An Evaluation of NLP Toolkits for Information Quality Assessment
Abstract

Documentation is often the first source, which can help user to solve problems or provide conditions of use of some product. That is why it should be clear and understandable. But what does “understandable” mean? And how to detect whether some text is unclear? And this thesis can answer on those questions.

The main idea of current work is to measure clarity of the text information using natural language processing capabilities. There are three global steps to achieve this goal: to define criteria of bad clarity of text information, to evaluate different natural language toolkits and find suitable for us, and to implement a prototype system that, given a text, measures text clarity.

Current thesis project is planned to be included to VizzAnalyzer (quality analysis tool, which processes information on structure level) and its main task is to perform a clarity analysis of text information extracted by VizzAnalyzer from different XML-files.

Keywords

Natural language processing analysis, information quality, clarity guidelines, natural language processing toolkits, graph format
Table of Content

1 Introduction ................................................................. 1
1.1 Problem and Motivation ............................................... 1
1.2 Goals and Criteria ...................................................... 1
1.3 Outline .................................................................. 2

2 Information Quality and Information Clarity ............................. 3
2.1 Focusing on the Meaning ................................................. 3
2.2 Avoiding of Ambiguity ...................................................... 3
2.3 Keeping of Elements Short ................................................. 4

3 Natural Language Processing Outline ....................................... 5

4 Information Clarity Analysis ................................................ 6

5 Evaluation of Natural Language Processing Toolkits .................... 8
5.1 Requirements Discussion and First Step of Evaluation ............... 8
5.2 Available Analyses of Apache OpenNLP and Stanford CoreNLP .... 8
5.3 Comparison of Apache OpenNLP and Stanford CoreNLP .......... 10

6 System Design and Implementation ......................................... 12
6.1 Project Tasks ................................................................ 12
6.2 Diagrams ................................................................... 12
6.3 Singularities of Design of Natural Language Processing Analysis .... 14
6.4 Design and Implementation of Configuration ......................... 15
6.5 Design and Implementation of Representation of Output Information ...... 15
6.6 Implemented Functionality ............................................... 17

7 Testing and Discussion of Results ........................................... 19
7.1 Testing process ................................................................ 19
7.2 Testing results .............................................................. 21

8 Conclusion and Future Work .................................................. 23
8.1 Conclusion .................................................................. 23
8.2 Future Work .................................................................. 23

References ........................................................................ 24
Appendices ........................................................................ 26
Appendix A Content of “README.txt” file ................................. 26
Appendix B Content of “lists.xml” file ..................................... 27
1 Introduction
Technical documentation is important because often it is the first source of information, which can help to user to solve problems or provide conditions of use of some product. That is why documentation should be clear and understandable. These concepts define information quality.

There are a lot of different type parameters or constructions, which can influence on text clarity, beginning from the most basic things such as spelling, punctuation, etc. and ending by syntax relations and sense of text. Thus it is important to define such parameters as many as possible and find ways to detect them in the text information.

In our opinion a good way to define the parameters or criteria is to use brainstorm based on different specific literature. For detection of these parameters the most real ways are implementation of such system on our own or using third-party tools. Fortunately, there is a field of computer science, artificial intelligence and linguistics, named Natural Language Processing, concerned with the interactions between computers and human (natural) languages. Specifically, it is the process of a computer extracting meaningful information from natural language input and (or) producing natural language output. That is exactly what we need because detection of parameters is based on text structure, which is presented by extracted information from language input. Thus searching of suitable natural language processing toolkit is more efficient and easy way than implementation of functionality in our own.

This chapter describes main tasks and goals of current thesis project. Also it describes all sections in common.

1.1 Problem and Motivation
One of the main problems is to define parameters, which can indicate a lack of quality. The main source for searching such criteria is [1]. Then we can discuss, which parameters could be formalized and define how. After this it is necessary to define, which text structures or constructions we need for analysis. For example, if we decide that long string of nouns (three and more) is one of lack quality parameter, we should be able at least to extract words or tokens and their part-of-speech tags from input text. Thus we can get requirements for natural language processing functionality and the last step is to find suitable for toolkit(s) these tasks.

Specifically, we want to to answer on open research questions:

- What measurable indicators of the text clarity exist?
- Which structures and attributes of natural language are required for measuring of the text clarity?
- Which natural language processing toolkits are best for getting required structures and attributes from previous question?

Text in common can be understandable (clear) or not. Answering the first question we want to find words, word constructions and phrases that are attributes of unclear text information. Example of structure from the second question is paragraph, sentence, word etc. Example of attribute from the same question is part of speech of some word, its lemma etc. The third question exists because implementation in our own of all required natural language tasks is too time consuming to implement on our own.

1.2 Goals and Criteria
The main goal of this thesis is to create tool prototype, which is able to measure text clarity. Thus we should follow the next steps:

- Find suitable and measurable indicators that determine if a text has high quality or not. We should find all (or major part of) words, word constructions and phrases that could do a text unclear for readers.
• Find the toolkit(s) that is the best match for getting of all information required for measuring of text clarity. The toolkits will be evaluated based on several criteria, for example how easy they are to integrate, how well they support different analyses, etc.
• Implement a prototype system that, given a text, measures text clarity.

After completing these steps we will have answers on our research questions. The last step is to evaluate our results and proof that this thesis makes sense. Thereto we will analyze clear and unclear text documentation by our prototype and compare results. If unclear text will contain much more bad words, word constructions and phrases than clear, we will succeed.

1.3 Outline
The rest of the thesis has the following structure. Chapter 2 introduces information quality and information clarity. Chapter 3 introduces natural language processing. Chapter 4 describes how we implement clarity analysis using natural language processing capabilities. Chapter 5 describes evaluation of toolkits with capabilities from chapter 4 and motivates our choice. Chapter 6 presents system design and implementation. Chapter 7 describes system testing and discusses results. Chapter 8 presents conclusion and discusses future work.
2 Information Quality and Information Clarity
Current chapter reviews in detail clarity quality statements that can be measured in some way.

“Clear information is information that users can understand the first time. They do not need to reread it to untangle grammatical connections, sort out excess words, decipher ambiguities, figure out relationships, or interpret the meaning. Clarity in technical information is like a clean window through, which you can clearly see the subject” [1]. English is an international language and the biggest part of the Web information is presented in this language instead of multiple languages. That is why it is very important to write information clear for everybody who can at least read in English. “Clear information requires close attention to elements such as words, phrases, sentences, lists etc. to make sure that each participates appropriately in the message” [1]. The most common way to get these elements from a text is to use natural language processing.

To make information clear, writer should follow these guidelines:
• Focus on the meaning.
• Avoid ambiguity.
• Keep elements short.
• Write cohesively.
• Present similar information in a similar way.
• Use technical terms only if they are necessary and appropriate.
• Define each term that is new to the intended audience. [1]

2.1 Focusing on the Meaning
Clear information requires that writer focuses on what he wants to say. What is the point? What does he want the users to do or to know?

When most people write, they need to warm up to a subject. They need to write a while before the words flow and they see what they need to say.

Writer should avoid:
• Imprecise words – “words, which do not have a clear meaning or are not clearly referring to the point” [1]. Examples of imprecise word phrases are: “have plans”, “hold meeting”, “give an answer” etc. Information is clearer when writer replaces them by “plan”, “meet”, “answer” respectively.
• Intensifying words – “such words are meant to intensify the meaning, but the meaning is clearer without them. In speaking, people tend to use these words liberally, but in writing, these words quickly lose their effect” [1]. Examples are: “absolutely”, “completely”, “simply”, “some”, “quite”, “totally”, “very” and others.
• Long sentences.

2.2 Avoiding of Ambiguity
Translators and nonnative speakers are more likely than native speakers to have difficulty when information is ambiguous. English is an international language and the biggest part of Web information is presented in this language instead of multiple languages. That is why it is very important to write in the way, all people who know English a little bit, can understand a writer:

To avoid ambiguity in what writer writes, he should follow these guidelines:
• Use words with clear meaning. “Words that have more than one meaning can be confusing when more than one meaning fits the context” [1]. Examples of such words are: “may”, “once”, “while” etc. It is better to use “can or might”, “after or when”, “although or whereas” instead respectively. “A word that can be used as more than one
part of speech can be confusing when the word is used in two different ways close together or when it is used with other such words” [1]. So it is also good to avoid such sentences as “Record the date of the record”.

**Avoid long strings of nouns.** It is better to not use three or more nouns in one string because it may confuse a reader. Examples of such phrases are: “input message destination transaction code”, “plan selection routine”, etc.

**Avoid vague references.** The noun is vague when “it is far from the pronoun, perhaps in another sentence” [1]. For example noun “boy” is vague in the next part of text: “Boy went to the shop. He bought some food.”

**Write positively.** “Putting more than one negative word in a sentence can make the sentence difficult to understand” [1]. Research shows that positive sentence is more understandable for readers than equivalent sentence with double negatives. Examples of such words are: “not unlike”, “not many”, “not the same”, etc.

2.3 Keeping of Elements Short

“Wordiness wastes reading time, space, and paper, and it sometimes buries the message. Writing concisely is especially necessary for information on the Web, where reading is 25% slower than for printed material” [1].

To keep elements short writer should follow next guidelines:

**Remove roundabout expressions and needless repetition.** Examples of such phrases are: “at present”, “due to the fact that”, “group together”, “in conjunction with” etc. It is better to use “now”, “because”, “group”, “with” instead respectively.

**Choose direct words.** Sometimes there are two English words that have the same meaning. Often one derives from Latin (longer word) and the other derives from Anglo-Saxon (shorter word). Usually the Anglo-Saxon word is more direct. Examples of such pairs are: “accomplish” – “do”, “perform” – “do”, “terminate” – “end” etc.

**To keep lists short.** “Long lists, especially long lists of tasks or subtasks, can overwhelm users” [1]. The traditional guidelines for lists are:

1) Seven items maximum for online information.
2) Nine items maximum for printed information. [1]
3 Natural Language Processing Outline

This chapter describes natural language processing and its main tasks.

Natural Language Processing (NLP) is a field of computer science, artificial intelligence and linguistics which helps to interact between computers and human (natural) languages. [2]

The development of natural language processing applications is a challenging because computer language completely differs from human. First one is often precise, highly structured and unambiguous. From the other side human language is often not precise, ambiguous and its linguistic structure can depend on many things such as slang, social context, regional dialects, etc. [3]. Natural language processing applications or toolkits are systems which help to translate human speech into computer and vice versa.

We will now review natural language processing tasks (or analyses) that we are using in this thesis. The most simple and common tasks are sentence segmentation and tokenization. Sentence segmentation (also known as sentence boundary disambiguation) finds the sentence boundaries in a text. Tokenization divides a text into separate words or punctuation. For example “Stockholm is the capital of Sweden” – is a sentence and “capital” is a token. These tasks are basis for other analyses.

When text is divided on sentences and tokens, we can do further analysis of each word, specifically named entity recognition, part-of-speech tagging and stemming.

Named entity recognition (NER) – “given a stream of text, determining, which items in the text map to proper names, such as people or places, and what the type of each such name is (e.g. person, location, organization).”[4] For example “John Smith” is a person; “West Europe” is a location.

Part-of-speech tagging – determination of the part of speech for each word. For example “she” is a pronoun, “green” is an adjective.

Stemming (or lemmatization) reduces inflected (or sometimes derived) words to their base or root form. For example the root form for words “was”, “is”, “am” is “be”. These analyses often need some dictionary or base.

The next task is parsing. It uses output of previous analyses. “Parsing determines the parse tree (grammatical analysis) of a given sentence. The grammar for natural languages is ambiguous and typical sentences have multiple possible analyses” [4]. In other words it shows relations between words in the sentence.

And the last analysis is co-reference resolution, which uses previous analyses with parsing. Co-reference resolution “determines, which words ("mentions") refer to the same objects ("entities") in a sentence or larger chunk of text” [4]. Anaphora resolution is a specific part of this task, and it is related with relations between nouns and pronouns. For example, we have such sentence: “John likes his job. He also likes computer games”. Words “his” and “he” should refer to “John”.

All these tasks are most common in natural language processing and are the major part of this thesis work.
4 Information Clarity Analysis

This chapter discusses which clarity guidelines (chapter 2) can be applied and how to extract information, required for implementation of selected information clarity analyses.

In chapter 2 we mentioned that for good text clarity writer should follow next guidelines: focus on the meaning, avoid ambiguity and keep elements short. So we decided to implement detection of such problems as:

- Imprecise words or phrases.
- Intensifying words or phrases.
- Ambiguous words or phrases.
- Long strings of nouns.
- Vague references.
- Negative words or phrases.
- Roundabout expressions and needless repetition.
- Indirect words.
- Long lists.

We also decided not to address detection of long sentences because the concept is too subjective and in different situations “long” has different meaning. At the moment we know what we want to implement and now we describe in which way we do this.

First of all for clarity measuring we should perform a group of similar tasks: to find imprecise, intensifying, ambiguous, negative, roundabout and indirect words or phrases in the text (as discussed in second section). And it is very hard task to detect these words and phrases because there are no such toolkits or databases that can entirely detect them. WordNet [17] is a powerful toolkit for definition of semantic relations between words but it can be only partially helpful for us because these words or phrases are often not a synonymous or have less noticeable semantic relations. So we decided that the best way is to “hardcode” lists of that groups using content from [1]. For example “may”, “once”, “while”, etc. are part of list of ambiguous words or phrases. In this situation task becomes much easier and we should just find words or phrases from lists in the text. Because lists contain content in initial form we need lemmatization or stemming functionality. As we mentioned in the chapter 3 stemming (or lemmatization) is a process of reducing of inflected (or sometimes derived) words to their stem, base or root form, which is generally a written word form. For example text can contain word “might” but its initial form is “may”.

Next problem, which we should detect, is long strings of nouns. Current problem is problem of ambiguity. As discussed in section 2.2 long string means three or more nouns in a row [19]. For counting of words we need sentence segmentation and tokenization functionality. These divide text on sentences and each sentence on words. Also in order to define part of speech of each word we need part-of-speech tagging functionality.

Next guideline is detection of situation when sentence contains few same words with different parts-of-speech tags. This guideline also extends problem of ambiguity. In order to find such words and compare their parts of speech we again need part-of-speech tagging and lemmatization or stemming functionality.

Next task is to find vague references. As discussed in section 2.2 vague reference means that noun and appropriate pronouns. In order to find vague references we need co-reference resolution functionality. It finds words, which relate to another words.

The last problem is long lists. As discussed in section 2.3 long lists are lists that contain more than seven items. This problem is a structural and entirely depends on
markup. So here we do not need natural language processing functionality but some basic patterns or constructions using clear Java language to detect such lists.

After describing all problems we can decide which natural language processing functionality we need and then evaluate different toolkits with respect to this information. So there is a list of functional requirements for toolkit(s):

- Sentence segmentation.
- Tokenization.
- Part-of-speech tagging.
- Lemmatization or stemming.
- Co-reference resolution.

This list is very important criterion in evaluation of toolkits, which is described, in the next chapter.
5 Evaluation of Natural Language Processing Toolkits

This chapter describes which toolkits were evaluated for natural language processing tasks, defines criteria of evaluation, winners and their advantages and limitations.

From the chapter 4 we got initial list of criteria for evaluation of toolkits. And now we need a list of candidates for evaluation. After discussions with my supervisors and some web searching we got such list:

- Apache OpenNLP. [6]
- Stanford CoreNLP. [7]
- LingPipe. [8]
- JTextPro. [9]
- General Architecture for Text Engineering (GATE). [10]
- LinguaStream. [11]

5.1 Requirements Discussion and First Step of Evaluation

The initial list of candidates can now be used to formulate and discuss requirements for toolkits. This will then define the suitable ones. In the end of section 4 we defined initial list of natural language processing tasks, which should support candidate (commonly or partially as exception). Now we want to define additional requirements, which are also important.

First such requirement is programming language. Because current master thesis is additional functionality for existing Java project “VizzAnalyzer” it is important to use toolkit, which provides APIs for the same language.

Next requirement is that toolkit should be “alive”. It is really a sign that project did not success if it is closed that is why we decided to not use APIs of such toolkits.

The last additional requirement is documentation. Each toolkit should be well documented. But this requirement is too subjective. So we decided that for us documentation is clear if we can use the most part of APIs using it (at least sentence segmentation, tokenization and part-of-speech tagging).

First step of evaluation is to compare toolkits by all criteria that are important for us. The comparison is presented in the table (we use our opinion and experience):

<table>
<thead>
<tr>
<th>Criteria</th>
<th>NLTK</th>
<th>LingPipe</th>
<th>JTextPro</th>
<th>LinguaStream</th>
<th>GATE</th>
<th>CoreNLP</th>
<th>OpenNLP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programming Language</td>
<td>Python</td>
<td>Java</td>
<td>Java</td>
<td>Java</td>
<td>Java</td>
<td>Java</td>
<td>Java</td>
</tr>
<tr>
<td>Documentation clarity</td>
<td>+</td>
<td>–</td>
<td>–</td>
<td>+</td>
<td>–</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Supports initial list of tasks</td>
<td>+</td>
<td>+/-</td>
<td>–</td>
<td>–</td>
<td>+/-</td>
<td>+</td>
<td>+/-</td>
</tr>
</tbody>
</table>

Table 5.1 First step of evaluation

Gray color in the table means that this result is not suitable for us. CoreNLP is the first candidate because it answers to all requerierments. OpenNLP does not support all mentioned natural language processing tasks but answers to all other requirements and thus we decided also include it in candidate group for further evaluation.

5.2 Available Analyses of Apache OpenNLP and Stanford CoreNLP

This section reviews in more detail our winners: Apache OpenNLP and Stanford CoreNLP. Here we want to describe, what analyses they support and which algorithms they use. Table 5.2 shows abilities of Apache OpenNLP and Stanford CoreNLP:
First of all we want to say that for each type of analysis by OpenNLP developer should include specific model. CoreNLP uses annotators for this purpose and it can include them all together. But in case of using OpenNLP developer can also train every model and add new information. CoreNLP does not support such ability. Also OpenNLP is not an academic project and unfortunately their documentation does not describe any algorithms, so in case of this toolkit we present only minimum information.

We start from the most basic analysis – **sentence segmentation**. CoreNLP does not describe algorithms for it and OpenNLP uses system that can detect that a punctuation character marks the end of a sentence or not. [13]

The next analysis is **tokenization**. CoreNLP uses tokenizer that was started as a PTB-style tokenizer (Penn Treebank tag set [18] – list of shortenings for parts of speech), but was extended since then to handle noisy and web text [7]. “OpenNLP offers multiple tokenizer implementations: whitespace tokenizer – a whitespace tokenizer, non-whitespace sequences are identified as tokens; simple tokenizer – a character class tokenizer, sequences of the same character class are tokens; learnable tokenizer – a maximum entropy tokenizer, detects token boundaries based on probability model” [13].

The next analysis is **lemmatization**. About CoreNLP we can only say that it supports this analysis. OpenNLP does not support it that is why for this toolkit instead of lemmatization we decided to use **Porter stemming algorithm** [16]. It is not as effective as CoreNLP lemmatizer but still it is better than nothing.

The next analysis is **part-of-speech (POS) tagging**. CoreNLP uses “maximum entropy-based part-of-speech tagger, which achieves superior performance principally by enriching the information sources used for tagging. In particular, they get improved results by incorporating these features: more extensive treatment of capitalization for unknown words; features for the disambiguation of the tense forms of verbs; features for disambiguating particles from prepositions and adverbs” [12]. OpenNLP also supports this analysis and it is available for other languages (German, Spanish, Swedish etc.). For English language both toolkits use Penn Treebank tag set [18].

The next analysis is additional for our project and it is **named entity recognition** (NER). CoreNLP Recognizes named (PERSON, LOCATION, ORGANIZATION, MISC) and numerical entities (DATE, TIME, MONEY, NUMBER). Named entities are recognized using a combination of three CRF (Conditional Random Fields – models for labeling sequences of tokens with tags drawn from a finite set) sequence taggers trained on various corpora, such as ACE (Automatic Content Extraction – technology to support automatic processing of human language in text form from a variety of sources) and MUC (Message Understanding Conference – information extraction technology). Numerical entities are recognized using a rule-based system [7]. All can we say about OpenNLP is that it also can detect named entities and numbers in text [13]. Also OpenNLP supports NER for a few languages.

<table>
<thead>
<tr>
<th>Ability</th>
<th>Apache OpenNLP</th>
<th>Stanford CoreNLP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sentence Detection</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Token Detection</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Lemmatization</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Part-of-speech Tagging</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Named Entity Recognition</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Co-reference Resolution</td>
<td>-</td>
<td>+</td>
</tr>
</tbody>
</table>

Table 5.2 Abilities of OpenNLP and CoreNLP
The last analysis is co-reference resolution. We do not describe parsing because both toolkits use it as a part of current analysis and we do not use parsing separately. Stanford CoreNLP uses a “simple co-reference architecture based on a sieve that applies tiers of deterministic co-reference models one at a time from highest to lowest precision. Each tier builds on the previous tier’s entity cluster output. Further, their model propagates global information by sharing attributes (e.g., gender and number) across mentions in the same cluster. This cautious sieve guarantees that stronger features are given precedence over weaker ones and that each decision is made using all of the information available at the time.”[14] OpenNLP also supports co-reference resolution but we did not find any documentation about it, so it is not possible to use it (at least for us). So in OpenNLP case for co-reference we have two alternatives: we can use CoreNLP APIs or we can find and use some other toolkit only for this task. Both ways have their limitations. For first alternative the problem is that it is impossible to integrate some input data in CoreNLP for co-reference resolution. So we can only do full analysis and apply co-reference resolution output from CoreNLP to tokens in OpenNLP. But in some situations they do different tokenization (for example “..” can be one or two tokens) and applying is impossible. For second alternative we still have the same problem with different tokenization mechanisms. Yes we can try to find toolkit that is much similar to Apache OpenNLP but here is another problem that we should include one more toolkit to our project. And it is only for one task! Choosing between two evils, we chose less. So in situation when Apache OpenNLP is a selected analyzer for co-reference resolution it also uses Stanford CoreNLP APIs. And this question is still opened for future work.

5.3 Comparison of Apache OpenNLP and Stanford CoreNLP
Here is described comparison of our two candidates by different criteria and define their advantages and limitations. For comparison we will use table presentation:

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Winner</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time and resource consuming</td>
<td>Apache OpenNLP</td>
</tr>
<tr>
<td>Required lines of code</td>
<td>Stanford CoreNLP</td>
</tr>
<tr>
<td>Analysis quality</td>
<td>–</td>
</tr>
<tr>
<td>Documentation</td>
<td>Stanford CoreNLP</td>
</tr>
</tbody>
</table>

Table 5.3 Second step of evaluation – comparison of OpenNLP and CoreNLP

First of all I want to say that it is only our own opinion obtained after using these two toolkits. Below each criterion is presented in more detail.

Time and resource consumption. Here we compare time that we need to wait while programs finish the same tasks on the same platform. And here Apache OpenNLP is much faster than Stanford CoreNLP. For example we started the same analyses on the same file with the approximate size 3 MB (on my laptop). OpenNLP finished work for nine seconds and CoreNLP spent around 50 seconds. Also CoreNLP has limitations for the size of analyzed file because of lack of heap memory. On my laptop with minimal heap memory in 256MB size is something around 10 MB. Heap memory size is a Java specific problem and if developer wants to use more memory than default one he should specify it manually using JVM options. OpenNLP has no such limitations (at least we tried it on files, bigger than 100MB), but it is just consuming more time. That is why here OpenNLP is a winner.

Required lines of code. Here we check how many lines of code we need to write for implementation of the same functionality. Using OpenNLP we should include and
initialize model for each type of analysis. That is we should write a lot of lines of code. Using CoreNLP we initialize model only once. That is why here CoreNLP is a winner.

Analysis quality. Here we check quality of different analyses, for example sentence segmentation, part of speech tagging, named entity recognition etc. Both toolkits are good and do not show obvious errors. That is why here is a draw.

Documentation. Both toolkits are well documented. They have good documentation about available analyses. It is pretty easy to use their APIs. But at the moment of evaluation Apache OpenNLP did not have information about co-reference resolution (but it is also part of their APIs) that is why we used CoreNLP for this task. And here CoreNLP is a winner.

In common we got one win for Apache OpenNLP against two wins for Stanford CoreNLP. Both toolkits are worthy candidates. Apache OpenNLP is very fast and it can process bigger text information (in comparison with Stanford CoreNLP). But from the other side Stanford CoreNLP can provide more documentation that is why we can use more analyses and more precisely measure information clarity. So both toolkits can be integrated in our prototype and used in different situations.
6 System Design and Implementation

VizzAnalyzer is a powerful quality analysis tool, which analyses information on structure level. Current thesis project is only small part of its functionality and performs clarity analysis using natural language processing capabilities of text information extracted by VizzAnalyzer from XML-files. As we decided previously for adding these capabilities we are using Apache OpenNLP and Stanford CoreNLP. For integration of current project with VizzAnalyzer it is necessary also to support a graph format of result representation. Current statements and other requirements help us to define project tasks, its architecture and implementation, which are described below.

6.1 Project Tasks

As mentioned in section 1.2 the main goal of this project is to implement a prototype system that, given a text, measures text clarity. For this purpose we should find suitable toolkit and using its APIs get natural language processing capabilities from text. But in section 3.3 we got result that both toolkits Apache OpenNLP and Stanford CoreNLP are useful and we decided to include both of them to project. Also user of our project for different purposes may decide to use only some types of analyses and we should support this decision. In addition there is one more requirement posed by client – is to support graph format of representation of output information. Following the goal of this thesis and these features, the next requirements were proposed to project:

• User should have abilities to set basic analyzer and configure it (user should be able to choose, which of natural language processing attributes he wants to get).
• Using APIs of chosen toolkits project should support such natural language processing tasks: sentence segmentation, tokenization, part-of-speech tagging, name recognition, co-reference resolution and lemmatization.
• Using bad clarity indicators (words, phrases etc.), project should support their detection in text.
• Project should support standard and graph representation of output information.
• Project should provide information about how to work with program.

First task was not required in the beginning of work, but after evaluation of toolkits we decided that it is very important because each toolkit supports various analyses on different level. We describe this task in more detail in 6.4.

In previous chapters we described motivation, design and implementation of second and third tasks. We also mentioned that we use Apache OpenNLP APIs [6] and Stanford CoreNLP APIs [7] for natural language processing purposes.

Like we said in introduction fourth task is also result of discussion. Because current project is part of “VizzAnalyzer”, it is necessary to get also graph representation for future purposes. For this task we decided to use yEd APIs [15] and it is described in more detail in section 6.5.

Fifth task is a trivial and does not need a big attention. Its implementation is just a creation of text file with helpful information.

6.2 Diagrams

Using described tasks we now can define requirements in use case diagram representation, discuss them and create a design. Use case diagram is presented on the figure 6.1.
There is only one role in our project – User. He can configure analyzer and run natural language processing and clarity analyses. He also can get result in two different ways: in standard (as text file) and in graph representation. In addition documentation about using of project is available.

Now, when we have project requirements and use case diagram we can define project classes and their relations. First of all we should have text object for storing analyzed text and its attributes. We decided that there is a need of three levels of deep: text, which is divided into sentences, and each sentence, which is divided into tokens (figure 6.2). For filling this object we should first configure and secondly run natural language processing analyzer. So we should have corresponding classes for configuration and for implementation of winner toolkits APIs. Next we should have some class-manager, which will start all process. After this we have text object filled by natural language processing capabilities. Finally we need class for clarity analysis and class for graph building.

Using information above, we built class diagram, which is presented on the figure 6.3. It presents just common view of classes and their relationships.
And now some words about current classes.

Starter. This class runs application and set configuration for system.

Configuration. This class configures and includes one of the existing toolkits.

ApacheOpenNLP. If Apache OpenNLP toolkit is configured as included analyzer than this class uses Stemmer for stemming functionality and by analyzing of incoming text fills Text->Sentence->Token hierarchy.

MyStanfordCoreNLP. If Stanford CoreNLP toolkit is included by configuration than this class analyses incoming text and fills Text->Sentence->Token hierarchy.

ClarityAnalysis. When Text->Sentence->Token hierarchy is filled out we can start clarity analysis and get output results.

GraphBuilder. If graph building functionality is included by configuration than this class builds graph representation of output results.

After discussion about project tasks and requirements we can in more detail review some of them. This review is presented in next sections.

6.3 Singularities of Design of Natural Language Processing Analysis

As mentioned previously we decided to divide text on sentences and each sentence on tokens. So project contains corresponding classes-containers with required fields and properties. Filling of them is work of natural language processing toolkits. In our case these are Apache OpenNLP and Stanford CoreNLP.

It would be logical to create some abstract parent class for all toolkits to define their behavior. In that way we could define all basic methods and be sure that each our toolkit class has the same methods. But mechanisms of analysis are very different, at least for Apache OpenNLP and CoreNLP toolkits. For example for Apache OpenNLP we should initialize models for each analysis we use and for Stanford CoreNLP we just add annotators of analyses we use to some instance. Also toolkits support different count of analyses and we cannot be sure about analyzers, which could be added in the future. In this situation all toolkit classes can have only one common method, which is called from the main class for starting analyses. That is why we decided to not use abstract class and underline a big difference between toolkits. So we are not using directly output of analyses but we fill our structures while analyzing. For example on sentence segmentation phase we fill sentence structures and on tokenization we fill token structures.
structures. If we need also to include co-reference resolution we also change appropriate fields of token structure.

In this situation we have interrelated between each other text structures (text, sentence, token) and not interrelated instances of toolkits. And it is not important, which form of output has current toolkit, only filled text, sentence, token structures and their fields are important. So we have kind of chain: text -> toolkit -> filled text structures. At the moment to include new toolkit to project we should create new class and provide some method, which will get at least a text information as parameter and return filled instance of “Text” object. Everything else fully depends on good clarity level of documentation and complexity of using current toolkit APIs.

6.4 Design and Implementation of Configuration
Here we describe what user is able to configure, why and how such ability is designed.

As it mentioned in chapter 5 we decided to include in project two toolkits: Apache OpenNLP and Stanford CoreNLP. First of all user should be able to select one of toolkit for natural language processing analyses. We also decided that Stanford CoreNLP is default analyzer.

The next reason for configuration is work with files. Because our project analyzes text files, user should set path to such file. Also in chapter 4 we decided to “hardcode” lists of imprecise, negative, etc. words. We think that the good way of doing that is to keep such lists in xml file. User should not create this file; he can just find it in root folder of our project source and set path to this file. The current content of that file is presented in Appendix B.

The next reason for configuration is that user can choose required for him natural language processing analyses. Of course sentence segmentation and tokenization are basic analyses and we cannot measure clarity without them. But for example in the most cases we do not need named entity recognition because it is not a part of clarity analysis but still it can be useful in graph representation. So user can switch on or off such natural language processing analyses: part-of-speech tagging, named entity recognition and co-reference resolution. By the way it is not recommended to use the last one for Apache OpenNLP because of problems described in chapter 4.

The last reason is to add ability for user to get graph representation of output results. And again graph representation could be helpful but not all time.

So user can configure basic analyzer, path to input text file, path to list with problematic words, include part-of-speech tagging, named entity recognition, co-reference resolution and graph building representation. Configuration class uses Java properties to store settings on hard disk. It contains different methods for adding all described information and string constants to use them as parameters for these methods. User can configure system through instance of main class “Starter”. More detailed information is available in Appendix A.

For adding new configurable parameters we should add some new string constants, which will define behavior and use existing methods with these as parameters or create new methods with another signature for saving settings to Java properties file.

6.5 Design and Implementation of Representation of Output Information
Our project is able to present output information in different ways. Here we review these ways in more detail.

We decided that default way of output presentation will be text information saved in file. This file contains two parts. First part presents common information about each problem we found in the text. Here is an example:
Number of ambiguous words/phrases: 6
Number of roundabout words/phrases: 1
Coreferences to other sentences: 6
Number of intensifying words/phrases: 7
Number of indirecting words/phrases: 1
Long string of nouns: 3
Number of negative words/phrases: 3

Figure 6.4 First part of Output Information

Figure 6.4 shows an amount of different clarity problems (type of problem and how many times we meet the given problem in the analyzed text). For example here is presented that current text contains six ambiguous, one roundabout, seven intensifying, one indirect, three negative words or phrases, six vague references (co-references to other sentences) and three long strings of nouns.

Second part of this file is detailed information about each problem. Here is an example:

```
Number of intensifying words/phrases_problem_11: very#1#the company now has some plans to extend the database significantly to include current photographs of employees and use the photographs as the basis for a very modern security system.
```

```
Number of intensifying words/phrases_problem_12: any#2#the system is specifically designed to protect secure areas of the company’s building from access by any people who really do not have authorization to be there.
```

Figure 6.5 Second part of Output Information

Second part is more complex and requires an explanation. So “Number of intensifying words or phrases” is a definition of problem, “...._11” or “...._12” is a number of problem. After colon we have three blocks divided by “#”. First block is a problematic word or phrase, second block is a sentence number in the text and the last block is an entire sentence. Such structure presents all requireds information but it is not perfect and can be changed in future.

Also all this information is duplicated in console (if user is using console application). It is important to note that output file is saved in the same path as input and have name “%input_file name%_res.txt”.

Second way of representation is a graph representation using yEd graph editor [15]. This view presents all possible information about text structures, clarity problems and their relations. For example we have such two small sentences: “The boy. He is strange.” Graph for this text is the next:

Figure 6.6 Graph Representation

Figure 6.6 presents text structure nodes ("Text", "Sentence0", "Sentence1" etc.). Each token node has complex name. It is divided on three blocks by “-“: first block presents token id, second block is token content (its text) and third block is its position in the current sentence. Also each such node can relate to other nodes, which are token attributes. On this picture each token node has relation with part-of-speech tag node
(using Penn Treebank tag set [18]). For example token with id #3, text “he” and which is the first word in sentence (0 position) has part-of-speech tag “PRP” what means “personal pronoun”. Clarity problem nodes have relations with corresponding sentence if it is applicable. In our case it is “Co-reference to other sentences” and it has not relation to concrete sentence. But all clarity problems with summarized count have relations to “Text” node. So on picture is presented that our text has one co-reference problem. This relation is also presented on the figure (token nodes with ids 3 and 1).

But graph representation by itself is not as important as yEd ability to export graph file and use it in other application. We implemented this feature in the way that graph file is saved in the same location as input file with name “%input file name%.gml”.

### 6.6 Implemented Functionality

In this paragraph are presented all implemented class methods and their description. This information is presented in table:

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Method or Group of Methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClarityAnalysis</td>
<td>readListsFromXML()</td>
<td>This method reads lists with problematic words and phrases (ambiguous, intensifying etc.) from xml file into collections</td>
</tr>
<tr>
<td></td>
<td>getCollection()</td>
<td>Returns collection by its number</td>
</tr>
<tr>
<td></td>
<td>addValuesToCollection()</td>
<td>Adds values to corresponding collection</td>
</tr>
<tr>
<td></td>
<td>setPhraseCount()</td>
<td>Counts problematic words and phrases and adds it to corresponding collection</td>
</tr>
<tr>
<td></td>
<td>setNegativePhraseCount()</td>
<td>Counts negative words and phrases and adds it to corresponding collection</td>
</tr>
<tr>
<td></td>
<td>findLongStringsOfNouns()</td>
<td>Finds strings with three or more nouns</td>
</tr>
<tr>
<td></td>
<td>findLongLists()</td>
<td>Finds lists with ten or more items</td>
</tr>
<tr>
<td>ApacheOpenNLP</td>
<td>analyze()</td>
<td>Runs sentence segmentation analysis and initiates tokens segmentation analysis</td>
</tr>
<tr>
<td></td>
<td>addNames()</td>
<td>Runs Named Entity Recognition (NER) analysis</td>
</tr>
<tr>
<td></td>
<td>doTagAnalysis()</td>
<td>Runs Part-of-Speech (POS) tagging analysis</td>
</tr>
<tr>
<td></td>
<td>doTokenization()</td>
<td>Runs tokens segmentation analysis and initiates NER, POS analyses and stemming</td>
</tr>
<tr>
<td></td>
<td>initialize…()</td>
<td>Group of methods, which initializes appropriate models for different analyses</td>
</tr>
<tr>
<td>MyStanfordCoreNLP</td>
<td>analyzeNLP()</td>
<td>Runs sentence and token segmentation, POS, NER and lemma analyses</td>
</tr>
<tr>
<td></td>
<td>analyzeCoref()</td>
<td>Runs co-reference analysis</td>
</tr>
<tr>
<td>Stemmer</td>
<td>A lot of methods</td>
<td>This class implemented by “Tartarus” and it realizes Porter stemming algorithm [16]</td>
</tr>
<tr>
<td>Configuration</td>
<td>addAnalysis()</td>
<td>Specifies type of included analysis</td>
</tr>
<tr>
<td></td>
<td>enterPath()</td>
<td>Enters path to test text file</td>
</tr>
<tr>
<td></td>
<td>enterXMLPath()</td>
<td>Enters path to XML file with problematic words and phrases</td>
</tr>
<tr>
<td></td>
<td>chooseAnalyzer()</td>
<td>Sets basic NLP analyzer (at this moment OpenNLP or CoreNLP)</td>
</tr>
<tr>
<td></td>
<td>includeGraphBuilder()</td>
<td>Includes graph representation of results</td>
</tr>
<tr>
<td></td>
<td>saveProperties()</td>
<td>Saves all specified properties</td>
</tr>
<tr>
<td>GraphBuilder</td>
<td>graphConstruction()</td>
<td>Creates graph instance and adds all nodes and edges</td>
</tr>
<tr>
<td></td>
<td>addCoreferences()</td>
<td>Adds co-reference relations between nodes into graph</td>
</tr>
<tr>
<td></td>
<td>graphDisplay()</td>
<td>Shows representation of graph instance</td>
</tr>
<tr>
<td>Starter</td>
<td>getProperties()</td>
<td>Gets specified earlier properties</td>
</tr>
<tr>
<td></td>
<td>startAnalysis()</td>
<td>Starts NLP analysis and measures clarity of tested file</td>
</tr>
<tr>
<td></td>
<td>printResults()</td>
<td>Prints results of clarity analysis and writes them to text file</td>
</tr>
<tr>
<td></td>
<td>readFromFile()</td>
<td>Reads text from specified file</td>
</tr>
<tr>
<td>Other methods</td>
<td>Provide a shell for Configuration class methods</td>
<td></td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>-----------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>Text Properties</td>
<td>This class is container for information about all text and list of sentences</td>
<td></td>
</tr>
<tr>
<td>Sentence Properties</td>
<td>This class is container for information about current sentence and list of tokens from this sentence</td>
<td></td>
</tr>
<tr>
<td>Token Properties</td>
<td>This class is container for information about current token</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.1 Project classes and methods
7 Testing and Discussion of Results

This chapter describes testing process of current master thesis project and its results.

7.1 Testing process

For testing this project we use three text files. We named them “bad”, “text1” and “text2”. First file, named “bad”, is a mixing of parts of text from different sources. In our opinion presented parts are not so clear and this text has bad level of clarity. “text1” includes part of text from real documentation and in our opinion it should be clearer than first one. Third file called “text2” also contains part of text from real documentation but has another source than “text1”. In our opinion it also should be clearer than “bad” and it is good point to compare clarity of documentation from different sources. So first of all we analyzed files in our own. In table 7.1 is presented real number of problems:

<table>
<thead>
<tr>
<th>Problem</th>
<th>“bad” file</th>
<th>“text1” file</th>
<th>“text2” file</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of ambiguous words or phrases</td>
<td>6</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>Number of roundabout words or phrases</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Vague references</td>
<td>3-6</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Number of intensifying words or phrases</td>
<td>7</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>Number of indirect words or phrases</td>
<td>1</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>Long string of nouns</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Number of negative words or phrases</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Number of imprecise words or phrases</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Common number of problems</td>
<td>24-27</td>
<td>13</td>
<td>14</td>
</tr>
</tbody>
</table>

Table 7.1 Real results

Detection of real count of problems is not so easy task because we should make this analysis with the minimal chance of mistake. But we cannot use any toolkits because we cannot guarantee high quality of their analyses. The good way in this situation is to analyze text in our own, searching all problems in the text by reading. Of course human brain also can do some mistakes but we can compare results of our analysis with results from other toolkits and extremely minimize chance of mistake. And now some words about analysis results. As we expected “bad” file has much more problems than other two and thus has a lower clarity level.

We also decided that it might be helpful to include some other information about each file in the table 7.2:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>“bad” file (Bytes)</th>
<th>“text1” file (Bytes)</th>
<th>“text2” file (Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>2662</td>
<td>1986</td>
<td>2284</td>
</tr>
<tr>
<td>Number of sentences</td>
<td>31</td>
<td>9</td>
<td>20</td>
</tr>
<tr>
<td>Longest sentence (length)</td>
<td>31</td>
<td>69</td>
<td>40</td>
</tr>
<tr>
<td>Words per sentence (avg)</td>
<td>15.71</td>
<td>40.56</td>
<td>21.7</td>
</tr>
</tbody>
</table>

Table 7.2 Size of source files

Obviously “bad” file is the biggest one and of course it may be the reason of the biggest count of problems. But this file contains almost twice as many problems as “text1” and “text2” files so size is not the reason of low clarity level. Also “text1” in average has almost as twice as more words in each sentence than other two files and the biggest sentence in this file contains 69 words. But previously we did not decide what is criterion of long sentence so we cannot mention this as a clarity problem. But still it is a good point for writer to think about text edition.
First of all we try to evaluate the results with no respect to toolkits. That is why we analyzed text files by two analyzers and present best results in the table 7.3:

<table>
<thead>
<tr>
<th>Problem</th>
<th>Result</th>
<th>CoreNLP</th>
<th>OpenNLP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Error number / correct matches / unnecessary / missing</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>“bad”</td>
<td>“text1”</td>
<td>“text2”</td>
</tr>
<tr>
<td>Number of ambiguous words or phrases</td>
<td>6/6/0/0</td>
<td>8/7/1/1</td>
<td>8/6/2/1</td>
</tr>
<tr>
<td>Number of roundabout words or phrases</td>
<td>1/1/0/0</td>
<td>0/0/0/0</td>
<td>0/0/0/0</td>
</tr>
<tr>
<td>Vague references</td>
<td>6/6/0/0</td>
<td>0/0/0/0</td>
<td>0/0/0/2</td>
</tr>
<tr>
<td>Number of intensifying words or phrases</td>
<td>7/7/0/0</td>
<td>1/1/0/0</td>
<td>5/5/0/0</td>
</tr>
<tr>
<td>Number of indirect words or phrases</td>
<td>1/1/0/0</td>
<td>4/4/0/0</td>
<td>0/0/0/0</td>
</tr>
<tr>
<td>Long string of nouns</td>
<td>5/4/1/0</td>
<td>0/0/0/0</td>
<td>0/0/0/0</td>
</tr>
<tr>
<td>Number of negative words or phrases</td>
<td>3/3/0/0</td>
<td>0/0/0/0</td>
<td>0/0/0/0</td>
</tr>
<tr>
<td>Number of imprecise words or phrases</td>
<td>0/0/0/1</td>
<td>0/0/0/0</td>
<td>0/0/0/0</td>
</tr>
<tr>
<td>Common number of problems</td>
<td>29/28/0/1</td>
<td>13/12/1/1</td>
<td>13/11/2/3</td>
</tr>
</tbody>
</table>

Table 7.3 Best analysis results

One can see that it is really possible to find almost all of clarity problems using Natural Language Processing toolkits. In the table is presented that the worst result in our examples is 79% (eleven matches and three missing problems) and it seems pretty good. The main reason of finding unnecessary and missing results is that there is no such toolkit with 100% guarantee of analysis correctness. The most often errors are related with part-of-speech tagging, lemmatization and co-reference resolution.

After presentation of best results we got after analysis of our files we now can show them with respect to Apache OpenNLP and Stanford CoreNLP. In the table 7.4 are presented results for both toolkits:

<table>
<thead>
<tr>
<th>Problem</th>
<th>CoreNLP</th>
<th>OpenNLP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Error number / correct matches / unnecessary / missing</td>
<td></td>
</tr>
<tr>
<td></td>
<td>“bad”</td>
<td>“text1”</td>
</tr>
<tr>
<td>Number of ambiguous words or phrases</td>
<td>6/6/0/0</td>
<td>8/7/1/1</td>
</tr>
<tr>
<td>Number of roundabout words or phrases</td>
<td>1/1/0/0</td>
<td>0/0/0/0</td>
</tr>
<tr>
<td>Vague references</td>
<td>6/6/0/0</td>
<td>1/0/1/0</td>
</tr>
<tr>
<td>Number of intensifying words or phrases</td>
<td>7/7/0/0</td>
<td>1/1/0/0</td>
</tr>
<tr>
<td>Number of indirect words or phrases</td>
<td>1/1/0/0</td>
<td>4/4/0/0</td>
</tr>
<tr>
<td>Long string of nouns</td>
<td>3/3/0/1</td>
<td>0/0/0/0</td>
</tr>
<tr>
<td>Number of negative words or phrases</td>
<td>3/3/0/0</td>
<td>0/0/0/0</td>
</tr>
<tr>
<td>Number of imprecise words or phrases</td>
<td>0/0/0/1</td>
<td>0/0/0/0</td>
</tr>
<tr>
<td>Common number of problems</td>
<td>27/27/0/2</td>
<td>14/12/2/1</td>
</tr>
</tbody>
</table>

Table 7.4 Analysis results with respect to toolkit
First of all we want to notice that these problems not always mean that text has bad clarity but they should attract the attention of writer and he can decide is that really a bad thing or not.

7.2 Testing results
And now we can discuss results. As follows from the table 7.4 they are sometimes entirely different for both toolkits and we have a few reasons for this. Firstly Apache OpenNLP has no own lemmatizer so we are using a stemmer that implements Porter’s algorithm and sometimes it shows not so good results. For example for word “copies” it defines lemma as “copi” and in the same time CoreNLP defines it correctly (“copy”). Secondly Apache OpenNLP does not have (at least we do not know how to use it) co-reference resolution analysis and thus we do not include it for current analyzer. Thirdly Apache OpenNLP’s part-of-speech tagging is worse than CoreNLP’s (it is only our opinion based on experiments) and that is why at least for “text1” file we have so many ambiguous problems. For example we can review such part of sentence: “If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy...”. The first word “copies” by OpenNLP is a noun and the last “copy” is a verb, what is incorrect. And now we will discuss results more precisely for each file.

For “bad” file they are different but CoreNLP shows more right results in common, than OpenNLP. But still they are not entirely correct because there are some errors, which both toolkits did not find. From the other side the main problem of OpenNLP here (excluding absence of co-reference resolution) is a not so good quality of lemmatization and as result this analyzer sometimes cannot find all problematic words or phrases from the lists. So in analysis of “bad” file the winner is CoreNLP.

For “text1” file CoreNLP again shows better results. It found has more matches with real in comparison with OpenNLP but still has some errors in co-reference resolution and ambiguous problems. Other toolkit here showed bad quality of part-of-speech tagging and found about 6 unnecessary ambiguous problems. But still it did not find three other ambiguous problems words or phrases. So for “text1” file CoreNLP again is a winner.

For “text2” file it is hard to say who is better because CoreNLP showed the worst result in comparison with previous files. It did not find all ambiguous problems and correct vague references. From the other side OpenNLP found more ambiguous problems and in common showed bigger number of matches. So here OpenNLP is a winner with very little advantage but still both toolkits showed satisfactory results (more than 60% matches).

After all previous work we also compared runtime for Apache OpenNLP and Stanford CoreNLP toolkits for obtained results in the table 7.5:

<table>
<thead>
<tr>
<th>Toolkit</th>
<th>“bad” file (s)</th>
<th>“text1” file (s)</th>
<th>“text2” file (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache OpenNLP</td>
<td>9</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>Stanford CoreNLP</td>
<td>49</td>
<td>68</td>
<td>63</td>
</tr>
</tbody>
</table>

Table 7.5 Comparison of runtime

Obviously Apache OpenNLP is faster more than in five times for “bad” file, in seven times for “text1” and almost in eight times for “text2”. That is in common this toolkit has a huge advantage in speed and, as we told earlier, this speed could be very helpful. But from the other side Stanford CoreNLP shows much more correct results while
searching clarity problems. So it just confirms our previous conclusions about speed and analysis quality for both of toolkits.

Also there is some singularity that CoreNLP spent more time for analysis of smaller file ("bad" is bigger almost on 700 KB than "text1" and almost on 400 KB than "text2"). Perhaps "text1" and "text2" have more complex structure or relationships between words than first file.

Despite the fact that Apache OpenNLP can analyze file in about five-seven times faster, at this time we are more interested in right results than in runtime (in reasonable terms). That is why we made a decision that Stanford CoreNLP should be basic natural language processing analyzer in our project.
8 Conclusion and Future Work
This chapter summarizes previous chapters, draws conclusions from the work done and presents ideas for future work.

8.1 Conclusion
In the first chapter we have three research questions and our work gives answers on them. First question is: “What measurable indicators of the text clarity exist?” We find a lot of these indicators, such as ambiguous, indirect, negative words and phrases, long strings of nouns, etc. More detailed information is presented in chapter 2. Also we described their implementation in chapter 4. Second question is: “Which structures and attributes of natural language are required for measuring of the text clarity?” We decided that we need to divide text on sentences and each sentence on tokens. Also we should at least be able to define part-of-speech and lemma of each word. More detailed information is described in chapters 2, 4 and 5. The third question is: “Which natural language processing toolkits are best for getting required structures and attributes from previous question?” We did evaluation of seven natural language processing toolkits by different criteria such as documentation clarity, supported tasks, etc. and found two suitable for us: Apache OpenNLP and Stanford CoreNLP. Both toolkits meet the requirements, defined in the introduction of chapter 5 and in section 5.1. Evaluation process is described in sections 5.1 – 5.3. Also we implemented prototype system that, given a text, measures text clarity. Furthermore clarity analysis prototype was tested. Project design and implementation is presented in chapter 6. Information about testing is described in chapter 7. Thus, we answered on the research questions and reached the goals, defined in sections 1.1 and 1.2 respectively.

Current thesis is useful for writers of technical documentation and may show some problematic points in the text after editing, which can be more understandable for readers. Project is configurable and supported by basic documentation. So it is easy to use it in other applications and choose analyses that are more suitable for user.

8.2 Future Work
Here we describe, which improvements or additions we may add to the project in future.

The most important part is to add multi-language support. Now project supports only English language. It would be nice to support natural language processing and clarity analyses of text information at least on Swedish language. But it is not easy task due this language is not as popular as English and it has a lot of differences with second one.

Another good part is adding of more toolkits. Now project uses APIs of two toolkits: Apache OpenNLP and Stanford CoreNLP. It would be better to find more toolkits that have other advantages and limitations than these two and provide to user more alternatives of natural language processing analyses.

Another way of improvement is to find new indicators of clarity measurement. After this we will be able to measure clarity more precisely.

One more way of improvement is to change default and graph representations. Now they are good enough for people who know their structure, but we could design them to be more clear and user-friendly.

This list could be bigger but here are presented the most important improvements at the moment.
References

Appendices

Appendix A Content of “README.txt” file
This file is a user guide for a project.

Including libraries:

1. Include to your project library textQualityAssessment.jar.
2. If you will use CoreNLP as a basic analyzer, include all jar files from lib/corenlp folder. If you will use OpenNLP as a basic analyzer, include all jar files from lib/opennlp folder. Or include jar files from both folders.
3. If you will want to get a graph with results also include grail.jar and yed.jar from root folder. (WARNING: Copy yed.jar to root folder of your project.)

Using code:

1. Create an instance of class Starter. All the following methods are methods of this instance.
2. Use method enterPath() to input path to analyzed file. (WARNING: it's mandatory). Example: enterPath("D:\test.txt")
3. Add to your project file lists.xml from root folder. It contains all problematic words/ phrases for clarity analysis. Use method enterXMLPath to input path to this xml file. (WARNING: it's mandatory). Example: enterXMLPath("D:\lists.xml")
4. CoreNLP is default analyzer. You may use method chooseAnalyzer() to select another analyzer. Possible arguments are constants of class Configuration: Configuration.CORENLP, Configuration.OPENNLP.
5. If you want to include in analysis part of speech analysis you may use method addAnalysisProperty() with argument Configuration.POS
6. If you want to include in analysis name recognition you may use method addAnalysisProperty() with argument Configuration.NER
7. If you want to include in analysis coreference resolution you may use method addAnalysisProperty() with argument Configuration.COREF
8. If you want to get a graph with relations between text structures you may use method includeGraphBuilder() without arguments.
9. Use method saveAnalysisProperty() without arguments to save your settings. (WARNING: it's mandatory)
10. Use method startAnalysis() without arguments to start program. (WARNING: it's mandatory)

Output results:

1. You will get text file with clarity problems of analyzed file in the same folder.
2. If you included graph builder you will get gml file with graph and will be opened YED editor for its representation.
Appendix B Content of “lists.xml” file

<?xml version="1.0" ?>
<lists>
  <impresizeList>
    <element>in agreement</element>
    <element>capable of</element>
    <element>carry out an inspection of</element>
    <element>conduct an investigation of</element>
    <element>do a verification of</element>
    <element>draw a conclusion</element>
    <element>give an answer</element>
    <element>give rise to</element>
    <element>have a requirement</element>
    <element>have knowledge of</element>
    <element>have plans</element>
    <element>has plans</element>
    <element>have the capability to</element>
    <element>hold a meeting</element>
    <element>keep track of</element>
    <element>make a distinction</element>
    <element>make a proposal</element>
    <element>make a suggestion</element>
    <element>make changes to</element>
    <element>make contact with</element>
    <element>perform the printing</element>
    <element>provide assistance</element>
    <element>reach a decision</element>
    <element>render inoperative</element>
    <element>serve to define</element>
    <element>show improvement</element>
  </impresizeList>
  <intensifyingList>
    <element>absolutely</element>
    <element>actually</element>
    <element>any</element>
    <element>basically</element>
    <element>certainly</element>
    <element>completely</element>
    <element>definitely</element>
    <element>fairly</element>
    <element>just</element>
    <element>of course</element>
    <element>particularly</element>
    <element>perfectly</element>
    <element>quite</element>
    <element>really</element>
    <element>significantly</element>
    <element>simply</element>
    <element>some</element>
    <element>specifically</element>
    <element>totally</element>
  </intensifyingList>
</lists>
<element>very</element>
</intensifyingList>

<ambiguousList>
  <element>as</element>
  <element>as long as</element>
  <element>in spite of</element>
  <element>may</element>
  <element>once</element>
  <element>on the other hand</element>
  <element>since</element>
  <element>through</element>
  <element>while</element>
</ambiguousList>

<roundaboutList>
  <element>at present</element>
  <element>at this point in time</element>
  <element>at that point in time</element>
  <element>a variety of</element>
  <element>due to the fact that</element>
  <element>during the course of</element>
  <element>for the most part</element>
  <element>for the purpose of</element>
  <element>given the condition that</element>
  <element>in an efficient manner</element>
  <element>in case of a</element>
  <element>in conjunction with</element>
  <element>in order for</element>
  <element>in order to</element>
  <element>in the event that</element>
  <element>of an unusual nature</element>
  <element>on account of the fact that</element>
  <element>with regard to</element>
  <element>adequate enough</element>
  <element>by means of</element>
  <element>create a new</element>
  <element>end result</element>
  <element>entirely complete</element>
  <element>every single</element>
  <element>exactly the same the same</element>
  <element>group together</element>
  <element>integral part</element>
  <element>involved and complex</element>
  <element>is currently</element>
  <element>new innovation</element>
  <element>one and only one</element>
  <element>period of time</element>
  <element>plan in advance, advance planning</element>
  <element>refer back</element>
  <element>repeat again</element>
  <element>share in common</element>
  <element>sequential steps</element>
</roundaboutList>
<element>subject matter</element>
<element>summary conclusion</element>
</roundaboutList>
<indirectList>
<element>accomplish</element>
<element>additional</element>
<element>construct</element>
<element>fabricate</element>
<element>discover</element>
<element>enumerate</element>
<element>initiate</element>
<element>commence</element>
<element>locate</element>
<element>majority</element>
<element>perform</element>
<element>possess</element>
<element>present</element>
<element>prior to</element>
<element>require</element>
<element>retain</element>
<element>terminate</element>
<element>transmit</element>
<element>utilize</element>
<element>employ</element>
<element>via</element>
<element>accelerate</element>
<element>align</element>
<element>defer</element>
<element>disconnect</element>
<element>exclude</element>
<element>review</element>
</indirectList>
<negativeList>
<element>not many</element>
<element>not the same</element>
<element>not different</element>
<element>not unlike</element>
<element>not able</element>
<element>not possible</element>
<element>not allow</element>
<element>not have</element>
<element>not accept</element>
<element>not#until</element>
<element>not#unless</element>
<element>n't many</element>
<element>n't the same</element>
<element>n't different</element>
<element>n't unlike</element>
<element>n't able</element>
<element>n't possible</element>
<element>n't allow</element>
</negativeList>
<element>n't have</element>
<element>n't accept</element>
<element>n't#until</element>
<element>n't#unless</element>
</negativeList>
</lists>