Rechtsdomäne zugeschnitten und ist bereits in der Lage, linguistische Informationen aus dem Text zu erkennen und anzuzeigen. Stell dir vor, die Anwendung würde in derselben Art und Weise die vorher erörterten Metainformation präsentieren. Findest du diesen Ansatz generell sinnvoll?

CS:
Ja, wenn ich z.B. bei einem Begriff mit einem Rechtsklick sofort die dazu gehörende Legaldefinition sehen könnte, wäre das sehr hilfreich und würde viel Zeit sparen. Neben Legaldefinitionen gibt es ja auch noch Begriffe, die nicht legal-definiert sind, sondern wo es wie bereits angesprochen unterschiedliche Auslegungen gibt. Besonders hilfreich wäre es in diesem Zusammenhang natürlich, wenn dann alle gebräuchlichen Definitionen mit deren Quellen angezeigt werden könnten. Das würde die Arbeit natürlich erheblich beschleunigen. Als Anwalt geht es unter anderem schon auch um Masse. Wenn man also auch ganz einfach alle Fundstellen in Kommentaren angezeigt bekäme und diese dann zur Stützung der eigenen Argumentation verwenden könnte, wäre das auch sehr hilfreich.

TW:
Fallen dir noch weitere Anforderungen, wie z.B. eigene Kommentare, an eine solche Software ein?

CS:
Eigene Kommentare wären jetzt keine Priorität für mich, da man das auch jetzt schon gut mit MS Word kann. Daher sollte die Software idealerweise auch Word-kompatibel sein. Ansonsten wäre es noch hilfreich, wenn ich gerade bei längeren Urteilen Unterüberschriften hätte und darauf klicken könnte, sodass nur die Abschnitte angezeigt werden, die mich gerade interessieren. Besonders interessant wäre natürlich eine Funktion, mit der der Text in entscheidungserhebliche Teile und entscheidungsunerhebliche Teile aufgeteilt wird. Ich denke aber, dass das nicht automatisiert machbar ist. Dazu ist ja dann schon ein Sprachverständnis auf höherem Niveau erforderlich.

TW:
Hättest du noch ein paar Vorschläge, wie die Benutzerfreundlichkeit erhöht werden könnte?

CS:
Ich finde das eigentlich schon sehr übersichtlich. Schön wären vielleicht noch weitere Spalten. Wenn z.B. auf einen Begriff klicke, wäre es gut, wenn dessen Definition nicht in einem flüchtigen Popup sondern in einer eigenen Spalte erscheint. Ansonsten wäre vielleicht noch gut, in einer weiteren Spalte einen anderen Text anzeigen zu können, sodass hierfür kein separates Browserfenster erforderlich ist.

TW:
Was müsste denn ein solches System können, um es auch für das Verfassen eigener Texte zu verwenden?

CS:
Ich glaube, realistisch gesehen werden die meisten Leute nicht darauf verzichten, ihre Texte in Word zu verfassen. Daher sollte es auch Word-kompatibel sein.

TW:
Dann sind wir auch schon durch. Vielen Dank nochmal, dass du dir die Zeit genommen hast.

CS:
Gern, ich hoffe, ich konnte dir weiterhelfen.

Riederer, Johannes – 30.07.2015

TW:

Guten Tag. Vielen Dank, dass du dir die Zeit für das Interview nimmst. Fangen wir gleich mit der ersten Frage an: Wie viel Zeit verbringst du ungefähr mit der Recherche von Rechtsliteratur (Gesetze, Urteile, Kommentare, etc.)?

JR:

Relativ viel, fast die Hälfte der Arbeitszeit.

TW:

Gibt es außer Gesetzen, Urteilen und Kommentaren noch weitere Literatur, die in diesem Zusammenhang relevant ist?

JR:

Es gibt noch Zeitschriften, Aufsätze, Monographien / Festschriften, Dissertationen.

TW:

Kann man grob sagen, wie sich der Zeitaufwand auf diese Literaturarten aufteilt?

JR:

Ausgangspunkt ist immer das Gesetz. Meistens schlage ich danach in Kommentaren nach und erst dann in weiterer Literatur wie z.B. Aufsätzen.

TW:

Welche Meta-Informationen fändest du hilfreich / würden dir Zeit sparen, wenn du mit Rechtsliteratur arbeitest?

JR:

Legaldefinitionen sind immer hilfreich. Die Systematik innerhalb eines Gesetzes wäre noch interessant, d.h. z.B. eine Anzeige, in welchem Buch und in welchem Abschnitt ich mich gerade befinde. Ansonsten werden Normen noch in Normen unterschieden, die eine unmittelbare Rechtsfolge beinhalten und solche, die lediglich Hilfsnormen sind und eine vorbereitende Wirkung auf die darauf folgenden Normen haben. Solche Definitionen, die jetzt nicht zwingenderweise Legaldefinitionen sind, nachzuschlagen ist immer etwas lästig.

TW:

Fändest du es hilfreich, unbestimmte Rechtsbegriffe wie „angemessen“ oder „adäquat“ zu markieren?

JR:

Eher weniger, diese Begriffe enthalten ohnehin kaum Informationsgehalt. Ich denke, eine Markierung würde keinen Mehrwert bringen.

TW:

Aus der Perspektive eines Verfassers von Texten, etwa einem Anwalt, der seine Argumentationsschrift formuliert: Wie hilfreich fändest du linguistische Informationen, etwa einen Lesbarkeitsindex?

JR:

Das ist eigentlich gar nicht verkehrt. Beim Schreiben eines Textes stellt sich immer die Frage, ob dieser dann auch verständlich formuliert wurde. Jeder hat ja letztendlich seinen eigenen Schreibstil wodurch am Ende die Verständlichkeit leidet. Was ich aber noch schlimmer finde, ist in diesem Zusammenhang, dass einem die eigenen Gedanken in diesem Moment ja völlig klar sind und man sie daher dann nicht mehr richtig zu Papier bringt. Ich weiß aber nicht, ob das mit Hilfe einer Software behoben werden könnte, da es hierbei ja um inhaltliche Lücken geht, die wohl kaum automatisiert erkannt werden können. Was ich auch sehr hilfreich finde, sieht man in Kopp-Kommentaren (Kommentare des Autors Ferdinand O. Kopp, Anm. d. Interviewers). Der hat hier so etwas Ähnliches wie Fußnoten, die den Randziffern entsprechen. D.h. man kann hier direkt vom Gesetzestext in die Stelle, an der im Kommentar darauf Bezug genommen wird. Das entspricht sozusagen einem Link in der Computerwelt. In Kommentaren von anderen Autoren steht so etwas leider wild durcheinander. So ein System wäre aber schön, wenn ich dann einfach auf den Verweis klicken kann und dann direkt bei der entsprechenden Randziffer lande.

TW:

Ich würde dir jetzt gern eine Webanwendung zeigen, die in einem Projekt im letzten Semester entstanden habe. Sie ist noch nicht auf die Rechtsdomäne zugeschnitten und ist bereits in der Lage, linguistische Informationen aus dem Text zu erkennen und anzuzeigen. Stell dir vor, die Anwendung würde in derselben Art und Weise die vorher erörterten Metainformation präsentieren. Findest du diesen Ansatz generell sinnvoll?

JR:
Ja, ich finde diesen Ansatz interessant.

TW:
Wofür würdest du ein solches System denn gern verwenden bzw. was soll es können.

JR:
Das hängt ein bisschen von der Art des Textes ab. Schön wären z.B. neben den Legaldefinitionen noch weitere Hilfsnormen, wie sie z.B. am Anfang des BGB stehen. Ich glaube aber, dass es sehr schwer ist, diese mit informatischen Möglichkeiten zu erkennen. Es tauchen zwar einige Begriffe immer wieder auf, aber im Gegensatz zu echten Legaldefinitionen folgen sie keiner einheitlichen Struktur. Viele Definitionen sind auch gar nicht niedergeschrieben sondern allgemein anerkannt, etwa der Begriff der Willenserklärung. Abgesehen davon habe ich mir z.B. einmal alle Anspruchsgrundlagen schwarz unterstrichen. Das wäre im Examen natürlich nicht zulässig, aber darum geht es hier ja nicht. Generell denke ich, dass gerade die Informationen, die man sich für die Examensvorbereitung in die Gesetzestexte schreibt – und noch mehr diejenigen, die man sich dort gern notieren würde, die aber im Examen nicht zulässig sind, - für den Anwender sehr wichtig sind und daher – soweit möglich – von der Software zur Verfügung gestellt werden sollten. Ein anderes Thema sind Rechtsfolgenverweisungen und Rechtsgrundverweisungen. Hier wäre es interessant zu wissen, ob bei einem solchen Verweis nur die Folge relevant ist oder auch die Voraussetzungen des referenzierten Paragraphen. D.h. eine Markierung von Tatbestandsmerkmalen und Rechtsfolgen wäre sicher nützlich. Gerade die Markierung von Tatbestandsmerkmalen wäre wohl das wichtigste.

TW:
Wie hilfreich fändest du eine Funktion, eigene Kommentare in der Rechtsliteratur zu hinterlassen?

JR:
Das wurde bei der Vorbereitung für das Examen auch gemacht, bis es verboten wurde. Also offenbar gibt es diesen Bedarf.

TW:

Hast du irgendwelche Vorschläge, wie man diese Anwendung benutzerfreundlicher gestalten könnte?

JR:

Etwas benutzerfreundlicher fände ich es, wenn die Kommentare nicht nur am Rand stehen, sondern man direkt sieht, mit welchem Textausschnitt der Kommentar verknüpft ist.

TW:

Du hast ja vorher noch vorgeschlagen, dass der eigene Standort innerhalb des Textes angezeigt wird. Eine Möglichkeit wäre z.B. eine Art Vogelperspektive auf den Gesamttext mit einer Markierung der aktuellen Stelle. Wie findest du diese Idee?

JR:

Naja, so ein Gesetz hat ja mehrere Hundert Seiten. Ich glaube, eine solche Vogelperspektive bietet noch keinen besonderen Mehrwert. Hilfreicher fände ich dagegen, wenn ein Inhaltsverzeichnis angezeigt wird, bei dem z.B. die Überschrift der aktuellen Stelle hervorgehoben wird. Eine andere Idee ist die Verwendung von Breadcrumbs innerhalb des Textes.

TW:

Gibt es noch weitere Funktionen, die du dir von einem solchen System wünschen würdest?

JR:

Unser Chef vergibt z.B. Tags, die mit einer Raute markiert sind, um die entsprechenden Stellen danach schnell ansteuern zu können. So ein Suchvorgang lässt sich mit informatischen Möglichkeiten sicher besser umsetzen, beispielsweise durch Setzen von Lesezeichen innerhalb des Texts.

TW:

Was müsste denn so ein System können, um es auch für das Verfassen eigener Texte zu verwenden?

JR:

Das kommt auch ein bisschen auf den Text an. Das wird sich in Kanzleien von der Wissenschaft unterscheiden.

Gerade in Kanzleien kommt es z.B. häufig vor, dass ein Mitarbeiter die erste Recherche macht und noch einige Fragen offen lässt, ein weiterer eventuelle Lücken schließt und ein dritter mit dem Ergebnis weiter arbeitet. Hier wäre es sehr hilfreich, wann alle am gleichen Text arbeiten können und man anschließend sieht wer was eingefügt bzw. geändert hat. Schön wäre es auch, wenn man zu dieser Änderung noch eine Notiz hinzufügen kann, in der die Änderung begründet wird. Da sich eine solche Begründung in der Regel auf entsprechende Fundstellen in Kommentaren, Urteilen, etc. stützt wäre es auch sehr gut, wenn diese dort gleich aufgelistet werden und verlinkt werden können, sodass der nächste Bearbeiter sofort sieht, welche Fundstellen schon berücksichtigt wurden und selbst nicht noch einmal analysiert werden müssen. Sollte man sich eine Fundstelle trotzdem nochmal ansehen wollen, müsste man dank der Verlinkung nicht mehr lang danach suchen. Eine solche gemeinsame Bearbeitung von Texten würde eine gemeinsame Wissensbasis zum Sachverhalt schaffen, sodass ein weiterer Bearbeiter schnell den roten Faden dazu findet.

TW:
Damit sind wir auch schon durch. Vielen Dank nochmal für deine Zeit.

JR:
Gern.

Figures

Figure 22: Screenshot of a previous project used as an interactive mockup in interviews
Source: Own illustration

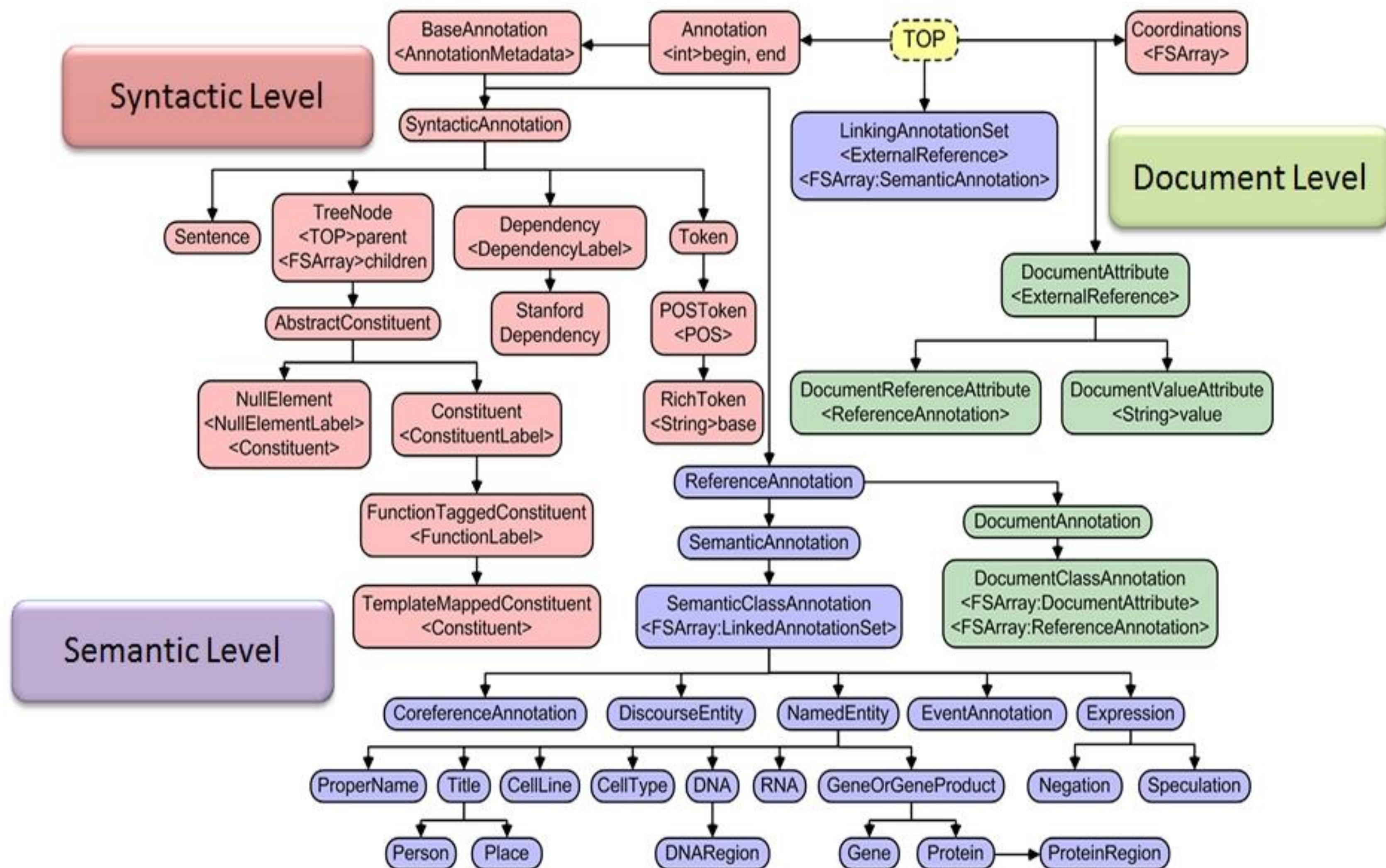


Figure 23: The type system of U-Compare
Source: (The National Centre for Text Mining - The University of Manchester, n.d.)

References

Afantenos, S. D., Petasis, G., & Karkaletsis, V. (2002). *Developers' Guide to Ellogon*. Retrieved from National Center for Scientific Research "DEMOKRITOS" website: http://www.ellogon.org/index.php/download/all-categories/category/7-ellogon-documentation-manuals?download=21:ellogondevelopersguide, last access on 31.08.2015

Allen, J. F. (2003). Natural Language Processing. In *Encyclopedia of Computer Science* (pp. 1218–1222). Chichester, UK: John Wiley and Sons Ltd. Retrieved from http://dl.acm.org.eaccess.ub.tum.de/citation.cfm?id=1074100.1074630, last access on 11.09.2015

Amstad, T. (1978). *Wie verständlich sind unsere Zeitungen?* (Dissertation). Universität Zürich.

Apache UIMA Development Community. (2014). *UIMA Asynchronous Scaleout: Version 2.6.0*. Retrieved from The Apache Software Foundation website: https://uima.apache.org/d/uima-as-2.6.0/uima_async_scaleout.pdf, last access on 26.09.2015

Apache UIMA Development Community. (2015a). *UIMA Tools Guide and Reference: Version 2.8.1*. Retrieved from The Apache Software Foundation website: https://uima.apache.org/d/uimaj-current/tools.pdf, last access on 09.10.2015

Apache UIMA Development Community. (2015b). *UIMA Tutorial and Developers' Guides: Version 2.8.1*. Retrieved from The Apache Software Foundation website: https://uima.apache.org/d/uimaj-current/tutorials_and_users_guides.pdf, last access on 26.09.2015

Appelt, D. E., & Onyshkevych, B. (1998). The Common Pattern Specification Language. In : *TIPSTER '98, Proceedings of a Workshop on Held at Baltimore, Maryland: October 13-15, 1998* (pp. 23–30). Stroudsburg, PA, USA: Association for Computational Linguistics. Retrieved from http://dx.doi.org/10.3115/1119089.1119095

Bamberger, R., & Vanecek, E. (1984). *Lesen, verstehen, lernen, schreiben: die Schwierigkeitsstufen von Texten in deutscher Sprache*. Wien: Jugend und Volk.

Bank, M., & Schierle, M. (2012). A Survey of Text Mining Architectures and the UIMA Standard. In Nicoletta Calzolari (Conference Chair), K. Choukri, T. Declerck, M. U. Doğan, B. Maegaard, J. Mariani, J. Odijk, S. Piperidis (Eds.), *Proceedings of the Eight International Conference on Language Resources and Evaluation (LREC'12)* (pp. 3479–3486). Istanbul, Turkey: European Language Resources Association (ELRA).

Basili, R., Di Nanni, M., & Pazienza, M. (1999). Engineering of IE Systems: An Object-Oriented Approach. In M. Pazienza (Ed.), *Lecture Notes in Computer Science. Information Extraction* (pp. 134–164). Springer Berlin Heidelberg. Retrieved from http://dx.doi.org/10.1007/3-540-48089-7_8

Besançon, R., Chalendar, G. de, Ferret, O., Gara, F., Laïb, M., Mesnard, O., & Semmar, N. (Eds.) 2010. *LIMA: A Multilingual Framework for Linguistic Analysis and Linguistic Resources Development and Evaluation.*

Bird, S., Klein, E., & Loper, E. (2009). *Natural language processing with Python*. Sebastopol, CA: O'Reilly Media, Inc.

Boitet, C., & Seligman, M. (Eds.) 1994. *The "Whiteboard" Architecture: A Way to Integrate Heterogeneous Components of NLP Systems*. Kyoto, Japan: Association for Computational Linguistics.

Bontcheva, K., Cunningham, H., Roberts, I., & Tablan, V. (2010). Web-based collaborative corpus annotation: Requirements and a framework implementation. *New Challenges for NLP Frameworks*, 20–27.

Bundesgerichtshof. (2015). *Entscheidungen des Bundesgerichtshofs ab 2000*. Retrieved from http://www.bundesgerichtshof.de/DE/Entscheidungen/EntscheidungenBGH/entscheidungenBGH_node.html, last access on 09.10.2015

Buyko, E., & Hahn, U. (2008). Fully embedded type systems for the semantic annotation layer. In *ICGL 2008-Proceedings of the 1st International Conference on Global Interoperability for Language Resources* (pp. 26–33).

Chalendar, G. de (2014). The LIMA Multilingual Analyzer Made Free: FLOSS Resources Adaptation and Correction and Evaluation (LREC-2014), Reykjavik, Iceland, May 26-31, 2014. In Nicoletta Calzolari (Conference Chair), K. Choukri, T. Declerck, H. Loftsson, B. Maegaard, J. Mariani, A. Moreno, J. Odijk, S. Piperidis (Eds.), *Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC'14)* (pp. 2932–2937). Reykjavik, Iceland: European Language Resources Association (ELRA). Retrieved from http://www.lrec-conf.org/proceedings/lrec2014/summaries/362.html, last access on 05.09.2015

Cohen, A. T. (1984). Data Abstraction, Data Encapsulation and Object-oriented Programming. *SIGPLAN Not*, *19*(1), 31–35. doi:10.1145/948415.948418

Comeau, D. C., Islamaj Doğan, R., Ciccarese, P., Cohen, K. B., Krallinger, M., Leitner, F., Lu, Z., Peng, Y., Rinaldi, F., Torii, M., Valencia, A., Verspoor, K., Wiegers, T. C.,Wu, C. H., Wilbur, W. J. (2013). BioC: a minimalist approach to interoperability for biomedical text processing. *Database*, *2013.* doi:10.1093/database/bat064

Copestake, A., Flickinger, D., Pollard, C., & Sag, I. A. (2005). Minimal Recursion Semantics: An Introduction: Research on Language and Computation. *Research Language Computation*, *3*(2-3), 281–332. doi:10.1007/s11168-006-6327-9

Cunningham, H. (2000). *Software Architecture for Language Engineering* (Ph.D). University of Sheffield. Retrieved from http://gate.ac.uk/sale/thesis/, last access on 01.05.2015

Cunningham, H. (2002). GATE, a General Architecture for Text Engineering. *Computers and the Humanities*, *36*(2), 223–254. doi:10.1023/A:1014348124664

Cunningham, H., Bontcheva, K., Tablan, V., & Wilks, Y. (2000). Software Infrastructure for Language Resources: a Taxonomy of Previous Work and a Requirements Analysis. *Proceedings of the 2nd International Conference on Language Ressources and evaluation (LREC-2)*. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.32.2089, last access on 28.04.2015

Cunningham, H., Humphreys, K., Gaizauskas, R., & Wilks, Y. (1997). GATE - a TIPSTER-based general architecture for text engineering. In *Proceedings of the TIPSTER Text Program (Phase III)* (Vol. 6).

Cunningham, H., Maynard, D., Bontcheva, K., Tablan, V., Aswani, N., Roberts, I., Gorrel, G., Funk, A., Roberts, A., Damljanovic, D., Heitz, T., Greenwood, M.A., Saggion, H., Petrak, J., Li, Y., Peters, W., Derczynski, L. (2014). *Developing Language Processing Components with GATE Version 8 (a User Guide)*. Retrieved from University of Sheffield, Department of Computer Science website: https://gate.ac.uk/sale/tao/tao.pdf, last access on 19.09.2015

Cunningham, H., Maynard, D., & Tablan, V. (2000). *JAPE: a Java Annotation Patterns Engine: Technical report CS-00-10.*

Cunningham, H., Tablan, V., Roberts, A., & Bontcheva, K. (2013). Getting More Out of Biomedical Documents with GATE's Full Lifecycle Open Source Text Analytics. *PLoS Comput Biol*, *9*(2), e1002854. doi:10.1371/journal.pcbi.1002854

Deutscher Bundestag. (2014). *Neue Ausgabe des Datenhandbuchs zur Geschichte des Deutschen Bundestages: Statistik zur Gesetzgebung*. Retrieved from https://www.bundestag.de/datenhandbuch/10, last access on 31.05.2015

Dimitrov, M., Cunningham, H., Roberts, I., Kostov, P., Simov, A., Rigaux, P., & Lippell, H. (2014). AnnoMarket – Multilingual Text Analytics at Scale on the Cloud. In V. Presutti, E. Blomqvist, R. Troncy, H. Sack, I. Papadakis, & A. Tordai (Eds.), *Lecture Notes in Computer Science. The Semantic Web: ESWC 2014 Satellite Events* (pp. 315–319). Springer International Publishing. Retrieved from http://dx.doi.org/10.1007/978-3-319-11955-7_41

Eckart de Castilho, R. (2014). *Natural Language Processing: Integration of Automatic and Manual Analysis.* Technische Universität Darmstadt.

Eckart de Castilho, R., & Gurevych, I. (2014). A broad-coverage collection of portable NLP components for building shareable analysis pipelines. In N. Ide & J. Grivolla (Eds.), *Proceedings of the Workshop on Open Infrastructures and Analysis Frameworks for HLT (OIAF4HLT) at COLING 2014.* Dublin, Ireland: Association for Computational Linguistics and Dublin City University.

Ferrucci, D., & Lally, A. (2004). UIMA: An Architectural Approach to Unstructured Information Processing in the Corporate Research Environment. *Natural Language Engineering*, *10*(3-4), 327–348. doi:10.1017/S1351324904003523

Ferrucci, D., Lally, A., Gruhl, D., Epstein, E., Schor, M., Murdock, W. J., Frenkiel, A., Brown, E. W., Hampp, T., Doganata, Y., Welty, C., Amini, L., Kofman, G., Kozakov, L., Mass, Y. (2006). Towards an interoperability standard for text and multi-modal analytics. *IBM Research Report*. Retrieved from http://domino.research.ibm.com/library/cyberdig.nsf/papers/1898F3F640FEF47E8525723C00551250/$File/rc24122.pdf, last access on 26.09.2015

Ferrucci, D., Lally, A., Verspoor, K., & Nyberg, E. (2009). Unstructured Information Management Architecture (UIMA). *OASIS Standard*. Retrieved from http://docs.oasis-open.org/uima/v1.0/uima-v1.0.pdf, last access on 25.09.2015

Flesch, R. (1948). A new readability yardstick. *Journal of applied psychology*, *32*(3), 221–233.

Gaizauskas, R., & Wilks, Y. (1998). Information extraction: beyond document retrieval. *Journal of Documentation*, *54*(1), 70–105. doi:10.1108/EUM0000000007162

Gambäck, B., & Olsson, F. (2000). Experiences of Language Engineering Algorithm Reuse. In *Language Resources and Evaluation.*

Genesereth, M. R., & Ketchpel, S. P. (1994). Software Agents. *Commun. ACM*, *37*(7), 48-ff. doi:10.1145/176789.176794

Gomaa, H. (1984). A software design method for real-time systems. *Commun. ACM*, *27*(9), 938–949. doi:10.1145/358234.358262

Götz, T., & Suhre, O. (2004). Design and implementation of the UIMA Common Analysis System. *IBM Systems Journal*, *43*(3), 476–489. doi:10.1147/sj.433.0476

Graß, T. (2015a). *Development of a web application to manage and edit semantically annotated texts*, Munich.

Graß, T. (2015b). *Development of a web application to manage and edit semantically annotated texts* (Master's thesis). Technische Universität München, Munich. Retrieved from https://wwwmatthes.in.tum.de/pages/1e8akj1lnw7wh/Master-s-Thesis-Thomas-Grass, last access on 11.09.2015

Grishman, R. (1996). TIPSTER Text Phase II Architecture Design. In *Proceedings of the TIPSTER Text Program: Phase II* (pp. 249–305). Vienna, Virginia, USA: Association for Computational Linguistics. Retrieved from http://www.aclweb.org/anthology/X96-1043, last access on 30.08.2015

Grishman, R. (2010). Information Extraction. In *The Handbook of Computational Linguistics and Natural Language Processing* (pp. 515–530). Wiley-Blackwell.

Griss, Martin, L. (1997). Software reuse architecture, process, and organization for business success. In *Computer Systems and Software Engineering, 1997, Proceedings of the Eighth Israeli Conference on* (pp. 86–89).

Guessoum, Z., & Briot, J.-P. (1999). From Active Objects to Autonomous Agents. *IEEE Concurrency*, *7*(3), 68–76. doi:10.1109/4434.788781

Gui, G., & Scott, P. D. (2006). Coupling and cohesion measures for evaluation of component reusability. In *Proceedings of the 2006 international workshop on Mining software repositories* (pp. 18–21). Shanghai, China: ACM.

Hahn, U., Buyko, E., Landefeld, R., Mühlhausen, M., Poprat, M., Tomanek, K., & Wermter, J. (2008). An overview of JCoRe, the JULIE lab UIMA component repository. In *Proceedings of the LREC* (Vol. 8, pp. 1–7).

Hahn, U., Buyko, E., Tomanek, K., Piao, S., McNaught, J., Tsuruoka, Y., & Ananiadou, S. (2007). An Annotation Type System for a Data-driven NLP Pipeline. In : *LAW '07, Proceedings of the Linguistic Annotation Workshop* (pp. 33–40). Stroudsburg, PA, USA: Association for Computational Linguistics. Retrieved from http://dl.acm.org/citation.cfm?id=1642059.1642064, last access on 01.10.2015

Hahn, U., Buyko, E., Tomanek, K., Piao, S., Tsuruoka, Y., McNaught, J., & Ananiadou, S. (2007). An UIMA Annotation Type System for a Generic Text Mining Architecture. In *UIMA-Workshop, GLDV Conference, April.* Retrieved from https://svn.apache.org/repos/asf/uima/site/trunk/uima-website/xdocs/downloads/gldv/gldv07-uima-hahn.pdf, last access on 01.10.2015

Heid, U., Schmid, H., Eckart, K., & Hinrichs, E. W. (2010). A Corpus Representation Format for Linguistic Web Services: The D-SPIN Text Corpus Format and its Relationship with ISO Standards. In Nicoletta Calzolari (Conference Chair), Khalid Choukri, Bente Maegaard, Joseph Mariani, Jan Odijk, Stelios Piperidis, Mike Rosner, Daniel Tapias (Eds.), *Proceedings of the Seventh International Conference on Language Resources and Evaluation (LREC'10)* (pp. 494–499). Valletta, Malta: European Language Resources Association (ELRA).

Heßler, K. (2015, July 3). Interview by T. Waltl [Transcript]. LMU, Munich.

Hevner, A. R., March, S. T., Park, J., & Ram, S. (2004). Design science in information systems research. *MIS Quarterly*, *28*(1), 75–105.

Hinrichs, E. W., Hinrichs, M., & Zastrow, T. (2010). WebLicht: Web-Based LRT Services for German. In *Proceedings of the ACL 2010 System Demonstrations* (pp. 25–29). Retrieved from http://www.aclweb.org/anthology/P10-4005, last access on 08.10.2015

Jackson, P., & Moulinier, I. (2007). *Natural language processing for online applications: Text retrieval, extraction and categorization*: John Benjamins Publishing.

Kano, Y., Baumgartner, W. A., McCrohon, L., Ananiadou, S., Cohen, K. B., Hunter, L., & Tsujii, J. (2009). U-Compare: share and compare text mining tools with UIMA. *Bioinformatics*, *25*(15), 1997–1998. doi:10.1093/bioinformatics/btp289

Kano, Y., Dorado, R., McCrohon, L., Ananiadou, S., & Tsujii, J. (2010). U-Compare: An Integrated Language Resource Evaluation Platform Including a Comprehensive UIMA Resource Library. In *LREC* (pp. 428–434).

Kano, Y., McCrohon, L., Ananiadou, S., & Tsujii, J. (2009). Integrated NLP Evaluation System for Pluggable Evaluation Metrics with Extensive Interoperable Toolkit. In : *SETQA-NLP '09, Proceedings of the Workshop on Software Engineering, Testing, and Quality Assurance for Natural Language Processing* (pp. 22–30). Stroudsburg, PA, USA: Association for Computational Linguistics. Retrieved from http://dl.acm.org/citation.cfm?id=1621947.1621951, last access on 01.10.2015

Kano, Y., Nguyen, N. L., Sætre, R., Yoshida, K., Miyao, Y., Tsuruoka, Y., Matsubayashi, Y., Ananiadou, S., Tsujii, J. (2008). Filling the Gaps Between Tools and Users: A Tool Comparator, Using Protein-Protein Interactions as an Example. In Russ B. Altman, A. Keith Dunker, Lawrence Hunter, Tiffany Murray, & Teri E. Klein (Eds.), *Biocomputing 2008, Proceedings of the Pacific Symposium* (pp. 616–627). World Scientific. Retrieved from http://psb.stanford.edu/psb-online/proceedings/psb08/kano.pdf, last access on 01.10.2015

Kluegl, P., Atzmueller, M., & Puppe, F. (2009). TextMarker: A Tool for Rule-Based Information Extraction. In C. Chiarcos, R. E. de Castilho, & M. Stede (Eds.), *Proceedings of the Biennial GSCL Conference 2009, 2nd UIMA@GSCL Workshop* (pp. 233–240). Gunter Narr Verlag. Retrieved from http://ki.informatik.uni-wuerzburg.de/papers/pkluegl/2009-GSCL-TextMarker.pdf, last access on 06.09.2015

Kluegl, P., Toepfer, M., Beck, P.-D., Fette, G., & Puppe, F. (2014). UIMA Ruta Workbench: Rule-based Text Annotation. In *Proceedings of COLING 2014, the 25th International Conference on Computational Linguistics: System Demonstrations* (pp. 29–33). Dublin, Ireland: Dublin City University and Association for Computational Linguistics. Retrieved from http://www.aclweb.org/anthology/C14-2007, last access on 02.10.2015

Kluegl, P., Toepfer, M., Beck, P.-D., Fette, G., & Puppe, F. (2015). UIMA Ruta: Rapid development of rule-based information extraction applications. *Natural Language Engineering*, *FirstView*, 1–40. doi:10.1017/S1351324914000114

Kontonasios, G., Korkontzelos, I., Kolluru, B., & Ananiadou, S. (2012). Adding text mining workflows as web services to the BioCatalogue. In *Proceedings of the 4th International Workshop on Semantic Web Applications and Tools for the Life Sciences* (pp. 50–57). London, United Kingdom: ACM.

Krieger, H.-U. (2003). SDL: A Description Language for Building NLP Systems. In : *SEALTS '03, Proceedings of the HLT-NAACL 2003 Workshop on Software Engineering and Architecture of Language Technology Systems - Volume 8* (pp. 83–90). Stroudsburg, PA, USA: Association for Computational Linguistics. Retrieved from http://dx.doi.org/10.3115/1119226.1119238

Leidner, J. L. (2003). Current issues in software engineering for Natural Language Processing. In *Proceedings of the HLT-NAACL 2003 workshop on Software engineering and architecture of language technology systems - Volume 8* (pp. 45–50). Association for Computational Linguistics.

Liddy, E. D. (2001). Natural Language Processing. In *Encyclopedia of Library and Information Science* (2nd ed.). N.Y.: Marcel Decker, Inc.

Manning, C. D., Surdeanu, M., Bauer, J., Finkel, J., Bethard, S. J., & McClosky, D. (2014). The Stanford CoreNLP Natural Language Processing Toolkit. In *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations* (pp. 55–60). Retrieved from http://www.aclweb.org/anthology/P/P14/P14-5010, last access on 01.10.2015

Marcus, M. P., Marcinkiewicz, M. A., & Santorini, B. (1993). Building a Large Annotated Corpus of English: The Penn Treebank. *Comput. Linguist.*, *19*(2), 313–330. Retrieved from http://dl.acm.org/citation.cfm?id=972470.972475, last access on 26.09.2015

Nazarenko, A., Alphonse, É., Derivière, J., Hamon, T., Vauvert, G., & Weissenbacher, D. (2006). The ALVIS Format for Linguistically Annotated Documents. *Computing Research Repository*, *abs/cs/060*.

Nelson, T. H. (1997). Embedded Markup Considered Harmful. *World Wide Web Journal*, *2*(4), 129–134. Retrieved from http://dl.acm.org/citation.cfm?id=274784.273632, last access on 02.10.2015

Ogren, P. V., & Bethard, S. J. (2009). Building Test Suites for UIMA Components. In : *SETQA-NLP '09, Proceedings of the Workshop on Software Engineering, Testing, and Quality Assurance for Natural Language Processing* (pp. 1–4). Stroudsburg, PA, USA: Association for Computational Linguistics. Retrieved from http://dl.acm.org/citation.cfm?id=1621947.1621948, last access on 26.09.2015

Pallotta, V., & Ballim, A. (2001). Agent-Oriented Language Engineering for Robust NLP. In A. Omicini, P. Petta, & R. Tolksdorf (Eds.), *Lecture Notes in Computer Science. Engineering Societies in the Agents World II* (pp. 86–104). Springer Berlin Heidelberg. Retrieved from http://dx.doi.org/10.1007/3-540-45584-1_7

Parnas, D. L. (1972). On the Criteria to Be Used in Decomposing Systems into Modules. *Commun. ACM*, *15*(12), 1053–1058. doi:10.1145/361598.361623

Petasis, G. (2010). Ellogon and the challenge of threads. In *Proceedings of the 17th Annual Tcl/Tk Conference (Tcl 2010).* Hilton Suites Chicago/Oakbrook Terrace, 10 Drury Lane, Oakbrook Terrace, Illinois, United States 60181. Retrieved from http://www.ellogon.org/petasis/bibliography/Tcl2010/EllogonAndThreads.pdf, last access on 31.08.2015

Petasis, G. (2014). The Ellogon Pattern Engine: Context-free Grammars over Annotations. In N. Calzolari, K. Choukri, T. Declerck, H. Loftsson, B. Maegaard, J. Mariani, A. Moreno, J. Odijk, S. Piperidis (Eds.), *Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC-2014), Reykjavik, Iceland, May 26-31, 2014* (pp. 2460–2465). European Language Resources Association (ELRA).

Petasis, G., Karkaletsis, V., Paliouras, G., Androutsopoulos, I., & Spyropoulos, C. D. (2002). Ellogon: A New Text Engineering Platform. *CoRR*, *cs.CL/0205017*.

Petasis, G., Karkaletsis, V., Paliouras, G., & Spyropoulos, C. D. (2003). Using the Ellogon Natural Language Engineering Infrastructure. In *Proceedings of the Workshop on Balkan Language Resources and Tools, 1st Balkan Conference in Informatics (BCI 2003).* Thessaloniki, Greece. Retrieved from http://www.ellogon.org/petasis/bibliography/BCI2003/BCI2003-Petasis.pdf, last access on 31.08.2015

Qureshi, P. A. R., Memon, N., & Wiil, U. K. (Eds.) 2011. *LanguageNet: A Novel Framework for Processing Unstructured Text Information*. Intelligence and Security Informatics (ISI), 2011 IEEE International Conference on.

Rak, R., Batista-Navarro, R., Rowley, A., Carter, J., & Ananiadou, S. (2013). Customisable Curation Workflows in Argo. In *Proceedings of the Fourth BioCreative Challenge Evaluation Workshop* (Vol. 1, pp. 270–278).

Rak, R., Carter, J., Rowley, A., Batista-Navarro, R. T., & Ananiadou, S. (2014). Interoperability and Customisation of Annotation Schemata in Argo. In Nicoletta Calzolari (Conference Chair), K. Choukri, T. Declerck, H. Loftsson, B. Maegaard, J. Mariani, A. Moreno, J. Odijk, S. Piperidis (Eds.), *Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC'14)* (pp. 3837–3842). Reykjavik, Iceland: European Language Resources Association (ELRA).

Rak, R., Rowley, A., Black, W., & Ananiadou, S. (2012). Argo: an integrative, interactive, text mining-based workbench supporting curation. *Database*, *2012.* doi:10.1093/database/bas010

Rak, R., Rowley, A., Carter, J., & Ananiadou, S. (Eds.) 2013. *Development and Analysis of NLP Pipelines in Argo.*

Riederer, J. (2015, July 30). Interview by T. Waltl [Transcript]. LMU, Munich.

Schäfer, Ulrich. (2006). Middleware for creating and combining multi-dimensional NLP markup. In *Proceedings of the 5th Workshop on NLP and XML: Multi-Dimensional Markup in Natural Language Processing* (pp. 81–84). Trento, Italy: Association for Computational Linguistics.

Schäfer, Ulrich. (2007). *Integrating Deep and Shallow Natural Language Processing Components - Representations and Hybrid Architectures* (Dissertation). Faculty of Mathematics and Computer Science, Saarland University, Saarbrücken, Germany. Retrieved from http://www.dfki.de/ uschaefer/diss/, last access on 16.09.2015

Schäfer, Ulrich (2008). Shallow, Deep and Hybrid Processing with UIMA and Heart of Gold. In *Proceedings of the LREC-2008 Workshop Towards Enhanced Interoperability for Large HLT Systems: UIMA for NLP, 6th International Conference on Language Resources and Evaluation* (pp. 43–50). ELRA.

Schmid, H. (1994). Probabilistic Part-of-Speech Tagging Using Decision Trees. In *International Conference on New Methods in Language Processing* (pp. 44–49). Manchester, UK. Retrieved from ftp://ftp.ims.uni-stuttgart.de/pub/corpora/tree-tagger1.pdf, last access on 07.10.2015

Schmid, H. (1999). Improvements in Part-of-Speech Tagging with an Application to German. In S. Armstrong, K. Church, P. Isabelle, S. Manzi, E. Tzoukermann, & D. Yarowsky (Eds.), *Text, Speech and Language Processing. Natural Language Processing Using Very Large Corpora* (pp. 13–26). Dordrecht: Kluwer Academic Publishers. Retrieved from ftp://ftp.ims.uni-stuttgart.de/pub/corpora/tree-tagger2.pdf, last access on 07.10.2015

Schor, M. (2003). *An Effective, Java-Friendly Interface to the CAS*. Retrieved from IBM Research Division website: http://domino.research.ibm.com/library/cyberdig.nsf/papers/D55875841121943E85256E78004BD826/$File/rc23176.pdf, last access on 09.10.2015

Settles, B. (2005). ABNER: an open source tool for automatically tagging genes, proteins and other entity names in text. *Bioinformatics*, *21*(14), 3191–3192. doi:10.1093/bioinformatics/bti475

Splinter, C. (2015, July 10). Interview by T. Waltl [Transcripts]. LMU, Munich.

Standish, T. A. (1984). An Essay on Software Reuse. *Software Engineering, IEEE Transactions on*, *SE-10*(5), 494–497. doi:10.1109/TSE.1984.5010272

Stefanini, M.-H., & Demazeau, Y. (1995). TALISMAN: A multi-agent system for natural language processing. In J. Wainer & A. Carvalho (Eds.), *Lecture Notes in Computer Science. Advances in Artificial Intelligence* (pp. 312–322). Springer Berlin Heidelberg. Retrieved from http://dx.doi.org/10.1007/BFb0034824

Stefanini, M.-H., & Warren, K. (1996). A Distributed Architecture for Text Analysis in French: An Application to Complex Linguistic Phenomena Processing. In : *COLING '96, Proceedings of the 16th Conference on Computational Linguistics - Volume 2* (pp. 1151–1154). Stroudsburg, PA, USA: Association for Computational Linguistics. Retrieved from http://dx.doi.org/10.3115/993268.993388

Steinberg, D., Budinsky, F., Merks, E., & Paternostro, M. (2008). *eclipse Modeling Framework*: Pearson Education.

Stenetorp, P., Pyysalo, S., Topi´c, G., Ohta, T., Ananiadou, S., & Tsujii, J. (2012). BRAT: a web-based tool for NLP-assisted text annotation. In *Proceedings of the Demonstrations at the 13th Conference of the European Chapter of the Association for Computational Linguistics* (pp. 102–107). Avignon, France: Association for Computational Linguistics.

Tablan, V. (2013). *Announcing the AnnoMarket News Pipeline*. Retrieved from https://annomarket.eu/news/2013/11/07/annomarket-news-pipeline/, last access on 08.10.2015

Tablan, V., Bontcheva, K., Roberts, I., Cunningham, H., & Dimitrov, M. (2013). AnnoMarket: An Open Cloud Platform for NLP. In *ACL (Conference System Demonstrations)* (pp. 19–24).

Tablan, V., Roberts, I., Cunningham, H., & Bontcheva, K. (2011). GATECloud. net: Cloud Infrastructure for Large-Scale, Open-Source Text Processing. In *UK e-Science All hands Meeting*. Retrieved from https://gate.ac.uk/sale/ahm11/ahm11-short-final.pdf, last access on 19.09.2015

Tablan, V., Roberts, I., Cunningham, H., & Bontcheva, K. (2012). GATECloud.net: a platform for large-scale, open-source text processing on the cloud. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, *371*(1983). doi:10.1098/rsta.2012.0071

The National Centre for Text Mining - The University of Manchester. (n.d.). *U-Compare Type System*. Retrieved from http://nactem.ac.uk/ucompare/documentation/type-system/, last access on 02.10.2015

Thompson, P., Kano, Y., McNaught, J., Pettifer, S., Attwood, T., Keane, J., & Ananiadou, S. (2011). Promoting interoperability of resources in meta-share. In *Proceedings of the IJCNLP Workshop on Language Resources, Technology and Services in the Sharing Paradigm (LRTS)* (pp. 50–58).

Toutanova, K., Klein, D., Manning, C. D., & Singer, Y. (2003). Feature-rich Part-of-speech Tagging with a Cyclic Dependency Network. In : *NAACL '03, Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology - Volume 1* (pp. 173–180). Stroudsburg, PA, USA: Association for Computational Linguistics. Retrieved from http://dx.doi.org/10.3115/1073445.1073478

Toutanova, K., & Manning, C. D. (2000). Enriching the Knowledge Sources Used in a Maximum Entropy Part-of-speech Tagger. In : *EMNLP '00, Proceedings of the 2000 Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora: Held in Conjunction with the 38th Annual Meeting of the Association for*

Computational Linguistics - Volume 13 (pp. 63–70). Stroudsburg, PA, USA: Association for Computational Linguistics. Retrieved from http://dx.doi.org/10.3115/1117794.1117802

Walter, S. (2010). *Definitionsextraktion aus Urteilstexten* (Dissertation). Universität des Saarlandes, Saarbrücken, Germany. Retrieved from http://www.coli.uni-saarland.de/bib/files/DissertationStephanWalter.pdf, last access on 20.07.2015

Wilcock, G. (2009). *Introduction to linguistic annotation and text analytics: Synthesis Lectures on Human Language Technologies*. San Rafael, CA: Morgan & Claypool Publishers.

Wolinski, F., Vichot, F., & Gremont, O. (1998). Producing NLP-based On-line Contentware. *Computing Research Repository*, *cs.CL/9809*.

Yimam, S. M., Eckart de Castilho, R., Gurevych, I., & Biemann, C. (2014). Automatic Annotation Suggestions and Custom Annotation Layers in WebAnno. In K. Bontcheva & Z. Jingbo (Eds.), *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics. System Demonstrations* (pp. 91–96). Stroudsburg, PA 18360, USA: Association for Computational Linguistics.

Yimam, S. M., Gurevych, I., Eckart de Castilho, R., & Biemann, C. (2013). WebAnno: A Flexible,Web-based and Visually Supported System for Distributed Annotations. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (System Demonstrations) (ACL 2013)* (pp. 1–6). Stroudsburg, PA, USA: Association for Computational Linguistics.

Yourdon, E., & Constantine, L. L. (1979). *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*: Prentice-Hall, Inc.

Zajac, R., Casper, M., & Sharples, N. (1997). An Open Distributed Architecture for Reuse and Integration of Heterogeneous NLP Components. In : *ANLC '97, Proceedings of the Fifth Conference on Applied Natural Language Processing* (pp. 245–252). Stroudsburg, PA, USA: Association for Computational Linguistics. Retrieved from http://dx.doi.org/10.3115/974557.974593