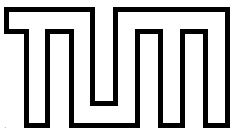


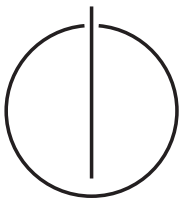
Technische Universität München

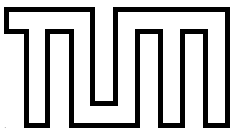
Fakultät für Informatik

**Master's Thesis in Information Systems**

# Development of a web application to manage and edit semantically annotated texts

Thomas Graß





Technische Universität München

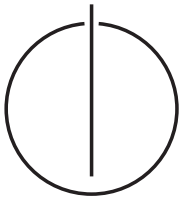
Fakultät für Informatik

**Master's Thesis in Information Systems**

# Development of a web application to manage and edit semantically annotated texts

# Entwicklung einer Web-Anwendung zur Verwaltung und Bearbeitung von semantisch annotierten Textsammlungen

| | |
|---|---|
| Author: | Thomas Graß |
| Supervisor: | Prof. Dr. rer. nat. Florian Matthes |
| Advisor: | Bernhard Waltl, M.Sc. |
| Date: | 15. September 2015 |



I confirm that this Master's Thesis is my own work and i have documented all sources and materials used.

Munich, 15. September 2015

. . . . . .
*(Thomas Graß)*

**Acknowledgments**

By submitting this Master's Thesis, a long time at the Technical University of Munich ends for me. I would like to say thank you to my advisor Bernhard Waltl for the great support during the period of this work. I would also like to thank my supervisor Prof. Dr. Florian Matthes for giving me the opportunity to write this Master's Thesis at his chair.

I want to say a special thank you to my family and friends, who supported me emotionally at any given time, especially my parents Elfriede and Elmar Graß and my sweetheart.

*Thomas Graß, 15. September 2015*

**Abstract**

Legal texts are developed for thousands of years to regulate the people's coexistence. During their development they became much more complex. Nowadays, legal texts are significantly more complex than they were in the past. It is hard to read and interpret them correctly.

With increasing computing capacity and many efforts in research, advanced text analysis frameworks did rapidly enhanced the last years. Part-of-speech tagging has been a highly manual task in the past, nowadays algorithms can perform this task with high precision automatically. Advanced text analysis can be used to support the process of understanding and interpreting complex legal texts. To perform advanced text analysis, frameworks must be tailored to a specific domain, such as the legal domain. In the first part of this Master's Thesis a research in the German legislation acquires the basic knowledge for further text analysis in the legal domain. The result of this research are data-models of different legal document types, namely legislative documents, jurisprudence documents and legal literature.

The second part of this work addresses the implementation of a generic importer for different types and different sources of legal documents. The implementation uses the knowledge of the prior research in the German legislation and is a necessary component for the next part, the handling of semantic text annotations.

Semantic text annotations are required to perform advanced text analysis and to persist the results of this computationally intensive process. This work discussed the benefits and drawbacks of the different types of semantic text annotations, in-line and stand-off semantic text annotations.

Finally, based on the the aggregated knowledge an application for generic semantic text annotation will be developed.

**Kurzfassung**

Seit Jahrtausenden werden Rechtstexte entwickelt, um das Zusammenleben zwischen Menschen zu regulieren. Während ihrer Entwicklung wurden diese immer komplexer. Rechtstexte sind heute deutlich komplexer als sie in der Vergangenheit jemals waren - es ist schwer diese zu lesen und deren Inhalt richtig zu interpretieren.

Der steigenden Rechenkapazität und großen Anstrengungen in der Forschung ist es zu verdanken, dass sich die Werkzeuge zur Durchführung von komplexen Textanalysen deutlich verbessert haben. In der Vergangenheit war Part-of-speech Tagging eine rein manuelle Aufgabe, das hat sich inzwischen geändert: Heutzutage übernehmen Algorithmen diese Aufgabe automatisch und mit hoher Präzision. Komplexe Textanalysen können das Verstehen und das Interpretieren von Rechtstexten unterstützen. Um komplexe Textanalysen durchführen zu können müssen die Werkzeuge auf die Anwendungsdomäne, wie die Rechtswissenschaft, zugeschnitten sein. Im ersten Teil dieser Masterarbeit werden Forschungen in der deutschen Legislative durchgeführt. Die Ergebnisse dieser Forschungsarbeiten sind Datenmodelle einzelner Typen von Rechtstexten, wie beispielweise Rechtstexte der Legislative oder Judikative.

Der zweite Teil dieser Arbeit beschäftigt sich mit der Implementierung eines Generic Importers mit dem es möglich sein soll, verschiedene Rechtstexte von unterschiedlichen Quellen und Dateiformaten zu importieren. Die Implementierung baut auf dem Wissen der durchgeführten Forschungsarbeiten auf und ist ein wichtiger Bestandteil des nächsten Teils: Semantische Text Annotationen.

Semantische Text Annotationen werden zum einen zur Durchführung von komplexen Textanalysen und zum anderen zum Persistieren dieser rechenaufwändigen Prozesse benötigt. Diese Arbeit debattiert über die Vor- und Nachteile der unterschiedlichen Arten von semantischen Text Annotationen: In-line und Stand-off Text Annotationen.

Am Ende dieser Arbeit wird auf Basis des aggregierten Wissens eine Applikation für generische semantische Text Annotationen entwickelt.

# Contents

# 1 Introduction

Nowadays most of legal texts like laws or judgements change each few years. When looking over the last decades, most of the changes did not make legal texts easier to read and to understand. The oldest legal text collection we know today is the "Code of Ur-Nammu". The "Code of Ur-Nammu" has been commissioned by king Ur-Nammu around the year 2100 B.C. This legal text collection consists of an accumulation of 32 laws (see [Rot95]). All those laws do describe what is not allowed and how the punishment for the criminal looks like. To receive an impression of the "Code of Ur-Nammu"'s laws, the first three paragraphs (see [Ger12]) of this legal text collection are described below:

1. *If a man commits a murder, that man must be killed.*
2. *If a man commits a robbery, he will be killed.*
3. *If a man commits a kidnapping, he is to be imprisoned and pay 15 shekels of silver.*
4. ...

Independently of the terrifying judgments of the "Code of Ur-Nammu", there is one good thing to talk about: The laws were easy to read and to understand (for those who were able to read). In the year 2100 B.C. a criminal knew that if he commits a murder, he will also get killed. This is precisely what was described in the legal text collection and we can discover some conditions, exceptions or relations to other legal texts.

More than 4000 years have passed since that time and today the "Code of Ur-Nammu" acts as a stone plate in a museum. Today's legal text collections consist in many cases of laws, judgements, conditions, exceptions, negations, negation's negations and relations to others of this species. Nowadays, legal texts are often hard to read and to understand. If you are asking a lawyer about what is allowed and what is not allowed, you won't probably receive a short and explicit anwser. But rather something of that nature: "Yes, but in combination with ... and in that case ... then maybe ...!".

Other people from outside the field would probably think that the lawyer is not doing a good job or that he is trying to make things complexer than they really are to increase his revenues. But a closer look on current laws illustrates that the lawyer is telling the truth: In most cases a lawyer can't make a clear decision between yes or no, or between guilty and innocent. A good example of a law that is very difficult to read represents the "ChemSanktionsV". A part of this law can be found in listing 1.1 and offers a brief insight.

Listing 1.1: Excerpt Law ChemSanktionsV, §8 (see [Deu15a])

```
§ 8 Ordnungswidrigkeiten nach der Verordnung (EG) Nr. 689/2008

Ordnungswidrig im Sinne des § 26 Absatz 1 Nummer 11 Satzteil vor Satz 2
des Chemikaliengesetzes handelt, wer gegen die Verordnung (EG)
Nr. 689/2008 verstößt, indem er vorsätzlich oder fahrlässig

1.
entgegen Artikel 7 Absatz 2 Unterabsatz 1 Satz 1 oder Satz 2, jeweils in
Verbindung mit Satz 3, jeweils auch in Verbindung mit Artikel 7 Absatz 4
Satz 1 oder Satz 2 oder Artikel 14 Absatz 1, die bezeichnete nationale
Behörde über die Ausfuhr einer Chemikalie oder eines Artikels nicht, nicht
richtig, nicht vollständig oder nicht rechtzeitig unterrichtet,

2.
entgegen Artikel 9 Absatz 1 Satz 1 erster, zweiter oder dritter Gedanken-
strich, jeweils in Verbindung mit Satz 2, 3 oder Satz 4, eine Information
über einen dort genannten Stoff, eine dort genannte Zubereitung oder einen
dort genannten Artikel nicht, nicht richtig, nicht vollständig oder nicht
rechtzeitig gibt,

3.
entgegen Artikel 9 Absatz 2 oder Artikel 10 Absatz 4 Unterabsatz 2 eine
dort genannte Information nicht, nicht richtig, nicht vollständig oder
nicht rechtzeitig zur Verfügung stellt,

[...]
```

## 1.1 Text analysis

However, the correct interpretation of legal texts is a challenging task for both, legislative authority and judicial power. Today, two possible ways exist to get rid of this drawback: The first possibility is to make the legal texts better to read and to understand. This can be done by going back decades or hundreds of years in the history of jurisprudence. But it seems to be obvious that this possibility certainly would not be the best way because nobody would favour outdated and terrifying legal texts like the "Code of Ur-Nammu" to be the hub of the jurisprudence. A better way would probably be to remain the originally legal texts and to use modern techniques, that support the reader by understanding the text's content. This may be effected by using high-class text analysis.

While languages did not change rapidly over the last years, possibilities to analyse them have rapidly evolved. High-class text analysis frameworks and tools allow to produce machines which are able to understand the text's content. Now, a machine that uses such tools is not only able to import texts and present them to the machine's user, it also interprets the content and presents it prepared and in a well understandable way to the machine's user.

High-class text analysis frameworks and tools should be efficient and correct (most important feature but often ignored) to work effectively and must be build for a specific domain. Specifically that means, that a high-class text analysis framework for Domain A will not work well for Domain B and that such a framework has to be tailored for its specific usage. However, this does not preclude the possibility to learn from each other. This feature of high-class text analysis frameworks is probably very important and supported their success over the last years.

In summary, we know now that

- high interest in understanding complex legal texts, like laws or judgments does exist,
- high-class text-analysis frameworks and tools that support interpreting and understanding the text's content do exist and
- each tool should be tailored for one specific domain.

The central question, which an attentive reader does ask subsequently, is the following: "Do text-analysis frameworks for legal texts exist?". To answer this question, we will take a closer look at the current market situation. This shows that there is actually no tailored text-analysis framework for legal texts, especially if German legal texts should be analyzed. For legal documents, written in American English, there is something that indicates small parts. The paper "Measuring the Complexity of the Law" of the Michigan States University was written in 2013 talking about the complexity of American laws and how it can be measured inter alia by text-analysis (see [DMK13]).

This unresolved gap should be closed. To achieve this aim, the project "LexAlyze" has been started in cooperation with the Technical University of Munich and the Ludwig Maximilian University of Munich in 2014. This interdisciplinary project involves legal sciences on one side and the informatics sciences on the other side [SEB15]. Both sciences research hand in hand to push the development in this area. A key part of this Master's Thesis deals with the development of a simple web application for managing semantically annotated texts. Semantic text annotation represents an essential and important part of text-analysis. The focus of this work is on raw texts of the legal domain that should be annotated.

Against this background, here are the central research questions which causes this Master's Thesis:

1. What kind of legal texts exist in the German legislation?
2. What is a way to implement a generic importer for legal texts that can easily be adapted?
3. What kind of semantic text annotations exist and what are benefits and drawbacks of those?
4. How to persist semantic text annotations in order to access them for further semantic processing?

The first research question concerns the research in the German legislation. Thereby, the result of this research will be presented as a descriptive list of legal text types.
The result of the first research question will be a significant element for the work on the next research question: The design and implementation of a generic importer. As the content needs to be analyzed within the web application, there has to be an interface to put content into it. Almost all contents already exist in digital manner and are available in different forms and characteristics. Therefore, a generic importer should be able to deal with all types of legal texts and prepare them for further text analysis.

Every research question of this Master's Thesis is build upon another. The third question concerns linguistic and semantic text annotations. After chapter four should be clear what kind of text annotations do exist and what the benefits and drawbacks of those are. As this Master's Thesis is named after this question this will be represented as a central issue and there will be a special focus on it.

High class text analysis is computationally intensive, each calculation needs some time and it would be regrettable if the results would get lost. This is the reason, why the last research question will deal with this issue.

# 2 Basic knowledge about German legislation

The chapter Introduction (see chapter 1) had briefly described that it is necessary to tailor a text analysis framework to a specific domain. This is an important element to get such a framework working and reach best results using it. To tailor such a framework, a fundamental knowledge about the domain must be build before start. As our domain is the German legislation, we will take a deeper look on it. This chapter should figure what kind of legal documents exists in the German legislation. Furthermore, the main characteristics of these should be presented in detail.

Legal documents are documents that persists determinations and decisions in the jurisdiction. All legal documents are foundation for judgements. A judge working in a democracy and constitutional state will use legal documents for research before passing a judgement. If there is something what is not written to legal documents and there is substantial public interest, the state will create new legal documents to help judges doing their work.

The legal documents in the German legislation can be divided in various types, namely:

- Legislative documents
- Judiciary documents
- Literature

## 2.1 Legislative documents

The making of law is a task performed by the parliaments of the Federal Republic of Germany. Therefore the most important organ of the legislative is the German Bundestag. In cooperation with the German Federation, the Bundestag decides in a legislative process on all laws (see [Bun15b], [SL10]). More precisely speaking, this means that the legislature has the power to enact, amend and to repeal public policy.

There are mainly two types of legislative documents in the German legislation:

- Laws
- Delegations

Laws are passed or changed by the parliamentary legislator. This procedure is set down in the German Constitution. Delegations are not passed or changed by the parliamentary legislator, but by the German Government. This is the most important difference between law and delegation. The authorization to pass of change delegations are given to the German Government by the parliamentary legislator. This authorization is not a “free ticket” for everything because it is strictly restricted to specific areas.

Because not only national laws exist in Germany, but also a lot of specific laws that exist in the 16 German states, it is hard to count all laws. To get an idea how much different laws and delegations exist, it is possible to take a look at the number of laws and delegations of the German Federal Republic: In the year 2009 there were 1.924 valid and existing laws and 3.440 valid and existing Delegations. These laws and delegations contained over 76.382 articles and paragraphs (see [Gmb10]). As mentioned, this numbers are only the amount of the laws and delegations existing in the German Federal Republic excluding all legislative documents existing int he 16 German states.

### 2.1.1 Legislative document structure

The basic structure of legislative documents, whether they are a law or a delegation, resemble each other. Each legislative document has an unique title and an unique short-title. These attributes are only unique within Germany but not unique abroad it. An example for the title is "Strafgesetzbuch", the short-title is "StGB". Other attributes are date-values like the promulgation-date and the last-edit-date, and an introduction or abstract about describing the focus and area of it. An example can be seen in listing 2.1.

A legislative document's first separation is the separation in several sections. Each section contains other sections and/or articles. The one and only attribute of a section is its title. Articles are the centerpiece of legislative documents. Articles contain the "real" content of the documents and contain the information about it. The articles themselves are not always easy to read and ofter not understandable, but the statement of these should be always the same: articles describe two things: rules and consequences. In each article a rule is set, which gives information about what is allowed and what is permitted in a specific situation. But just setting a rule is not enough, consequences must be defined. Each article contains information about what will happen when violating the rule. The construction of an article shows many similarities with the rules set in the "Code of Ur-Nammu" (see chapter 1).

Listing 2.1: Excerpt Strafgesetzbuch (StGB), (see [Deu15b])

```
STRAFGESETZBUCH (StGB)

Ausfertigungsdatum: 15.05.1871
Neugefasst: 13.11.1998, Zuletzt geändert: 21.01.2015

1. Abschnitt: Das Strafgesetz

Erster Titel: Geltungsbereich

§1 Keine Strafe ohne Gesetz: Eine Tat kann nur bestraft werden, wenn
die Strafbarkeit gesetzlich bestimmt war, bevor die Tat begangen wurde.

§2 Zeitliche Geltung: (1) Eine Tat kann nur bestraft werden, wenn die
```

```
Strafbarkeit gesetzlich bestimmt war, bevor die Tat begangen wurde. (2) Wird
die Strafdrohung während der Begehung der Tat geändert, so ist
das Gesetz anzuwenden, das bei Beendigung der Tat gilt. [...]
```

To summarise, the two kinds of legislative document in the German legislation are law and delegation. A legislative document itself has attributes describing it. Unique attributes are the title and the short-title. Each legislative document contains sections which can contain other sections (subsections) or articles. That's how legislative documents looks in a "real" ("perfect") world, which is in our context the German legislation. Further work is necessary to use this knowledge for the implementation of an application which should work with it: software modeling.

There are many explanations for the term "software modeling". A very intuitive explanation for "software modeling" is that it is an "abstraction" of the real world (see [Wan08]). This means, that a software modeler tries to simplify areas, situations or topics so that it is possible to generate a model. This model can be both, as detailed or as simplified as possible. It depends on the field of application or the recipient (reader) of the model. While the one readers are happy with a very simple entity-relationship-model without any attributes in the entities, the others needs more detailed information in each entity. This is exactly what will be done now in this Master's Thesis. A look at illustration 2.1 shows how a law and a delegation are modeled with the facts and basic conditions the attentive reader know yet.

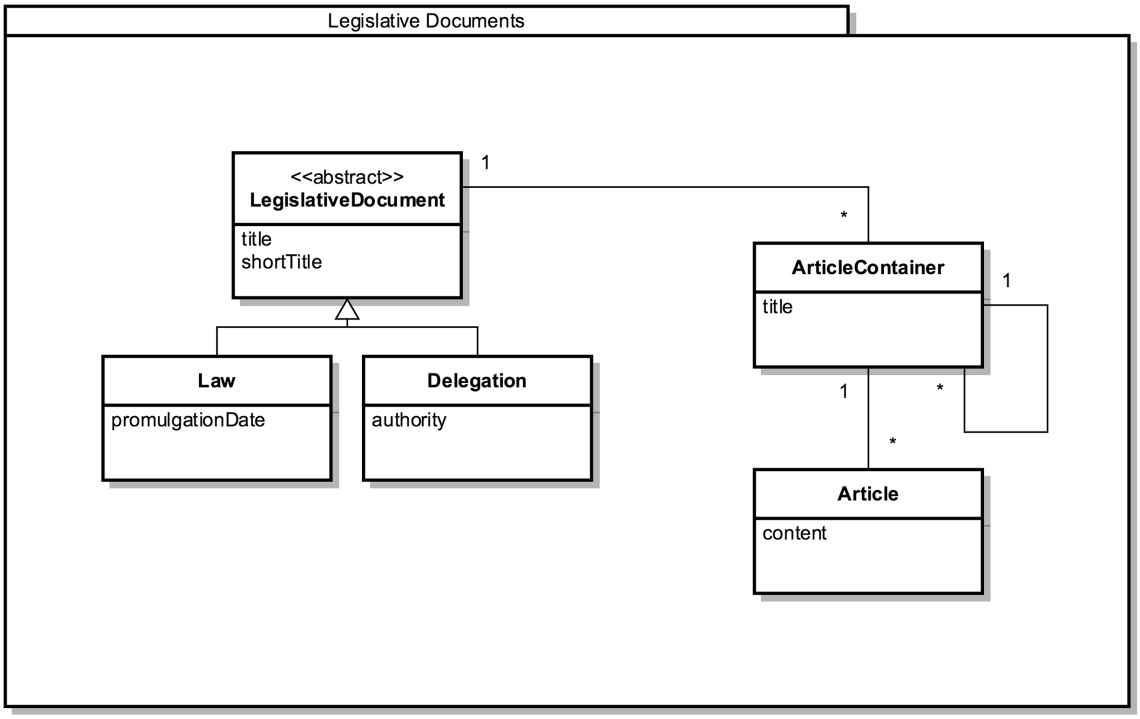



Figure 2.1: Entity-relationship-diagram: LegislativeDocument

We use single inheritance to model our legislative documents. As known, law and delegation are both legislative documents in real world. Therefore, we inherit the classes Law and Delegation as subclasses from the superclass LegislativeDocument. Because a title and a short-title are used by law and delegation in exact the same context, these attributes will be provided by the superclass LegislativeDocument. Law and Delegation have also additional variables like the promulgationDate in case of the class Law and authority in case of the class Delegation. Authority gives information about the source of the delegation and is not provided by law, promulgationDate gives information about the date a law was first promulgated. A delegation has no need for a promulgation date.

Copied from "real world", our model is taking shape. As next part we take a look on the nested section and subsection-construct. Each legislative document contains sections, which are able to contain other sections (subsections). There is no limit on the deepness in this nested construct, this means that it is theoretically possible that a section contains a subsection, a subsection contains a sub-subsection, and so forth. In our entity-relationship-model we are now introducing the entity ArticleContainer. We renamed it from section/subsection/... to ArticleContainer because we do not want to restrict it too much to the term "section" and leave a margin for other developments.

A LegalDocument contains many ArticleContainers, therefore there is an one-to-many relationship between the LegalDocument and the ArticleContainer. Because each section should be able to contain subsections, subsections sub-subsections and so forth, an ArticleContainer has an one-to-many relationship to itself. With this construct, it is easily possible to rebuild the real world's scenario. As attribute each ArticleContainer has a title.

To finalize the model of a legislative document, only one but important entity is missing: the Article. The Article contains just content and is related to an ArticleContainer. Each ArticleContainer can contain many Articles. Therefore, it is related with an one-to-many relationship.

After modeling, it is important to validate the result. One common way to proof the correctness is to rebuild the "real world" with the abstracted model: Let's try it with the "Strafgesetzbuch (StGB)" (see listing 2.1). The illustration 2.2 shows an object-diagramm, populated with the data of the "Strafgesetzbuch (StGB)". As it can be observed, the main object is of type Law, it's super-class is LegislativeDocument and its values (title, short-title and date values) are set. The object has a relation to "1. Abschnitt: Strafgesetz" (type ArticleContainer). The "1. Abschnitt: Strafgesetz" is the parent object of "Erster Titel: Geltungsbereich" (as well of type ArticleContainer). Finally, this object contains the two articles and their content. The last check to validate a model's correctness is to compare the "real" world with its reflexion in the model. If there are no more or no less parts, the abstracted model should be correct. Only one further remark: Depending how detailed the model is, it is possible that some parts are missing, which are not needed.

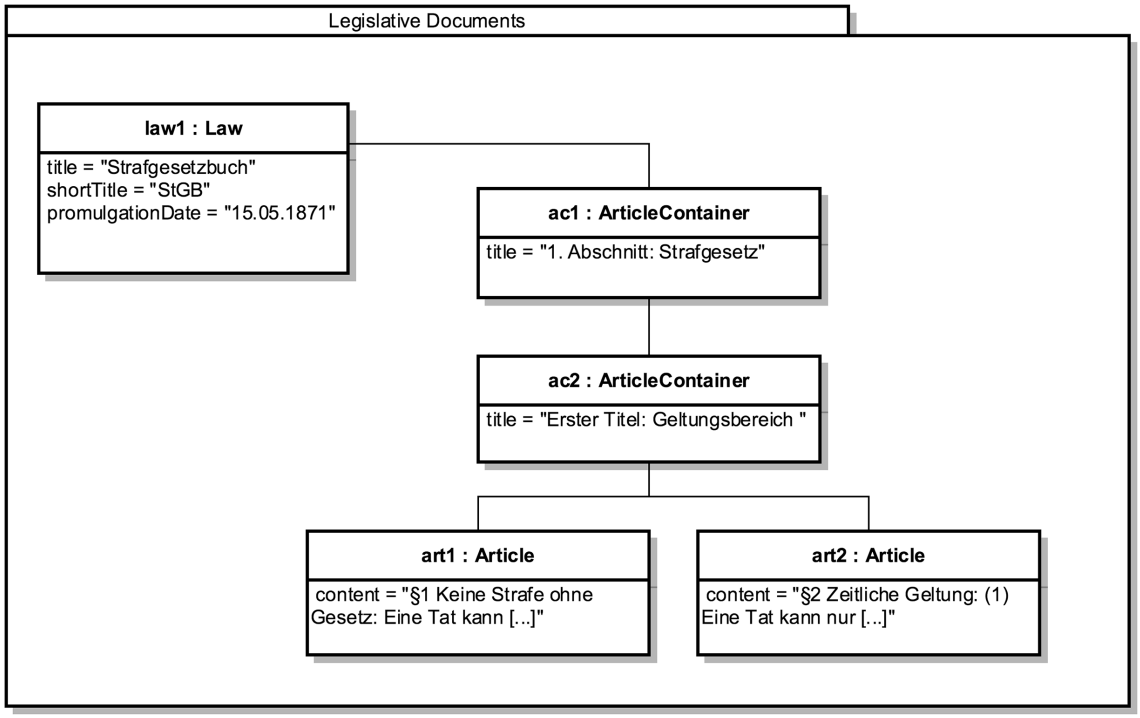



Figure 2.2: Object-diagram: Strafgesetzbuch (StGB), Law

## 2.2 Judiciary documents

Judicial authorities exists since the dawn of human civilization. While the emperors were the only responsible people which had the right to dispense justice, the responsibility for justice changed through the years continuing till today. The increasing realization of the principle of the separation of powers made it possible, that in most countries, especially in European countries, justice and government are strictly separated (see [Wik15a]). In Germany all courts are either federal courts (in German "Bundesgerichte") or state courts (in German "Landesgerichte"). Federal courts are the highest level of courts in the German justice and only eight of them exist: Bundesverfassungsgericht (BVerfG), Bundesgerichtshof (BGH), Bundesarbeitsgericht (BAG) Bundesfinanzhof (BFH), Bundessozialgericht (BSG), Bundesverwaltugnsgericht (BVerwG), Bundespatentgericht (BPatG) and two Truppendienstgerichte (TDG Nord, TDG Süd). Only a small part of judgments are given in the highest level, most of all judgements are given in the state courts. There are 89 state courts in Germany, whereby the court's responsibility depends on the case (see [dJufV15]). Every case starts at the lowest level. Only if there is an appeal, it is possible that a higher authority like the federal courts rescan the judgement and confirm or withdraw it.

In the previous section, we talked about the outcome of the legislation. As explained, law and delegation are the most important kinds of legal documents in the legislation. As can be guessed, the legislation is not the only area with special legal documents. The equivalent to law and delegation in the judiciary are

- Judgements
- Decisions

Judgements are the result of a court case. It is created by a judge and contains the complete outcome of a proceeding. There is no court case without a judgment in Germany and hopefully in most countries on earth. In the middle of each court case is one legal entity which is accused. In Germany a legal entity can be a human, a club or a joint-stock company, a cooperative or something similar, but it is necessary that one or more human beings are concealed behind. Roughly explained, a judgement can have three outcomes: a positive, a negative or a partly positive/negative (from the accused's point of view). Positive means that the accused granted a full discharge by the court, negative means that the accused is guilty. The last outcome can be that the accused is not in all points guilty, which can be seen positive or negative, depending whichever way the accused looks at it.

In case that the accused is guilty or partly guilty, this person gets a penalty. A penalty in Germany can have many facets: The spectrum ranges from admonishments to life sentence. What penalty an accused gets depends not only on the crime but also on the accused himself. A "good" accused, which never did a crime before, often gets a much softer penalty than an other. This penalty will be a decisive part of the legal document judgement.

A decision is not the result of a court case. To make a decision, no lawsuit is needed. The decision-maker is ever a judge or a group of judges. When making a decision, the decision-maker judges based on currently known facts only. Only if highly relevant and the judge is not able to make a decision, a court case will be created. For that, a result of the decision can be to make a court case.

Summarized, both kinds of legal document judgement and decision are written by a judge. They are both the result of a decision-making process. The first difference between both kind is that the judgement is the result of a court case while the decision is the result of judge and a decision-making process without an court case. It is often the case that the decision leads to a court case, in which the result is a judgement. The second difference between a judgement and a decision is that a judgement contains a penalty if the judgement is negative (from the accused's point of view).

### 2.2.1 Judiciary document structure

In the parent section it had been elaborated that the legal document kinds judgement and decisions are very similar. This similarity can not only be seen by the intent and purpose of both. The similarity also reflects in the basic structure of both legal document kinds. The first similarity will probably cause a little confusion: There is no default format for those kinds of legal documents. The situation reminds a little on the wild west. A short research in the German court's databases shows, that nearly every judgement and every decision has a different shape.

What are the reasons for this? The central reason for this is certainly that there are a huge amount of different courts in Germany. As mentioned there are totally eight federal and 89 state courts (in 2015, see [dJufV15]). There is no central template, each court is able

to make his own and it seems that they make use of this with pleasure. Within one court, the decisions and judgement often have a default form, but not always a similar structure. Each judge is able to explain in his own words the basis and thought processes when making a decision or a judgement. Furthermore, the court cases differ greatly from each other. While in some court cases one single page is more than enough to explain the court case, the judgment and the penalty (if the accused is guilty), there are bigger court cases where hundreds of pages are needed to explain everything related to the court case.

What can be done to find a structure? As explained there is no default structure when looking detailed on the different kinds of legal documents judgment and decision. To find nevertheless a structure it is necessary to go one or more steps further back and take a look from another, more abstract point of view. Each judiciary legal document starts with a heading: "Urteil" (in English "judgement") or "Beschluss" (in English "decision"). Apart from that, this is the first decision support when a reader of a judiciary legal document wants to know if he reads a judgement or a decision.

The next important information of the legal documents judgement or decision are the subject and the date information: which legal entity is the accused and what is the date of this judgement? Furthermore, the name and the address of the responsible court is immediately evident. This should be the most important meta information about this kinds of legal documents. But the most important part is still missing: the content. Each judgement or decision is separated in a various number of paragraphs. It seems that most of the courts try to write their documents with patterns. A often used pattern is to write first about the penalty, secondly about offence and finally about the reasons or a conclusion of the whole judgement or decision. That will become apparent when taking a look at the excerpt Urteil (BGH) (see 2.2). The "Bundesgerichtshof" have handed down a judgement. What is this judgement about is not relevant, but in this listing it gets a bit clearer how such a judgement look like.

Listing 2.2: Excerpt Urteil (BGH), (see [Bun06])

```
BUNDESGERICHTSHOF
IM NAMEN DES VOLKES

URTEIL
VI ZR 259/05
Verkündet: 21. November 2006

im Rechtsstreit: XYZ

Zur Frage, unter welchen Voraussetzungen in der Meldung einer Presseagentur
unter namentlicher Benennung des Betroffenen über dessen Abberufung als
Geschäftsführer wegen nachhaltiger Störung des Vertrauensverhältnisses mit
einem Großteil der Mitarbeiter berichtet werden darf.

Der VI. Zivilsenat des Bundesgerichtshofs hat auf die mündliche Verhandlung
vom 21. November 2006 durch die Vizepräsidentin Dr. Müller und die Richter
```

Dr. Greiner, Wellner, Pauge und Stöhr
für Recht erkannt:
Auf die Revision der Beklagten wird das Urteil des 10. Zivilsenats des Kammergerichts vom 7. November 2005 aufgehoben. Die Berufung des Klägers gegen das Urteil des Landgerichts Berlin vom 17. August 2004 wird zurückgewiesen.
Der Kläger trägt die Kosten des Rechtsstreits.
Von Rechts wegen

Tatbestand:
1. Der Kläger, der seit dem 10. Juli 2000 Geschäftsführer der Klinikum N. GmbH war, die drei Krankenhäuser in Brandenburg mit ca. 900 Mitarbeitern betreibt, verlangt von der beklagten Presseagentur Unterlassung einer identifizierenden Berichterstattung unter Nennung seines Namens über die Tatsache und die Umstände seiner Abberufung im Juni 2002.

2. [...]

Entscheidungsgründe:
Das Berufungsgericht hat einen Unterlassungsanspruch des Klägers analog § 1004 Abs. 1 Satz 2 BGB i.V.m. § 823 Abs. 1, Abs. 2 BGB, Art. 1 Abs. 1, 2 Abs. 1 GG im Sinne des Klagebegehrens als begründet erachtet, weil die angegriffene Agenturmeldung, mit der die Beklagte unter Nennung des Namens des Klägers über dessen Abberufung als Geschäftsführer der Klinikum N. GmbH im gesamten Raum Berlin-Brandenburg und damit überregional berichtet habe, den Kläger in seinem allgemeinen Persönlichkeitsrecht verletze [...]

Now it should be understandable how a judiciary legal document look like. As we did it in the section before, we try to abstract the "real" world and create a model to work with.

Like we did it with the legislative documents, we use single inheritance to model our jurisprudence documents. Judgements and decisions are both kinds of jurisprudence documents. First we create a superclass JurisprudenceDocument. The second step is to create two classes Judgement and Decision which are both inherit from the superclass JurisprudenceDocument. As next step we have to decide what kind of attributes are only relevant for judgements, what attributes are only relevant for decisions and what kind of attributes are relevant for both, judgements and decisions. The attributes in this intersection will be available in the superclass JurisprudenceDocument, the other attributes only in the class Judgement or respectively in the class Decision.

There are much attributes in the intersection and further in the superclass: The first one is an information about the court: what court is responsible for this legal document and who did the decision or judged it? The next attribute is the date of the document, which should be equivalent to the date of decision or date of judgement. This should be the most important attributes in our abstracted superclass JurisprudenceDocument. The next step is to take a look at the attributes of the class Judgment and Decision. Depending on the number of observed judgments or decisions, new possible attributes will appear again and

again. As we want to build an abstract model that fits best for our case, it is not necessary to go too much in detail and search for fancy attributes used by single courts. The only one attribute, which is provided in each judgement is the number of transaction. The number of judgement is an unique number like an ID. A decision has something similar, the number of decision. The central question should be: Why is it necessary to create this attribute in each class and do not create something like a transaction number in the superclass? This can be done and would probably be a valid way. But as a judgement is just similar but not equivalent to a decision, the number of judgement and the number of decision is not the same, it has just the same idea behind. So it is recommended to strictly separate them.

After classes and attributes for the most important meta-data are created, the whole content must be modeled. As explained, the content of a jurisprudence legal document do not have a fancy structure. It is just text, separated in paragraphs. There are now three possible ways to add the content to our structure:

- attribute content
- relation to paragraphs
- relation to a nested construct

The first one would be a valid way: Just add an attribute content to our superclass JurisprudenceDocument, which contains the whole content of the legal document. The advantage would be the easiness, but the big disadvantage top it: we will loose the structure, and not mentioned yet, relations between different legal document parts are not able to be build (one part of legal document A refers to an another part of legal document B or something similar).

The next valid way is to create an entity Paragraph and relate it with an one-to-many relation to the JurisprudenceDocument, that means that one JurisprudenceDocument contains many Paragraphs. It would be possible to model most of all judgements and decisions with this construct, as long as they have no nested structure. Thinking big, there are as well legal documents that only have a few pages, but there will be also legal documents with hundreds of pages and much more content. In such legal documents, nested structures (section/subsection/ and so forth) will probably occur. To take this into account, the third valid way would be the best: relation to a nested construct.

Because reinventing the wheel should never be intent and purpose, we copy our nested structure from the model of the other legal document's model. So we introduce the class ArticleContainer with an one-to-many relation to our JurisprudenceDocument and the class Article with an one-to-many relation to ArticleContainer. Furthermore, we add an relation from one ArticleContainer to many ArticleContainers. A nested structure is now available.

The whole entity-relationship-diagram can be seen in figure 2.3. To finalize our modeling process, the next step will be to make a short proof of concept. Does our model work with "real" world-data or is it too abstract? Let's take a look at the object-diagram in illustration 2.4. The excerpt of the BGH-judgement (see listing 2.2) is used for this proof of concept.

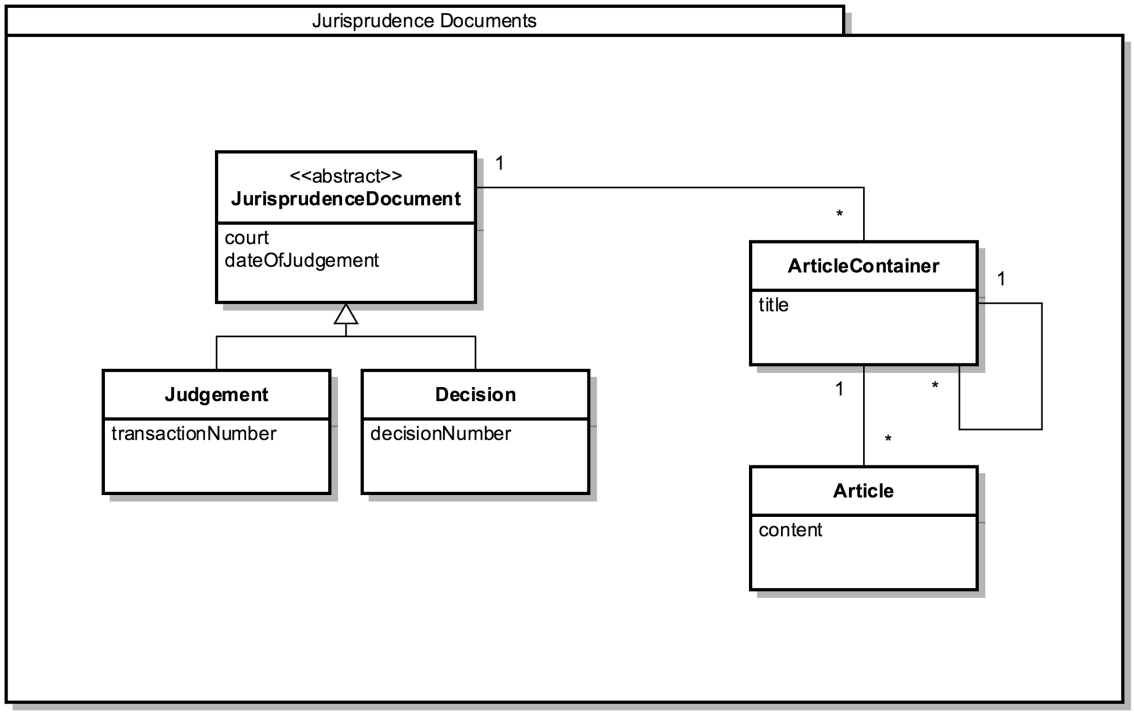



Figure 2.3: Entity-relationship-diagram: JurisprudenceDocument

As this is a judgement and not a decision, a new object of type judgement is created. Its attribute, the number of judgement is as well set (set to "VI ZR 259/05") as the inherited attributes of the superclass (for example: the courtName is set to "Bundesgerichtshof", the date is "21. November 2006", and so forth). The object is related to three objects of type ArticleContainer, which contain the articles of the example-listing.

It looks like nearly all parts of the content are still available in the proof of concept. So the reverse way should be also available, the "real" world document seems to be able to re-build out of the model's data, without loosing information about it. So the proof of concept should have the result that it is correct. The attentive reader of this Master's Thesis can create his own picture of it: just take any judgment or decision and proof it.

## 2.3 Literature

The last important legal document type is literature. It differs basically from legislative and jurisprudence documents. In most cases it is not written by an authority, but by normal authors. In highly simplified terms and in the context of this Master's Thesis, a literature is nothing else than a non-fiction book. The content of a literature is not telling a fictive story, but it is dealing with cases, facts, sticking points or something else of the legislative or jurisprudence.

There is not that one type of legal literature but several types exists. The most common

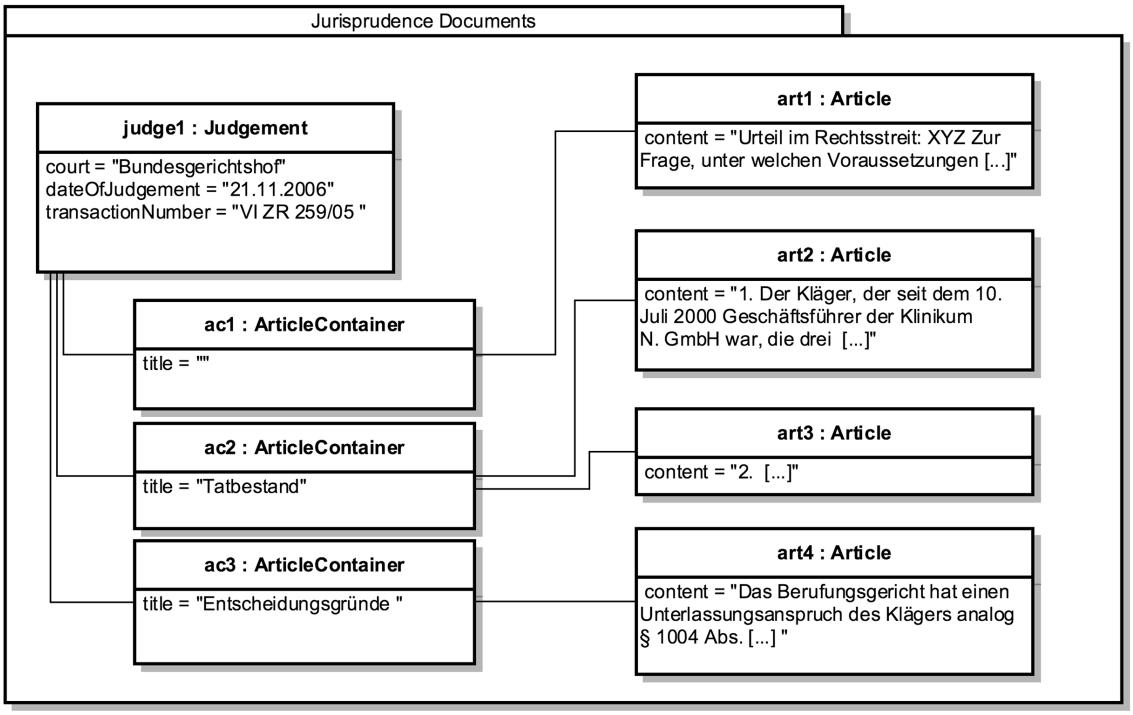



Figure 2.4: Object-diagram: Bundesgerichtshof Judgement

types of legal literature are textbooks and papers. It is not really countable how many of them exist worldwide, but it is interesting to get an idea. One possible way to get a number is to take a look in public libraries, the German National Library should work well for this short research. The German National Library is the central archival library of the Federal Republic of Germany. It archives all types of literature written in German language whether the literature was written in Germany or not. Looking for the German word “Recht” will return a list of 210.682 documents referencing this word (Query-Date: July 2015). This is just the number of legal documents literature in German language which is registered in the German National Library. The number of legal documents literature should be considerably greater.

### 2.3.1 Literature document structure

Like in the previous subsections of the other legal document types “legislative document” and “jurisprudence document” we try to abstract the “real” world to a model. To do this, it is necessary to take a closer look at the legal document type literature. As it is impossible to research all 210.682 documents of the German National Library, which are referenced to the term “Recht”, within the scope of this Master’s Thesis, we take a look at the structure of one single example. Listing 2.3 shows an excerpt of one of the most common German legal literature. The excerpt shows the title and parts of the table of contents.

What can be observed is that each legal literature, independent of its type has a title. In the listing the whole title is “ALLGEMEINE RECHTSLEHRE - Ein Lehrbuch”. The

title itself is part of the literature's meta-information containing information about the literature itself but not content. As well as a title, each type of legal literature has one or more authors. In case of the example, the authors "Dr. Klaus F. Röhl" and "Dr. Hans Christian Röhl" are represented with more information about their work (e.g. "em. o. Professer für Rechtssoziologie [...]"). In case of this example, this additional information is part of the meta-information, but this should not be the rule for all types of legal literature. In most cases the full name with the academic title as prefix or suffix should be enough.

If the legal literature is a publication, it gets its own International Standard Book Number ("ISBN"). As the name of this unique number suggests, that it is a world wide guilty and standardized number which is referenced to exact this one and only publication. If one literature is published with exact the same content as ebook and as printed edition, the same book would have two different ISBNs. Since 2007 an ISBN contains 13 digits, which can be divided in five parts (see [fdBD12]):

- prefix
- group-number
- publisher
- title number
- check sum

In the example below, the ISBN is "978 3 8006 4098 0" and the prefix is "978", which represents a "book", the group-number is "3" which means in case of the prefix that this literature is written in German language. The code "8006" is the unique number of the publisher "Verlag Franz Vahlen", the fourth part is the number of this title. It is unique within one publisher. Last but not least the check sum "0" fulfill the 13 digits of the ISBN. It is calculated out of the first 12 digits and helps software and hardware (e.g. ISBN-(barcode)-readers) prechecking the ISBN when dealing with it. The correct and complete composition of the ISBN is not part of this work, but such a short excursion seems to be interesting for the readers of this thesis. Maybe someone of the readers may remember it when holding a publication in their hands.

The last part of the meta-information is like the ISBN only available when it is published: the name and address information about the publisher. In the example of the listing 2.3, the publisher is "Verlag Franz Vahlen" and its address information is "Munich".

Now that we have looked at the literature's meta information, it is time to focus on the literature's content. The listing 2.3 shows the first part of table of content of the literature "ALLGEMEINE RECHTSLEHRE - Ein Lehrbuch". The first thing that immediately strikes the eye is the nested structure. The whole literature is separated into single chapters. In the example, there are four chapters: two pre-chapters "Vorwort", "Literatur und Abkürzungen", and two content-chapters "1. Kapitel Einleitung", "2. Kapitel Begriffe vom Recht und Begriffe im Recht". For the purpose of this Master's Thesis, there will be no differences between pre-chapters and content-chapters, because it doesn't matter if the chapter contains seemingly "irrelevant" content. But what matters is that the chapters can contain

subchapters and the subchapters themselves can be parent of another subchapters, and so forth. It is a nested structure.

Even if it is not clearly visible in the listing, but it should be clear that the example is just the table of contents. The chapters do not contain only subchapters, they do also contain content (what should be clear for all people which had read a book in their life). A Legal literature is not really different to an ordinary literary like a roman or any other kind of books. But it is important to introduce this structure to this work because there are big differences to the other introduced types of legal documents.

Listing 2.3: Excerpt Allgemeine Rechtslehre - Ein Lehrbuch von Prof. Dr. Klaus Friedrich Röhl, Hans Christian Röhl, (see [DKFR08])

```
ALLGEMEINE RECHTSLEHRE
Ein Lehrbuch

von
Dr. Klaus F. Röhl
em. o. Professor für Rechtssoziologie und Rechtsphilosophie
an der Ruhr-Universität Bochum

und

Dr. Hans Christian Röhl
o. Professor für Staats- und Verwaltungsrecht, Europarecht und
Rechtsvergleichung an der Universität Konstanz

3., neu bearbeitete Auflage

Verlag Franz Vahlen München 2008
ISBN 978 3 8006 4098 0

[...]

INHALT

Vorwort
Literatur und Abkürzungen
1. Kapitel Einleitung
 § 1 Gegenstand, Ziel und Methode der Allgemeinen Rechtslehre
  I. Aufgaben der Allgemeinen Rechtslehre
  II. Von der Rechtsphilosophie zur Rechtstheorie
  III. Von der Rechtstheorie zur Allgemeinen Rechtslehre
  IV. Gegenstand und Methoden der Allgemeinen Rechtslehre
  V. Theorien hinter der Theorie
```

```
  VI. Ockham's Razor
2. Kapitel Begriffe vom Recht und Begriffe im Recht
 § 2 Recht als Kommunikation
  I. Der Begriff des Rechts als Definitionsproblem
  II. Sprache und Medien
  III. Vom Linguistic Turn zum Pictorial Turn
  IV. Funktionen von Text- und Bildkommunikation
  V. Spekulative Prognosen

[...]
```

Now that we know how a legal literature look like, it is time to abstract the "real" world to a model.

It does not harm to remind ourselves how we did the abstraction in past creating a model for the other types of legal documents: The legislative and the judiciary documents. When creating the model for those two types we had in both cases a special case: Both, the legislative document and the judiciary document are available as different kinds. As a reminder: there are two legislative document kinds, the law and the delegation and there are two judiciary documents, the judgement and the decision. When creating the model for those legal document types we decided in both cases to use single inheritance to model this situation. For both types we created a super-class ("LegislativeDocument" and "JudiciaryDocument") and created subclasses for each kind which inherit from the super-classes (Law and Delegation inherit from super-class LegislativeDocument, Judgement and Decision inherit from JudiciaryDocument). With this design we reached an optimum solution and best abstraction from the real world.

This is how it is solved for the other types of legal documents. One possible way to create a model for the legal document type literature is to copy, paste and modify the already created model so that it fits with literature instead of a legislative or judiciary document. If we want to do it like that, we will also use single inheritance. Therefore we first have to create a superclass "LiteratureDocument" and second we create subclasses that inherit from the superclass. One subclass could be "Book", an another subclass "Paper".

The big benefit of this kind of model-creation is, that we have an equality to the already created models of the legislative and judiciary documents. Already known patterns and know-how can be reused. So far so good, but the reality is a little bit different. We would not try to count benefits if there were no existing drawbacks. The single inheritance is a great possibility to describe an "is-a"-relationship. For sure, a "book" is a literature, as well as a "paper" or something similar. But the question is if it is necessary to know this. The best abstraction of the "real" world is not the most detailed. On the contrary: the best abstraction is the most simplified model. Most simplified means that it is as detailed enough that we can work with it in our purpose. If we are the manager of a huge library, it would be important to us to know if a literature is a book, a paper or something else. But in our context, this is useless information. The model would be a bit overengineered.

A more simplified version can be to create a class “Literature”. The class “Literature” is the container for all meta-information. Therefore, we add variables for the meta-information we found in our example listing 2.3: title, author, publisher and ISBN should the most important of all meta-information for further processing.

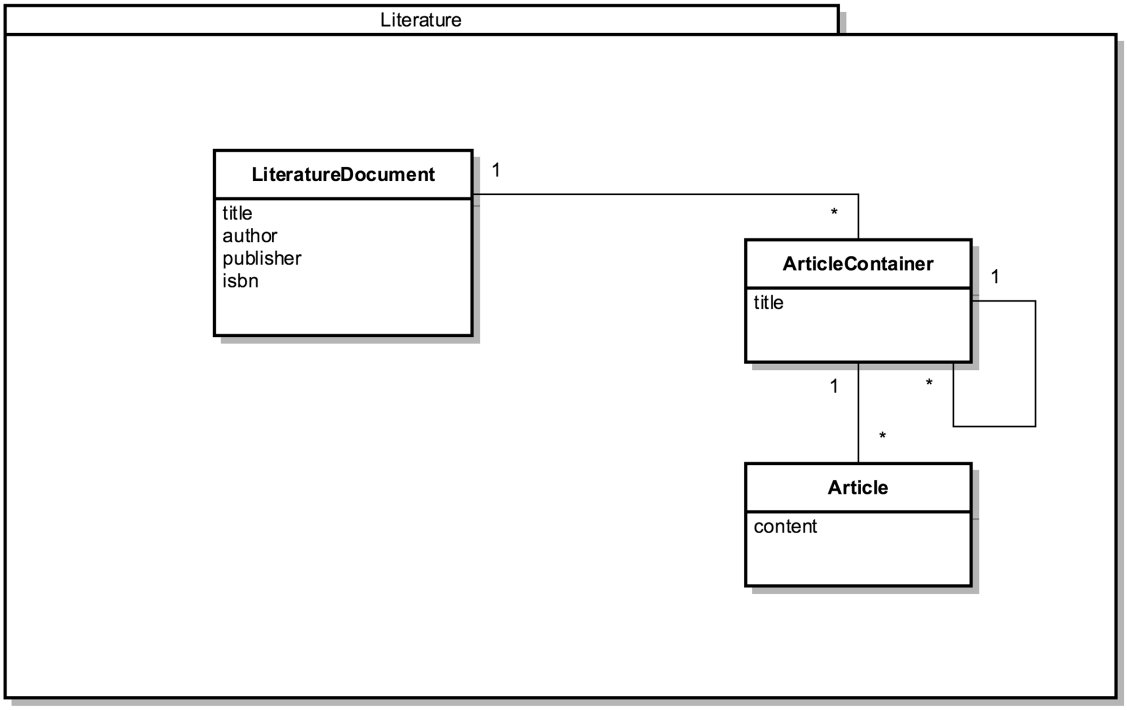



Figure 2.5: Entity-relationship-diagram: LiteratureDocument

The next step in the abstraction and model-creation process is to model the nested content. Once again, we can make use of existing knowledge. The nested structure of the legal document type literate consists of chapters, subchapters, subsubchapters and so forth. Each chapter contains a list of subchapters and content. Each subchapter is a chapter again. The same construct can be found in laws as well as in delegations. To reproduce the real world we created an object “ArticleContainer” and created an one-to-many relationship between the one LegalDocument and many ArticleContainers. To reproduce textual content we created an object “Article” and created an one-to-many relationship betweeen one ArticleContainer and many Articles. To finalize the model and to create a nested structure, one relationship is left: the one-to-many relationship between an ArticleContainer and itself. As mentioned, this structure should be the same, so we reuse this construct but replace the first relationship between the LegalDocument and the ArticleContainers with an one-to-many relationship between the new object Literature and the ArticleContainer.

The entity-relationship-diagram (see 2.5) shows the result of our modeling process. We finalize our modeling process by validating it. To perform the validation process, we choose our example of listing 2.3 and try to create it as an instance of our model. The object-diagram (see 2.6) shows how the example literature “Allgemeine Rechtslehre - Ein Lehrbuch” fits on our model. On top, an instance of the object “Literature” has been created. Its title is the title of the example “Allgemeine Rechtslehre - Ein Lehrbuch”. All other attributes are set too, the author is set to “Dr. Klaus F. Röhl, Dr. Hans Christian Röhl”, the attribute isbn has the value “978 3 8006 4098 0” and the publisher is “Verlag Franz Vahlen München”. At this point it can be observed what abstraction means and what effects simplifying has. The “real” world example literature provides more information about the authors (background information about their work and their motivation writing this literature). After the abstraction process, only their names (and academic titles) survived. For this Master’s Thesis we probably need to know the names of the authors, so the names should be enough for our purpose.

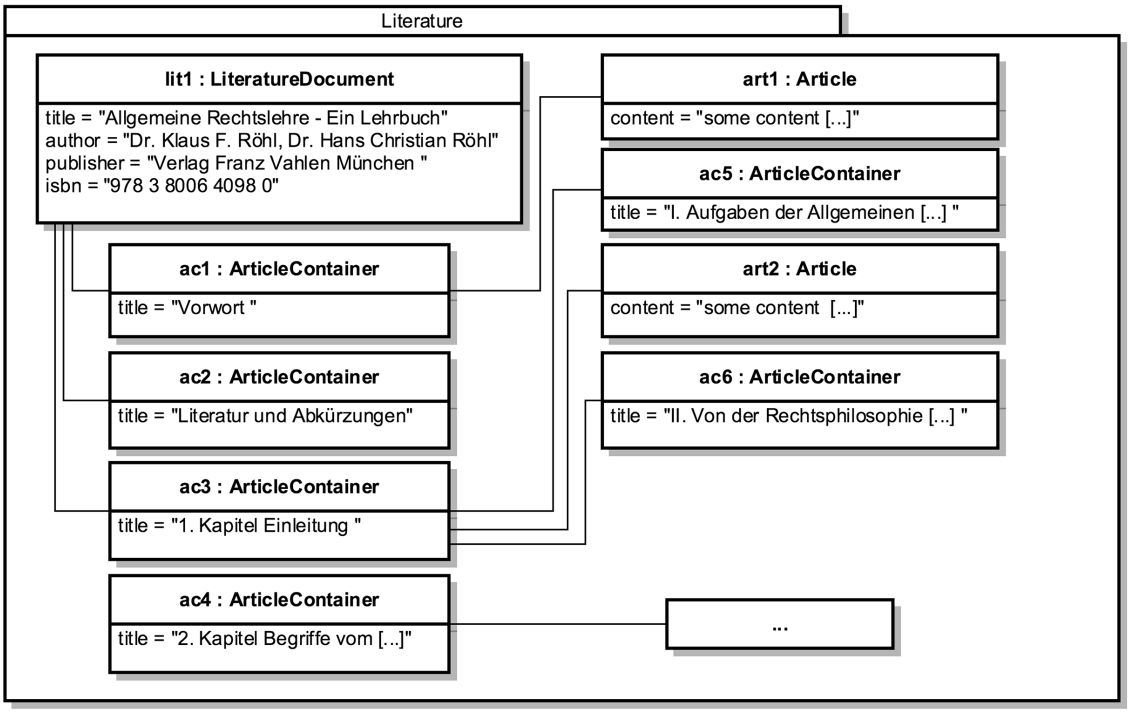



Figure 2.6: Object-diagram: Allgemeine Rechtslehre - Ein Lehrbuch : Literature

Validating a concept is as or even more important than finding a concept. It is important to remain realistic: in this example and this special case it was clear that only the name of the authors remain and the attention was probably a little bit exceeded. But this was just a more conspicuous case. When walking threw the modelling process there are stumbling blocks, and some of them are not visible at the beginning. Such stumbling blocks can occur

when dealing with different data types or something similar. It is important to pay special attention and doing a validation process.

Exactly this validation process will be now continued. After all important meta-information are now available, only the content is missing in our validation process. As it can be observed in diagram 2.6 there are four instances of ArticleContainer that are in a direct relationship to the instance of the literature. This are the chapters (pre-chapter and content-chapter) which are on the first level, the root level. Each of the ArticleContainers contain the chapter's title. The ArticleContainer with the titles "1. Kapitel Einleitung" and "2. Kapitel Begriffe vom Recht und Begriffe im Recht" are the parent of other ArticleContainers, that represents that they are the parent chapter of other chapters. Furthermore, each ArticleContainer has a relationship to instances of the object Article. Each instance of the object Article is a part of textual content, like a section or one or more paragraphs within a chapter. As this is just textual content and everyone of the Master's Thesis readers should be able to imagine how a literature's content look like, the diagram shows not real content of the example literature but a remark "some content".

This is the end of the validation of the literature. We demonstrated that the model works with real content and with this abstract and simplified model this should work not only with the example but also with every literature else.

## 2.4 Aggregation of knowledge

Summing up the last sections it can be said that we researched the most important types and kinds of legal documents in the German legislation. The most important types are

- Legislative document
- Judiciary document
- Literature

Legislative documents can be roughly divided in two different kinds, law and decision, as well as judiciary documents, which can also be divided in two different kinds, judgment and decision. Literature is literature and will not be divided within the context of this Master's Thesis.

Due to the introduction and explanation of the different legal document types and its kinds we created for each type an entity-relationship-diagram which is in every case a simplified abstraction of the real world. The result of this processes can be found in the diagrams 2.1 (legislative documents), 2.3 (judiciary documents) and 2.5 (literature). While modelling we tried to prevent re-engineer the wheel again and again. This become visible when taking a look at the relation of the types to their content. In each of the models the same nested construction for the ArticleContainer and Articles can be found. Here, the special attention will start to pay off.

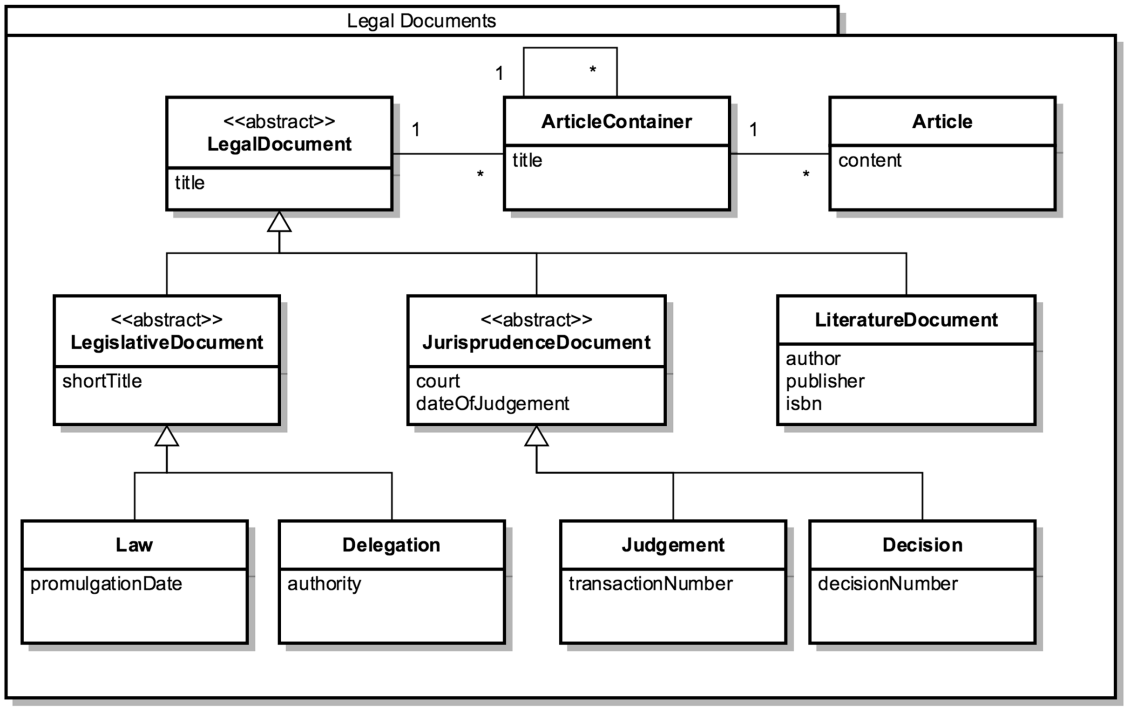



Figure 2.7: Entity-relationship-diagram: legal document

It is now time to fit it all together. What is the right way to do this? We will use a well-known principle and create a new entity-relationship model. The first entity what will be placed on it is a super-class which should allow other entities to demonstrate their togetherness. The entity LegislativeDocument as well as the entity JudiciaryDocument and the Literature should fit together. The new super-class will be named to its common denominator: "LegalDocument". LegislativeDocument, JudiciaryDocument and Literature are extending the super-class "LegalDocument". Because the entities had attributes what are now re-engineered in each object again and again, the attributes which are in an intersection and which have the same function will climb up a ladder to the new object and removed in the subclasses. The attribute, which is in an intersection is title.

As the ArticleContainer/Articles-construct is the same in each model, we will reuse it one time and remove all other occurrences. The relationship from one LegislativeDocument, one JudiciaryDocument or one Literature to the many ArticleContainers will be now created from the new object LegalDocument to the ArticleContainer, so that there is an one-to-many relationship again.

That's all for now. The result of the aggregation of all models can be observed in the new entity-relationship-diagram 2.7. Compared to the diagrams 2.1 (legislative documents), 2.3 (judiciary documents) and 2.5 (literature) every attribute and relation should be available again and no parts are missing right now. It should be clear that the model should be the same and should not has a different behavior. Because no important part had been

changed, we will forgo validating this model again. If some readers want to validate it, it is up to each themselves, but the validation of the single parts of the model in the previous sections should be enough validation.

This is now the end of this chapter. What should the attentive reader of this Master's Thesis takeaway? First it should be clear what different types of legal documents exist in the German legislation and what is their main purpose. Secondly, he should be able to delimit the different types of legal documents and their different purpose. Thirdly the simplified and abstracted structure of each of this types should be visible and it should be clear how the aggregated entity-relationship-model (see 2.7) has been formed. This model will be important to follow the further explanations and elaborations of the next chapters.

# 3 Importing and representation of data

The previous chapter "Basic Knowledge about the German legislation" (see chapter 2) had described in detail how the German legislation is constituted and what are the German legislation's jobs to be done. To fulfil the requirements of the German legislation's jobs, several types of legal documents have been developed in the past years, decades and centuries. In the previous chapter we analyzed the most important types of legal documents and their different kinds and abstracted them to a model, which can be used to develop different kind of software. In the final step, we aggregated the different results of the model building process and developed one single model, which covers all types of legal documents. The final model can be found in diagram 2.7 in chapter 2.

As we are now able to map and hold data from the "real" world in our model there is one important thing which should not be unattended: How should data be added to access them? Basically it is possible to distinguish between three types of adding data to an information system:

- manually
- automatically
- semi-automatically

Adding data manually means to type in data to an information system. Every document, including its collection of paragraphs, sentences, words and single characters, must be typed in character by character. It is a little bit comparable with working on this Master's Thesis. This Master's Thesis and the single parts of it you are currently reading are typed in an information system (in this special case a LaTeX-Workbench). There is no way to do this automatically, even if I would do this step by copy and paste it would still be a manually action. After finishing the manually type-in and starting the LaTeX-"engine", the information system uses the typed-in characters to calculate and generate a new, well-structured version of this Master's Thesis containing the just added parts.

Manual input means a lot of work has to be done by human. Because human power is very expensive in measurable objectives like salary as well as in unmeasureable objectives like lifetime, it should be used economically. Manual input should be done if and only if the workflow does not enable something better. Thinking about the example of working on this Master's Thesis, the workflow researching and writing about the results can only be done step by step and out of the mind of the Master's Thesis writer. It should be clear that this type of workflow does not enable automatically input of data.

Adding data automatically means to use a system doing the work for someone. It is important that the system is not a human being, because otherwise it would be manual action

done automatically for another. Examples for automatically done actions can be seen all around us. A simple and catchy example for those can be an ip-webcam, which takes periodically and automatically pictures and upload them to a webserver. The webserver uses this automatic input and places the pictures on a webpage. This input is done fully automatically. Another catchy example can be found in modern smartphones and GPS-devices, or some cars and trucks. While using GPS navigation systems, some of them do not only receive GPS-signals to calculate routes inside the navigation system but do also send data to a collection point. This data can contain things like the current position and the current speed. At the collection point, a system uses the data to calculate if there is a traffic jam or something else which may hinder free movement and sends back better, alternative routes to the GPS navigation system users. This is another example of automatic data input.

Semi-automatic input is an mixture of both, manual and automatic input. This means that data, which can be added automatically to an information system will be added this way. All other data will be added manually. Another shape of adding data semi-automatic is to add manual a source which can be used for the automatic input of data.

## 3.1 Challenges and difficulties while importing data

As defined, there are three possible ways to import data: Import manually, automatically and semi-automatically. The optimum way is to import data automatically and without any manual input. There is no need to say that a fully automatically import is only possible in a perfect world. Unfortunately we do not live in a perfect world and a computer scientist developing an automatically import struggles daily with some difficulties like:

- different file formats
- different content formats
- different character sets
- different authors
- inconsistency and missing data

**Different file formats**

While importing data automatically one part is obligatory: the import-source. It does not matter what type of import is done or what the intent or purpose of this import is, if it's done automatically there is always an import-source. Let us create a simple and catchy example: Textual content should be imported from a list of files containing textual content. The authors of this files wrote a book with three chapters: The list of files are containing three different textual files: Chapter1.txt, Chapter2.docx, Chapter3.pdf. All of this files should be imported by a simple application. It is noticeable that the authors wrote the first chapter with a simple text-application like Notepad or similar, then the author wrote the second chapter with Microsoft Word and the last chapter is a text document converted to PDF (portable document format). Even if all documents are containing just unformatted

text, this can be a huge problem (see [AF05]). The computer scientist, who develops a system for the automatic import of such files must consider all different formats of the source files, while ordinary text-files can be imported in most programming languages, "higher" formats like *.docx or *.pdf needs plugins or libraries to read out content of them.

**Different content formats**

If source-files have the same file format or a system is able to readout the different file formats, the first hurdle is clear but the next hurdle is waiting: The format of the content. There are thousands of different ways structuring textual content in a file. While sometimes content is structured just in paragraphs, there are many cases in which content is structured with an markup language like XML or TEX. As there is an innumerable amount of variations, the different content formats can be a big challenge for computer scientists developing an automatically importer.

**Different character sets**

But those who think that using the same format should prevent any difficulties are false. Even if using the same format different content will be read out. In most cases different content is the result of importing "the same" file, different character sets may cause the problems. When using only the default characters a-z, A-Z and 0-9 problems with different character sets may not be visible. In most cases the problems may become visible when special characters like $, §, & or similar are in the textual content. Especially the German language is causing problems because most textual contents contain vowel mutations like ä, ö, ü, Ä, Ö, Ü which can be equated with special characters. The visible results of character set problems differs, sometimes characters disappears, in other cases characters will be replaced with totally different characters, which often is the worse case (see [Kor01]).

**Different authors**

When examining carefully the first three difficulties "different file format", "different content format" and "different character sets" it becomes apparent that the root cause is usually created by the author himself. These are all difficulties which could be prevented by doing things better. These difficulties could be eliminated by saving data in the same file format and taking care that the content in the source files have the same structure with the same character sets. But this is another example of the perfect world. But those problems often appear again and again when there are different authors providing content. It is usually the unknowingness in combination with the inattention that let occur the problem. Thinking about the example with the three authors writing the book's three chapters: They probably had taken the same file format if they had known that this could be a problem or they did not know that several file formats exists.

**Inconsistency and missing data**

But "different file format", "different content format", "different character sets" and "different authors" are not the only problems when importing data automatically: Again and again problems occur while importing content caused by missing data or invalid and inconsistent

structured source-files. Such errors occur exactly at the point where authors are using the right technology in a wrong way. A good example is an author providing data structured with a markup language like XML: In XML each markup which is opened needs to be closed. If author forgets to close the markup, the XML file is invalid and for most importers not readable anymore. It may happen that a system needs to know for each source file an attribute like the title of a book. If the author forgets to provide it and this attribute is required, an importer may have problems. Another example is that sometimes data is provided but the format is not valid: Thinking about the import of a book and the author needs to provide the publication date as date with format (YYYY-MM-DD, e.g. 2015-05-03). If the author does not forget to provide it but provides "3rd May 15", an importer may stop working.

Above we gave an overview of the most important difficulties while importing data. There will be the one or the other computer scientists throwing one's hands up in horror thinking that there are many other aspects which may be difficult. Yes they are right, but most difficulties which occur can be put in one of those categories. As the estimation of security issues while importing data is not the main topic of this Master's Thesis, this should answer the majority of all questions.

## 3.2 Importing legal documents

Table 3.1: Overview of a few legal documents of the German legislation

| **Name** | **Type** | **Source** | **Format(s)** |
|---|---|---|---|
| Strafgesetzbuch (StGB) | Law | gesetze-im-internet.de/stgb | xml, pdf, epub |
| Einkommenssteuergesetz (EStg) | Law | gesetze-im-internet.de/estg | xml, pdf, epub |
| Bürgerliches Gesetzbuch (BGB) | Law | gesetze-im-internet.de/bgb | xml, pdf, epub |
| Straßenverkehrs-Gesetz (StVG) | Law | gesetze-im-internet.de/stvg | xml, pdf, epub |
| Straßenverkehrs-Ordnung (StVO) | Law | gesetze-im-internet.de/stvo | xml, pdf, epub |
| Aktiengesetz (AktG) | Law | dejure.org/gesetze/AktG | html |
| Urteile des BGH | Judgement | juris.bundesgerichtshof.de | pdf |
| Entscheidungen des BGH | Decision | juris.bundesgerichtshof.de | pdf |
| Urteile des LG München | Judgement | openjur.de | pdf, tex, xml, json |
| ... | ... | ... | ... |

For this Master's Thesis we need to import data as a basis for our text analysis. As we have discussed in the previous section, we must assume that there will be difficulties when importing data automatically. There are several legal documents which needs to be imported. Table 3.1 provides an overview of some legal documents.

Table 3.1 is just a set of a few legal documents of the German legislation. The first legal documents are all published laws which can be accessed on the official platform "gesetze-im-internet.de" which is an offer of the German Federal Government. The German Federal Government tries to offer all available and published laws on this platform, but only in the newest version. Old versions are not available on "gesetze-im-internet.de". The examples were elected because they should be the most important one for private citizens. The German Federal Government tried hard to offer the Laws in different formats. By default a law is available as XML, PDF and EPUB (see [BD15]). XML is a markup language which allows the structuring of content and is a well-known and often used format (see [BPSM+98]). EPUB is an open standard which has been developed especially for e-books (see [Gar11]). Modern e-book readers use EPUB as default file format.

Usually if a computer scientist gets a list of different file formats is ask what file format he prefers to use for import, the computer scientist would answer XML. Before starting the development of an automatically importer this would have been also my favorite answer. But the German Federal Government hat a little bit have changed my opinion. The XML-interface they provide does not use XML in the standard way. For some reasons they totally ignore the most important benefit of XML: the possibility to structure nested content. It is absolutely weird that they developed a new format by misusing a well-known and easy to use format which does the same before misusing it.

Listing 3.1: Excerpt example Strafgesetzbuch, gesetze-im-internet.de - XML, (see [Deu15b])

```
<?xml version="1.0" encoding="UTF-8" ?>
<dokumente builddate="20150619215005" doknr="BJNR001270871">
  <!-- more norms before -->
  <norm builddate="20150619215006" doknr="BJNR001270871BJNG003702307">
    <metadaten>
      <jurabk>StGB</jurabk>
        <gliederungseinheit>
          <gliederungskennzahl>020010030</gliederungskennzahl>
          <gliederungsbez>Dritter Titel</gliederungsbez>
          <gliederungstitel>Gefährdung des demokratischen
                      Rechtsstaates</gliederungstitel>
        </gliederungseinheit>
    </metadaten>
        <textdaten>
               <text format="XML">
               <Content>[...] <!-- a lot of content --></Content>
               </text>
        </textdaten>
  </norm>
  <!-- more norms after -->
</dokumente>
```

The listing 3.1 displays an excerpt of the Strafgesetzbuch as XML. The format of the XML is exactly the format the German Federal Government offers. As it can be seen, the XML starts with an opening markup “dokumente”, which should indicate that a new legal document starts. This markup has attributes with an builddate, which provides information about the render-date of this markup and an platform-wide unique document number (doknr). Each legal document has a long list of children with the markup name “norm”.

Now the weird concept of this XML starts. A norm can be nearly everything, for example a paragraph of the legal document or a container enclosing it. No matter what it is, they are all children of the markup “dokumente” and they are siblings among themselves. So how it is possible to build a nested structure with this flat architecture? To do this, they add to all markups which should be any type of container a markup “metadaten/gliederungseinheit”. The markup “gliederungseinheit” contains information about the container, which is something like a chapter, a section or similar. If a “norm” contains “metadaten/gliederungseinheit” it is a container. All norms which occur as siblings after this norm and which have no markup “metadaten/gliederungseinheit” are paragraphs of this container. The magic of this concept is done by the markup “gliederungskennzahl”.

The length of the “gliederungskennzahl” provides information about the current deep of the nesting if it’s divided by three. Let’s take the “gliederungskennzahl” of the example in listing 3.1 “020010030”:

$$nestingdeep(x) = length(x)/3 \tag{3.1}$$

$$nestingdeep(020010030) = 3 \tag{3.2}$$

After calculating the current deep of the nesting, the “gliederungskennzahl” gives some more information when playing around with it. The first step is to split the whole “gliederungskennzahl” in parts of length three. The result of this is an array with three parts “020”, “010” and “030”. This single parts can be interpreted with the following logic: “020” means that this container is on the highest level (root-level) in the second chapter. With this logic it is possible to have maximum 99 chapters on the same level (would be “990”). The previous chapter on the same level is a sibling with the the number “010”, the next chapter on the same level would have the number “020”. The first two characters of this number indicate the number of the chapter, the last character is a placeholder for variants of the same chapter. A variant of this chapter could be for example “2.a” or “2.b”. It is possible to have at maximum 10 variants of one chapter (0-9). Keep in mind that this is not the same as a subchapter of the current chapter. Subchapters (chapters on the second level) are part to the second part of the “gliederungskennzahl”. As before it is again possible to have up to 99 different subchapters in each chapter. The third part of the “gliederungskennzahl” is also the same logic.

Summarized: The “gliederungskennzahl” of the example in listing 3.1 is “020010030”. As the length divided by three is three, we know that the container’s position in the legal document is on the third layer (sub sub chapter). Splitting the “gliederungskennzahl” in three

parts allows us to analyze its right position: “020” indicates that the chapter is 2, “010” that the subchapter is 1 and finally “030” that the sub sub chapter is 3.

The other markups located enclosed by the markup “metadata” give information about the concrete title of this container, in case of the example “Dritter Titel - Gefährdung des demokratischen Rechtsstaats”.

As visible they misused the XML-standard and developed their own standard to structure and nest the content. It is not visible what was the intent and purpose of this complex structuring, but a better idea would be to structure nest the “norm”’s in each other. But it is how it is, an importer must handle such source-files as well as “pretty” and well-formatted XML-sourcefiles.

As mentioned all laws of the German legislation are provided by the German Federal Government on their portal “gesetze-im-internet.de”. As laws are generally open documents for everyone (otherwise people are not able to know if something is permitted or not), laws can be found on other platforms too. Because this Master’s Thesis should not be an advertisement for one single platform, exemplarily let us talk about another platforms offering nearly the same content. The first item in the legal document overview-table 3.1 which is not provided by “gesetze-im-internet.de” is the Aktiengesetz (AktG). With this example we introduce a comparative platform “dejure.org”. Dejure is a free accessible database containing more than 270 Laws and more than 1.000.000 Decisions. This numbers are based on information provided by Dejure on their website (Accessed 28th July 2015).

All laws are accessible as HTML on Dejure’s portal “dejure.org”. Unfortunately there is no export function so using the provided content needs parsing the content from their website. This is currently the only way to get the content. There are three big benefits compared with the portal “gesetze-im-internet.de”: The first benefit is that the content is rendered in a very pretty format, so that the use can enjoy working with it. The second benefit is that it not only provides Laws but also a huge amount of decisions. The last benefit is, that it is never good to have only one single and monopolistic source. It seems that all German laws are accessible on “dejure.org” as well as on “gesetze-im-internet.de” too. But there is one big drawback on which should be keeped an eye on: As the German Federal Government is not the institution behind “dejure.org”, there is no guarantee that the information provided by “dejure.org” is up-to-date or correct. Keep that in mind before robbing a bank.

The judgements and decisions of the Bundesgerichtshof (BGH) in the legal document overview table 3.1 are all provided by the Bundesgerichtshof itself on their platform “juris.bundesgerichtshof.de”. They offer a free database of all judgements and decisions they made since the year 2000. All previous judgements and decisions must be ordered and will by provided postal in a printed version. Unfortunately the Bundesgerichtshof has only one displayable version of their judgements and decisions which is a PDF format. In principle PDF is one of the best format for displaying and sharing this type of legal documents, because this are all documents which should be displayed on all devices in the same format and quality. For our purpose PDF is not the best format. What we want to do is to read out the content automatically and access it within our structure. The problem is, that PDF is not a file format which saves it’s content in an structured and hierarchical architecture like a markup

language does. This means that we have to parse the content from the PDF-document what needs a lot of work and can be very error-prone. To get an idea what the first steps of the read-out process are, the illustration 3.1 shows a decision of the Bundesgerichtshof which was provided as PDF on their platform "juris.bundesgerichtshof.de". As it can be observed a PDF of the Bundesgerichtshof is able to contain both, raw text and images like the German Federal Eagle.

# BUNDESGERICHTSHOF

## BESCHLUSS

**X ZB 1/15**

vom
30. Juni 2015

in dem Rechtsbeschwerdeverfahren

Nachschlagewerk: ja
BGHZ: nein
BGHR: ja

Flugzeugzustand

PatG § 1 Abs. 3 Nr. 1, § 4; EPÜ Art. 52 Abs. 2 Buchst. a, Art. 56

a) Mathematische Methoden sind im Hinblick auf § 1 Abs. 3 Nr. 1 PatG nur dann patentierbar, wenn sie der Lösung eines konkreten technischen Problems mit technischen Mitteln dienen.

Figure 3.1: Original PDF, decision Bundesgerichtshof (see [Bun15a])

It is necessary to extract the content from the PDF before analyzing and parsing it. To do this extracting part it is not necessary to reinvent (see [TH00]) the wheel because there are a few libraries, frameworks and tools which takes the PDF as input and doing some magic stuff to extract the PDF's content to raw text. To demonstrate what is the result of such an

extraction process, the listing 3.2 shows an excerpt of the result of the extraction process. This extractionn process has been done by using Apache PDFBox. Apache PDFBox is an open source tool for the work with PDF documents (see [Fou15]). It is written in Java and allows users to extract, create, split, merge or convert PDF's in own Java applications as well as in command line. One drawback, which should be visible at first sight is that it is not possible to extract images. Images will be ignored when extracting text, but this should be no drawback for our purpose. As PDF is not a structured or hierarchical format every text of a PDF document will be extracted and added to the result. It is not easy to interpret whether it is real content or other elements. Taking a deeper look in the result shows one example: "- 2 -" is not real content, it is the page number of which is located in the PDF on the top of the second page. As the extraction process takes all content of the PDF as it is, we are running into several problems with automatic or manual word breaks.

Listing 3.2: Excerpt extraction decision Bundesgerichtshof PDF using Apache PDFBox (see [Bun15a])

```
BUNDESGERICHTSHOF
BESCHLUSS
X Z B 1 / 1 5
vom
30. Juni 2015
in dem Rechtsbeschwerdeverfahren
Nachschlagewerk: ja
BGHZ: nein
BGHR: ja
 Flugzeugzustand
PatG § 1 Abs. 3 Nr. 1, § 4; EPÜ Art. 52 Abs. 2 Buchst. a, Art. 56
a) Mathematische Methoden sind im Hinblick auf § 1 Abs. 3 Nr. 1 PatG nur
dann patentierbar, wenn sie der Lösung eines konkreten technischen Pro-
blems mit technischen Mitteln dienen.
b) Eine mathematische Methode kann nur dann als nicht-technisch angesehen
werden, wenn sie im Zusammenhang mit der beanspruchten Lehre keinen
Bezug zur gezielten Anwendung von Naturkräften aufweist.
c) Ein ausreichender Bezug zur gezielten Anwendung von Naturkräften liegt
vor, wenn eine mathematische Methode zu dem Zweck herangezogen wird,
anhand von zur Verfügung stehenden Messwerten zuverlässigere Erkennt-
nisse über den Zustand eines Flugzeugs zu gewinnen und damit die Funkti-
onsweise des Systems, das der Ermittlung dieses Zustands dient, zu beein-
flussen.
d) Ein Gegenstand, der neu ist und auf erfinderischer Tätigkeit beruht, kann
nicht allein deshalb als nicht patentfähig angesehen werden, weil er im Ver-
gleich zum Stand der Technik keinen erkennbaren Vorteil bietet.
BGH, Beschluss vom 30. Juni 2015 - X ZB 1/15 - Bundespatentgericht

- 2 -
Der X. Zivilsenat des Bundesgerichtshofs hat am 30. Juni 2015 durch
```

```
den Vorsitzenden Richter Prof. Dr. Meier-Beck, die Richter Dr. Grabinski,
Dr. Bacher und Dr. Deichfuß sowie die Richterin Dr. Kober-Dehm
beschlossen:
Auf die Rechtsbeschwerde wird der am 23. Oktober 2014 verkün-
dete Beschluss des 17. Senats (Technischen Beschwerdesenats)
des Bundespatentgerichts aufgehoben.
Die Sache wird zur anderweiten Verhandlung und Entscheidung
an das Patentgericht zurückverwiesen.
```

The last example of the legal document overview table 3.1 are judgements of the Landgericht München which are provided by the free open platform "openJur.de". openJur is provided by a openJur e.V., a registered association (see [oe15]). They try to collect as much legal documents as possible from different sources like Landgerichte, Bundesgerichte or laws of the German Federal Government. One of the most worthy benefits of openJur are their efforts in offering a wide range of different display/export formats: openJur offers each legal document as PDF, TEX, MARKDOWN, JSON or XML. Because we already talked about XML and PDF, we will now take a look on JSON as a format.

Listing 3.3 shows an excerpt of one judgement spoken by the Landgericht München. This judgement was loaded from openJur.de as JSON. JSON was developed for the easy exchange of data between systems (see [Cro06]). As it can be observed, "openJur.de" did a good job and created a well-formatted and clear structured JSON-file. The JSON-file contains both, raw textual content and meta-information about the judgement. The provided meta-information in the listing 3.3 gives information about the court ("gericht: LG München I") and the court's identification number ("aktenzeichen: 37 0 11843/14"). OpenJur also adds a reference to the location of this JSON-file ("permalink_json: http://openjur.de/u/771387.json") what can be really helpful especially when source document suddenly gets important again. The textual content in this JSON-file is separated in three main parts ("tenor", "tatbestand" and "gruende"). In each of the parts special characters are escaped and linebreaks or paragraphsbreaks are done with "\n or \n \n".

Listing 3.3: Excerpt export Judgement from openJur as JSON, (see [I15])

```
{
 "aktenzeichen": "37 O 11843/14",
 "gericht": "LG München I",
 "datum": "27.05.2015",
 "dokumenttyp": "Urteil",
 "fundstelle": "openJur 2015, 9396",
 "permalink_json": "http://openjur.de/u/771387.json",
 "permalink": "http://openjur.de/u/771387.html",
 "tenor": "I. Die Klage wird abgewiesen.\n\nII. Die Kosten des Rechtsstreits
 trägt die Klagepartei.\n\nIII. Das Urteil ist gegen Sicherheitsleistung i.H.v.
 110 % des jeweils zu vollstreckenden Betrages vorläufig
 % vollstreckbar.Beschluss:\n\nDer Streitwert für das Verfahren wird\n\n- für
 % den Zeitraum bis 14.11.2014 auf 1.000.000.- Euro\n\n- und ab dem 15.11.2014
```

```
% auf 1.100.000.- Euro festgesetzt.",
"tatbestand": "Die Parteien streiten um die Zulässigkeit eines
Werbeblockers.\n\nDie Klägerinnen gehören zur ...Gruppe. Die Klägerin zu 2
betreibt die im Klageantrag (Anlage 1) genannten Internetseiten (...-Seiten)
wie beispielsweise www...de und www...de für Unternehmen der ...-Gruppe, die
Klägerin zu 1 vermarket Werbung auf den ...-Seiten. Die ...-Seiten sind für die
Nutzer überwiegend kostenlos und werden im Wesentlichen durch die von der
Klägerin zu 1 vermarktete Werbung finanziert.
[...]",
"gruende": "Die zulässige Klage ist nicht begründet, weder in den
Hauptanträgen, noch in ihren Hilfsanträgen.\n\nDabei war über die Klageanträge
in ihrer geänderten Fassung zu entscheiden. In diesem Zusammenhang kann offen
bleiben, inwieweit es sich tatsächlich jeweils um eine Änderung des
Streitgegenstandes handelt. Die Beklagten haben sich jedenfalls rügelos auf die
zuletzt gestellten Anträge eingelassen,
[...] ",
"gruende_isexcerpt": false, "lizenz": "ODbL", "lizenz_hinweis": "Enthält Daten
von openJur, die unter der Open Database License (ODbL) veröffentlicht wurden."
}
```


OpenJur is really a good source when well-formatted legal documents are needed, because all export file formats openJur offers can be helpful for the development of an automatically importer and further developments. But openJur has the same big drawback like "dejure.org" has: As this platform is not provided by official site (German Federal Government), there is no guarantee that the data on this platform are correct and complete. In a perfect world, the German Federal Government takes a look on other platforms like openJur and reinvent their current export concept or add new export functionality to their current range.

## 3.3 Development of a generic importer

In the section before we provided an overview of different sources for legal documents which are available for the German legislation. We have found that most of the different sources provide a few file formats for exporting the legal documents. In most cases it is not required to parse content from websites, but the different export formats need a special handling when an importer should be implemented for those. As there is no standardized file format for legal texts and there are sources that misuse existing file formats, there will be no real chance to write only one single importer per file format, it is necessary to write an importer for each file format and for each source.

The goal of this section should be the development of an import-structure, that makes the most of the current entangled and tricky situation. To start the development of an import-structure, it is good to summarize the main findings of the research results and the desired functions of the importer:

- three main legal document types legislative document, jurisprudence document and literature
- different sources for all imports

- mainly three different important file formats: XML, PDF, JSON

The first function should be clear enough and should not take any further explanation. Every of the legal document types must handled itself because of the different intent and purpose. The second function has been researched in the previous section: As there is no common denominator, we have no chance to use one importer of the first source on unmodified on an other. This does not exclude that there is potential in learning how we can do the things. The third functions build on the information of the previous section. As the small research shows, there are mainly three different file formats that will be useful for our importer. XML and JSON should be the easier file formats, PDF will be a little bit tricky. The table 3.1 shows that there are more export file formats, but we do not want to parse HTML documents when there are better sources with better file formats which have been developed for data exchange.

The entity-relationship-diagram in illustration 3.2 shows the structure of our importer for legal documents. As it can be observed there is one on top the the whole structure: LegalDocumentImporter. This class is superclass for the classes LawImporter, JudgementImporter and LiteratureImporter. As it can be observed, we put different legal document types visible in this structure. With this structure it is possible to reuse functions and knowledge of other importers.

As we talked about, it is not possible to develop just one importer doing all the stuff for all sources. It is even not possible to write just one single importer for one legal document type. Therefor, it is necessary to make the classes LawImporter, JudgementImporter and LiteratureImporter become a superclass too. This classes are the superclasses for the real importer classes, which are able to do the importing automatically. We must code one importer class for each legal document type and each source. As this can be from one to an infinite number of different importers, there are a handful of them to show exemplary how this structure will look like. In the illustration 3.2 there are four different importer classes: “GermanLawsImporter”, “AktGLawsImporter”, “BGHJudgementsImporter” and “LGMJudgementsImporter”. The importer class “GermanLawsImporter” is the importer class which imports legal documents of type laws from the official source of the German Federal Government “gesetze-im-internet.de”. As the attentive reader may remember, the German Federal Government offers the laws as XML. Read-out and parse XML needs a bit of work but is basically nor problem. As a good computer scientist does not want to do things twice, we do create an interface-class XMLImporterInterface, which contains all the relevant functions to read-out and parse an XML. This interface-class will be implemented by our importer class “GermanLawsImporter”.

Exactly the same will be developed for each of the other importers, but for each importer there will be a decision from which superclass the importer should inherit and which importer-interface should be implemented. The importer “BGHJudgementsImporter” is the importer for all jurisprudence documents of the Bundesgerichtshof. Therefor it inherits the class JudgementImporter and implements the class PDFImporterInterface.

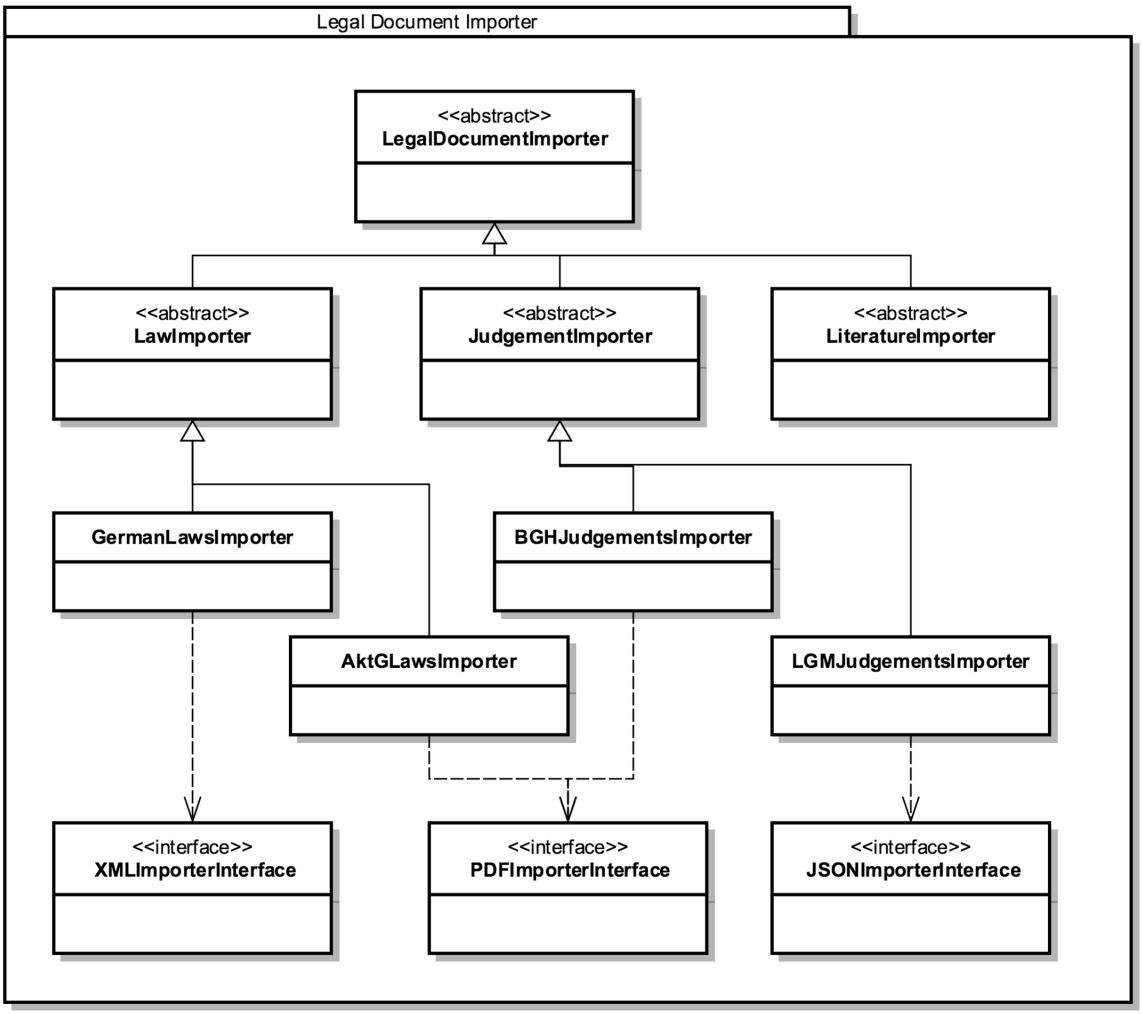



Figure 3.2: Entity-relationship-diagram: legal document importer

To show this importer structure with more details I would like to transfer the entity-relationship-diagram 3.2 into Java to see how such a structure will look like within an application. Therefor, let's take a look at table 3.2. The table shows the structure of the complete importer within a Java-application.

There are probably thousands of possible ways to make the structure of the diagram 3.2 working within a Java-application. The idea behind the solution visible in table 3.2 is to build it as generic and easy to extend as possible. This package/file-structure displays the developed importer-structure as clear as possible and is quite easy to teach to practised computer scientists. Every single package and file used by the importer is located within the package "importer". In the root of this package there are the generic files of the importer located: "LegalDocumentImporter.java" - the superclass of each importer. This class is basically a conglomeration of the basic functions/variables each importer needs. The most important

Table 3.2: Legal Document Importer within an Java application

| **Path/Name** | **extends** | **implements** |
|---|---|---|
| */importer* | | |
| LegalDocumentImporter.java | - | - |
| XMLImporterInterface.java | - | - |
| PDFImporterInterface.java | - | - |
| JSONImporterInterface.java | - | - |
| LawImporter.java | - | - |
| JudgementImporter.java | LegalDocumentImporter | - |
| LiteratureImporter.java | LegalDocumentImporter | - |
| */importer/laws* | | |
| GermanLawImporter.java | LawImporter | XMLImporterInterface |
| AktGLaw.java | LawImporter | PDFImporterInterface |
| */importer/laws/germanlaw* | - | - |
| Content.java | - | - |
| Document.java | - | - |
| ExternalReference.java | - | - |
| ... | ... | ... |
| */importer/laws/aktglaw* | - | - |
| - | - | - |
| */importer/judgements* | - | - |
| BHGJudgements.java | JudgementImporter | PDFImporterInterface |
| */importer/judgements/bghjudgement* | - | - |
| ParsingBGHContentHelper.java | - | - |

are a list of source-files with the associated function "setImportFiles(ArrayList<File>files)", and a list of resulting legal Documents "getLegalDocuments()". It is important to set in focus that this class is only the basis for each importer, this class do not provide any working or processing functions.

Processing functions can be found in the next files: The importer interfaces for the different file formats. Each of this importer interfaces ("XMLImporterInterface.java", "PDFImporter-Interface.java", "JSONImporterInterface.java") are providing the main functionalities and are blueprints for the real importer classes. Each of this interfaces has a blueprint function "public boolean doImport() throws Exception". This is the processing function which should do the complete magic within the importers. For computer scientists which are working later on the project, this function should be a "blackbox". It is not relevant for them how this function works exactly, but they get *true* if everything works and *false* including an Exception if something went wrong.

As the different types of LegalDocuments can have their individual stuff, there are three different superclasses which extend the LegalDocumentImporter: “LawImporter.java”, “JudgementImporter.java” and “LiteratureImporter.java”. All of these importers are fitted with necessary attributes which are necessary for processing the type of legal document.

After the preparations are done, we are now able to implement the real importer classes. Let us take a deeper look on one of them: “GermanLawImporter.java”. The importer class for the German laws extends the class “LawImporter.java” and implements the interface “XMLImporterInterface.java”. The created real importer class “GermanLawImporter.java” has after creation, without any further actions, and at least an array of source-documents of the superclass “LegalDocumentImporter.java”, an array of results as well inherited from the superclass “LegalDocumentImporter.java”, a few additional variables and functions of the “GermanLawImporter.java” and most important: A blueprinted function “public boolean doImport() throws Exception”, which must be implemented for this special case. Within this elaboration we will not take a deeper look in the processing function, because there is no additional value. As each source and each legal document type needs a special handling, the processing can not be used “one-to-one” within other importers.

Some importers may be very easy to implement, especially when they are a clear and easy to use structure. Unfortunately there are many other importers which need pretty lot of work and therefor many other classes helping to get the job done. To make the computer scientist’s work a little bit easier and more comprehensible, each real importer gets its own package, where everything is allowed. It is a playground for each computer scientists, there is no rule that it must be used: Everything is possible, nothing is necessary. As it can be observed in the table 3.2 the computer scientist who implemented the importer for the German laws used this possibility and added a lot of files to the playground-package “/importer/laws/germanlaws”. The table 3.2 shows just an excerpt of the package. Theoretically it would be possible to implement all the stuff within the real importer class “GermanLawImporter.java”. The computer scientist who implemented the importer for the Aktiengesetz “AktG” did exactly this: The whole importer including the whole stuff is implemented within only one file “AktGImpoter”. Even if the package is empty, it is good to create it to have the similar structure everywhere.

## 3.4 Limitations of generic importers

In the previous section we developed a generic importer for legal documents of the German legislation. As it would take a few Master’s Thesis to implement importers for all different legal documents and sources of the German legislation, we did not implemented them all. However, we created a foundation for the further implementation of these.

The development and the use of a generic importer has benefits and drawbacks. The benefits should be self-explanatory:

- automation of recurrent tasks
- exclusion of manually done errors

- using shared knowledge
- straight forward implementation
- structured procedure during implementation
- usage of blueprints

The drawbacks are a little bit trickier and not always visible at first sight:

- loss of specific data
- error prone when manipulating the foundation
- limited scope

It is normal that good things bring along drawbacks. But drawbacks are not that bad, when make the best out of it. A list of drawbacks can be very useful to set and define limitations on something, and limitations are both, good and bad. Good means that there are borders within a computer scientist can let steam off without losing sight of the main objective. Bad means that there are borders which sets clear limits and stops to computer scientist's possibles and creativity.

The loss of specific data is one of the drawbacks that sets a limitation. When we researched the "real" world legal documents we saw that the transfer of this legal documents abstracts the legal documents when transfer it to our model. This is exactly what a generic and automatic importer will do. It takes a look on the source-file and reads-out the information of this source-file we implemented in the import-function. Even if a source-file would give us additional or better information, our predefined implemented importer will ignore it. This data is lost within our scope. If we only implemented one specific and non generic importer, it would be on the first sight the same, but if only one source-file must be handled, the specific importer is a lot easier to reimplement if there are better or additional information. When implementing a generic importer it is not that easy and there are probably information which are not provided by all other sources.

The loss of specific data is in any case a limitation to our generic importer. To answer the question if this limitation is good or bad for our application: It depends. On the one hand there will be the one or other information which is really pity when it's getting lost. But on the other hand most of the additional information will not be used within our scope, so it is really unnecessary to handle and keep them. For any computer scientist taking a look at the work as a whole it will be much easier to understand it when not every single confetti is implemented.

The KISS-principle (keep it small and simple, see [TH00]) that we not exactly meet but in which our generic importer structure goes has the drawback that it is very error prone. Especially when adding, modifying or removing functions or variables in the foundation. The deeper the manipulations are done, the bigger the negative and unwished effects. For example: When editing the class "LegalDocumentImporter.java", nearly all classes which extends from this superclass and the classes extending this classes are endangered to throw errors after "LegalDocumentImporter.java" has been manipulated.

This big drawback sets the next limitation of a generic importer: It is not (easily) possible to reimplement stuff of the parent importer classes without editing all importers extending them. It must be keeped in mind that the classes of the foundation must be implemented as clear as possible and as generic as possible. In a perfect world, nothing must be manipulated in this classes after the first real importers have been developed. But this is just wishful thinking: In real world this classes will be manipulated because there will rise new requirements or there are bugs or something similar.

# 4 Semantic text annotation

After starting a new chapter the first thing should always be to take a look back on the covered topics. In the first chapter we talked about the history of legal documents and about the analysis of textual content. In the second chapter we took a deep look in the German legislation and analyzed what types of legal documents and what kinds of these exists and how the structure of those looks. At the end of the second chapter we fitted all those together to one single model representing the most important legal documents of the German legislation. In the previous chapter, we developed a generic importer to get data from various sources and different types to get content automatically into our scope and application.

So far we did no analysis or manipulation on the legal document's data. The only thing we did until now is that we receive the raw data. When analysing the raw content with high class text analysis functions or something similar, there will be a lot of meta information that has to be combined with the raw text. This is exactly the intent and purpose of this chapter. In this chapter we research possibilities to fit the raw important content with additional information. Please keep in mind that the text analysis is a big research field and is not part of this Master's Thesis. What is part of this Master's Thesis is how to support the further work of text analysis experts by creating a possibility to prepare the text for further analysis or adding the results of the analysis to the text.

Adding semantic annotations means adding additional information to textual content. It is not regulated what kind of additional information can be added. Furthermore, there is no special case what happens with the additional information (see [EMSS00], [OMS+06]). There are many examples which are using semantic annotations to add information to its content. Nowadays, one of the most used formats for adding additional information to textual content is HTML. Hypertext Markup Language (HTML) is used to add structural information to textual content in order to make the content visible in a structured way in a webbrowser. It was introduced in 1992 and is by now available in the fifth version. HTML adds additional structural information by adding markups to the raw text. The markups enclose whole words, sentences, paragraphs, documents or other parts of the resulting web page. The webbrowser reads the HTML-files and renders the raw file in the human-readable and defined structure (see [RLHJ+99], [Hic12]).

Listing 4.1: Example HTML-document displaying basic structure of each HTML-document.

```
<html>
<head>
 <title>Document-Title</title>
</head>
<body>
 <h1>This is the first chapter.</h1>
 <p>This is a lot of text written in one paragraph.</p>
 <p><strong>This text should be bold, it is important.</strong></p>
 <p>This is <strong><i>bold and italic.</i></strong></p>
</body>
</html>
```

The example in the listing 4.1 shows an example of how textual content can be structured. The raw textual content of this document is:

"This is the first chapter. This is a lot of text written in one paragraph. This text should be bold, it is important. This is bold and italic."

To the raw text, additional information about its parts have been added. First of all the first part "This is the first chapter." is enclosed by the markup "h1" which indicates that the enclosed content is the heading 1 of this text. Next, the single paragraphs become enclosed by the markup "p". This forces the enclosed content to be part of a paragraph. HTML does also offer styling information that add styling to the content. An example in the listing 4.1 is the markup "strong" that forces the content to be displayed in bold. Styling or structuring-markups can also be combined with other markups when they do not escape each other. It is for example possible to combine the markup "strong" with the markup "i" which leads to the content being displayed in italic.

After adding the additional structuring and styling information on the raw text, the whole construct will be embedded in the default minimum HTML-structure. The default minimum HTML-structure starts with the markup "html" and contains at least a markup "head" and a markup "body". The markup "head" contains meta-information about the complete HTML-document like the "title". The "body" contains everything of the content. This is the position for our part, the markups "body" will enclose it.

The result of the complete process, which is basically raw content structured with markups, is not visible humans eyes. It is necessary that a webbrowser reads and renders the content. Performing this process on the example of listing 4.1, the resulting text looks as follows:

# This is the first chapter.

This is a lot of text written in one paragraph.

**This text should be bold, it is important.**

This is ***bold and italic.***

Styling content means not only to change the display of the document. There are also HTML-markups and commands that manipulate, give the content itself and give different content to the reader of the document. One of these commands can be added as CSS (Cascading Stylesheet, see [CR12]) style information to an HTML-paragraph "p":

Listing 4.2: Example Manipulating HTML-markups and commands

```
[...]
<p style="text-transform: uppercase;">Make this content Uppercase.</p>
<p style="text-transform: lowercase;">Make this content Lowercase.</p>
<p style="text-transform: capitalize;">Make this content Capitalized.</p>
[...]
```

If the webbrowser receives the paragraph-markup "p" with a style-attribute, additional styling of the content can be done. The listing 4.2 uses this style attribute to add manipulative styling-commands. The CSS-command "text-transform" edits the enclosed content. The styling-command "text-transform: uppercase" manipulates the enclosed content and converts every character within the markup's scope to uppercase. The result of the first line would be "MAKE THIS CONTENT UPPERCASE". The converse styling-command to "text-transform: uppercase" is "text-transform: lowercase". This command converts the enclosed content to lowercase characters, the result of the second line is: "make this content lowercase". To complete the example in the listing 4.2, the last possible text transformation command is "text-transform: capitalize", which manipulates the content to be capitalized: "Make This Content Capitalized.".

It is debatable whether the raw content "Make this content Uppercase." is really different to the transformed content "MAKE THIS CONTENT UPPERCASE.". If somebody without a background in computer science were to take a look at this example, he would likely say that these have the same content. Is he really right with this statement? In the human's interpretation this is the same content but for machines, the both phrases are completely different from each other. Characters like A, a or B, b are for a machine not the same: they are different characters. If it is necessary to check whether a phrase one and phrase two are the same or not, many computer scientists transform the text to lowercase or uppercase and match than the lowercase or the uppercase-version of both string phrases with each other.

## 4.1 Types of semantic text annotations

The introduction of the Hypertext Markup Language in the previous section should be enough to get an idea for what semantic text annotations are used for. HTML is just one example - there are hundreds of other purposes for semantic text annotations. As mentioned, the main intent and purpose of semantic text annotations is to provide additional information to raw text.

In the example of the previous section, the additional information was add as markup in the raw text. But this is not the only possible way to create semantic text annotations. There are mainly two types (see [Wil09]):

- in-line-annotation
- stand-off-annotation

In-line annotations are exactly how HTML works: Additional information is embedded in the text: in-line. Stand-off annotations leave the raw content as it is and create a second file containing additional information about the raw content file. It is comparable with a phonebook containing address information about a small village: The phonebook would be a stand-off annotation because there is information like "Doe, John: Main Road 123, 56789 City". With the information of the phonebook, the reader of the phonebook would be able to identify the person who lives in the house with the address "Main Road 123". The in-line annotation in this example would be the name plate with the name "John Doe" on the house. A visitor of the house would be able to identify the person who lives in the house. To be honest: When using the phonebook as an addressbook, someone wants to know where one person lives and not the other way around. But it is not easy to find a real world example outside information systems that makes the two main types of semantic annotations tangible.

We will now take a deeper look on semantic annotations. Because we already identified the two main types of semantic text annotations "in-line-annotation" and "stand-off-annotation", we will know discuss the both separated from each other.

### 4.1.1 In-line-annotations

Let's start with the in-line-annotations. We already showed one example using in-line-annotations to add additional information to raw textual content. As mentioned in-line annotation means adding semantic text annotations to the raw text-file itself (see [Wil09]). An especially famous example which thrives of this principle is the Hypertext Markup Language we talked about. The drawback of HTML is that it's semantic language is limited. Adding additional markups is not allowed within HTML.

Is this the end of using a markup language like HTML? Yes for HTML it is the end, because the markups provided by HTML are not enough to support the further work of computer scientists who want to work with the content and want to add their information to it. But there is a very similar semantic language which allows computer scientist to use their own vocabulary and add custom markups to raw content: XML (see [BPSM$^+$98]). The eXtensible Markup Language does not have its own vocabulary and it does not provide any markups

itself. XML has been developed to add hierarchical information on raw textual content. We already debated about XML when researching different sources of legal documents in the chapter before. XML was introduced in 1998 and therefor more than six years later than HTML. What does a XML-document look like? We already saw one when taking a look at the source import files on the official platform of the German Federal Republic "gesetze-im-internet.de" (see 3.1). But as mentioned, when talking about it, the German Federal Republic is not famous for creating well-formed XML documents which use all features of this markup language.

To see at least one well-formed instance of XML that uses the advantages of this markup language, let's introduce a new example.

Listing 4.3: Adding additional information with XML

```
<sentence>
  <subject>
       <firstname>Homer</firstname>
       <lastname>Simpson</lastname>
  </subject>
  <noun>likes</noun>
  <adjective>blue</adjective>
  <object>jeans</object>
  <punctuation>.</punctuation>
</sentence>
```

The example of the listing 4.3 shows a valid, well-formed XML that adds additional information to the raw text:

"Homer Simpson wears blue jeans."

Two additional types of information have been added to the raw text of the example in the listing 4.3: Additional information about things in the content and hierarchical information. The whole structure starts with the markup "sentence" which indicates that a new sentence starts. Every content enclosed by the start-markup "<sentence>" and the end-markup "</sentence>" is part of the same sentence. Keep in mind that the markup "sentence" is not part of any vocabulary provided by XML, because XML does not provide any vocabulary. But XML allows every vocabulary as long as it is well formatted and is opened by a start-markup and closed by an end-markup. If, and only if a the content enclosed by a markup is empty, then it is allowed to combine the start-markup and the end-markup to one single markup. For example: If a sentence does not contain any content (it's empty), then the markup "<sentence/>" is enough. It would be allowed to take the start-markup followed immediately by the end-markup too, but a good computer scientist tries to save unnecessary memory usage everywhere.

Let's go one step deeper in the example of the listing 4.3: A minimal sentence in the English language, as well as in the German language, contains at least a subject and a noun. As it can be observed, the example sentence contains a subject and a noun, too. The subject

"Homer Simpson" is enclosed by its own markup as well as the verb "likes". Because this is a sentence with a little bit more information than a minimal sentences like "He goes.", the sentence has an object with an adjective describing the object with a detail. The object and the adjective are enclosed by its own markups as well. XML allows unlimited interleaving: there is no maximum limit of nesting levels. Each object is allowed to contain an unlimited amount of other objects. We tried to make this visible by adding more additional details to the subject. The subject "Homer Simpson" received additional markups "firstname" and "lastname", giving the complete subject information about the single parts of it.

XML has significantly more interesting functions which makes it one of the strongest languages for semantic text annotations. One of these is the possibility to add more attributes on the markups itself. Like the style-attribute in the HTML-example, attributes can be added in the same way. One example of such attributes can be to add a type-information on subjects, information about times to the verb and information about pricing to the object.

Listing 4.4: Adding additional information with XML

```
<sentence>
  <subject type="person">Homer Simpson</subject>
  <verb times="present">likes</verb>
  <adjective>blue</adjective>
  <object min-price="50.00" max-price="120.00">jeans</object>
  <punctuation>.</punctuation>
</sentence>
```

The example in the listing 4.4 is similar to the example of 4.3, but it is fitted with more additional information added as attributes to the already added markups. Now a person reading this example knows that the subject "Homer Simpson" is a person and currently likes blue jeans which cost between 50 and 120 monetary units. One big benefit is that the raw content's structure is visible and comprehensible to the readers without any further assistance or conversion. Also, the content itself (not only it's structure) stays understandable. The reader just has to hide the markups within his head or by the using one of countless existing software for displaying XML-files.

After we counted only benefits of XML, we talk about the drawbacks of a markup-language like XML. XML works great as long as the XML files are well-formed. Well-formed means amongst other things that:

- every markup that starts will be closed
- enclosed markups must be opened and closed within the enclosing markup
- disallowed characters

A complete and more detailed list of well-formed characters can be found in [w3s15]. We already talked about the first point. Every markup that is opened must be closed. We extended this short explanation by the special case of empty markups where we explained that in this case the start- and the end-markup is allowed to be combined to one single markup.

The next point sounds easier than it really is: "Enclosed markups must be opened and closed within the enclosing markup". Most of the readers of this Master's Thesis perhaps think that this can't be a problem, but with the well-formed aspect XML destroys maybe some future ideas. But let's start at the very beginning: XML works as designed, it allows to add structure to raw textual documents. Furthermore, this structure allows to create and display relations between each part of a document to other parts. Each part of a document is allowed to be parent and is therefor allowed to contain other parts of the document. The children of a parent are siblings to each other. When thinking about relations that are possible to create in a well-formed XML structure, there are many parable examples: Most computer scientists think about a family tree with parent and children relationships, but as there are many dubious family relationships, a better parable example would probably be an ordinary tree. Each tree has a trunk which is the root. Each trunk has many branches, which can have both: branches and leaves. A leaf is the end, a branch can contain other branches and leaves.

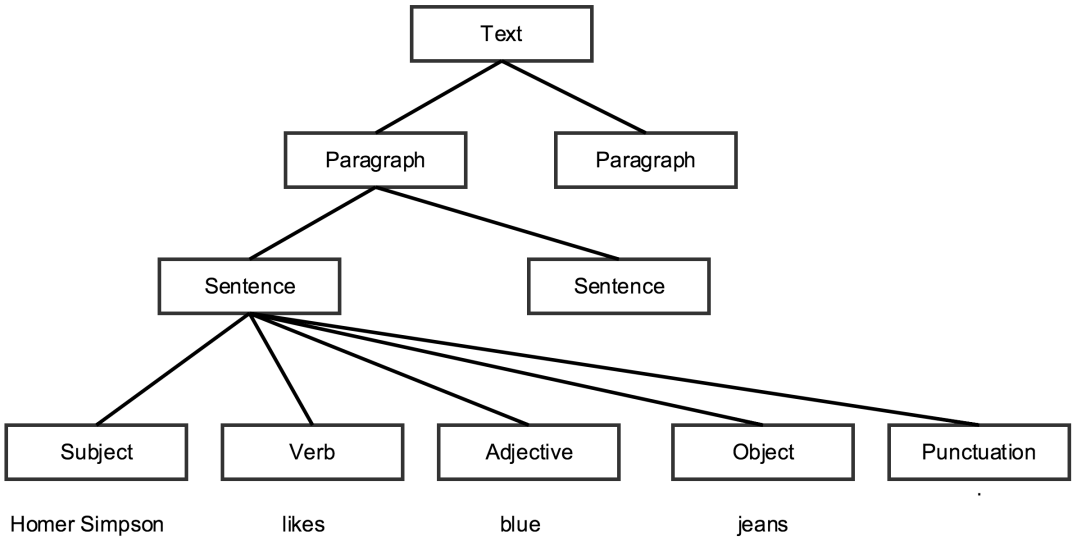



Figure 4.1: Example sentence "Homer Simpson likes blue jeans." as tree.

Everything that can displayed as a tree is possible to be well-formed in an XML-structure. Illustration 4.1 shows the sentence "Homer Simpson likes blue jeans" which has been already brought into an XML-structure in listing 4.3. We run into problems with XML when we cannot depict the problem definition as an ordinary tree. This happens exactly when one leaf tries to be related to many branches. In nature, a branch can have many leaves but not the other way round.

Take a look at the following sentence:

"This text starts cursive, then it will be bold too and ends just bold."

When we try to make this sentence visible, it would look something like this:

"This text starts *cursive, then it will be* ***bold too and ends*** **just bold.**"

The main question is now: How can this sentence be structured by using the markups "sentence", "bold" and "cursive"? Listing 4.5 shows how this would look like when using XML. As it can be observed, this structure is not able to be read using default XML-parsers, because basic XML rules are violated. The problem is that XML does not support overlapping markups (see [w3s15], with an eye on linguistic annotations: [Wil09], [GB]). Nested markups are okay, but they are not allowed to overlap each other.

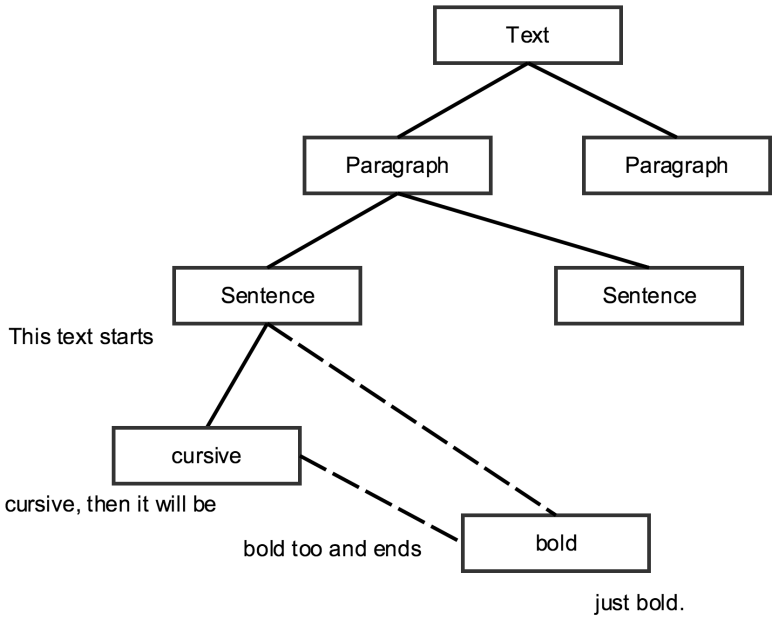



Figure 4.2: Conflicting sentence as tree.

Imagine this were a tree: it would look like the illustration 4.2. As it can be observed in this illustration, the leaf sticks on two branches, what is a clear violation of natural rules and hence a violation of XML-rules.

Listing 4.5: Violating basic XML-rules by overlapping

```
<sentence>
  This text starts
  <cursive>
    cursive, then it will be
    <bold> bold too and ends
  </cursive>
          just bold.
    </bold>
</sentence>
```

This is not the end of using XML as semantic text annotation language. The question should be: It is necessary to support overlapping annotation or is the default hierarchy enough for this purpose? If it can be excluded that overlapping annotations will occur, XML is a good choice and should be followed up. If overlapping annotations are a necessary part, other possible workarounds have to be found. As we do not want to set a limit on further computer scientists working with this basis, we will try to analyse a few of workarounds for this problem definition. There are two possible ways to support both, the usage of XML and overlapping annotations:

- split and nest the annotations in each other
- create an own language which looks a bit like XML

The first way to support both, needs a bit of work when creating the XML and a lot of work when parsing and understanding it later on. When choosing this way, the first step is to analyse the XML for overlapping markups. Non overlapping markups will be left as they are. An overlapping markup can be found when a new markup is opened and another markup is closed before this markup is closed. In the listing of the XML violation 4.5, we will notice this when the end-markup "</cursive>" occurs. What will be done in this case is, that before closing the markup "cursive", the markup "bold" will be closed by using the end-markup "</bold>" followed by the end-markup "</cursive>". Immediately after this, the markup "bold" must be opened again to keep everything as it is. The result of this investigation can be observed in listing 4.6.

Listing 4.6: Resolve XML-Violation

```
<sentence>
  This text starts
  <cursive>
    cursive, then it will be
    <bold> bold too and ends</bold>
  </cursive>
  <bold>
     just bold.
  </bold>
</sentence>
```

It should be understandable that the XML in the listing 4.6 is a good and solid workaround if overlapping annotations should be created. But unfortunately this needs a lot of work in both directions, the creation and the parsing of the XML. Connected parts will be torn away from each other, it could be possible that the error rate rises with such operations.

The second way is the creation of an own language which supports overlapping annotations. We create our own language which forgoes the nested structuring but uses XML-markups to add additional information to the raw text.

Listing 4.7: Resolve XML-Violation with own language

```
<sentence length="70" />
  This text starts
<cursive length="43" />
  cursive, then it will be
<bold length="28" />
  bold too and ends just bold.
```

Listing 4.7 shows how our own and new language works. It adds markups wherever a new part starts and sets the length of the area in which this markup is valid. The first markup "<sentence length='70' />" defines that a new sentence starts. Immediately after this markup the next 70 characters are part of the sentence. This counting method only works if all markups are not counted, but only the raw text. For this example, 70 characters include the whole sentence including all spaces and punctuations. The next markup "<cursive length='43' />" includes the whole phrase "cursive, then it will be bold too and ends ". Finally the last markup "<bold length='28' />" includes the rest of the sentence. It should be clear that this is a working workaround too which do not break any XML-rule. But is this a good solution? It is a working solutions that allows overlapping semantic text annotations. It is easy to create such a file and easy to parse it again. But one thing that keeps it away from being the optimal solution: let's remember back in the last chapter when talking about legal documents that are provided as XML from the German Federal Government on their own platform. In that chapter we charges the German Federal Government to ignore all best practices and guidelines of XML. When creating our own solution we are not better than them. The best practice is to use existing standards and not to create own fancy solutions. Thinking about this circumstance, we should definitely think about a better solutions.

The last drawback which needs to be talked about builds on the knowledge of the last explanations: As in-line semantic text annotation works by adding additional information to a raw textual content, it is necessary to add those information in a form that can be clearly separated from the raw textual content when parsing it at a later time. It can be imagined that adding markups that starts with angle brackets "<" and end with angle brackets ">" are only able to be separated from the raw textual content, if these characters are not occurring in the content. There are also a lot of other characters which are forbidden in the raw textual content.

Generally, it is not problem to escape or refactor the raw textual content before turning it into an XML-file, but this work must be done and this is unperformant. It is not the

work itself that makes it unattractive, but it is the fact that raw textual content that is correct, must be touched and modified to make it possible to add additional information (see [Wil09]). It should be forgo to manipulate content, and this is the reason why we will take a look on the other semantic annotation type “stand-off annotation”.

### 4.1.2 Stand-off-annotations

After we talked about in-line semantic text annotations and we analysed the benefits and drawbacks of those, it is now time to analyse the next main type of semantic text annotations: Stand-off semantic text annotation.

Stand-off semantic text annotations are the opposite of in-line semantic text annotations. While in-line means that additional information about a raw text is added to the raw text itself, stand-off means that no information is added to the raw text itself. To create a semantic annotation, a second file will be created that contains only the additional information about the raw text file (see [Wil09]). Summarized: Doing semantic text annotations using stand-off semantic text annotations needs at least two files:

- raw text file
- annotation file

Let’s take a look at the raw text file in listing 4.8. This raw text file contains an excerpt of an tech article about scientists researching the possibilities of a computer program and the awakening of them to real life organisms.

Listing 4.8: Excerpt of an tech article ([Rev14])

```
The Avida project began in the late 1990s, when Chris Adami, a physicist, sought
to create computer programs that could evolve to to simple addition problems and
reproduce inside a digital environment. Adami called these programs "digital
organisms.".

Whenever a digital organism replicates, it has a chance to alter the program of
the newly created offspring. In this way, the programs mutate and evolve. The
goal of the Avida program is to create a model that could simulate the
evolutionary process.

Initially, the digital creations were unable to process numbers in any way. But
Adami designed Avida to reward digital organisms that were able to work with the
numbers in some way. The digital organisms that could process numbers were
allowed to reproduce in higher numbers. In only six short month, the primitive
program had evolved a number of mechanisms to perform addition. And, most
surprisingly, not all of the digital creatures performed addition in the same
way.
```

Principally it doesn't matter what the exemplary raw text in listing 4.8 is about. The example has been selected not only because it regards to computer sciences, but mostly because it allows to show many different examples when creating a stand-off annotation.

Semantic text annotation, whether it is added in-line or stand-off, allows everything to be annotated. In this Master's Thesis we mostly used the same example with sentence, noun, verb and so forth because these are annotations which do not need any special knowledge in high class text analytics. We will use that now, too, but fitted with a few information of this high class text analytics.

The listing 4.9 shows a stand-off annotation file which could look like a stand-off annotation file that is really in use. As it can be observed this file is a well-formed XML file with an easy hierarchy. The whole XML is separated into the root level in different markups which are annotating different things in the raw text file 4.8: "paragraphs", "sentences" and "nouns". Keep in mind that this is just an excerpt and not every annotation file contains exactly these annotations - it strongly depends on the use case.

Listing 4.9: Example stand-off semantic text annotation file.

```
<paragraphs>
       <paragraph no="1" start="0" end="251" fre="41.8" avg-wps="13.00" />
       <paragraph no="2" start="253" end="504" fre="48.8" avg-wps="14.33"/>
       <paragraph no="3" start="506" end="977" fre="48.4" avg-wps="15.40"/>
</paragraphs>
<sentences>
       <sentence no="1" start="0" end="200" fre="26.3" wps="32"/>
       <sentence no="2" start="201" end="251" fre="34.1" wps="7"/>
       <sentence no="3" start="252" end="362" fre="38.8" wps="18"/>
       <sentence no="4" start="363" end="408" fre="71.8" wps="8"/>
       <!-- ... -->
</sentences>
<nouns>
       <noun no="1" start="4" end="9" is-complex="true" syllables="3" />
       <noun no="2" start="11" end="18" is-complex="false" syllables="2" />
       <noun no="3" start="48" end="53" is-complex="false" syllables="1" />
</nouns>
<!-- ... -->
```

For the use case of our example with two files, the raw tech article of listing 4.8 and the semantic text annotation file 4.9 we want to add additional information for single paragraphs, single sentences and nouns. When taking a look at the first child element "paragraph" of the parent markup "paragraphs", it can be observed that there are different attributes added to it:

- no : int
- start : int

- end : int
- fre : float
- avg-wps : float

The first attribute "no" is just an ascending order of the occurrence in the raw file. There are use-cases in which this can be useful, because XML itself dos not ensure the order when dealing with it. It strongly depends on the used tools and libraries which assits in the creation and parsing of the files.

The attributes "start" and "end" are both numbers which are indicating the location of the current paragraph in the content. As you can see the first "paragraph" in the semantic text annotation file is ranging from start-index 0 to end-index 251, the second from 253 to 504 and finally the third from 506 to 977. The numbers refer to the start of the raw text file, they are not in a relation to each other. In this example we are using the start-index and the end-index to indicate the location in the whole content. Another valid solution would be to use the start-index in combination with the length of the paragraph. It's up to the computer scientist what he prefers and what makes more sense to support the final application. If the only use case of this numbers is to find out the length of the paragraphs, the second solution would be the better one because one calculation less is needed ($end - start = length$). But the question should be: Is one subtraction for modern computer systems really that much effort that it is worth to think about it?

The attribute "fre" means the key performance indicator "Flesch Reading Ease". The Flesch Reading Ease is one of the most famous key performance indicators giving information about the difficulty of a reading passage (see [Fle48], [FJP51]). It has been developed by the Austrian-born Rudolf Franz Flesch, who emigrated to the United States of America during World War Two in the year 1938 (see [Wik15b]). The Flesch Reading Easy only works for texts in English language and the result is quite easy to calculate:

$$\Delta ws = (\frac{\text{nr of words}}{\text{nr of sentences}}) \tag{4.1}$$

$$\Delta sw = (\frac{\text{nr of syllables}}{\text{nr of words}}) \tag{4.2}$$

$$FRE = 206.837 - 1.015 * \Delta ws - 84.6 * \Delta sw \tag{4.3}$$

The formula to calculate the Flesch Reading Ease takes the ratio between the number of words in a text as well as the the number of sentences ("How long are the sentences?"), the ratio between the number of syllables as well as the number of words ("How long are the words?") and amplifies them by multiplying these values with fixed boosting variables (1.015, 84.6). These variables are the result of Flesch's researches. Both results are subtracted from a fixed start value, which is a result of Flesch's researches too. The result is the Flesch Reading Ease, which should be a float-score between 0 and 100, a list of all scores can be found in table 4.1.

Table 4.1: Results, Flesch Reading Ease (see [Bra10])

| **Max** | **Min** | **Comment** |
|---|---|---|
| 100.00 | 90.00 | Very Easy (Easily understood by an average 11-year old student) |
| 90.00 | 80.00 | Easy |
| 80.00 | 70.00 | Fairly Easy |
| 70.00 | 60.00 | Normal (Easily understood by 13 to 15 year old students |
| 60.00 | 50.00 | Fairly Difficult |
| 50.00 | 30.00 | Difficult |
| 30.00 | 0.00 | Very Difficult (best understood by college graduates) |

As we do now have a few ideas about the Flesch Reading Ease and how high class text analysis can look like, let's go back to our example with the raw tech article of listing 4.8 and the semantic text annotation file 4.9. To finalize the paragraphs and to get the link between the new knowledge about the Flesch Reading Ease and the example: The first paragraph has a Flesch Reading Ease of 41.8, the second of 48.8 and the third of 48.4. As it can be read in the table 4.1, all of these paragraphs are difficult to read and understand - the first one is a little bit harder to read than the others.

The last attribute of each paragraph is the "avg-wps", which is the "Average Words per Sentence". This is exactly the same as $\Delta ws$ of the Flesch Reading Ease-calculation. It is just another key performance indicator which can be used for the final application.

The next part of the semantic text annotation file 4.9 is the annotations of sentences. It is exactly the same hierarchical structure as the paragraphs-structure: the XML file contains a markup "sentences" which has child-elements "sentence". Each sentence in this example has the attributes:

- no : int
- start : int
- end : int
- fre : float
- wps : float

As it follows the same idea as explained for the paragraphs, we will not get to deep in this thematic. But we talk about one interesting thing which can be observed in this example: As it can be counted, the first two sentences are part of the first paragraph. Interestingly, the Flesch Reading Ease of both sentences (26.3, 34.1) is worse than both sentences in the paragraph combined (41.8). The first sentence has even the Flesch Reading Ease "Very Difficult", but in combination with the second sentence it's "Difficult". It is a sign that key performance indicators must be interpreted with caution.

The very last part of the example semantic text annotation file 4.9 is the list of the nouns. Each noun has the following attributes to work with:

- no : int
- start : int
- end : int
- is-complex : boolean
- syllables : int

The first attributes "no", "start" and "end" are equal to the attributes of sentences and paragraphs. As the Flesch Reading Ease is not significant for single words, the example has a few more attributes. The first attribute "is-complex" is a boolean value which gives information about the complexity of a word: If a word is complex (hard to understand), the value is *true*, otherwise it is $false$. The logic how this attribute can be calculated must be developed, but this is not part of this Master's Thesis. If an interested and motivated reader wants to implement such an attribute, the next attribute "syllables" can be a good foundation to start with.

That was one way how semantic text annotation by adding a stand-off file can be done. It is not necessary that the annotation file must be created as XML. Generally every file format is possible when there is an application which can analyse it. If there are only simple text annotations without any additional attributes, for example only the type of a word must be added (noun, verb, adjective, adverb, ...), the stand-off annotation file can be a CSV file (Comma-separated values, see Listing 4.10 for a short example) as well as a table in a database or something similar. Keep in mind that it is important that creating, reading and modifying can be done easily and in a performant manner.

Listing 4.10: Example of an random CSV Text annotation file

```
type;start;end
noun;0;5
noun;11;17
noun;37;42
verb;7;9
[...]
```

If using XML, even the structure is up to the computer scientist. The file can have a nested structure as in listing 4.11, but with the restriction that overlapping annotations are not possible, because the XML itself must be well-formed and valid. The benefit when creating an XML file with this structure is, that things that belong together are together (like a word in a sentence, a sentence in a paragraph, or similar). Maybe there is no scenario in the application that must be developed where overlapping is an issue, then it is worth thinking about it.

Listing 4.11: Example xml stand-off, nested semantic text annotation file.

```
<paragraphs>
 <paragraph no="1" start="0" end="251" fre="41.8" avg-wps="13.00">
  <sentences>
   <sentence no="1" start="0" end="200" fre="26.3" wps="32">
    <nouns>
         <noun no="1" start="4" end="9" is-complex="true" syllables="3" />
         [...]
    </nouns>
   </sentence>
   <sentence no="2" start="201" end="251" fre="34.1" wps="7">
    [...]
   </sentence>
   [...]
  </sentences>
 </paragraph>
 <paragraph no="2" start="253" end="504" fre="48.8" avg-wps="14.33">
 [...]
 </paragraph>
 [...]
</paragraphs>
```

### 4.1.3 Evaluation between in-line and stand-off

Summarized, semantic text annotation is about adding additional information to an existing raw text. In the last two sections of this chapter we talked about the different types of semantic text annotation and introduced two different types:

- in-line
- stand-off

Both types, in-line semantic text annotation and stand-off semantic text annotation enhance the revenue of raw textual content. Generally it strongly depends on the requirements what type the computer scientist should use to develop his application. To support the decision-process, it's good to summarize the benefits and drawbacks of both types and compare them with each other.

The table 4.2 shows a listing of different aspects, which can be useful to develop an application for adding semantic text annotation to a raw text. All of these aspects are rated in this table by how both types support these aspects and how they take them into account. Comparisons can be done in several ways, in this confront the single position ("How does X support Y?") are rated with checkmarks "✓". Up to three of them can be added to each point, which means that it is fully supported by this type. Zero checkmarks means, that there is no support.

Table 4.2: Compare in-line semantic text annotation with stand-off semantic text annotation

| | **In-line** | **Stand-off** |
|---|---|---|
| Adding additional Information | ✓✓✓ | ✓✓✓ |
| Usage of existing techniques (XML, CSV etc.) | ✓✓ | ✓✓✓ |
| Do not touch raw texts, leave them as they are | - | ✓✓✓ |
| Do not manipulate (edit) raw texts | ✓ | ✓✓✓ |
| Allow overlapping annotations | ✓ | ✓✓✓ |
| Use only one file for adding additional information | ✓✓✓ | - |
| Annotations are human readable | ✓✓✓ | - |
| Analytics of annotations | ✓ | ✓✓✓ |
| $\sum$ | 14 | 18 |
| Allocation | 43.75% | 56.25% |

Let's see what the results of this comparison are. First of all, as mentioned both types are supporting adding additional information to a raw text. Therefor, both types receive best marks. Theoretically, this aspect can be dismissed in this table, but as this is the most important aspect, it is displayed.

The daily work of every computer scientist should be to prevent him and his colleagues from reinventing the wheel again and again. Sometimes it is not easy to find things that already exist and fully fit into the application's context. In this case, it is quite easy to find things that supports the purpose of our application. As mentioned, in-line semantic text annotation is using XML, but it does not use the full function range of it. Stand-off semantic text annotation uses more of the function range of XML compared to in-line. When doing stand-off annotation, other techniques like CSV and so forth could be used, too. The is the reason why in-line receives just two out of three checkmarks and stand-off reaches best marks again.

The next two aspects on table 4.2 can be discussed together. When adding additional information as semantic text annotations on a raw text, it is like looking on and talking about a work of art in a gallery: It is forbidden to touch it. It is the same with semantic text annotations, it's better to not write stuff directly in the raw text, because there is always a danger that something goes wrong. When using in-line semantic text annotation naturally it can't be avoided that the raw text file gets touched and fitted with additional content. That is the reason why in-line does not receive any checkmarks. Stand-off annotations do not touch the raw content, that's minimizes the risk and that's a good thing which leads to the three checkmarks. Adding in-line annotations does not necessarily manipulate the content, but when using in-line annotations there can always be a conflict with special characters. These special characters must be either removed or escaped, which means that manipulation of the content itself is within the realms of possibility, that's the reason why in-line gets a reduction of checkmarks at the aspect "Do not manipulate (edit) raw texts".

The next aspect on the table 4.2 is "Allow overlapping annotations". The question is whether overlapping annotations are really important and necessary in the application or not. There

will be lots of applications in which overlapping annotations are not an issue. But if the decision is to use a technique which does not allow overlapping annotations and at a later time overlapping annotations must be possible, the computer scientist will have some troubles. Be careful when making the decision, changes at later times are more expensive. Overlapping annotations are possible with both, in-line and stand-off semantic text annotations. But it is tricky to handle them using in-line annotations and they need a bit or work to make it possible. Therefore it's rated with one checkmark. Stand-off can handle them without big restrictions.

Until now, stand-off semantic text annotations leads the comparison. But the next two aspects will probably help in-line semantic text annotations to make up leeway. When adding additional information to a raw text using stand-off semantic text annotations, it is not possible to use just one single file. There are at least two different files necessary, the raw text file and the annotation file containing the information about the first file. As mentioned in an other aspect, it is good that stand-off annotations do not touch the raw file. But when working with stand-off semantic text annotations, there must be ever at least two files handled. One file is of no worth when the other file is corrupt, missing or just changed. Therefore, stand-off annotations receive no checkmarks, but in-line annotations get best results.

Another aspect that is not optimal when using stand-off annotations to add additional information to a raw content is that the additional information is not human readable and interpretable. To read and interpret annotations as a human, there are at least two steps needed:

1. Take a look in the annotation file, remember the start-index and the end-index of an annotation
2. Take a look in the raw text file and start counting character by character until the start-index is reached

Although it is possible to do that manually, it is impractical. If thinking about a longer text, it is absolutely realistic to have annotations starting at position 137603 and ending at position 144833. Counting manually from zero to this start-position is nearly impossible to do it without any error. That is the big advantage of in-line semantic text annotations. Annotations are exact at the position in the raw text where they start and where they end. This makes it possible to read and interpret annotations as human being without the support of any computer system.

There are a lot of specific application cases where it is necessary to make analytics covering the whole annotations. A catchy example would be to make an analysis of how many nouns are in a document, how long the nouns are in average and how many words or characters are between the single nouns. If developing a single application that makes this analysis using stand-off annotation, a student studying informatics or something similar can develop this within a few hours. It will be enough to supply the student just with the annotation file, he will not have to know what's in the raw text or what the raw text is about. When the semantic text annotation is done using in-line annotations, it is not that easy, because the annotations are build around and in between raw content. It will be a lot of additional

work to do analytics.

Summarized, the in-line semantic text annotations reaches 14 checkmarks, the stand-off semantic text annotations reaches 18 checkmarks. Naturally a decision for a technique relates strongly to the intended purpose and the rating itself. But with the aspects and the rating, it would be better to use stand-off semantic text annotations because it has more benefits than in-line semantic text annotations.

# 5 Development of an application for semantic text annotation

In the previous chapter we analyzed semantic text annotations and introduced different types of them. There are mainly two different types of semantic text annotations: in-line and stand-off. We discussed the benefits and drawbacks of those and compared the two types with each other. The result of the comparison was that stand-off semantic text annotation is more conformable for further usages and does allow more efficient analysing and easier handling.

What we did until yet concerns building basic knowledge: we researched about persisting raw text as well as persisting additional information. What has not been mentioned yet is how applications can be developed which use those semantic text annotations.

In this chapter we want to create a simple application to show how such an application can be developed and what single parts are necessary. What we will do in this chapter is that we start from scratch and develop an application step by step:

- application's purpose, scope and definition
- read in raw data and stand-off annotations
- work with annotations in the application
- create and style output including annotations

## 5.1 Application's purpose, scope and definitions

What we want create is a simple application that makes it possible to display different types of semantic text annotations to the end-user. It does not matter what kind of semantic text annotations will be displayed and what kind of raw data will be fitted with additional information. The illustration 5.1 shows what should be developed within this chapter. On the left side there is a raw text file and a related stand-off annotation file. On the right side of the illustration there is the result, which should be an application displaying the raw text including the semantic text annotations with all available additional information.

The black box in the middle of the illustration is the scope of this application. The goal of this chapter is, that every computer scientist is able to develop his own application with the information that are provided in this chapter. It is not necessary that the output looks exactly like it will be done here, but the basic idea should be transmitted. We define the following conditions for our application:

- raw data is provided by a raw text file (".txt")

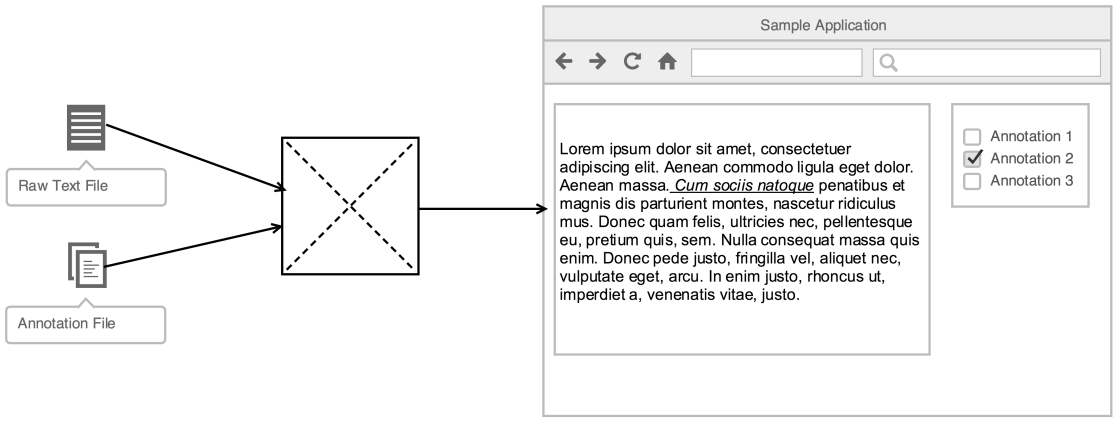



Figure 5.1: Illustration of application's workflow

- stand-off annotation file is provided as XML with predefined structure and hierarchy
- the output is a basic web-application using following techniques
  - HTML5
  - CSS
  - jQuery
- the complete processing is done using Java

This application should be a showcase open for further developments. Therefor, we restricted the usage of techniques to the minimum which is necessary to create a web-application. The usage of frameworks like "Play Framework", "Symphony" or "WebObjects" is renounced intentionally. But we decided to make this application to be a web-application because web-applications have a lot of benefits compared to common applications. A few of those are (see [DN14], or [Wik15c]):

- platform independent
- requirements are available of the box
- working together with shared data
- updates must only be done server-side

But as this Master Thesis is first not about web-applications and second not an advertisement for them, that should be enough information to start. The conditions and the context of our application are defined.

## 5.2 Read in raw data and stand-off annotations

In the last section we defined using Java for processing the data. Theoretically, this would be possible to be done with any other programming language too, but Java is good language

to explain what is happening within the code as most of computer scientists knows Java or a programming language that works and looks a bit like it. What will be developed within this chapter will not use specific Java-stuff which is only available in this language: it should be possible to implement the same with any other programming language.

The first thing we need for this application is a raw text file and a stand-off semantic annotation file. The raw text-file is visible in listing 5.1, the stand-off semantic annotation file is visible in listing 5.2.

Listing 5.1: Raw text file, rawText.txt, (content from [Wik15c])

```
A web-application or web-app is any program that runs in a web-browser. It is
created in a browser-supported programming language (such as the combination of
JavaScript, HTML and CSS) and relies on a web-browser to render the application.

Web-applications are popular due to the ubiquity of web-browsers, and the
convenience of using a web browser as a client to update and maintain
web-applications without distributing and installing software on potentially
thousands of client computers is a key reason for their popularity, as is the
inherent support for cross-platform compatibility. Common web-applications
include webmail, online retail sales, online auctions, wikis and many other
functions.
```

Listing 5.2: XML annotation file, stand-off.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<annotations>
        <paragraph start="0" end="238" fre="55.2" />
        <paragraph start="239" end="698" fre="17.0" />
        <sentence start="0" end="71" fre="73.2" />
        <sentence start="71" end="238" fre="42.5" />
        <sentence start="239" end="588" fre="5.0" />
        <sentence start="588" end="698" fre="28.1" />
        <noun start="2" end="17" syllable="5" />
        <noun start="21" end="28" syllable="2" />
        <noun start="36" end="43" syllable="2" />
        <noun start="59" end="70" syllable="3" />
        <!-- ... -->
</annotations>
```

The raw text-file in listing 5.1 contains the first part of an article about web-applications (see [Wik15c]), the content should be perfect to show and test the main functionality of the application. The annotation-file in listing 5.2 is a simple XML-file, with a root node “annotations” which encloses all annotations of different types. As mentioned in the previous chapter, a stand-off annotation file like this can have any structure. For this application we

make our life easier and use a flat hierarchy without any nesting. All of the annotations are on the same level and siblings of each other.

As it can be observed, all of the annotations are nodes of a specific type “paragraph”, “sentence” and “noun” and contains at least two attributes “start” and “end”. This attributes indicate the annotation’s position in the raw text file. The annotation’s start is counted from position zero in the raw-text as well as the annotation’s end, which is counted from zero too. The length of the annotation can be calculated by subtracting the start from the end. All other attributes can be added for the specific case and type of the annotation. The annotation of type “paragraph” for example has a specific attribute with name “fre” for the Flesch Reading Ease (see [Fle48]). The same attribute can be found in the annotation “sentence”. As the Flesch Reading Ease only makes sense when more than one word can be analyzed, the attribute “fre” can not be found in annotations of type “noun”. To have additional information in this example for all types of annotations, the annotation of type “noun” has an attribute “syllable”, which contains the number of syllables for a noun.

The contents in the listings 5.1 and 5.2 are the contents we need to have within our application. To read in the contents in our application, we need two import-functions, one for the raw text and one another for the XML-stand-off annotation file. Listing 5.3 shows a simple function importing raw text from a text-file. The import is quite easy and adds line by line of the file’s content to a variable of type String. When finished properly, the function returns the file’s content.

Listing 5.3: Import raw text from raw-text file (.txt).

```
/**
 * Import a raw text-file
 * @param rawTextFile
 * @return the file's content as String, null if error
*/
public static String readRawTextFromFile(File rawTextFile) {
 try {
  BufferedReader in = new BufferedReader(new FileReader(rawTextFile));
  String s = "";
  String line;
  while((line = in.readLine()) != null) {
   s+=line;
  }
  in.close();
  return s;
 } catch (Exception e) {
  e.printStackTrace();
 }
 return null;
}
```

Importing raw text of an ordinary text file is quite simple and should be able to be done by every computer scientist. The more challenging question is how an XML-annotation-file can be imported and processed. Listing 5.4 shows how an XML-annotation file like the one in listing 5.2 can be imported and processed. First of all, a new ArrayList containing objects of type "Annotation" is created. "Annotation" is a custom type which will be created in our application. Later on we will talk about this type, how it works and what attributes and functions it has. For this section it is enough to know that each instance of this type represents a single annotation like a noun which starts at position 2, or a sentence which starts at position 0 or a paragraph which starts on the same position, too.

Listing 5.4: Import stand-off annotation file (.xml).

```
/**
 * Readin Stand-off Annotation File
 * @param xmlFile
 * @return a list of objects of type Annotation
 */
public static ArrayList<Annotation> readStandOffAnnotationFile(File xmlFile) {
 ArrayList<Annotation> annotations = new ArrayList<Annotation>();
 try {
  DocumentBuilderFactory dbFactory = DocumentBuilderFactory.newInstance();
  DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();
  Document doc = dBuilder.parse(xmlFile);
  doc.getDocumentElement().normalize();

  NodeList annotationNodes =
     doc.getElementsByTagName("annotations").item(0).getChildNodes();
 for(int i = 0; i < annotationNodes.getLength(); i++) {
  Node annotation = annotationNodes.item(i);
  if(!annotation.getNodeName().equals("#text")) {
   NamedNodeMap attr = annotation.getAttributes();
   int startIndex = 0;
   int endIndex = 0;
   Map<String, Object> htmlAttr = new HashMap<String, Object>();
   Map<String, Object> dataAttr = new HashMap<String, Object>();

   for(int k = 0; k < attr.getLength(); k++) {
    if(attr.item(k).getNodeName().equals("start")) {
     startIndex = Integer.parseInt(attr.item(k).getNodeValue());
    } else if(attr.item(k).getNodeName().equals("end")) {
     endIndex = Integer.parseInt(attr.item(k).getNodeValue());
    } else {
     dataAttr.put(attr.item(k).getNodeName(), attr.item(k).getNodeValue());
    }
   }
```

```
   htmlAttr.put("class", "annotation " + annotation.getNodeName());

   Annotation newAnnotation = new Annotation(annotation.getNodeName(),
    startIndex, endIndex, Annotation.ELEMENT_TYPE_SPAN, htmlAttr, dataAttr);
   annotations.add(newAnnotation);
   }
  }
 } catch (Exception e) e.printStackTrace();
 return annotations;
}
```

The next function's part in listing 5.4 concerns read-in and parsing of the XML-file. Therefor, we use existing knowledge of different Java-libraries which are mostly available in the packages "org.w3c.dom.*" and "javax.xml.parsers.*". The type "DocumentBuilder" in combination with the "DocumentBuilderFactory" parses the raw XML-file and creates a nested structure of the types "NodeList" and "Node". This nested structure will be passed from the first to the last node. Each node contains the information of one markup, for example "<paragraph start='0' end='238' fre='55.2' />".

Next, the attributes of each node will be processed (for example start, end, fre, et cetera). We defined for our stand-off annotation file that each annotation markup contains at least the information about it's location "start" and "end". Therefor, this two information will be read-in directly and added to the variables "startIndex" and "endIndex". All additional attributes, like the "fre", will be added dynamically to the HashMap "dataAttr" with the name of the attribute as key and the value as value. As it can be observed, only the markup's attributes "start" and "end" must be available, there is no minimum or maximum or even a restriction on special attribute-names ore attribute-types. Every attribute can be added to the stand-off-annotation file 5.2 without touching the import-function in listing 5.4 again.

At the import function's end, a new object of type "Annotation" will be created and added to the list. As mentioned, we will talk about the type "Annotation" later on. But for now the "Annotation"'s constructor can be checked: As it can be seen, next to startIndex and endIndex, the constructor needs an element-type as well as a list of html-attributes "htmlAttr" and a list of data-attributes "dataAttr".

## 5.3 Work with annotations in the application

In the last section we introduced two different types of import-functions, one for the raw text and one another for importing and processing the stand-off annotation file from XML. As it is the computer scientist's daily work to read in raw text files in various forms and characteristics, this was quite easy. The other import function is a little bit more trickier because the content of an XML must not only be added line by line to a string variable - it must be preprocessed and various objects must be created.

When explaining the function in listing 5.4 we mentioned that we developed a custom type "Annotation" for representing annotations in our application. An object of type "Annota-

tion" is exactly one semantic text annotation referencing one location in the raw text. The type "Annotation" has the following variables:

- name : String
- startIndex : int
- endIndex : int
- startIndexReal : int
- endIndexReal : int
- htmlElementType : String
- htmlAttributes : Map<String, Object>
- dataAttributes : Map<String, Object>

The type "Annotation" should assist the conversion from annotations provided by the annotations file (for example see listing 5.2) and the raw text file (for example see listing 5.1) to a HTML-document that is a combination of both, raw text and semantic text annotations containing additional information about parts of the text. Listing 5.5 shows the first paragraph of listing 5.1 which has already been fitted with the first annotation of listing 5.2.

Listing 5.5: Add information with html-markups

```
<span class="annotation paragraph" data-fre="55.2" >
 A web-application or web-app is any program that runs in a web-browser. It is
 created in a browser-supported programming language (such as the combination of
 JavaScript, HTML and CSS) and relies on a web-browser to render the
 application.
</span>
```

As it can be observed in listing 5.5 the result of this conversion should be a wrapping html-element enclosing the part which should be annotated. In this example the html-element is a "span" with a html-class "annotation paragraph" and a HTML5-data-attribute "data-fre" containing the Flesch Reading Ease of the enclosed content.

After we presented the result, we are now able to take a deeper look in the type "Annotation". The attribute "name" is an information about the type of the annotation (like "paragraph", "sentence" or something similar). The variables "startIndex" and "endIndex" are the annotation's location in the raw and unmodified document. "Raw and unmodified" are in this case important terms, because this is the differentiation to the next two variables "startIndexReal" and "endIndexReal": This two variables are the location of the annotation the the results document at time of the adding process. If this differentiation between the first indicating variables and the second indication variables are not enough explanation, please take a look at listing 5.6.

Listing 5.6: Add markups step by step to html.

```
<!-- 1st step -->
The blue house has a green carpet.

<!-- 2nd step -->
<span class="annotation sentence">
 The blue house has a green carpet.
</span>

<!-- 3rd step -->
<span class="annotation sentence">
 The blue
 <span class="annotation noun">house</span>
 has a green carpet.
</span>

<!-- 4th step -->
<span class="annotation sentence">
 The blue
 <span class="annotation noun">house</span>
 has a green
 <span class="annotation noun">carpet</span>.
</span>
```

Table 5.1: Adding annotations step by step

| step | name | startIndex | endIndex | startIndexReal | endIndexReal | added |
|---|---|---|---|---|---|---|
| 1 | sentence | 0 | 34 | 0 | 34 | |
| | noun | 9 | 14 | 9 | 14 | |
| | noun | 27 | 33 | 27 | 33 | |
| 2 | sentence | 0 | 34 | 0 | 34 | ✓ |
| | noun | 9 | 14 | **43** | **48** | |
| | noun | 27 | 33 | **61** | **67** | |
| 3 | sentence | 0 | 34 | 0 | 34 | ✓ |
| | noun | 9 | 14 | 43 | 43 | ✓ |
| | noun | 27 | 33 | **99** | **105** | |
| 4 | sentence | 0 | 34 | 0 | 34 | ✓ |
| | noun | 9 | 14 | 43 | 43 | ✓ |
| | noun | 27 | 33 | 99 | 105 | ✓ |

In the “1st step” of adding semantic text annotations to the raw text, the text is just raw. As it can be observed in this sample, there are three annotations which are going to be added in the next three steps. Now have one eye look at the listing and the other eye look on the table 5.1. In the first step, the “startIndex” and “endIndex” of all three annotations are equal to their real index “startIndexReal” and “endIndexReal”. In the “2nd step” the first annotation “sentence” is added to the raw text enclosing the whole content. As it can

be seen, adding the other annotations to their initial location would not be correct anymore. To place them now at the correct position, the indexes must be updated. As the start of the markup "" with the length 34 has to be added to the start of the raw text, the location for the other to annotations starts and ends now 34 characters later. The "startIndexReal" and the "endIndexReal" will now be incremented by this value.

In the "3rd step", the next annotation "noun" will be added. The correct location of it is the just updated "startIndexReal" and "startIndexReal". This annotation is now enclosing the raw text "house". What will now have to be done is that the remaining annotations, which are not added yet will be updated. As the start-markup "" as well as the end-markup "" is added to the text before the remaining annotation starts and ends, we will need the full length of both (which is 38) to update the "startIndexReal" and the "endIndexReal" of the last annotation. In the "4th step" the last remaining annotation "noun" is added to the raw text enclosing the word "carpet". As there is no remaining annotation, the complete process is done.

This work is done by a function "updateAnnotation" which is part of the type "Annotation". The implementation of this function can be observed in listing 5.7. This function does exactly what have been done in the last example. If a new annotation is added, this function checks if the "startIndexReal" and "endIndexReal" must be updated or not. Therefor, it compares the initial location "startIndex" and "endIndex" of the current annotation with the added annotation "addedAnnotation". If the start-markup and/or the end-markup of this added annotation changes the real index of the current annotation, the real index-variables "startIndexReal" and "endIndexReal" will be updated by incrementing the length of the markups.

Listing 5.7: Create HTML-markups

```
/**
 * Update the startIndexReal and endIndexReal depending where the added
 * Annotation is placed in the raw text
 * @param addedAnnotation
*/
public void updateAnnotation(Annotation addedAnnotation) {
 if(addedAnnotation.startIndex()<=startIndex()) {
  _startIndexReal += addedAnnotation.htmlAnnotationStart().length();
  _endIndexReal += addedAnnotation.htmlAnnotationStart().length();
 } else if(addedAnnotation.startIndex()<endIndex()) {
  _endIndexReal += addedAnnotation.htmlAnnotationStart().length();
 }

 if(addedAnnotation.endIndex()<=startIndex()) {
  _startIndexReal += addedAnnotation.htmlAnnotationEnd().length();
  _endIndexReal += addedAnnotation.htmlAnnotationEnd().length();
```

```
 } else if(addedAnnotation.endIndex()<endIndex()) {
  _endIndexReal += addedAnnotation.htmlAnnotationEnd().length();
 }
}
```

The next variable of the type “Annotation” is “htmlElementType” which was already visible in the last example 5.6 as element type “span”. The “htmlElementType” can probably be every allowed element type which is available in HTML, for example “div”, “span”, “p”, “b” and so forth. It is up to the application and the computer scientist what element type will be used: for our application’s purpose “span” does the job. Because the list of potential useful HTML element types is restricted, the most important types are available as static variables in the type “Annotation”:

- public static final String ELEMENT_TYPE_SPAN = ’span’;
- public static final String ELEMENT_TYPE_DIV = ’div’;
- public static final String ELEMENT_TYPE_B = ’b’;
- public static final String ELEMENT_TYPE_I = ’i’;

The last variables of the type “Annotation” are the “htmlAttributes” and the “dataAttributes” which are both of type “Map<String, Object>”. The basic idea behind this two maps is that attributes can be added to the HTML markups in the result. There is a differentiation of attributes that can be added to markups in HTML5: those attributes which are part of the language HTML and those attributes that can be added as additional data-attributes containing different types of data. Attributes that are part of the language HTML are for example “class”, “style”, “id” or something similar. A complete reference of HTML and the allowed markups can be seen in [Hic12]. All other attributes containing additional information about the annotation (like Flesch Reading Ease) are added to the map “dataAttributes”. All this attributes will be preprocessed to “data-KEY=’VALUE’”, for example: “data-fre=’53.3’”. Listing 5.8 shows the two functions which are creating the start- and the end-markup for the annotation. This two functions are part of the type “Annotation”.

Listing 5.8: Create HTML-markups

```
public String htmlAnnotationStart() {
 String html = "<" + htmlElementType + " ";
 if(_htmlAttributes!=null) {
  for(String key : htmlAttributes.keySet()) {
   html+= key + "=\"" + htmlAttributes.get(key).toString() + "\" ";
  }
 }
 if(_dataAttributes!=null) {
  for(String key : dataAttributes.keySet()) {
   html+= "data-"+ key + "=\"" + dataAttributes.get(key).toString() + "\" ";
  }
```

```
 }
 html += ">";
 return html;
}

public String htmlAnnotationEnd() {
 return "</"+htmlElementType+">";
}
```

## 5.4 Create and style output including annotations

In the last section we introduced our type "Annotation" and explained what kind of variables and what main function this type has. We talked in detail about the concept of incrementing indexes and created a function doing this work for us. As a result we introduced the functions which creates the HTML start-markup and the end-markup which should enclose content of the raw text in the resulting HTML. All we talked about in the previous sections were single parts of the final result. Now it is time to assemble the puzzle.

What will be part of this section is that we first create a Java-class which is doing the complete process functions by using our custom type "Annotation" and creates a HTML-document which contains all semantic text annotations in the content as HTML-markups. In the second part of this section the markups should be made a little bit interactive and visible to the end-user by styling it with CSS and creating a simple user interface using jQuery.

As mentioned, what we need is a new Java-class which is doing the complete process function. For this we create a new Java-class "HTMLAnnotator", which contains exactly two functions:

- public static String createAnnotatedHtml(
  String rawText, ArrayList<Annotation>annotations)
- private static ArrayList<Annotation>_cleanOverlappingAnnotations(
  ArrayList<Annotation>annotations)

The first function has an access specifier "public" which indicates that this method is visible for each computer scientist who wants to use our class, "static" means that it can be used without instancing this class. If a computer scientist wants to use this function, he would just call "HTMLAnnotator.createAnnotatedHTML(rawText, annotations)" and gets an annotated HTML as result. The function's content itself is a blackbox for further computer scientists, but not for the readers of this Master's Thesis, which will now take a look in this function.

Listing 5.9: Implementation of createAnnotatedHTML

```
/**
 * Create an html, that contains the raw text fitted
 * with rendered annotations of the List
 * @param rawText : a raw Text
 * @param annotations : a List of Annotation's
 * @return an html
*/
public static String createAnnotatedHtml(String rawText,
                                         ArrayList<Annotation> annotations) {
 String htmlText = rawText;
 annotations = _cleanOverlappingAnnotations(annotations);
 ArrayList<Annotation> notAddedAnnotations =
          (ArrayList<Annotation>) annotations.clone();

 for(Annotation newAnn : annotations) {
  notAddedAnnotations.remove(newAnn);
  for(Annotation anAnn : notAddedAnnotations) {
   anAnn.updateAnnotation(newAnn);
  }
  String htmlPartBefore =
   htmlText.substring(0, newAnn.startIndexReal());
  String htmlPartEnclosed =
   htmlText.substring(newAnn.startIndexReal(), newAnn.endIndexReal());
  String htmlPartAfter =
   htmlText.substring(newAnn.endIndexReal(), htmlText.length());

  htmlText = htmlPartBefore +
             newAnn.htmlAnnotationStart() +
             htmlPartEnclosed +
             newAnn.htmlAnnotationEnd() +
             htmlPartAfter;
 }

 return htmlText;
}
```

Listing 5.9 shows what is in the blackbox of the function's implementation "createAnnoatedHTML(rawText, annotations)". First of all, the raw text of the input parameter "rawText" is added to a new variable "htmlText". The content of this new variable will be fitted with HTML-annotations step by step within this function. Next, the annotations will be put as input parameters to the second function of this class "_cleanOverlappingAnnotations(annotations)". As this is a core functionality of this class, we will take a look later on in this function. But to provide a little piece of information: This function explores the list of annotations for overlapping annotations. Overlapping annotations will be splitted and nested to kill overlappings of them. The result of this function

is an ArrayList of “Annotation”’s which do not overlap each other: it will be possible to generate valid and well-formatted HTML.

This ArrayList “annotations” will be cloned and saved to the variable “notAddedAnnotations”. The array will contain all annotations, which do not have been added and which “startIndexReal” and “endIndexReal” must be updated when adding an annotation to the resulting HTML. At the beginning, all annotations are in the array. While iterating annotation for annotation of the ArrayList “annotations”, first the current annotation will be removed from the “notAddedAnnotations”. This prevents the current annotation being updated. All other annotations will be updated by calling the function “anAnn.updateAnnotation(newAnn)”, how this function works can be observed in listing 5.7.

After removing the current annotation “newAnn” from the ArrayList “notAddedAnnotations” and updating all other not added annotations, it is time to add the current annotation “newAnn” to the resulting HTML. This is a little bit tricky, but should not seems like magic for experienced computer scientists. It is important that we keep in mind that we have to add for each annotation two parts to the raw text: the start-markup and the end-markup. This two markups will enclose the part of content they annotate. To do that, we have to split the “htmlText” in three parts: “htmlPartBefore”, “htmlPartEnclosed” and “htmlPartAfter”. In most cases, all of this parts should contain content, there are only three different situations in which a part can be empty:

- the new annotation starts at 0 ⇒ htmlPartBefore = ”
- the new annotation contains no content ⇒ htmlPartEnclosed = ”
- the new annotation ends at end ⇒ htmlPartAfter = ”

A combination of different situations is realistic, for example if an annotation starts at position 0 and ends at the end of the raw text. Theoretically, a combination of all three situations is possible when adding an annotation starting at position 0 and ending at position 0 in an empty text. But this is not really a practicable situation.

The whole process of adding a new annotation to a text ends with assembling all parts together to one single string. Therefor, the three parts will be fitted with the markups: that’s it. If this is repeated for each annotation, the function returns a HTML containing raw text and HTML markups.

#### Handling overlapping annotations in HTML

As mentioned, we want to take a deeper look how overlapping annotations can be handled when rendering annotations in HTML for display purpose. Remembering back when talking about saving annotations using in-line semantic text annotations, we resolved the problem easily by recommending the computer scientist to use stand-off semantic text annotations instead of in-line. As this was for saving semantic text annotations a proper workaround, this tweak will not work when rendering it to HTML. We have to get rid of overlapping annotations, for this we introduce the custom method “_cleanOverlappingAnnotations(annotations)”, which is already used, but not listed in detail within this chapter. As this method is a little

bit longer than other methods which have been discussed in the scope of this Master's Thesis and it is not fundamental but only helpful, the function's content is not listed but the single steps will be discussed.

Roughly explained, this method does the following two steps:

- check array if two annotations overlap each other
- create four new annotations, two for the first annotation and two for the second annotation
- kill the two old overlapping annotations

Take a look at the illustration 5.2, which shows this process graphically with one example. As it can be seen, the sentence contains two overlapping annotations A and B.

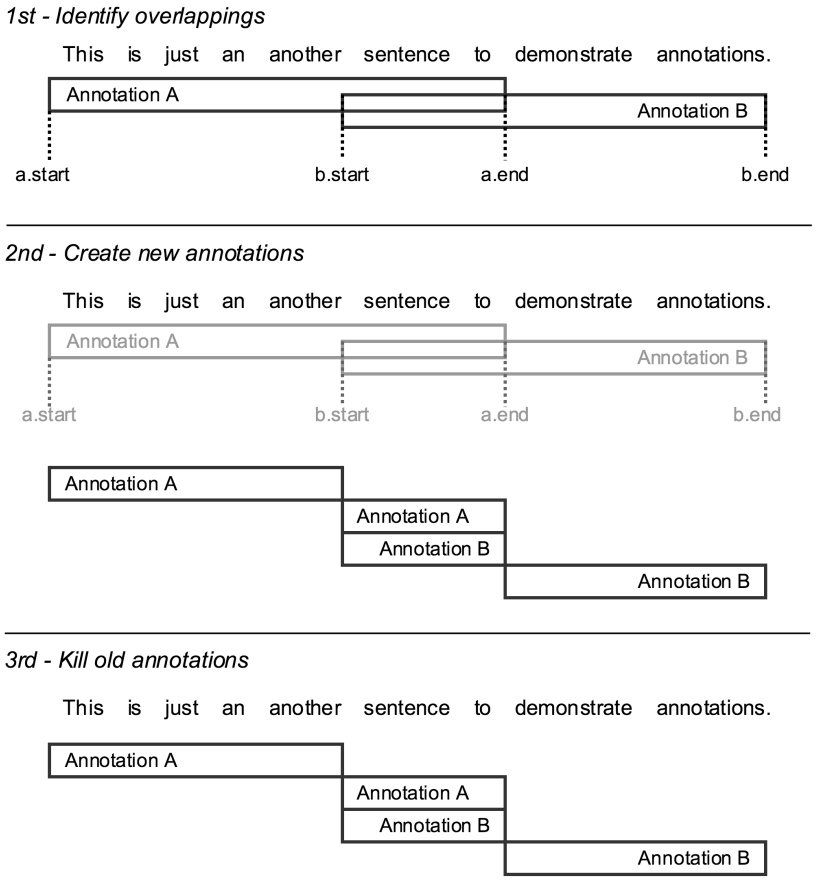



Figure 5.2: Remove overlapping annotation, graphical workflow.

The problem with this overlapping annotation is, that the annotation B starts within the annotation A and ends outside of it. Interpreted the other way round: Annotation A starts outside of annotation B and ends within it. What will be done next is, that four new annotations will be create with the following attributes:

1. Annotation A:
   - start: a.start
   - end: b.start
2. Annotation A:
   - start: b.start
   - end: a.end
3. Annotation B:
   - start: b.start
   - end: a.end
4. Annotation B:
   - start: a.end
   - end: b.end

The last step in illustration 5.2 shows the final action of this cleaning-process: The two old overlapping annotations will be killed. This part should be fixed now. This is exactly what the method "_cleanOverlappingAnnotations(annotations)" does.

#### Graphical user inferface

Our application is taking shape: we created a custom type "Annotation" that represents one annotation within our Java-application. Further, we implemented methods that can import the raw text file as well as the XML annotation file and creates a String containing the raw text and an ArrayList containing objects of type "Annotation", created with the information of the XML annotation file. Finally, we created a Java-class "HTMLAnnotator" that has one public method to render the objects of type Annotation as HTML and fits it into the raw text. The method uses an other method in the same class HTMLAnnotator that cleans overlapping annotations before fitting them into the raw text.

The result of this whole process is something astounding: A well-structured HTML-content as it can be seen in listing 5.10.

Listing 5.10: Result of creating process: an annotated HTML

```
<span class="annotation paragraph" data-fre="55.2" ><span class="annotation
sentence" data-fre="73.2" >A <span class="annotation noun" data-syllable="5"
>web-application</span> or <span class="annotation noun" data-syllable="2"
>web-app</span> is any <span class="annotation noun" data-syllable="2"
>program</span> that runs in a <span class="annotation noun" data-syllable="3"
>web-browser</span>.</span><span class="annotation sentence" data-fre="42.5" >
It is created in a browser-supported <span class="annotation noun"
data-syllable="6" >programming language</span> (such as the <span
class="annotation noun" data-syllable="4" >combination</span> of <span
class="annotation noun" data-syllable="3" >JavaScript</span>, <span
class="annotation noun" data-syllable="4" >HTML</span> and <span
class="annotation noun" data-syllable="3" >CSS</span>) and relies on a <span
class="annotation noun" data-syllable="3" >web-browser</span> to render the
<span class="annotation noun" data-syllable="4"
>application</span>.</span></span>
<!-- ... -->
```

In the computer scientist's view, the result may be impressive. But all that glitters is not gold: An end-user can not use this result to work with, because it is hard to read and nearly impossible to interpret or understand. To make this piece of code read- and understandable, support by computer science is necessary. What will be part of this section is the creation of a simple graphical user interface to display such piece of code in a way, the annotations are readable for the end-user. For our application, let's define the specifications and acceptance criteria:

- raw text is still readable
- annotations are visible by coloring
- each type of annotation has its own color
- all additional information of an annotation can be seen by clicking on the annotation
- annotations can be displayed or hidden by the end-user itself

The first acceptance criteria of our application is quite easy to implement. The whole HTML annotated content of listing 5.10 will be enclosed by a simple HTML construct. As this content is already listed, only a very small part of it that is visible within the HTML construct of our application in listing 5.11.

Listing 5.11: Application's HTML

```
<!DOCTYPE html>
<html>
<head lang="en">
 <meta charset="UTF-8">
 <title>sample application</title>
```

```
 <script src="jquery-1.11.3.min.js"></script>
 <script src="annotating.js"></script>
 <link href="style.css" type="text/css" rel="stylesheet"/>
</head>
<body>
 <div id="content">
  <!-- ANNOTATED HTML -->
  <span class="annotation paragraph" data-fre="55.2" ><!-- ... -->
 </div>
 <div id="toolbar">
  <h2>Annotations</h2>
 </div>
</body>
</html>
```

As it can be observed, the content is enclosed by its own div with id "content". Additionally, the body contains an another div with id "toolbar". This construct will present the content to the end-user, but without any styling. What the end-user will see in his webbrowser is just the raw text. The next step is to make application displying colors indicating the annotation. The simplest way to do this would be to add an CSS with styling commands for each type of annotation, like it can be seen in listing 5.12.

Listing 5.12: Simple coloring of annotations using CSS

```
.annotation .paragraph {
 background-color: #ffff00;
}

.annotation .sentence {
 background-color: #cecece;
}

/* ... */
```

But as this is not a generic way and does only support known annotations to be displayed in our application, we are going to use jQuery, a javascript library, to color our annotations generically. To do this, let's take a look at the jQuery/javascript-code in listing 5.13.

Listing 5.13: Generic coloring of annotations.

```
var COLORLIST_ANNOTATIONS = ["#D5FFC5", "#FFFDC6", "#FFDCAC", ...];

$(document).ready(function() {
 var classes = [];
```

```
$(".annotation").each(function() {
 if($.inArray($(this).attr("class").replace("annotation ",""), classes)==-1) {
  classes.push($(this).attr("class").replace("annotation ", ""));
 }
});

for(var i = 0; i < classes.length; i++) {
 $("<style type='text/css'> ."+classes[i]+"{
    background-color:"+COLORLIST_ANNOTATIONS[i]+";}
   </style>").appendTo("head");
 }
});
```

The jQuery/javascript-code in listing 5.13 first creates a global array which contains a list of hexadecimal color codes that should be used. Theoretically, it would be possible to use random color codes, but there is the danger that some background-colors make black text hard to read, or colors are to equal to each other. The list of color codes contains only "light" colors that make the text easy to read. When the document is ready, the first thing is that all classes, which contains an "annotation" (for example "annotation paragraph", "annotation sentence", ...) will be loaded. Each class will be checked for the second part of the class name, which indicates the type of the annotation. This type will be put in the array "classes" if the array do not already contains this class. In the case of our example content, the list of classes would contain "paragraph", "sentence" and "noun". For all of this classes a CSS rule, which contains one of the colors of the COLORLIST_ANNOTATIONS-array as background-color will be created in the head of the HTML-document. When taking a look in the result, the end-user should see a white page, which contains text with colored annotations.

The next acceptance criteria of our application is waiting impatiently to be implemented: The end-user should be able to see additional information of an annotation (for example the Flesch Reading Ease, nr of syllables, ...) when clicking or hovering it. Let us first take a look at the rendered and annotated HTML-content in listing 5.10. We added additional information to the enclosing HTML-elements "span" by using the HTML5-improvement "data-* attributes". This in combination with the name of the annotation's type should be visible to the end-user. We could now make our life easy and use the title-functionality of HTML. This would mean that all data is put in an additional attribute "title" to each span, like it is visible in listing 5.14.

Listing 5.14: Tooltip using title-attribute

```
<span class="annotation paragraph" data-fre="55.2" title="paragraph: fre=55.2">
<!-- ... -->
```

The result would be, that it is visible in most browsers as hovering event. Unfortunately there are at least two drawbacks:

- it is not possible to style them without additional tools/tweaks/...

- title-hovering only works on desktop-browsers and do not work with touch-events by default

The second one would be forgivable, the first one is really worse and not nice for the end-user's experience. The easy way will not work properly, so we have to make a good solution and create our own info-popup. As the types of annotations are generically, the additional attributes should be generic, too. We have to create a jQuery/javascript-code, which creates an info-popup containing all additional information about an annotation when clicking on it. Take a look in the executive summmary in listing 5.15. This code listing shows an example what should be created when click on an annotation. In combination with a little bit CSS-styling, this will look like an info-popup.

Listing 5.15: Info-Popup with HTML

```
<div class="infopopup">
 <header>paragraph</header>
 <ul>
  <li>fre: 55.2</li>
 </ul>
</div>
```

What we need is a function that automatically generates such a piece of HTML-code for any annotation with any type of additional information and adds it to the page. Styling will be done using CSS. Such an info-popup should only be created when clicking an annotation and should be removed/hided when leaving the annotation with the mouse or any other leave-gesture (like double-click/tap, pressing the "Any"-Button, or similar). In the listing 5.16, such a function is implemented using jQuery/javascript.

Listing 5.16: Info-Popup creation using jQuery/javascript

```
$(".annotation").click(function(event) {
 if(!$(this).hasClass("disabled")) {
  event.stopPropagation();
  var infopopup = "<div class=\"infopopup\" style=\"top: "+
    (event.pageY+25)+"px; left: "+event.pageX+"px\">";
  infopopup += "<div class=\"header\">"+
    $(this).attr("class").replace("annotation ","")+"</div><ul>";

  for (var key in $(this).data()) {
   infopopup += "<li>" + key + ": " +
      $(this).data()[key] + "</li>";
  }
  infopopup += "</ul></div>";
  $("body").append(infopopup);
```

```
 }
});
$(document).mousemove(function() {
 $(".infopopup").remove();
});
```

The whole listing is splitted in two different user-interactions: The first one is the click on an annotation: as it can be observed, the function starts creating step by step the info-popup's HTML. The first exciting part is that the click-position (X and Y) is read-out to position the info-popup absolutely in the HTML-document. This means that the info-popup occurs exact at the position where the click-event occurred. The second exciting part of it can be seen in the middle of the function, when the data-attributes will be read-out and printed using key/value as list. The second interaction will be fire when moving the mouse again: The function removes all occurrences of class "infopopup" in the HTML. The info-popup is away.

The last acceptance criteria is that the end-user should be able to show and/or hide annotations to support better analysis and make a better and clearer presentation of the data. It should be a basic filter. We already prepared the HTML-file of our application as it can be seen in 5.11: There exists a div with the id "toolbar" as sibling of the div which encloses our annotated content (id "content"). The div "toolbar" contains only a heading, the rest has to be implemented. We want to carry on a generic way within our application, that means that any markups with any attributes can be used. Remembering back, the creation of the annotated HTML allows all types of annotations as long as there is an existing start-index and an existing end-index defined for each annotation. The coloring of the annotations itself in the application is generic as well, only the set of available colors is predefined to use only colors, which do not make text hard to read.

Our filter should be created generically, showing all different types of annotations and allows the user to show or hide them separately. We want to implement this filter by using the HTML-input type "checkbox": each type of annotation (for example "paragraph", "sentence" or "noun") should have its own checkbox. There are two parts which must be implemented: the first part is the creation of the checkboxes itself, the second part are the actions for show/hide the annotations when checking or unchecking the checkboxes. Listing 5.17 shows how the first part, the creation of the checkboxes, can be done using jQuery/javascript in a generic way.

Listing 5.17: Creation of generic toolbar using jQuery

```
var COLORLIST_ANNOTATIONS = ["#D5FFC5", "#FFFDC6", "#FFDCAC", ...];

function initToolbar() {
 var classes = [];
 $(".annotation").each(function() {
  if($.inArray($(this).attr("class").replace("annotation ",""), classes)==-1) {
   classes.push($(this).attr("class").replace("annotation ", ""));
  }
```

```
 });

 for(var i = 0; i < classes.length; i++) {
  addCheckboxToToolbar(classes[i], COLORLIST_ANNOTATIONS[i]);
 }
}

function addCheckboxToToolbar(name, color) {
 var container = $('#toolbar');
 $('<div>').appendTo(container);
 $('<input />', { type: 'checkbox', id: 'cb_anno_'+name, value: name,
   checked: 'checked', class: 'filter-annotation-cb'}).appendTo(container);
 $('<label />', { 'for': 'cb_anno_'+name, text: name, style: 'border-bottom: 2px
 solid ' + color }).appendTo(container);
 $('</div>').appendTo(container);
}
```

The result of the function should be a list of checkboxes, which should look like listing 5.18. For each type of annotation exact one checkbox should exist for filtering. Each checkbox should get its own label containing the name of the annotation, the label itself will be underlined with the color of the annotation. This is done by the function "initToolbar()" in combination with the already known array of available colors "COLORLIST_ANNOTATIONS" and the helper function "addCheckboxToToolbar(name,color)".

Listing 5.18: List of checkboxes in toolbar

```
<div id="toolbar">
 <h2>Annotations</h2>
 <div>
  <input type="checkbox" id="cb_anno_paragraph" value="paragraph"
       checked="checked" class="filter-annotation-cb" />
  <label for="cb_anno_paragraph" text="paragraph"
       style="border-bottom: 2px solid #D5FFC5" />
 </div>
 <!-- ... -->
</div>
```

Listing 5.18 shows the result of the creation process: A list of checkboxes with colored labels. The very last but very important thing what is missing to fulfill all requirements of our application is the interactivity of those checkboxes to make them work as a filter. To to this, we must implement two parts: an observer that checks if a checkbox have been checked or unchecked, and an action that makes annotations in the content visible or not. Annotations will be visible to the end-user by adding a background-color to the enclosing element. The easiest way is to remove the background-color of an annotation when it should be hidden, and add it again when it should be visible. To make this a little bit more efficient, we create a new CSS-class with the name ".disabled" which looks like listing 5.19. This CSS-class defines a rule that says, if an element have the class ".disabled", the background-color should be transparent. "!important" forces this rule to be applied in each case.

Listing 5.19: Class disabled, CSS rule for hiding annotations

```
.disabled {
 background-color: transparent!important;
}
```

The observe-function in listing **??** triggers all change-actions (check or uncheck) of the filter-checkboxes with the class ".filter-annotation-cb". Each time a checkbox becomes checked or unchecked, this observe-function "fires" and add the class ".disabled" to the annotations, which forces their background-colors to be invisible or remove this class to revert their background-colors to their initial value.

Listing 5.20: Observer for generic checkboxes in toolbar.

```
$(document).ready(function() {
 $(".filter-annotation-cb").change(function() {
  if($(this).is(":checked")) {
   $("." + $(this).val()).removeClass("disabled");
  } else {
   $("." + $(this).val()).addClass("disabled");
  }
 });
});
```

That should be enough to pass the last acceptance criteria Annotations can be displayed or hidden by the end-user itself. We talked a lot about our application, what functions it should have and how this application is implemented. But how does the result look like? What does the end-user will see when using our application? Let's lift the curtain: illustration 5.3, 5.4, 5.5 and illustration 5.6 show four screenshots of the final result.

As it can be observed, the application have two main parts, the left one shows the content in an annotated or non annotated state. The right part contains the generic filter that allows the end-user to make single annotations visible of hidden. The annotations and the associated filters have the same generic color. Clicking on any visible annotation opens an info-popup at the location of the click-event containing a generic List of additional attributes.

With this application in combination with the Java-application for creating the annotated HTML-content, it is easily possible to use all types of annotations and present them with simple interactivity to the end-user. For simplicity we used an easy to understand text in this example, but as this whole application is generic, it not depends on the text nor on the types of annotations.

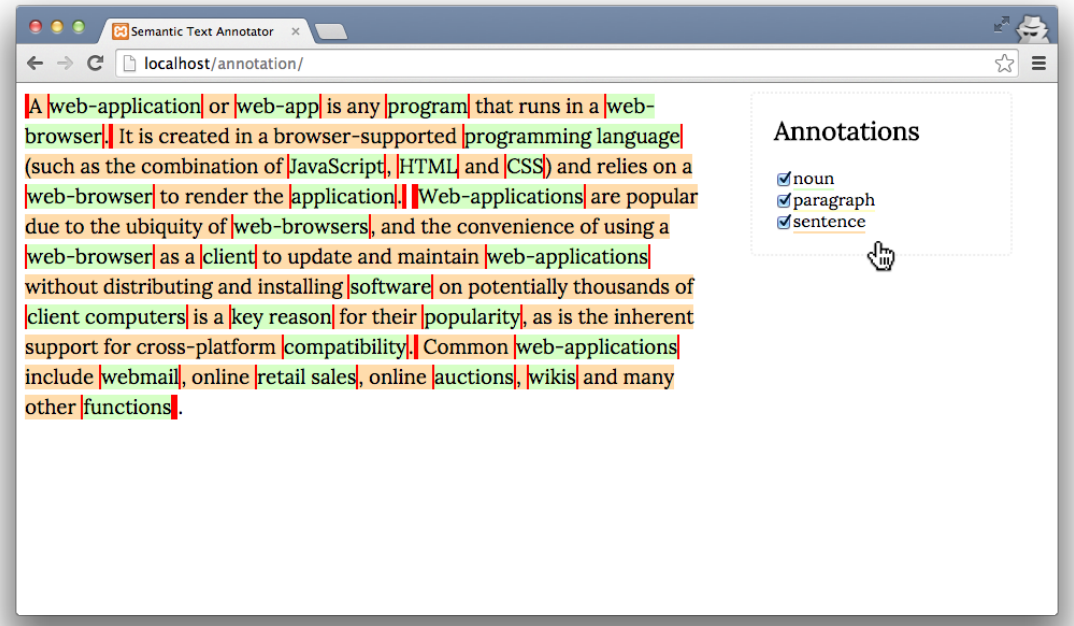



Figure 5.3: Screenshot application: All annotations visible.

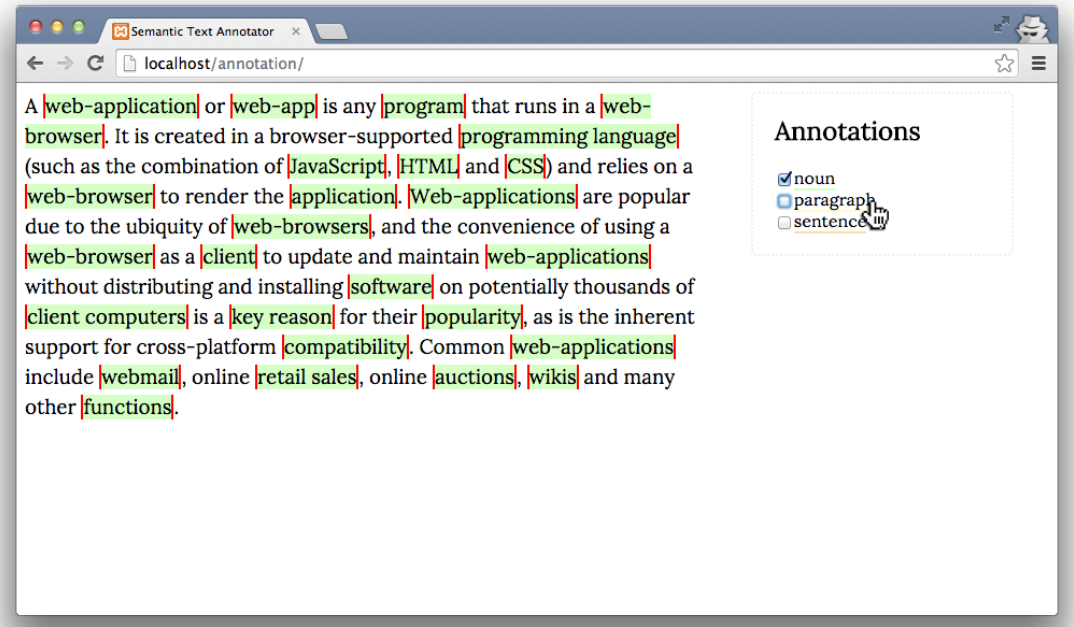



Figure 5.4: Screenshot application: Only annotations “noun” visible.

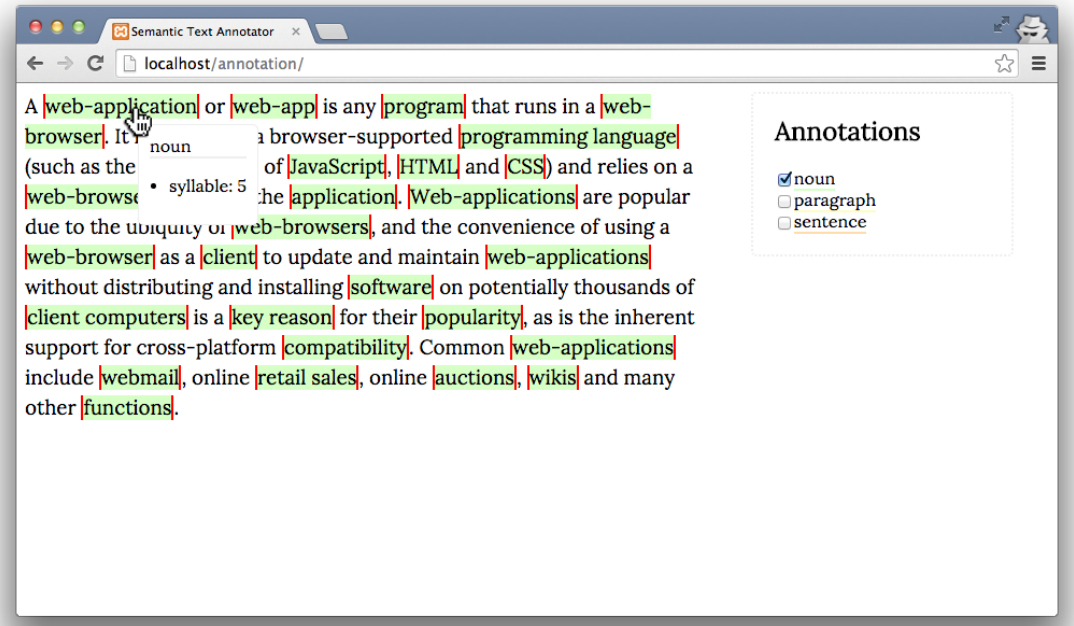



Figure 5.5: Screenshot application: info-popup “noun” visible.

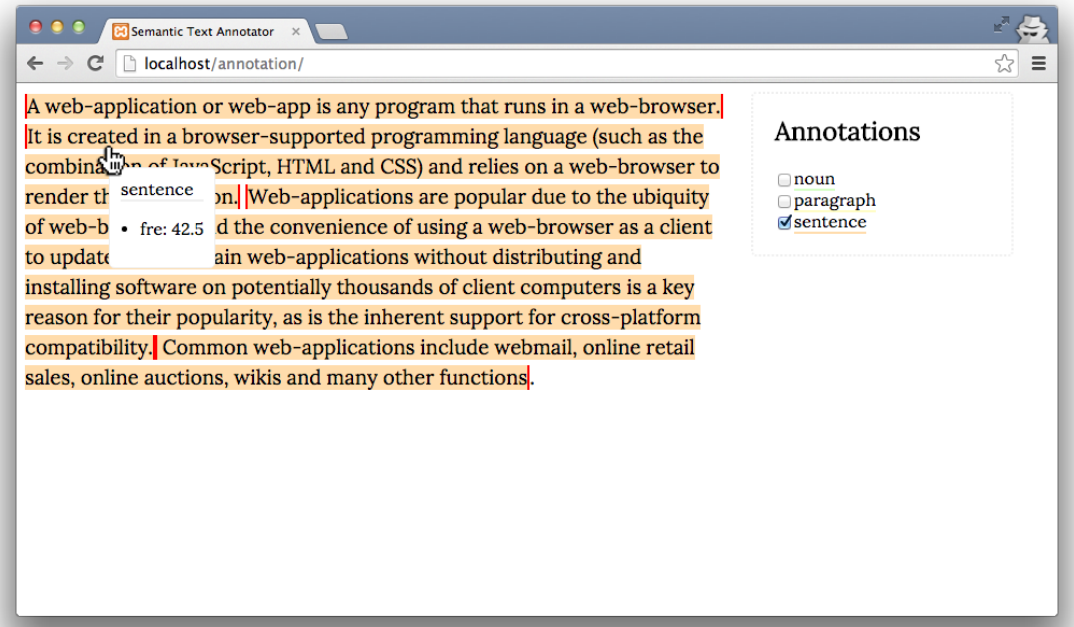



Figure 5.6: Screenshot application: info-popup “sentence” visible.

# 6 Summary and outlook

Now the time has come to open a final chapter of the Master's Thesis. Within this final chapter you can find a summary of the work's previous chapters with the aim to create a comprehensive and big picture of what have been researched during the previous pages.

A common way to summarize such a work is to bring back to mind what the intent and the purpose of this Master's Thesis has been. The aim of this thesis was to answer the following initial research questions:

- What kind of legal texts exist in the German legislation?
- What is a way to implement a generic importer for legal texts that can easily be adapted?
- What kind of semantic text annotations exist and what are benefits and drawbacks of those?
- How to persist semantic text annotations in order to access them for further semantic processing?

As the attentive reader should know, this are exactly the research questions we mentioned at the end of the first chapter.

The second chapter of this work was devoted to the first research question "What kind of legal texts exist in the German legislation?". To answer this research question we took a closer look on the German legislation and pointed out that basically three types of different legal texts do exist in the German legislation: legislative documents, judiciary documents and legal literature. For each one of the first two types we could find two different kinds: law and delegation as legislative documents, judgement and decision as judiciary documents. All these types and kinds have been transferred from "real"-world by abstraction to data models. The knowledge of all data models have been aggregated to one big single data model which maps all important types and kinds of legal texts that exist in the German legislation.

We used this aggregated knowledge to focus on the next research question "What is a way to implement a generic importer for legal texts that can easily be adapted?" in the third chapter. We analyzed how an importer should be developed to allow the import of different types of legal texts. To start this development, we analyzed most important sources of different legal documents and talked about benefits and drawbacks of the different sources and about different file formats they provide. As result, we presented a model that can easily be adapted for new sources and new file formats. In doing so, we considered very important to show interest in the limitations of generic importers. When developing in a "generic" way, abstraction and reduction are always issues. These issues generate limitations and show that not everything is possible, for example: some data get dropped when they get imported with

a generic and not with an individual or manual import.

The next big topic of this Master's Thesis addresses the issue semantic text annotation. This topic took the greatest attention of the complete work. When working on chapter four, the central issue was answering the research question "What kind of semantic text annotations exist and what are benefits and drawbacks of those?". Answering this research question, we went hand in hand with answering the last research question "How to persist semantic text annotations in order to access them for further semantic processing?". When researching about different types of semantic text annotations, we explored that all semantic text annotations can be separated in two different types: those semantic text annotations which will be embedded in the raw text and those which are added as an additional file which contains the annotations themselves. To call them by name: in-line semantic text annotations and stand-off semantic text annotations. We talked about the benefits and drawbacks of those and compared them to each other. The result showed that in our case stand-off semantic text annotations have more revenue than in-line semantic text annotations. But it is up to the application case which type of annotation should be used. Theoretically, both types can be useful because they have their own charm.

During the research of various types of semantic text annotations we also talked about the persistence of those. We mentioned as well, that the choice of the type sets the way of persisting the semantic text annotations. When deciding to use in-line semantic text annotations, all annotations will fit in the raw text directly. In this case, it is up to the application and depending on what kind of persistence the application uses to persist the raw text. The persistence of the annotation does not need any special handling because it is a by-product of the raw text. When deciding to use stand-off semantic text annotations, a second file is necessary for saving the annotations of the first file. Theoretically, the content of this file can consist as any format, but XML will be a good choice for further processing. If the raw text is persisted as ordinary text file, it will be good, in addition save the annotations as a file. Using the same way for persistence will make further work easier. If the raw text is part of a database, the annotations can be saved as XML in the database. Theoretically, the structure of an annotation file can be mapped in the database as well. As in conclusion and as shown in chapter four of this Master's Thesis, the persistence of semantic text annotations is an issue of the individual case and the computer scientist's preference. All types of persistence have their own charm, benefits and drawbacks. Before should be evaluated what type will work for the application case.

In the case of chapter five, we decided to use ordinary files for persisting the data of our application. The raw text has been saved as text-file (".txt") and also the annotation file has been saved as XML-file (".xml") at the hard disk. During chapter five, we firstly created a simple Java-application to read out the raw text and the annotation file and to create a HTML-string, which contains the raw text fitted and enclosed with the annotations using HTML5-elements and -attributes. This annotated HTML was used within an HTML5 page, was generically colored and got some interactivity functions for the end-user's experience. Thereby, it has been very important to develop all parts of the application in a generic way. Specifically that means, that it is neither restricted to a special application case nor to any types of annotations or additional attributes. If more or other annotations will be added to the annotation file, the application is able to read-out and display those information cor-

rectly to the end-user. In the implementation of the annotated HTML-file we solved one annoying and hindering issue: Overlapping annotations. To get rid of this problem, we found a solution working in two steps: identify and solve overlapping annotations.

## 6.1 Future work

Now, only one question still remains open: what are the next steps and what will happen in the future? Unfortunately the human being has no clairvoyant abilities, but it should be clear that this Master's Thesis is just a single part of the foundation.

This Master's Thesis should support further works on this project. One further work is another Master's Thesis that will end in October 2015 and researches high class text analysis applied on the legal domain. This Master's Thesis written by Tobias Waltl at the Technical University of Munich will use the results and hints of this work for further processing. Special attention will be paid on the research on different types of Legal Texts and the creation, usage and persistence of semantic text annotation. Therefore, the Master's Thesis of Tobias Waltl will partly build up on this Master's Thesis.

If high class text-analysis will be tailored on the legal domain in the German legislation, next steps will be to take a look on the legal domain from other countries. Probably, the same result could be used when such things should be introduced in other countries with German language like the Republic of Austria, the Principality of Liechtenstein or the Swiss Confederation. If this also works out in countries with other languages and different legal systems, has to be examined by further research. Except from the research of the German legislation in chapter two, this Master's Thesis should be independent from both, language and country. Although it was not an explicit part of this work to include also other languages next to German, but the procedure will probably work in other circumstances equally.

# List of Figures

# Bibliography


[AF05] Caroline Arms and Carl Fleischhauer. Digital formats: Factors for sustainability, functionality, and quality. In *Archiving Conference*, volume 2005, pages 222–227. Society for Imaging Science and Technology, 2005.

[BD15] Juris Bundesrepublik Deutschland. Hinweise. `http://www.gesetze-im-internet.de/hinweise.html`, 2015. [Online; accessed 2015-08-17].

[BPSM+98] Tim Bray, Jean Paoli, C Michael Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible markup language (xml). *World Wide Web Consortium Recommendation REC-xml-19980210. http://www. w3. org/TR/1998/REC-xml-19980210*, 16, 1998.

[Bra10] Kalli Bravos. Readability tests and formulas. `http://www.ideosity.com/ourblog/post/ideosphere-blog/2010/01/14/readability-tests-and-formulas`, 2010. [Online; accessed 2015-08-08].

[Bun06] Bundesgerichtshof. Urteil - vi zr 259/05. 2006.

[Bun15a] Bundespatentgericht. Beschluss - x zb 1/15. 2015.

[Bun15b] Deutscher Bundestag. Adoption of legislation - no lawmaking without parliament. `https://www.bundestag.de/htdocs_e/bundestag/function/legislation`, 2015. [Online; accessed 2015-08-13].

[CR12] Ingo Chao and Corina Rudel. *Fortgeschrittene CSS-Techniken*. Galileo Press, 2012. ISBN 978-3-8362-1695-1.

[Cro06] Douglas Crockford. The application/json media type for javascript object notation (json). 2006.

[Deu15a] Bundesrepublik Deutschland. Chemikalen-sanktionsverordnung, §8 ordnungswidrigkeiten nach der verordnung (eg) nr. 689/2008. `http://www.gesetze-im-internet.de/chemsanktionsv/__8.html`, 2015. [Online; accessed 2015-08-17].

[Deu15b] Bundesrepublik Deutschland. Strafgesetzbuch (stgb). `http://www.gesetze-im-internet.de/stgb/`, 2015. [Online; accessed 2015-07-07].

[dJufV15] Bundesministerium der Justiz und für Verbraucherschutz. Gerichte des bundes und der länder am 23. februar 2015. 2015.

[DKFR08] Hans Christian Dr. Klaus Friedrich Röhl. *Allgemeine Rechtslehre - Ein Lehrbuch*. Verlag Franz Vahlen München, Frankfurt am Main, 2008. ISBN 978-3-8006-4098-0.

[DMK13] Michael J. Bommarito II Daniel Martin Katz. Measuring the complexity of the law: The united states code. 2013.

[DN14] About.com Daniel Nations. Web applications. `http://webtrends.about.com/od/webapplications/a/web_application.htm`, 2014. [Online; accessed 2015-08-08].

[EMSS00] Michael Erdmann, Alexander Maedche, H-P Schnurr, and Steffen Staab. From manual to semi-automatic semantic annotation: About ontology-based text annotation tools. In *Proceedings of the COLING-2000 Workshop on Semantic Annotation and Intelligent Content*, pages 79–85. Association for Computational Linguistics, 2000.

[fdBD12] ISBN-Agentur für die Bundesrepublik Deutschland. *ISBN-Handbuch*. MVB Marketing- und Verlagsservice des Buchhandels GmbH, Frankfurt am Main, 2012. ISBN 978-3-7657-3278-2.

[FJP51] James N Farr, James J Jenkins, and Donald G Paterson. Simplification of flesch reading ease formula. *Journal of applied psychology*, 35(5):333, 1951.

[Fle48] Rudolph Flesch. A new readability yardstick. *Journal of applied psychology*, 32 (3):221, 1948.

[Fou15] The Apache Software Foundation. Apache pdfbox - a java pdf library. `http://pdfbox.apache.org/index.html`, 2015. [Online; accessed 2015-08-03].

[Gar11] Matt Garrish. *What is EPUB 3?* " O'Reilly Media, Inc.", 2011. ISBN 978-1-449-31454-5.

[GB] Stefan Th Gries and Andrea L Berez. Linguistic annotation in/for corpus linguistics. *Handbook of Linguistic Annotation*.

[Ger12] Ellen Gerwitz. *Ancient and American History I, Textbook*. Honour of Kings, Washington D.C., 2012. ISBN 978-1-300-62264-2.

[Gmb10] LexXpress GmbH. Mehr gesetze und verordnungen. `http://www.presseanzeiger.de/pm/Mehr-Gesetze-und-Verordnungen-317121`, 2010. [Online; accessed 2015-07-27].

[Hic12] Ian Hickson. A vocabulary and associated apis for html and xhtml. *World Wide Web Consortium, Working Draft WD-html5-20120329*, 2012.

[I15] LG München I. Lg münchen i · urteil vom 27. mai 2015 · az. 37 o 11843/14. `https://openjur.de/u/771387.html`, 2015. [Online; accessed 2015-08-11].

[Kor01] Jukka K Korpella. A tutorial on character code issues. 2001.

[oe15] openJur e.V. Informationen über den verein und die rechtsprechungsdatenbank. `https://openjur.de/i/willkommen.html`, 2015. [Online; accessed 2015-08-11].

[OMS+06] Eyal Oren, Knud Möller, Simon Scerri, Siegfried Handschuh, and Michael Sintek. What are semantic annotations. *Relat'orio t'ecnico. DERI Galway*, 2006.

[Rev14] Princeton Review. *Cracking the TOEFL iBT*. Princeton Review, 2014. ISBN 978-0307945600.

[RLHJ+99] Dave Raggett, Arnaud Le Hors, Ian Jacobs, et al. Html 4.01 specification. *W3C recommendation*, 24, 1999.

[Rot95] Martha T. Roth. *Law Collections from Mesopotamia and Asia Minor, Volume 6*. Scholars Press, Atlanta, 1995. ISBN 0-7885-0104-6.

[SEB15] Technical University Munich SEBIS. Lexalyze - interdisciplinary research program. `http://www.en.lexalyze.de/`, 2015. [Online; accessed 2015-08-17].

[SL10] Frank Sobolewski Susanne Linn. *The German Bundestag, Functions and Procedures*. NDV Neue Darmstädter Verlagsanstalt, Rheinbreitbach, 2010. ISBN 978-3-87576-655-4.

[TH00] David Thomas and Andy Hunt. *The pragmatic programmer*. Addison-Wesley, 2000. ISBN 978-0-201-61622-4.

[w3s15] w3schools.com. Xml document types - well formed xml documents. `http://www.w3schools.com/xml/xml_doctypes.asp`, 2015. [Online; accessed 2015-08-14].

[Wan08] Yingxu Wang. *Software Engineering Foundations*. Auerbach Publications, Danvers, 2008. ISBN 978-0-8493-1931-0.

[Wik15a] Wikipedia. Gericht. `https://de.wikipedia.org/wiki/Gericht`, 2015. [Online; accessed 2015-06-25].

[Wik15b] Wikipedia. Rudolf flesch. `https://en.wikipedia.org/wiki/Rudolf_Flesch`, 2015. [Online; accessed 2015-06-27].

[Wik15c] Wikipedia. Web application. `https://en.wikipedia.org/wiki/Web_application`, 2015. [Online; accessed 2015-07-23].

[Wil09] Graham Wilcock. *Introduction to Linguistic Annotation and Text Analysis*. Morgan & Claypool Publishers, 2009. ISBN 978-1-59829-738-6.