COMPANO

Textual Comparison of XML Documents with Annotations

Bachelor Thesis

Bruno Pfeiffer
0717311
bruno.pfeiffer@gmail.com
Vienna UT

Philipp Waibel
0716754
philipp.waibel@gmail.com
Vienna UT

Supervisor:
Dr. Katharina Kaiser

November 16, 2011

Abstract

Several research instutitions enable researchers and interested persons to access documents concerning the latest medical guidelines by the researchers / interested parties. These electronic documents are often annotated with additional information. When a new version of a text is released, one wants to automatically integrate the exisiting annotations and see the differences between the original and the new document. The paper concerns itself with the creation of a program that satisfies the following:

1. Comparison of textual content of the two HTML documents and highlighting of the differences.
2. Migration of the annotations in document α into the new document β.
3. Additional highlighting for the case that an annotation was integrated into the new document β.
4. The solution is programmed as an Eclipse RCP Plug-in.

This thesis treats the process of defining a sensible approach to the problem and creating a software that achieves this, the challenges overcome, the lessons learned and ideas of how this problem could otherwise have been solved (this is being covered in another IEG project[1]). We have chosen to dub the application *Compano* as an abbreviation of *Comparison with Annotations*.

[1]See list of projects at: `http://www.cvast.tuwien.ac.at/node/73`

Contents

1 Introduction

The following chapter gives the reader a deeper understanding of the project, the surrounding tools and processes, as well as the motivation for the project.

1.1 Motivation

The purpose behind this project is derived from the process that the group must follow. The process is graphically described in figure 1 (p. 4). The goal of the project is to simplify or even automate the comparison and migration process. Times change, and so do processes and formats. In earlier times, the documents were delivered in HTML format, making them easy to edit and extract or add information. The documents are now delivered in PDF format: This makes it impossible (without further tools) to access and manipulate the actual data. This, of course, has to be considered in the group's work process. Accordingly, a new step was introduced to convert the documents back into their HTML form: See figure 3 (p. 5). The documents delivered by the conversion process are of poor quality and complicate textual comparison and migration of annotations. This leads to high human resource costs which are sought to be reduced.

1.2 Project Description

The online description of the institute for this project was as follows: Text documents are often annotated with additional information. When a new version of a text is released, one wants to automatically integrate the existing annotations and see the difference between the original and the new document. The goal is to implement a program with the following characteristics:

1. Comparison of textual content of the two HTML / XHTML documents and highlighting of the differences.
2. Integration of the annotations in document α into the new document β.
3. Special highlighting for the case that an annotation was integrated into text that was changed in the new document.

1.2.1 Supporting Tools

The tool used for manipulating the HTML documents (i.e. adding information in the form of annotations) is DELT/A[2]. The application allows users to load an XML file, inspect it in various forms, add *annotation* tags and map these to other XML documents. This tool is analysed in more detail in chapter 2 (p. 7).

1.2.2 Regular Updates: HTML Documents

During analysis of the files we received for evaluation, we determined that there were two phases: (a) The HTML phase, followed by (b) the PDF phase.

[2]See `http://ieg.ifs.tuwien.ac.at/projects/delta/`

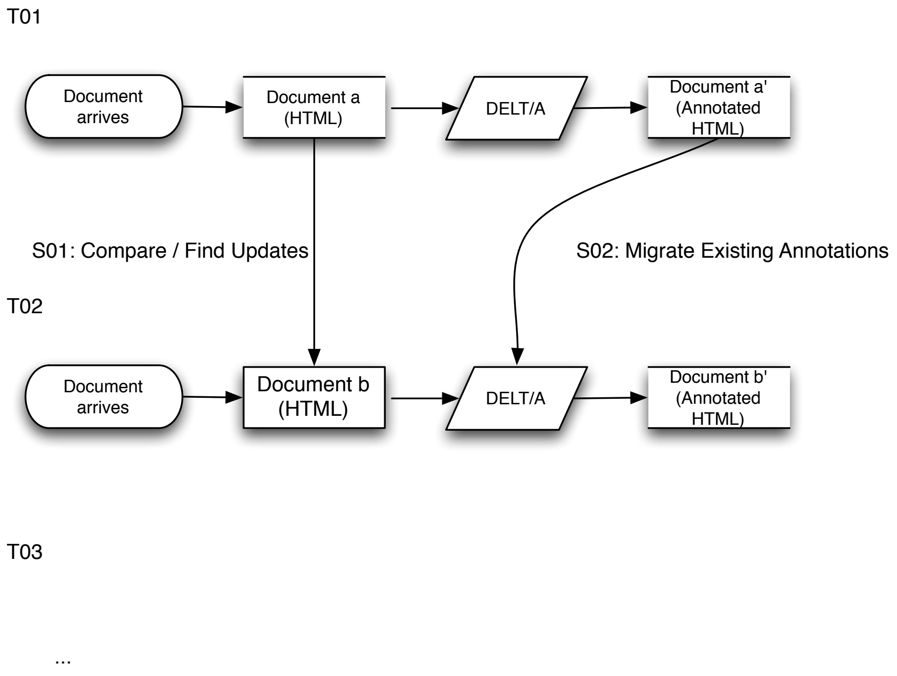

T01
Document arrives
Document a (HTML)
DELT/A
Document a' (Annotated HTML)
S01: Compare / Find Updates
S02: Migrate Existing Annotations
T02
Document arrives
Document b (HTML)
DELT/A
Document b' (Annotated HTML)
T03
...

Figure 1: Annotation Process of IEG

During the HTML phase, the document was delivered in HTML format and also contained a markup signalling changes over the time intervals in a specific span *update* class, for example:

```
<span class="update05">serum biochemistry and</span>
```

Figure 2: Example: Update Class in HTML documents.

This fact makes finding annotations in the text as simple as searching XML tree of the document for any element of type span with attribute *class="update"*. Highlighting the changes would also have been a simple task: Simply add font markups within the update class tags.

1.2.3 Regular Updates: PDF Documents

The documents are now delivered only in PDF form, making them extremely complicated to compare or annotate. Due to this, they are converted back into HTML format for processing. The *update* class is not exported into HTML. Instead, the PDF document contains a marker pointing at a piece of text stating *"Update 200x"*. This is not the most effective form of highlighting, especially

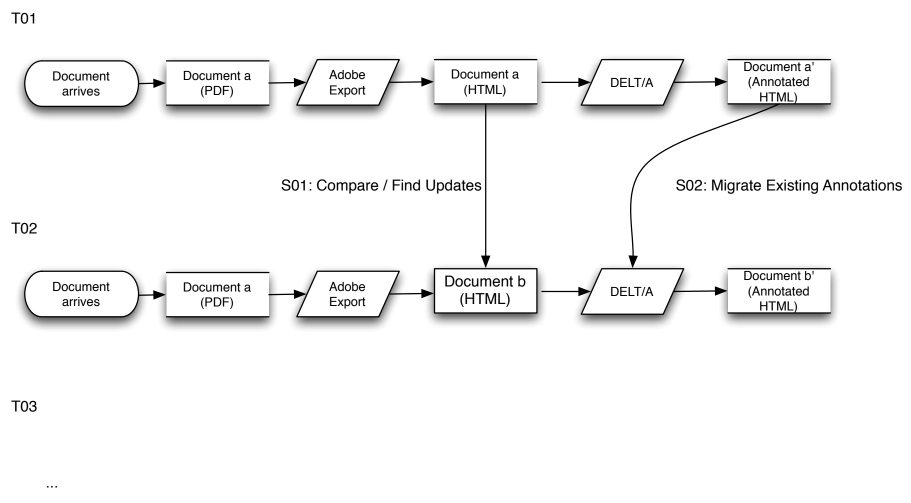
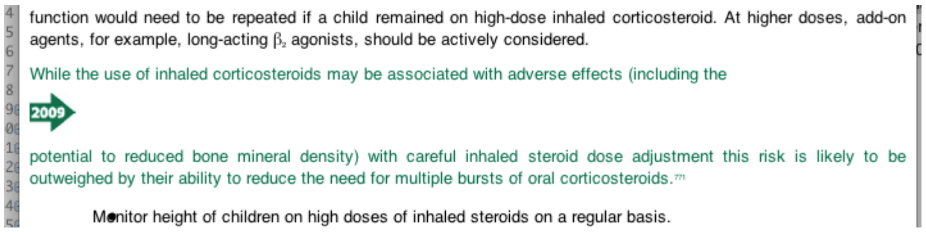


Figure 3: Annotation Process of IEG, with PDF.

once converted to HTML, since any relation of the data is lost (see 4 (p. 6)).

```
<IMG src="images/section 4_img_7.jpg">
<P>
<SPAN>
potential to reduced bone mineral density) with careful inhaled
    steroid dose
adjustment this risk is likely to be outweighed by their ability to
     reduce the
need for multiple bursts of oral corticosteroids.
</SPAN ><SPAN><Sup>771 </Sup></SPAN >
</P>
```


function would need to be repeated if a child remained on high-dose inhaled corticosteroid. At higher doses, add-on agents, for example, long-acting β_2 agonists, should be actively considered.

While the use of inhaled corticosteroids may be associated with adverse effects (including the

potential to reduced bone mineral density) with careful inhaled steroid dose adjustment this risk is likely to be outweighed by their ability to reduce the need for multiple bursts of oral corticosteroids.[771]

M●nitor height of children on high doses of inhaled steroids on a regular basis.

Figure 4: Top: Excerpt of HTML generated from PDF, shortened for readability. Bottom: Display version of code. (Note that the code does not match the shown image completely.)

2 Research & Related Work

The first description yielded three main topics for more intensive research: (1) text comparison algorithms, (2) the annual updating of the documents and (3) the handling & cleaning of XML documents. The results and insights are described in the following subsections.

2.1 Text Comparisons

During the search for an algorithm that would compare texts and deliver the differences in a satisifying technological fashion, we evaluated several algorithms. Specifically, we were looking for an algorithm that would perform the following:

1. Take two strings $\alpha, \beta \in \Sigma^*$ as parameters.
2. Return an ordered set Λ, consisting of tuples in the form $(\chi \in \Sigma^*, \delta \in \{equal, inserted, deleted\})$.
3. The elements in Λ are the result of the substrings of α & β being textually and sensibly compared, with δ being set accordingly.

Our research yielded several types of comparison methods and algorithms, ranging far and wide in applicable formats and quality of results. The algorithms that showed the most promise (i.e.: its input is two sequences of plain text, first test results were satisfactory) are described below. However: In the end, none of the algorithms delivered usable results in actual testing, see chapter 2.1.4 (p. 8) for details.

2.1.1 Levenshtein Distance

The first named algorithm for measuring the *delta* between two strings was the *Levenshtein Distance*. A general summary is as follows:

> The Levenshtein distance is the result of an algorithm that describes the difference between two strings A and B. The is also known as an *edit distance* and refers to the minimum number of operations required to transform A into B. [13]

The problem with this algorithm lies in its return value: The Integer value of the distance between two strings does not contain enough information to perform the consequent operations we required: i.e. knowing where and what the differences are.

2.1.2 UNIX diff Utility

The traditional UNIX diff Utility was the next algorithm that crossed our path. This classic algorithm delivers acceptable results for suitable data. However, two limitations caused it to be discarded: The use of the utility would have required a direct system call to the utility, which causes the application to depend on the underlying operating system. The other problem this would have caused is that the return value would have been a simple string representation of the result: Without effort in further processing, this data is not sufficient to fulfill the requirements.

> "In computing, diff is a file comparison utility that outputs the differences between two files. It is typically used to show the changes between one version of a file and a former version of the same file. Diff displays the changes made per line for text files. Modern implementations also support binary files.The output is called a "diff", or a patch, since the output can be applied with the Unix program patch. The output of similar file comparison utilities are also called a "diff". Like the use of the word "grep" for describing the act of searching, the word diff is used in jargon as a verb for calculating any difference."
>
> [11]

This left us with an additional requirement: The algorithm would have to be implemented in Java, so that objects could easily passed between the algorithm and our application.

2.1.3 Google diff-match-patch Utility

Google provides a utility called *diff-match-patch*, which is available as a Java API[3]. It implements *Myer's diff Algorithm*:

> This algorithm was implemented in a search for the problems of (a) finding the longest common string $\alpha \in A, B$ with $A \neq B$ and (b) findin the smallest number of operations necessary to convert A into B. These two problems have been shown to be equivalent. [14]

This API delivered suitable results when fed with reasonable data and was tested most actively during the implementation phase.

2.1.4 Results

All algorithms were plugged in via a Java interface and the delivered results evaluated: The Google algorithm (chapter 2.1.3 (p. 8)) delivered the best results for suitable texts. However, due to the heavily malformed data[4], the algorithm was largely ineffective: The results were always close to 100% textual difference, even though the textual content was equal for an estimated 92-97%. The problem occurs due to two factors:

1. The algorithms work in a *line-based* fashion, examining the contents of all pairs of lines (α', β') for equality.

[3]See `http://code.google.com/p/google-diff-match-patch/`
[4]See figures 6 (p. 11), 4 (p. 6) for examples.

2. The data delivered by the PDF export did not guarantee the integrity & coherence of the original content (see figure 6 (p. 11)).

This led to the following: The source document consisted of lines α, β, χ, the target document consisted of $\alpha', \alpha'', \beta', \beta'', \chi', \chi''$. Let $*$ denote the concatenation of strings, then the following held for the documents:

$$\begin{aligned}\alpha &= \alpha' * \alpha'' \\ \beta &= \beta' * \beta'' \\ \chi &= \chi' * \chi''\end{aligned}$$

However, the algorithms can and may not assume this concatenation (in fact separation of α' and α'' leads to a newline character separating the string elements), and so have to mark these occurrences as changes. In practice, the functionality provided no use or additional information, and was therefore removed from the final product.

2.2 Document Parsing and Handling

As described in 1.2 (p. 3), a main part of the solution requires importing, handling and manipulating data from XML documents. Thus, it was important to deeply research and evaluate existing frameworks / APIs which could be applied. A major part of our research dealt with the question of how to handle the HTML files correctly.

2.2.1 Parser Models

The first subquestion was which parser model is best suited for the required task. There are three types of models:

Object Model Parser Mirrors the entire XML tree in memory, enables random access. Example implementations are *DOM* [3] and *JDOM*.

Push Parser Run through the XML document once, only allowing sequential access. So-called *callback* methods are executed for every occurring element. An example implementation is *SAX* [8].

Pull Parser These parsers also only pass through the document once and allow for sequential access. The application has more freedom in analysis of the structure of the document and how to react to these circumstances. An example is the *StAX* parser [9].

Since it is essential that the software has random access to the XML elements, we chose to work with the DOM parser. The random access to individual parts of the document is primarily important for finding text sequences in the destination document which must be surrounded by annotations.

A Document Object Model (DOM) is an API which parses an entire XML document and constructs an in memory representation thereof. This model can then be read, edited and written back into an XML file by the DOM API's capabilities.

2.2.2 Document Form

The document's form was the next matter of concern. If the document is well formed, which is the case with XML files, the DOM parser can read the document without complications. A document is well formed if it satisfies the following rules:

- The document contains at least one element
- Each element has an opening tag and a closing tag (or a opening & closing tag)
- All tags must be nested properly

The exact definition of well-formed can be found here [12].

With HTML files, however, the matter is quite different: They do not *need* to be well-formed. This means that problems may occurr when attempting to parse HTML files with the DOM parser. For example, a missing tag or parent node make XML documents invalid, while an HTML document remains intact.

As a result, we searched for a DOM parser that is able to deal with these inconsistencies. The algorithm needed to be able to ascertain that the document is well-formed; if this is not the case, the parser should rewrite the data in the correct form, then perform the actual parsing process.

2.2.3 Choosing a DOM Parser

A number of APIs exist that perform the requirements listed above - and so a number of APIs were tested in operation, starting with the standard DOM API delivered with the Java Programming language. The APIs listed below are the ones that proved most usable, including the actual choice of API.

The first parser that was tested and found to be satisfactory was *JTidy*. The API claims the following:

> "JTidy is a Java port of HTML Tidy, a HTML syntax checker and pretty printer. Like its non-Java cousin, JTidy can be used as a tool for cleaning up malformed and faulty HTML. In addition, JTidy provides a DOM interface to the document that is being processed, which effectively makes you able to use JTidy as a DOM parser for real-world HTML." [5]

The alternative, which also became the API that was in fact used, was the *CyberNeko* HTML parser.

> "NekoHTML is a simple HTML scanner and tag balancer that enables application programmers to parse HTML documents and access the information using standard XML interfaces. The parser can scan HTML files and 'fix up' many common mistakes that human (and computer) authors make in writing HTML documents. NekoHTML adds missing parent elements; automatically closes elements

with optional end tags; and can handle mismatched inline element tags." [6]

Both parsers are able to read the HTML and evaluate if the data is well-formed. If errors exist in the scanned HTML documents, both parsers can resolve them to some degree. The reason that CyberNeko was chosen is that this technology is also employed by the DELT/A [15] software, and thus leads to increased standardization and reduced maintenance efforts.

2.2.4 Document Characteristics

In this phase of research, we analysed the HTML documents, specifically the ones generated from PDF files.

While analysing the documents, we noticed that the PDF to HTML converting program adds unnecessary *span* and other tags to the file. These unused tags split coherent texts, at times even individual letters. These splits occurr completely at random: There is no discernable algorithm that could be applied to reverse the process. See figures 5 (p. 11) and 6 (p. 11). Any identifiable unnecessary tags were to be removed before parsing: This would reduce memory requirements, speed up parsing, and simplify editing.

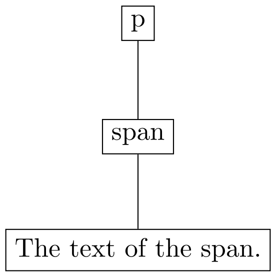


Figure 5: Normal HTML span

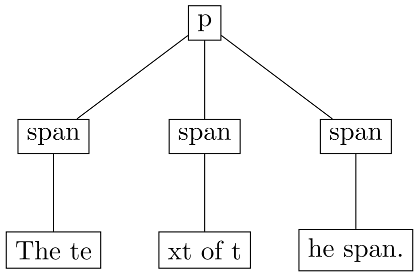


Figure 6: HTML span exported from PDF

2.3 Definition: Annotation

This section gives a brief explanation and definition of an *annotation*, so that the reader may better understand how to work with them. The definition related here is as it is used by the DELT/A software (see chapter 1 (p. 3)).

An annotation in an HTML document is an element of type *"a"*, whose descendants contain the annotated text. An annotation element has the following properties:

- The attribute *id*, whose value must match the regular expression "delta:[0-9]+".
- The (conditional) attribute *class*. This property marks the begin and the end of an annotation. It can take the values *sep* for a single node annotation, *sep1* for the begin of an multiple node annotation and *sep2* for the end of a multiple node annotation.
- The element may be at any location within the structure. Specifically, it need not have a *span* or text element as its immediate descendant.

Some example figures have been included. Please see figures 7 (p. 12), 8 (p. 13), 9 (p. 13).

```
<a id="delta:4" class="sep"><span>The annotated text</span></a>
```

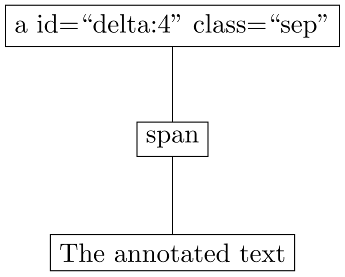


Figure 7: Basic Annotation Tag (code & graphic)

```
<a id="delta:4"><span>First part</span><span>Second part</span></a>
```

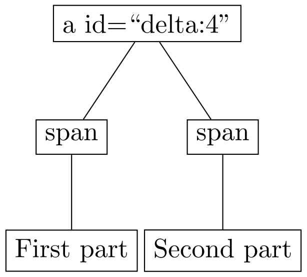


Figure 8: Basic Annotation Tag (code & graphic), with multiple children

```
<element>
<a id="delta:4" class="sep1"><span>Annotation part 1</span></a>
<a id="delta:4"><span>Annotation part 2</span></a>
<a id="delta:4" class="sep2"><span>Annotation part 3</span></a>
</element>
```

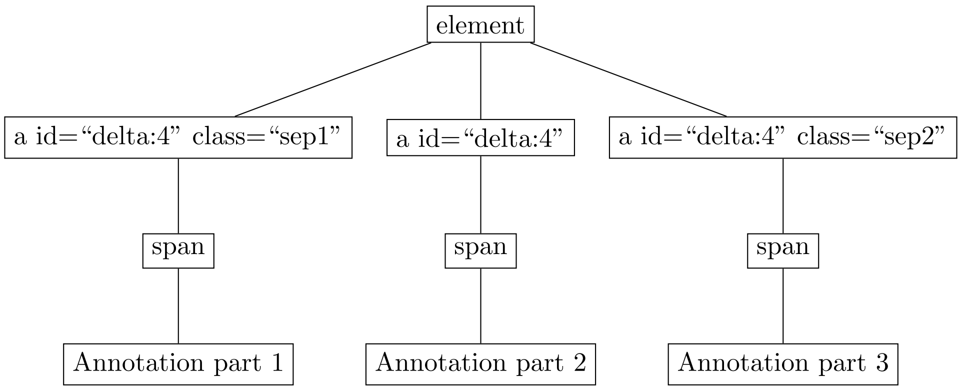


Figure 9: Single Annotation spread across several non-related texts

3 Approach

The approach to the project consisted of five main steps: Identification, Consideration, Analysis, Strategy and Execution (as shown in figure 10 (p. 14)). Each steps serves as input for the next step in the process. In the *Identification* phase, the major components, the general process and the surrounding challenges are identified. With this input, one can begin to *consider* the existing knowledge about the topic and understand what has been previously accomplished. If any substantial research on the topics has been done and similar solutions have been developed, one can *analyse* these to determine whether or not they are applicable to the specific task at hand. Last but not least, one sets the *strategy* to close the gaps: What needs to be created, what can be reused and who will be responsible? The final step is the actual execution of the strategy.

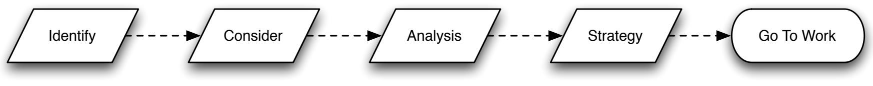


Figure 10: Approach: A 5-Step Process

3.1 Identification

The first step called for an investigation of the known facts and making first decisions: What should the process of the solution look like? What will we research? What do users expect from the software? etc.

3.1.1 Process

Figure 11 (p. 15) shows the process that was identified after gaining an understanding of the work that is to be supported and gaining deeper knowledge of the named functional requirements. The functional description of the created software is best described in graphs - please see appendix A, figure 32 (p. 40) for a detailed process overview. Nevertheless, a few words are spent here describing the application's typical usage; the individual *Use Cases* are described in detail in section A.2 (p. 48). The main functionalities that were specified & implemented are as listed below:

1. Opening Documents
2. Automatic Comparison of Documents[5]
3. Automatic Placement of Annotations
4. Manual Placement of Remaining Annotations
5. Saving the Result

The steps are typically performed in the listed order, although other variations can apply (complete manual placement, deletion of annotations).

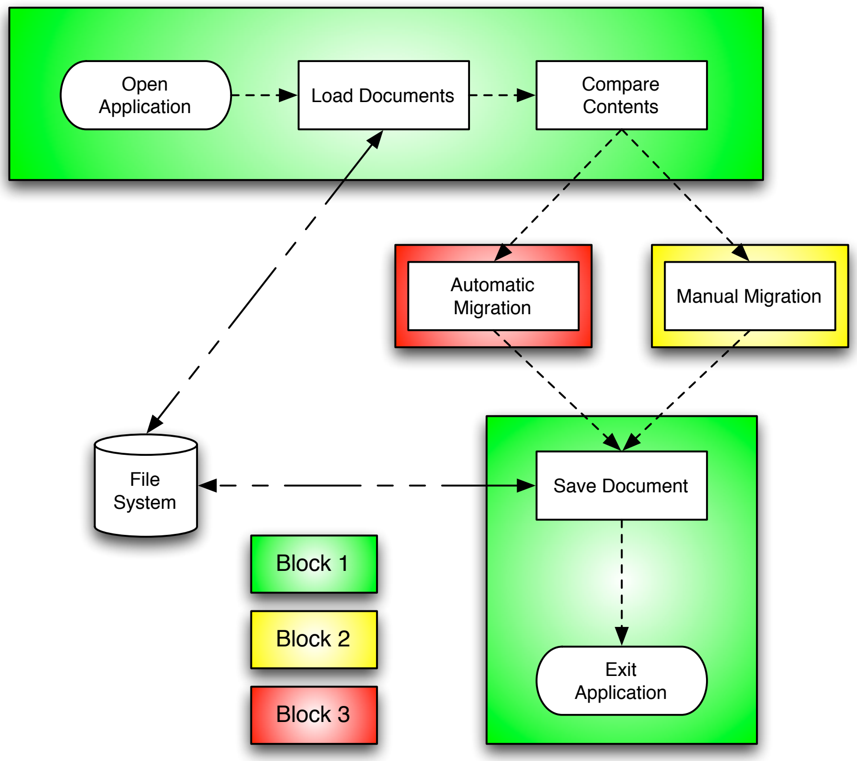


Figure 11: Definition of Process and Modules

3.1.2 Surrounding Challenges

The immediate challenges became apparent during an investigation of the surrounding systems and the required functionalities. The problem of dealing with

[5]This was implemented, but removed from the final product. See chapter 2.1.4 (p. 8) for further details.

a number of character encodings was one, the corrupted data delivered by PDF exports another. An algorithm to compare strings also needed to be found.

3.2 Consideration & Analysis

The input from the *Identification* step allowed us to begin researching topics in more detail. The results are described in section 2 (p. 7). It proved to be the case that no previous algorithms, save an HTML parser, existed that could be re-applied to the situation. With this knowledge, we could now go forward with the last planning phase: *Strategy*.

3.3 Strategy

Any project, albeit in this case a very small one, requires planning and certain levels of project management. A specific benefit of a small team is flexibility: Changes to organisation, standards, etc. are quickly discussed, agreed upon and implemented. This allows the team to basically rely on the trial-and-error method to determine the best way to do anything, since the involved time scales are miniscule. Because it does not force a lot of planning and assumptions about future risks in the implementation, it allows the team to *solve problems as they emerge* and in any way; meaning that sticking to individual milestones is not as important as the *big picture*.

The project plan is located in the appendix A, figure 30 (p. 38). This plan was meant to be a *best estimate*, since we could not foresee the results derived from the research or the availability of the performing resources (i.e. vacations, absences, etc.). As described above, we also did not want to spend more time planning than acting. The absence of a formal deadline allowed us to plan as we went.

3.3.1 Conception

To create the project plan, we split the project into its isolated steps:

Specification The goal of this phase is to gain a general understanding of the problem and specify the actual deliverables. After an overall picture, use cases and test cases are designed in standard templates that are meant to help guide and support the actual implementation process. It also gives the team an even more in-depth understanding of how users would like to interact with the system: This is a valuable input for designing the user interface as a mock-up. Once these facts have been gathered and set, it is important to consider the technical aspects of the functionality and formally specify these before beginning the implementation process, so as to ensure that certain paradigms are adhered to and questions can be clarified before hand. The result of this phase is a detailed specification accepted by all stakeholders.

Research Once the specification has been accepted, it is vital to perform research on existing solutions and previous implementations to gain an understanding of best practices and potential problems. The areas for research are described in chapter 2 (p. 7). In this case, the results from research were planned to be the base functionality of the solution, and thus had to be evaluated in minute detail to ensure their quality.

Implementation Upon completion of the research phase, it is possible to begin the actual implementation of the solution. This process was split into three logical blocks of implentation: *General instantiation, Manual placement of Annotations* and *Automatic Placement of Annotations*. The blocks are constructed in a *bottom-up* fashion, i.e. each block was a pre-requisite for its immediate follower. This approach required all the groundwork to be performed before focusing on the more detailed algorithms. A major reason to proceed in this manner was that the experience gained in the simpler algorithms can immediately be applied to the more difficult tasks, leading to less false assumptions.
The blocks were defined as listed below and is also graphically represented in figure 11 (p. 15):

1. The focus of the first block lay in instantiating the project and realising the major *look & feel* parts of the solution with little of the actual functionality. The steps involved are: Setting up the environment (eclipse, subversion, etc.), creating a mock GUI from the design specification, using the research results to implement two parsers (one that parses the pure text of the involved XML documents, one that parses the pure text plus the annotations), implementing the *diff* algorithm for use with the results of the previous & current version[6], implementing the target use cases for block 1 (see appendix A, figure A.2 (p. 48)) and last of all: Getting the acceptance for the first block.
2. Block number 2 focused on the target use cases and the functionality around manually placing annotations, as well as reviewing items from block 1. The manual placing of annotations is one of the core functionalities that would require large amounts of specification, given the data the solution would have to handle. This part of the implementation would deal with the interfacing the view and the core part of the solution to allow selected texts to be passed to the core for placing the annotations around them.
3. The last block revolved mainly around automating the placement of the annotations, reviewing the code from block 2 and finalizing the solution from a functionality aspect.

Testing & Evaluation Once finished with the general implementation, the solution was to be tested thoroughly. Tests were to be written and agreed upon, any identified software bugs would be corrected. The tests can be

[6]This was dropped from the final solution.

found in appendix A, figure A.2 (p. 52). As a final step, the solution would be demonstrated and accepted.

Documentation & Finalizing With the solution finished, the documentation of the final solution was to be created and any unfinished *pretty touches* to be implemented. The surrounding documentation is to consist of UML diagrams, test cases and brief functional descriptions.

The modules were put in sequence, estimates for total duration of each individual module made. Based on known resource availability, the modules were mapped to one or more calendar weeks. The effort estimation was made with a generous buffer to account for the unforeseen challenges that would be encountered.

3.3.2 Dependencies & Risks

The identification of dependencies - and the resulting risks - was simplified by the fact that all milestones were put in sequence (e.g. no parallel processing was planned). In other words, all blocks depended only on their immediate predecessor. This can be seen in figure 30 (p. 38).

3.3.3 Plan & Actual

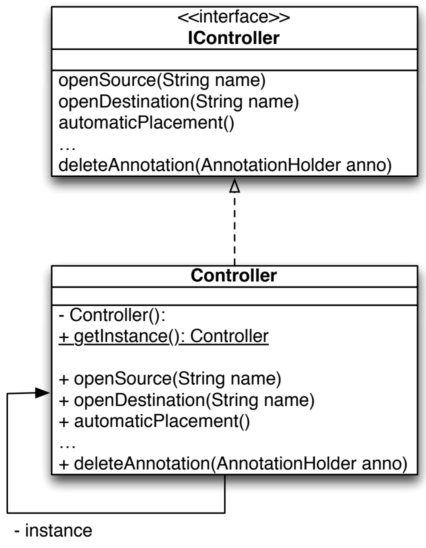
In the end, the project plan proved to hold true for a large part of the project. Only a few long term absences (due to other business) caused delay that was unforeseeable.

4 Implementation

This chapter describes how the solution was implemented, what challenges were faced, what designs & patterns were used, as well as what a number of the technical background pre-requisites were.

4.1 Architecture

The core element of the software is the *Controller* class. This class contains methods for each action the user can perform. The responsible *ActionHandler* calls these methods. The controller also holds references to the view and to the model. It is designed using the singleton pattern, so that all ActionHandlers have access to the same Controller instance. The Controller also implements the IController interface. Figure 12 (p. 19) shows the Class Diagramm of the Controller and Figure 13 (p. 20) the connections of the controller with the other parts of the software.



<<interface>>
IController
openSource(String name)
openDestination(String name)
automaticPlacement()
…
deleteAnnotation(AnnotationHolder anno)
Controller
- Controller():
+ getInstance(): Controller
+ openSource(String name)
+ openDestination(String name)
+ automaticPlacement()
…
+ deleteAnnotation(AnnotationHolder anno)
- instance

Figure 12: UML of Controller Class

For all annotations in the source document a new instance of the AnnotationHolder class will be generated. In this object the individual nodes of the annotation and the *id* are stored. Furthermore, the AnnotationHolder also holds the information concerning whether the annotation is already inserted in the destination document or not. Beyond the common *getter* and *setter* methods the class provides, methods to easily access the start node of the annotation, the previous node and the next node of a given node in the annotation were implemented. If there is no previous or next node, the return value of the method

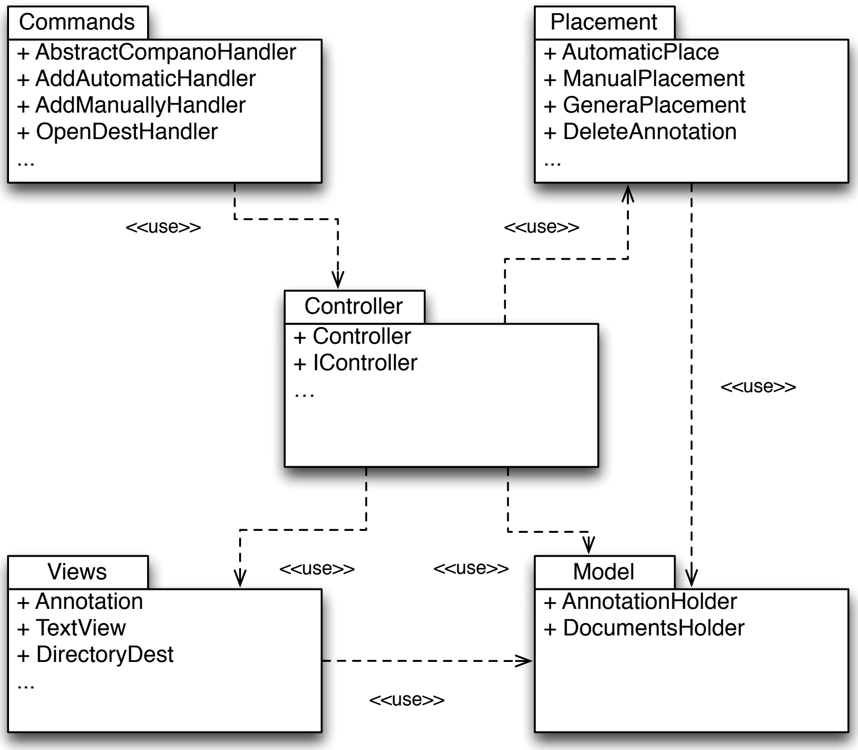


Figure 13: Package diagram

is *null*. With this method it is easier to search for the correct place in the destination document during the automatic placement procedure. Figure 14 (p. 21) shows the Class Diagramm of the *AnnotationHolder*.

The three tasks (1) add annotations automatically, (2) add annotation manually and (3) delete annotation were realized in the package named *src/placement*, each with a separate class. For a better understanding of the operations *add annotation manually* and *add annotations automatically* see the chapters 4.3 (p. 22) and 4.4 (p. 24) respectively.

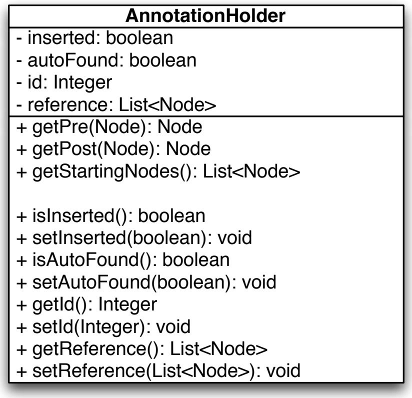


Figure 14: UML of AnnotationHolder Class

4.2 User Interface

The user interface, shown in figure 15 (p. 22), was built using Eclipse's RCP Framework, so that the program can easily be integrated into existing software at the institute. Another reason to use RCP is that it is possible to create user interfaces in a quick and uncomplicated way, so that programmers can focus on the core tasks of the program.

The interface consists of a perspective with the following views for interaction with the user:

Source Document The older version of the document containing the existing annotations.

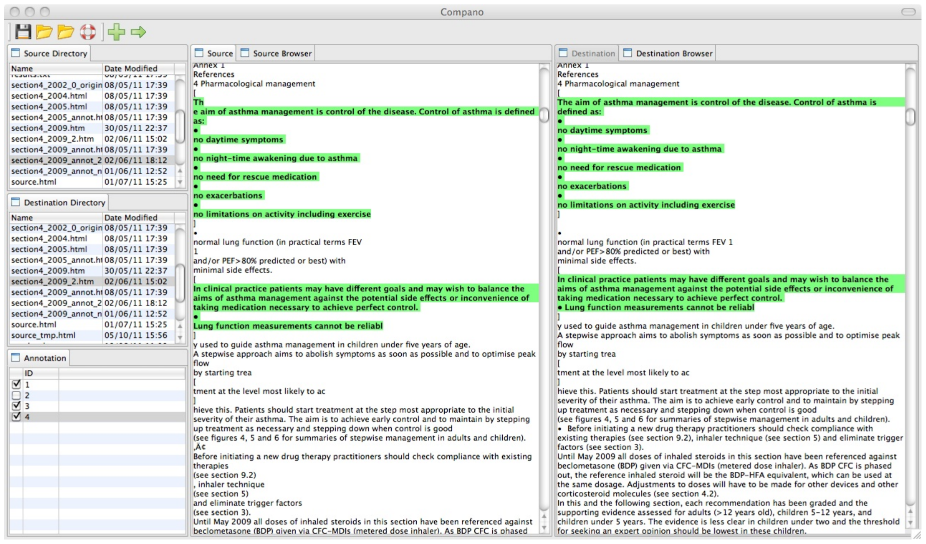
Destination Document The new document, in which annotations are to be integrated. When the user adds an annotation manually, he can mark the position where the annotation is to added in this view. After adding, an annotation it will be marked in this view.

Annotation List of annotations in the source document and whether they are placed or not. When manually adding an annotation, the user has to select the annotation in the *Annotation* view.

Source Folder This view shows all files of the selected directory. After selecting a file, it will shown in the *Source Document* view.

Destination Folder Same as destination folder view, but the file will shown in the *Destination Document* view.

The documents can be displayed as plain text or in an integrated HTML browser using the tabs in the source and target document views.



Name	Date Modified
section4_2002_0_origin	08/05/11 17:39
section4_2004.html	08/05/11 17:39
section4_2005.html	08/05/11 17:39
section4_2005_annot.ht	08/05/11 17:39
section4_2009.htm	30/05/11 22:37
section4_2009_2.htm	02/06/11 15:02
section4_2009_annot.ht	08/05/11 17:39
section4_2009_annot_2	02/06/11 18:12
section4_2009_annot_n	01/06/11 12:52
source.html	01/07/11 15:25

Name	Date Modified
section4_2002_0_origin	08/05/11 17:39
section4_2004.html	08/05/11 17:39
section4_2005.html	08/05/11 17:39
section4_2005_annot.ht	08/05/11 17:39
section4_2009.htm	30/05/11 22:37
section4_2009_2.htm	02/06/11 15:02
section4_2009_annot.ht	08/05/11 17:39
section4_2009_annot_2	02/06/11 18:12
section4_2009_annot_n	01/06/11 12:52
source.html	01/07/11 15:25
source_tmp.html	05/10/11 15:56

	ID
✓	1
☐	2
✓	3
✓	4

```
Annex 1
References
4 Pharmacological management
[
Th
e aim of asthma management is control of the disease. Control of asthma is defined
as:
•
no daytime symptoms
•
no night-time awakening due to asthma
•
no need for rescue medication
•
no exacerbations
•
no limitations on activity including exercise
]
•
normal lung function (in practical terms FEV
1
and/or PEF>80% predicted or best) with
minimal side effects.
[
In clinical practice patients may have different goals and may wish to balance the
aims of asthma management against the potential side effects or inconvenience of
taking medication necessary to achieve perfect control.
•
Lung function measurements cannot be reliabl
]
y used to guide asthma management in children under five years of age.
A stepwise approach aims to abolish symptoms as soon as possible and to optimise peak
flow
by starting trea
[
tment at the level most likely to ac
]
hieve this. Patients should start treatment at the step most appropriate to the initial
severity of their asthma. The aim is to achieve early control and to maintain by stepping
up treatment as necessary and stepping down when control is good
(see figures 4, 5 and 6 for summaries of stepwise management in adults and children).
‚Äc
Before initiating a new drug therapy practitioners should check compliance with existing
therapies
(see section 9.2)
, inhaler technique
(see section 5)
and eliminate trigger factors
(see section 3).
Until May 2009 all doses of inhaled steroids in this section have been referenced against
beclometasone (BDP) given via CFC-MDIs (metered dose inhaler). As BDP CFC is phased
```

```
Annex 1
References
4 Pharmacological management
[
The aim of asthma management is control of the disease. Control of asthma is
defined as:
•
no daytime symptoms
•
no night-time awakening due to asthma
•
no need for rescue medication
•
no exacerbations
•
no limitations on activity including exercise
]

•
normal lung function (in practical terms FEV 1
and/or PEF>80% predicted or best) with
minimal side effects.
[
In clinical practice patients may have different goals and may wish to balance the
aims of asthma management against the potential side effects or inconvenience of
taking medication necessary to achieve perfect control.
• Lung function measurements cannot be reliabl
]
y used to guide asthma management in children under five years of age.
A stepwise approach aims to abolish symptoms as soon as possible and to optimise peak
flow
by starting trea
[
tment at the level most likely to ac
]
hieve this. Patients should start treatment at the step most appropriate to the initial
severity of their asthma. The aim is to achieve early control and to maintain by stepping
up treatment as necessary and stepping down when control is good
(see figures 4, 5 and 6 for summaries of stepwise management in adults and children).
•  Before initiating a new drug therapy practitioners should check compliance with
existing therapies (see section 9.2), inhaler technique (see section 5) and eliminate trigger
factors (see section 3).
Until May 2009 all doses of inhaled steroids in this section have been referenced against
beclometasone (BDP) given via CFC-MDIs (metered dose inhaler). As BDP CFC is phased
out, the reference inhaled steroid will be the BDP-HFA equivalent, which can be used at
the same dosage. Adjustments to doses will have to be made for other devices and other
corticosteroid molecules (see section 4.2).
In this and the following section, each recommendation has been graded and the
supporting evidence assessed for adults (>12 years old), children 5-12 years, and
children under 5 years. The evidence is less clear in children under two and the threshold
for seeking an expert opinion should be lowest in these children.
```

Figure 15: User Interface

The menu (shown in figure 16 (p. 22)) is divided into one part for the file I/O operations, such as “save destination file”, “open source directory”, “open destination directory” and “show help” and one for the placement operations “add annotation manually” and “add annotations automatically”.

For the implementation of the interface, the *MVC Model - View - Controller* pattern was applied. Exclusively SWT components were used in the design since they are integrated part of the RCP framework.

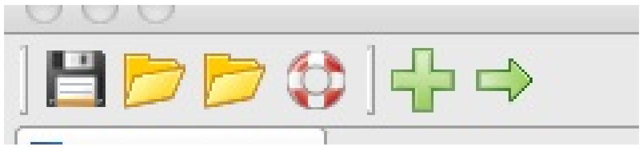


Figure 16: Software Menu, from left to right: Save Target Document. Open Source Directory. Open Target Directory. Show Help. Manual Annotation Placement. Automatic Annotation Placement.

4.3 Manual Annotation Placement

Figure 21 (p. 26) shows the inital situation of the documents. The left side shows the source file with the annotations (shown as a delta node) and the right side the destination file without the annotation.

For manual placement, only the text position (the start and end point within the text) and the id of the annotation is known. So, at the beginning of the algorithm, it is necessary to collect all text nodes in the destination file that have to be marked by the annotation. Since it is also possible that the start or the end point of the selected text is located in the middle of the string of a node, it is also necessary to get the exact point of the start or end point in the string. Figure 17 (p. 23) shows the position finding algorithm.

```java
public List<StringNodePair> findPosition(List<StringNodePair>
    destText, Point point) {

  List<StringNodePair> returnList = new ArrayList<StringNodePair>()
      ;

  boolean setPos = false;
  int counter = 0;
  int counterOld = 0;
  for(StringNodePair node : destText) {
    counterOld = counter;
    counter += node.getString().length();

    if(counter >= point.x) {

      if(setPos == false) {
        // start of the selected text in the snp
        posInSnp.x = point.x - counterOld;
        setPos = true;
      }

      returnList.add(node);
    }
    counter++;
    if(counter >= point.y) {
      // end of the selected text in the snp
      posInSnp.y = point.y - counterOld;
      break;
    }


  }
  return returnList;
}
```

Figure 17: Code for the position finding algorithm

In case the start or end point is located in the middle of a node, that node has to be split so that the to-be-surrounded string is the child of a single node. This is necessary to place the annotation at the desired position. It is possible that the selected text is located at the start, end or middle of the node, so there are 3 cases that the splitting algorithm had to deal with. Furthermore the algorithm has to guarantee that the structure of the DOM tree stays intact. To guarantee this, the algorithm divides the string into the needed text parts and then generates new nodes out of the strings. Figure 19 (p. 25) shows a code snippet of the third case where the text, which should be annotated, is in the middle of the String. Figure 34 (p. 42) structure of the Node before and after the method.

After the splitting process, the new annotation nodes can be placed. There are two cases for this operation. In the first case, there is only one node that must be surrounded by an annotation and in the second case there are several nodes that must be surrounded by the annotation. To add an annotation, it is necessary to create a new annotation node element. The new annotation node will replace the current node in the tree, and the current node will become the child of the annotation node. With this process, the integrity of the DOM-tree is ensured. Figure 20 (p. 25) shows the replacement algorithm for manual annotation placement.

The code of the main method of the manual placement of the annotations is shown in figure 18 (p. 24).

```java
public List<StringNodePair> manualPlacement() {

  // gather all nodes for the annotation
  FindPosition find = new FindPosition();
  List<StringNodePair> snps = find.findPosition(destText,
      selectedPoint);
  Point posInSnp = find.getPosInSnp();

  // if it is needed split the nodes
  int lengthLastSnp = snps.get(snps.size()-1).getString().length();
  if(posInSnp.x != 0 || posInSnp.y != lengthLastSnp) {
    snps = splitSnp(snps, posInSnp);
  }

  // finaly place the annotation
  PlaceAnno mp = new PlaceAnno(snps, anno, destDoc);
  destText = mp.placeDeltas();

  return destText;
}
```

Figure 18: Manual annotation placement

In figure 33 (p. 41), the process of inserting the annotation manually is shown in a sequence diagram.

4.4 Automatic Annotation Placement

Simply speaking, automatic placement of an annotation element is the same as in chapter 4.3 (p. 22), with the addition of iterating through annotations and texts in the search of matches, as seen in the following pseudocode (figure 22 (p. 26)). Typically, however, “the devil’s in the details”. And so it was.

```
public StringNodePair splitTextNodesAndAdd(String anno,
    StringNodePair snp) {
  Node cur = snp.getNode();
  Node par = cur.getParentNode();

  String total = cur.getTextContent();
  Node annoNode = doc.createTextNode(anno);

  if (!anno.equals(total)) {
    String[] split = total.split(anno);

    if (total.startsWith(anno)) {
      // ... text at the beginning of the string
    } else if (total.endsWith(anno)) {
      // ... text at the end of the string
    } else {
      Node mid = annoNode;
      Node before = doc.createTextNode(split[0]);
      Node after = doc.createTextNode(split[1]);

      par.replaceChild(after, cur);
      par.insertBefore(mid, after);
      par.insertBefore(before, mid);
    }
    snp.setN(annoNode);
    snp.setS(anno);

  }
  return snp;
}
```

Figure 19: Code to split a textnode

```
private void placeSingleDelta() {
  Node theNode = snps.get(0).getNode();
  String id = "delta:" + anno.getId();
  placeBeginningBracket(theNode, id);

  Element newNode = doc.createElement("a");
  newNode.setAttribute("id", id);

  Node parent = theNode.getParentNode();
  Node replacedNode = parent.replaceChild(newNode, theNode);
  newNode.appendChild(replacedNode);

  placeEndBracket(newNode, id);
}
```

Figure 20: Code for the placement of a single annotation

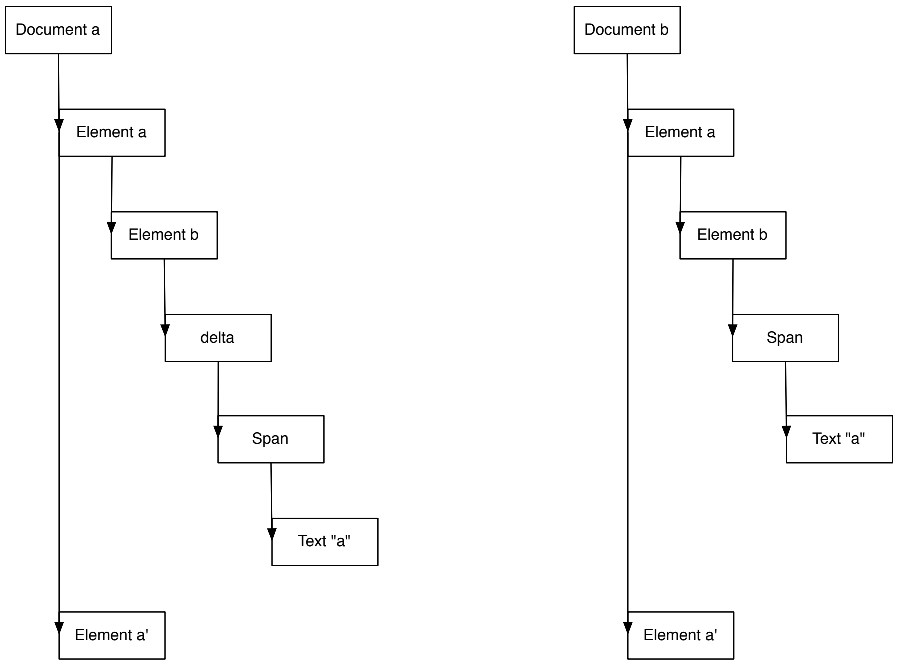


Figure 21: Initial Situation of Documents (simplified)

```
For Annotation a in sourceDocument.getAnnotations() {
  For Text t in targetDocument.getTexts() {
    // If the texts match...
    If (t == a.text) {
      // ... surround the text with an ...
      // ... annotation whose basis is 'a'
      t.surroundWithAnnotation(a);
    }
  }
}
```

Figure 22: Pseudocode for the automatic placement of annotations

4.4.1 Simple Annotations

Figure 35 (p. 43) shows the possible cases when trying to automatically insert annotations into an XML structure. The annotation originally surrounded the string "$\alpha\beta\chi$". The algorithm now has to search for adjacent nodes whose textual content contains the required string. Once the adjacent nodes have been found, the XML structure of the target document must be updated, so that the string "$\alpha\beta\chi$" is the child of a single node. Figure 36 (p. 44) shows the restructuring that must take place. The last step is the insertion of the annotation as the span element's parent; the DOM API provides simple methods for performing this operation.

4.4.2 Spread Annotations

The matter of spread or multi-annotations is a very different one. Figure 24 (p. 29) shows the object network that needs to be created when mapping existing annotations to text in a target document. Especially the (m, n) connection between Annotations and StringNoderPairs is a tricky algorithm.

The first level of iteration concerns the existing annotations that are to be automatically placed. Within this iteration, the first order of business is to immediately check if the specific annotation (in the source document) has already been placed (automically or manually); if this is the case, further processing of the annotation is skipped and the iteration continues.

If the specific annotation has not been placed, the algorithm begins an iteration over the annotation's (multiple) *starting node* elements. The definition of *starting node* is given in chapter 2.3 (p. 12). For each starting node, the algorithm generates a string consisting of the current annotation-part's nodes and a string of the destination document. Then, a standard text comparison of these two strings returns the exact position of the annotation-part's nodes in the destination document. With this position information, the algorithm calls the manual placement methods to place the annotations. Figure 23 (p. 28) shows the algorithm after the check if the annotation has already been placed.

```
List<Point> annoPoints = new ArrayList<Point>();
List<Node> startNodes = anno.getStartingNodes();

String pureText = getPureText();

// search for all start nodes of the annotation
for (Node startNode : startNodes) {
  String annoText = getFullAnnoTest(startNode, anno);

  Point annoPoint = new Point(0, 0);
  annoPoint.x = pureText.indexOf(annoText);
  annoPoint.y = annoPoint.x + annoText.length();

  if (annoPoint.x != annoPoint.y && annoPoint.x >= 0 && annoPoint.y
      >= 0) {
    annoPoints.add(annoPoint);
  } else {
    annoPoints.clear();
    break;
  }
}

/** if the algorithm found all start nodes add the annotation
 * by calling the manuall placement methods.
 */
if (!annoPoints.isEmpty()) {
  for (int i = annoPoints.size() - 1; i >= 0; i--) {
    Point annoPoint = annoPoints.get(i);
    GeneralPlacement gp = new GeneralPlacement(destText, destDoc);
    destText = gp.generalPlacement(false, annoPoint, anno);
  }
}
```

Figure 23: Code for the automatic placement of an annotation

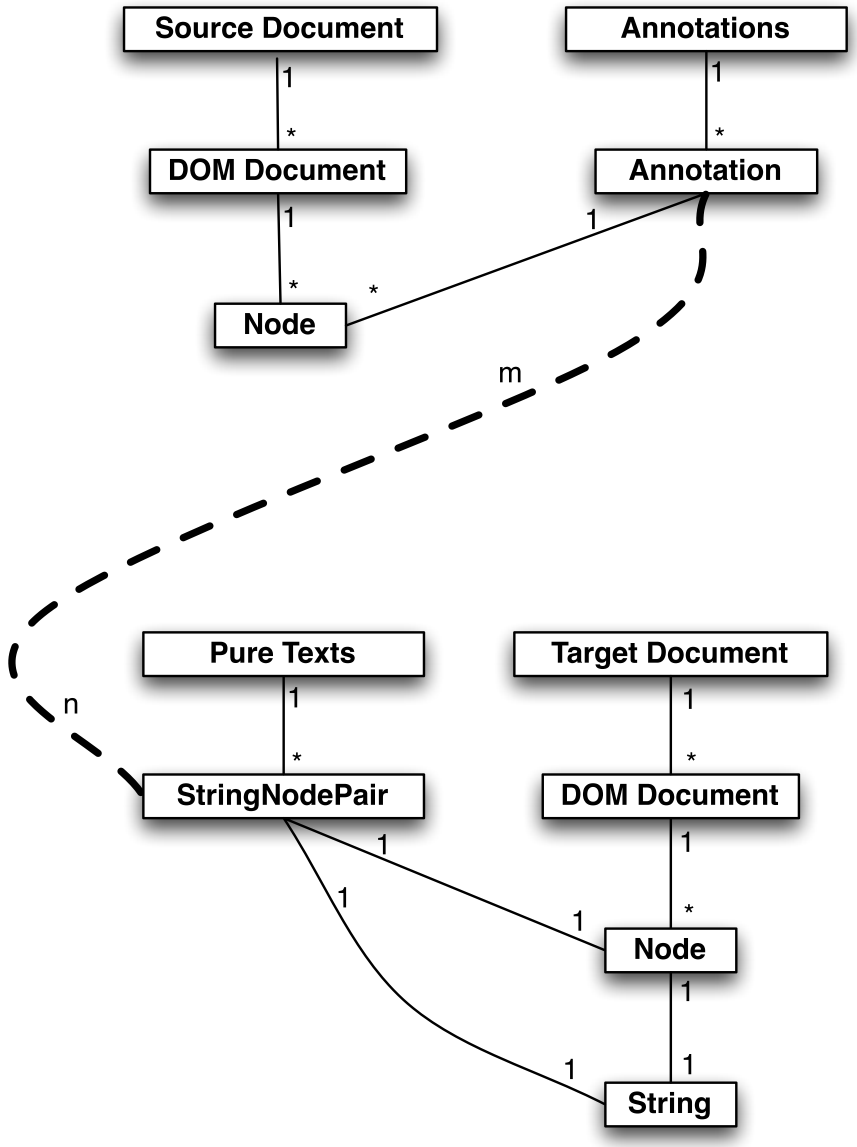


Figure 24: Object Network for Automatic Placement

5 Conclusion & Evaluation

This chapter describes the results of the research and an evaluation thereof.

5.1 Lessons Learned

A number of lessons were learned over the course of the project; some of them quite expected, other not so much. The large surprises were related to HTML, character sets and PDF exports and are described below.

5.1.1 HTML

The processing of HTML through programming languages is a common procedure today. However, these procedures mostly deal with *pretty*, specifically, not corrupted through various character sets and insufficient formatting styles, HTML code. This was not the case, as described in chapters 2.2 (p. 9), 5.1.3 (p. 32). Therefore, the first goal of processing was to *clean* the exisiting code to bring into a more usable form.

A number of utitilies that clean up the formatting and what could partially be termed “abuse” exist. These utitlies are very practical and available in portations to a number of programming languages. A program that specifically helped in the analysis of the HTML was *JTidy*[7]: The tool parses HTML documents and strips them of unnecessary information while restructuring them for better readibility and usability. After processing, the data quality had improved, but was still insufficient for further processing. This is due to the fact that the software aims to keep the content of the code equal to its previous version. Not helpful in this case: Empty tables, random spacing, etc. do not need to kept in a document; they only complicate handling the data and provide no information. For this reason, we decided to implement our own functionality to perform further cleaning. Through analysis, trial & error, we concluded that the regular expressions in figure 25 (p. 30) rendered the document's data in a state sufficient (but not excellent) for further processing.

```
<SPAN[a-zA-Z="0-9,:#;'\n\.\-\_ ]*>
</SPAN\n?>
<span[a-zA-Z="0-9,:#;'\n\.\-\_ ]*>
</span\n?>
<TD[a-zA-Z="0-9,:#;'\n\.\-\_ ]*/>
<TR>\n*</TR>
<TABLE>\n*</TABLE>
\n+
<a id="delta\:[0-9]+"*>\n*</a>
```

Figure 25: Regular Expressions denoting unnecessary strings

[7]See `http://jtidy.sourceforge.net/`

It is sensible to include an explanation of what the strings in figure 25 (p. 30) denote in prose and why they can be removed:

Lines 1,2,3,4 These expression represent *span* elements, which are additional formatting information without content. Specifcally the newline characters caused problems in parsing and formatting.

Line 5 This string represents an empty table data cell. It contains no information nor any children and thus can be removed before parsing.

Line 6 The empty table row, denoted by this expression, is not necessary. Again, the newline characters also caused problems during parsing.

Line 7 As a super class of the two above named entities, the empty table is also not necessary and only causes white space in the displayed document.

Line 8 A sequence of newline characters indicate only white space in the document; this is of no use for processing.

Line 9 This is an annotation as described in chapter 2.3 (p. 12). However, the annotation surrounds only a string of newline characters and therefore provides no information.

5.1.2 Character Sets

The character encoding is a mapping between single characters and numerical values that represents this symbol in a file or data stream. The signs and symbols are stored as numerical values to allow an easy and consistent way of storing and transmitting them. It is important to use always the same character set, otherwise the characters can be misinterpreted. For example the numeric value of the character Ü is in the character set ISO-8859-1 220 and in the character set EBCDIC the numeric value 220 represents the symbol } [2].

By using a different character set, it is possible that some characters of the text could be misinterpreted and so that the texts are no longer comparable. In our case, this problem occurs when the newer file was converted out of a PDF and the older file is still an HTML.

For example it is possible that during the process of converting a PDF to an HTML and manipulate that HTML in the program Delt/A a "•" can change to another character and so it is no longer comparable to the "•" in the original file.

The lifeline of a "•" is shown in Figure 26 (p. 32) - 28 (p. 33) and in table 1 (p. 31).

PDF	After converting PDF → HTML	Shown in Delt/A	After saving with Delt/A
•	•	,Ä¢	&#8218;&#196;&#162;

Table 1: Lifeline of a "•" character

```
<TR>
<TH style="width:13px; height:16px">
<SPAN style="font-family:'sans-serif', 'Wingdings', sans-serif;
    font-size:10pt; font-weight:normal; color:#000000">
•
</SPAN>
</TH>
<TH colspan=2 style="width:517px; height:16px">
<SPAN style="font-family:'sans-serif', 'CG Omega', sans-serif; font
    -size:10pt; font-weight:normal; color:#000000">
no daytime symptoms
</SPAN>
</TH>
</TR>
```

Figure 26: HTML file after converting from PDF.

```
<tr>
<th  style="height: 18px; width: 13px">
<span>
<font  face="sans-serif, Wingdings, sans-serif"  color="#000000"
    size="10pt">
‚Ä¢
</font>
</span>
</th>
<td  style="height: 18px; width: 517px"  colspan="2">
<span>
<font  face="sans-serif, CG Omega, sans-serif"  color="#000000"
    size="10pt">
no exacerbations
</font>
</span>
</td>
</tr>
```

Figure 27: HTML file open with Delt/A

5.1.3 PDF Exports

The most important lesson of this project: *Do not try to work with HTML data exported from PDF files.* A different approach is to perform the necessary operations directly in the PDF format, without using the existing “Export to HTML” functionality as a pre-cursor.

5.1.4 Low Quality Exports

The export functionality provided by the Adobe[8] PDF software is not sufficient. Detailed documentation on the matter was not found, but it’s assumed that the

[8]See `http://www.adobe.com/`

```
<tr>
<th style="height: 16px; width: 13px">
<span>
<font face="sans-serif, Wingdings, sans-serif" color="#000000" size
    ="10pt">
&#8218;&#196;&#162;
</font>
</span>
</th>
<th style="height: 16px; width: 517px" colspan="2">
<span>
<font face="sans-serif, CG Omega, sans-serif" color="#000000" size=
    "10pt">
no daytime symptoms
</font>
</span>
</th>
</tr>
```

Figure 28: HTML file saved with Delt/A

algorithm works with a *visual* analysis of the document. The following problems occur:

Characters The algorithm mistakenly interprets some characters (β, for instance) as other characters ("B"). It also interprets some formatting characters (such as bullet points) as textual characters. Another inaccuracy is the placement of some characters in relation to the others: Sometimes characters are placed in superscript or subscript falsely.

Formatting The algorithm attempts to deliver an HTML document that matches the exact display of the original. This leads to several empty *table* tags, among other things. Oher constructs are also misintepreted: Lists are apparently not recogznized at all, other times paragraphs are split into tables.

Logical The two possibilities named above lead to the separation of data that belongs together logically (i.e. content that would be grouped together by e.g. a *p* (paragraph) element). This leads to problems down the road when searching for specific strings in the document.

Of course, a visual analysis of a document will never be as perfect as the analysis of its formal declaration would be. However, this amount of *data destruction* makes it very difficult to enable any further processing.

5.1.5 Other APIs

A number of APIs that allow the direct manipulation of PDFs from a programming language obviate the HTML export. Only a brief evaluation of the individual APIs was done (specifically, the APIs were not tested in practice):

iText "iText is a library that allows you to create and manipulate PDF documents. It enables developers looking to enhance web- and other applications with dynamic PDF document generation and/or manipulation." [4]

This library is available open source and commercial form; it has been used in larger projects with suitable results by companies and research institutes.(See [4])

Asprise Java PDF Library "Asprise offers PDF writer and reader library (with text extract function) [...]. Portable Document Format (PDF) is a file format widely used for all kinds of documents. With Asprise Java PDF library, you can easily create, manipulate (read and write), disassemble PDF files easily. You can also use it to extract text and then index the text extracted for search." [1]

This API offers another approach entirely, allowing for extraction of the pure text elements for further processing. This was attempted in this particular project as well, but was not possible with the HTML data (see chapter 2.2 (p. 9)). A downside to this API is its commercial aspect[9].

5.2 Evaluation

5.2.1 Delivered Application

General The solution covers 100% of the agreed tests and a large part of the defined business process as shown in figure 29 (p. 35). The *comparison of documents* module was not included in the final version for valid reasons: The differences in formatting and character encodings made the texts seem as if there were almost no common passages.

The general finding of the research & project is that reverting PDF data to other formats for further processing is an effort not worth pursuing: The amount of research that must be done to implement successful algorithms is exponentially higher than when working with other standard formats or (possibly) work directly with the PDF data via libraries (such as listed in 5.1.5 (p. 33)).

Manual Migration The algorithm allows the user to manually place the annotation as desired, including multi-annotations by repeating the manual placement for the same annotation. The user is even allowed to place an annotation *within* another annotation: This may or may not be troublesome for other application.

Automatic Migration The delivered algorithms provide the user with support in the migration of existing annotations. The automatic placement capabilities aim toward delivering absolutely no false positives, i.e. the automatic migration functions only if a perfect match is located in the

[9]Of course, this is typically also an indication of a certain level of quality.

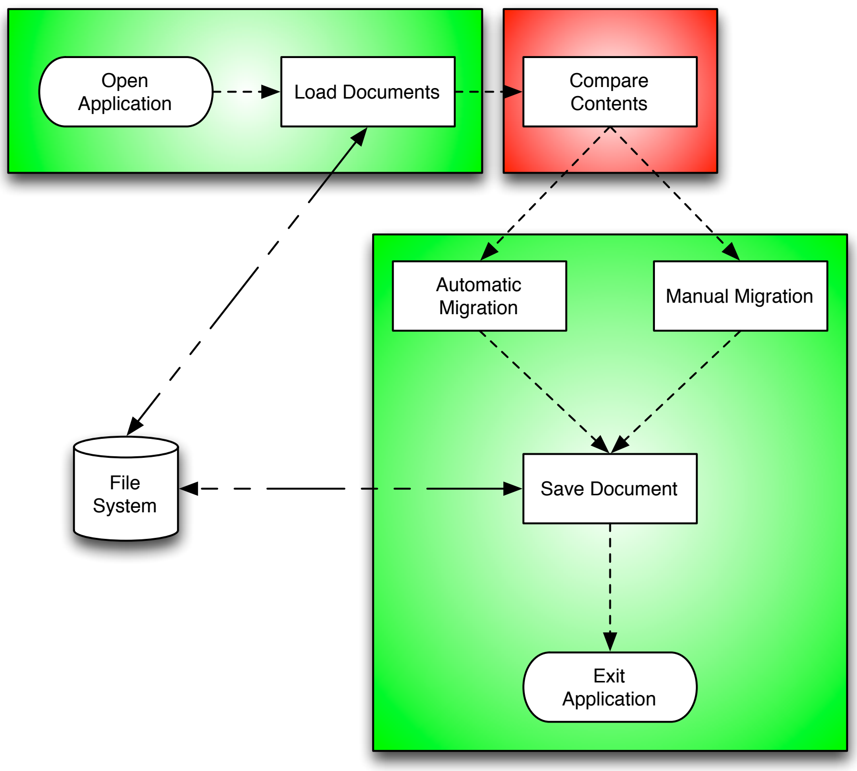


Figure 29: Implemented Modules (Legend: Green = Implemented / Red = Not Implemented)

target document. This has two consequences: (1) The chances of the algorithm migrating 100% of the annotations sinks drastically, however (2) the user does not need to spend time performing post-processing and review work. This trade-off could also have been tilted in the other direction (placing *all* annotations and forcing post-processing), however after brief evaluation this was considered to create more work than it saved.

5.2.2 Manual Test Driven Development

To ensure goal-oriented development, we decided to work with the *Test-driven Development* technique:

> “Test-driven development (TDD) is a software development process that relies on the repetition of a very short development cycle: first the developer writes a failing automated test case that defines a desired improvement or new function, then produces code to pass

> that test and finally refactors the new code to acceptable standards.” [10]

After a brief analysis, we realised that automated tests would not work well with this project: Too many of its aspects were purely visual and (due to the malformed data, see chapter 2.2 (p. 9)) not testable. This is why we decided to formulate the tests in prose before starting development on an issue and then working toward that goal with manual testing.

5.2.3 Pair Programming

The style of prgoramming we mainly pursued was pair programming:

> “Pair programming is an agile software development technique in which two programmers work together at one workstation. One types in code while the other reviews each line of code as it is typed in.” [7]

This approach came quite naturally: A two person team almost automatically starts programming this way. It also served our paradigm *Communicate Everything*. During the project, we found that several of the more complicated topics, as described in chapters 4.3 (p. 22) & 4.4 (p. 24), were solved with much greater ease than when we had tried working out the problems on our own.

5.2.4 Common Sense Project Management

Common Sense project management is a concept we thought up after realising that even agile project management bore too much formality, which would have swamped the project. Another reason to strip formalities was that events such as formal *update meetings* only make sense if members of the team work on individual tasks and not, as in our case, on one specific task. Over time, we found the procedure below to be most effective. It specifically allowed us to work in irregular sessions and treat each session as a closed-off entity that contained everything the project (and project management in general) required. Please see appendix A, figure 31 (p. 39) for details.

5.3 Conclusion

Due to insufficient time and resources, it was not possible to test the application in *real life*. However, as the results of the research showed that this type of solution *can* work, but that it is not the optimum, it is not wise to invest large efforts in this direction.

A Appendix

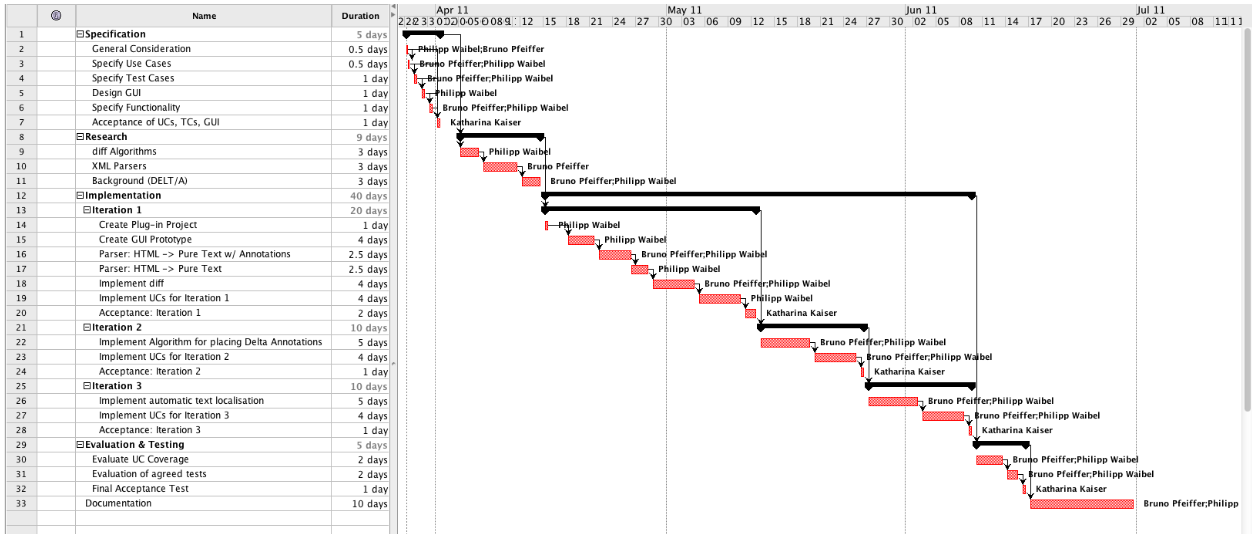
A.1 Tools

The tools and technologies that make up the foundation of the solution are:

1. Eclipse RCP
2. Oracle Java v1.6.0
3. CyberNeko HTML Parser

A.2 Figures & Documents

The following figures serve to better illustrate certain parts of research, project & solution.



		Name	Duration
1		⊟Specification	5 days
2		General Consideration	0.5 days
3		Specify Use Cases	0.5 days
4		Specify Test Cases	1 day
5		Design GUI	1 day
6		Specify Functionality	1 day
7		Acceptance of UCs, TCs, GUI	1 day
8		⊟Research	9 days
9		diff Algorithms	3 days
10		XML Parsers	3 days
11		Background (DELT/A)	3 days
12		⊟Implementation	40 days
13		⊟Iteration 1	20 days
14		Create Plug-in Project	1 day
15		Create GUI Prototype	4 days
16		Parser: HTML -> Pure Text w/ Annotations	2.5 days
17		Parser: HTML -> Pure Text	2.5 days
18		Implement diff	4 days
19		Implement UCs for Iteration 1	4 days
20		Acceptance: Iteration 1	2 days
21		⊟Iteration 2	10 days
22		Implement Algorithm for placing Delta Annotations	5 days
23		Implement UCs for Iteration 2	4 days
24		Acceptance: Iteration 2	1 day
25		⊟Iteration 3	10 days
26		Implement automatic text localisation	5 days
27		Implement UCs for Iteration 3	4 days
28		Acceptance: Iteration 3	1 day
29		⊟Evaluation & Testing	5 days
30		Evaluate UC Coverage	2 days
31		Evaluation of agreed tests	2 days
32		Final Acceptance Test	1 day
33		Documentation	10 days

Apr 11
May 11
Jun 11
Jul 11
Philipp Waibel;Bruno Pfeiffer
Bruno Pfeiffer;Philipp Waibel
Bruno Pfeiffer;Philipp Waibel
Philipp Waibel
Bruno Pfeiffer;Philipp Waibel
Katharina Kaiser
Philipp Waibel
Bruno Pfeiffer
Bruno Pfeiffer;Philipp Waibel
Philipp Waibel
Philipp Waibel
Bruno Pfeiffer;Philipp Waibel
Philipp Waibel
Bruno Pfeiffer;Philipp Waibel
Philipp Waibel
Katharina Kaiser
Bruno Pfeiffer;Philipp Waibel
Bruno Pfeiffer;Philipp Waibel
Katharina Kaiser
Bruno Pfeiffer;Philipp Waibel
Bruno Pfeiffer;Philipp Waibel
Katharina Kaiser
Bruno Pfeiffer;Philipp Waibel
Bruno Pfeiffer;Philipp Waibel
Katharina Kaiser
Bruno Pfeiffer;Philipp

Figure 30: Project Plan

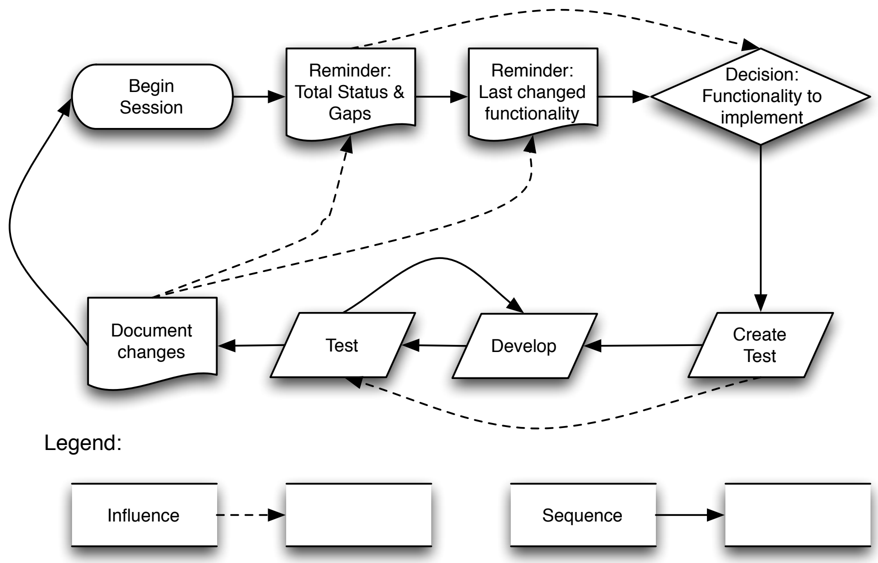

Begin Session
Reminder: Total Status & Gaps
Reminder: Last changed functionality
Decision: Functionality to implement
Create Test
Develop
Test
Document changes
Legend:
Influence
Sequence

Figure 31: Common Sense Process

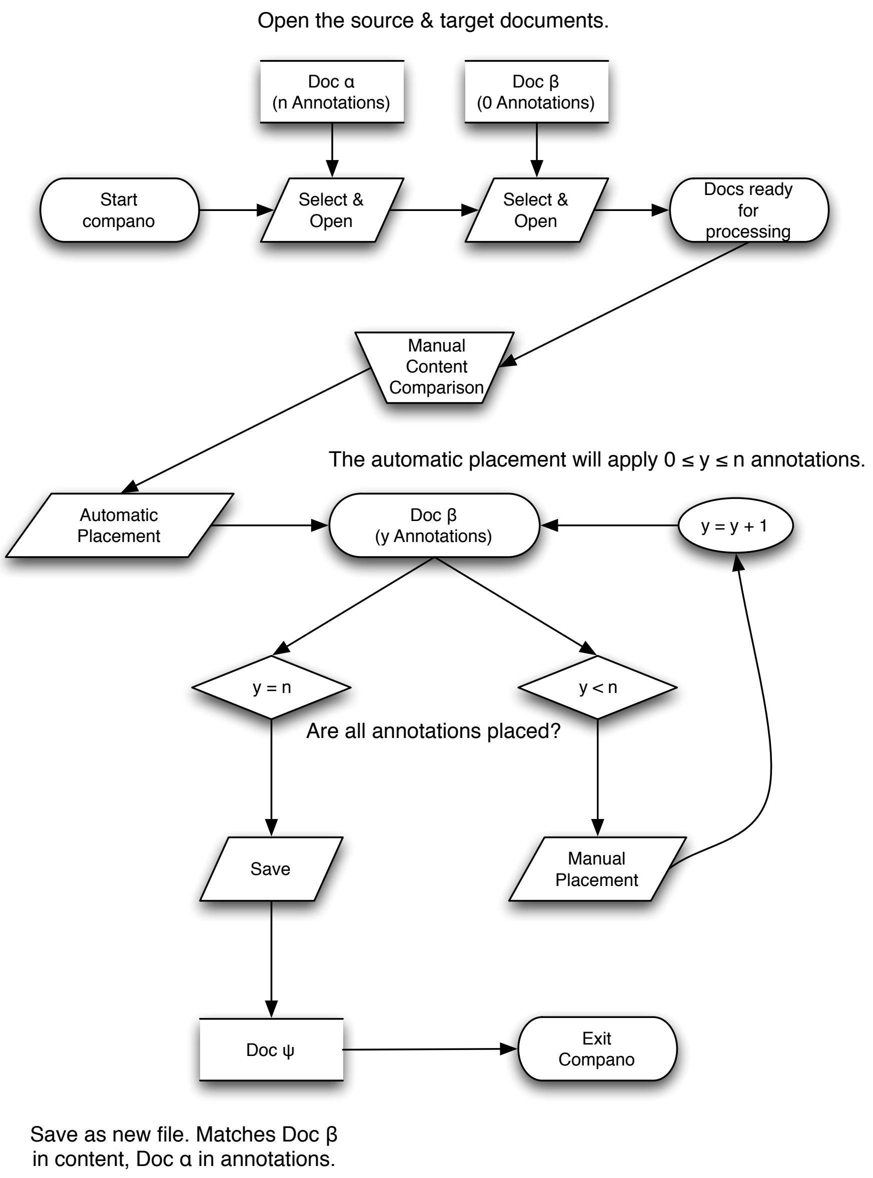

Open the source & target documents.
Doc α (n Annotations)
Doc β (0 Annotations)
Start compano
Select & Open
Select & Open
Docs ready for processing
Manual Content Comparison
The automatic placement will apply 0 ≤ y ≤ n annotations.
Automatic Placement
Doc β (y Annotations)
y = y + 1
y = n
y < n
Are all annotations placed?
Save
Manual Placement
Doc ψ
Exit Compano
Save as new file. Matches Doc β in content, Doc α in annotations.

Figure 32: Typical Usage Steps

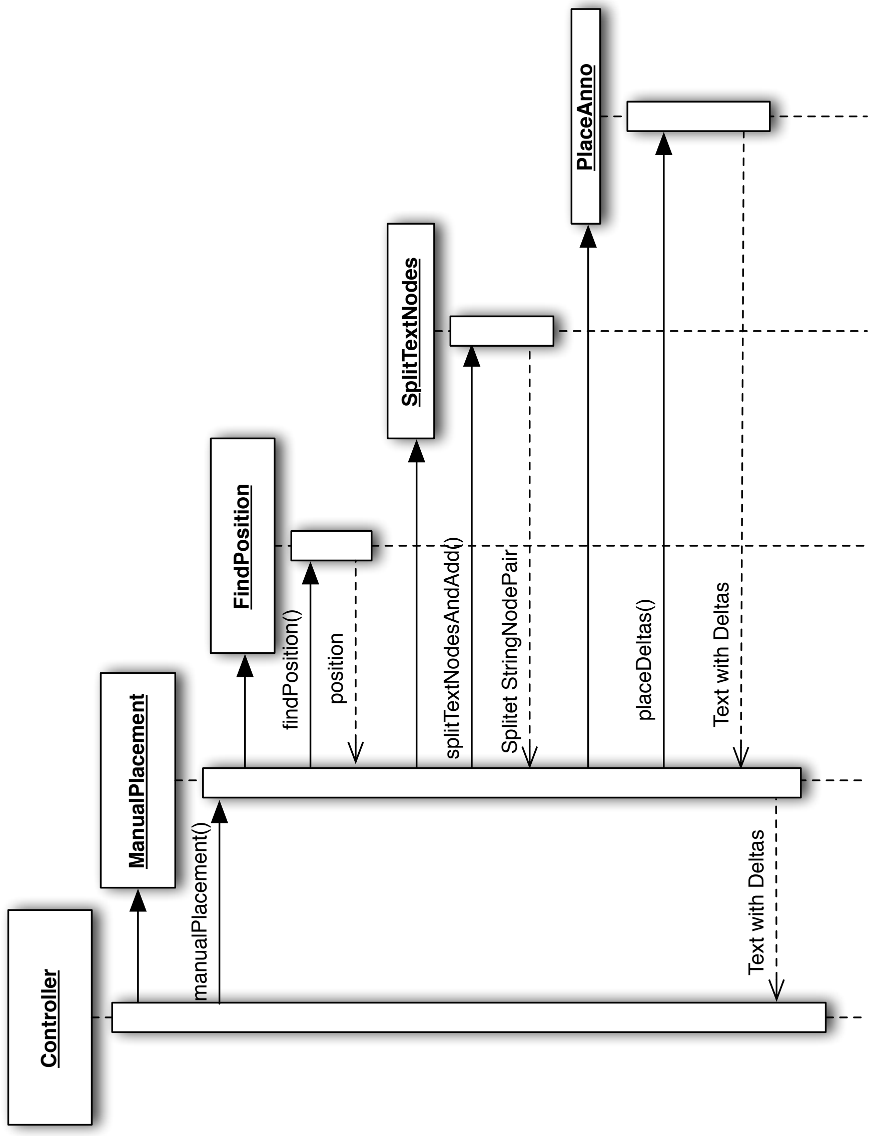


Figure 33: Sequence diagram of manual annotation placement

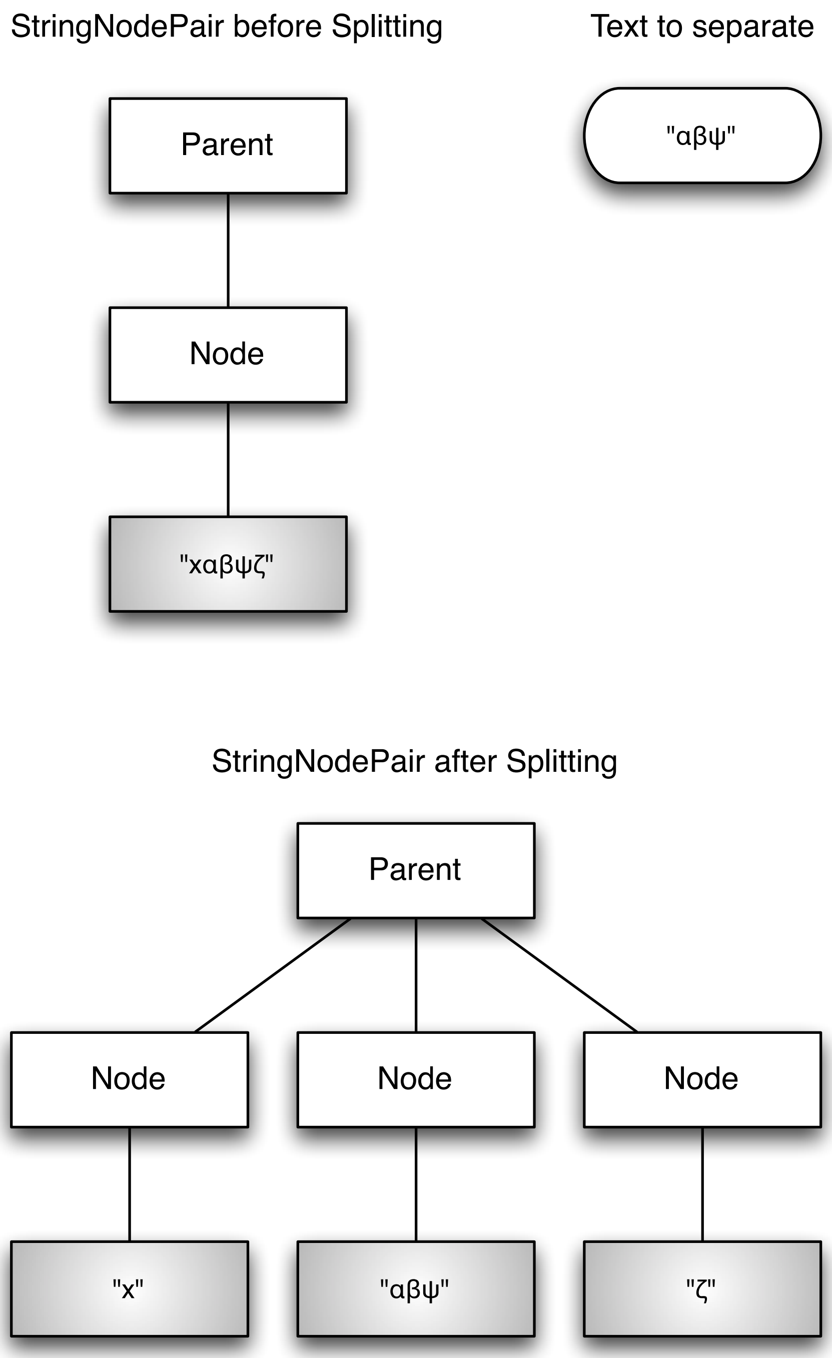


Figure 34: Before and after split a StringNodePair

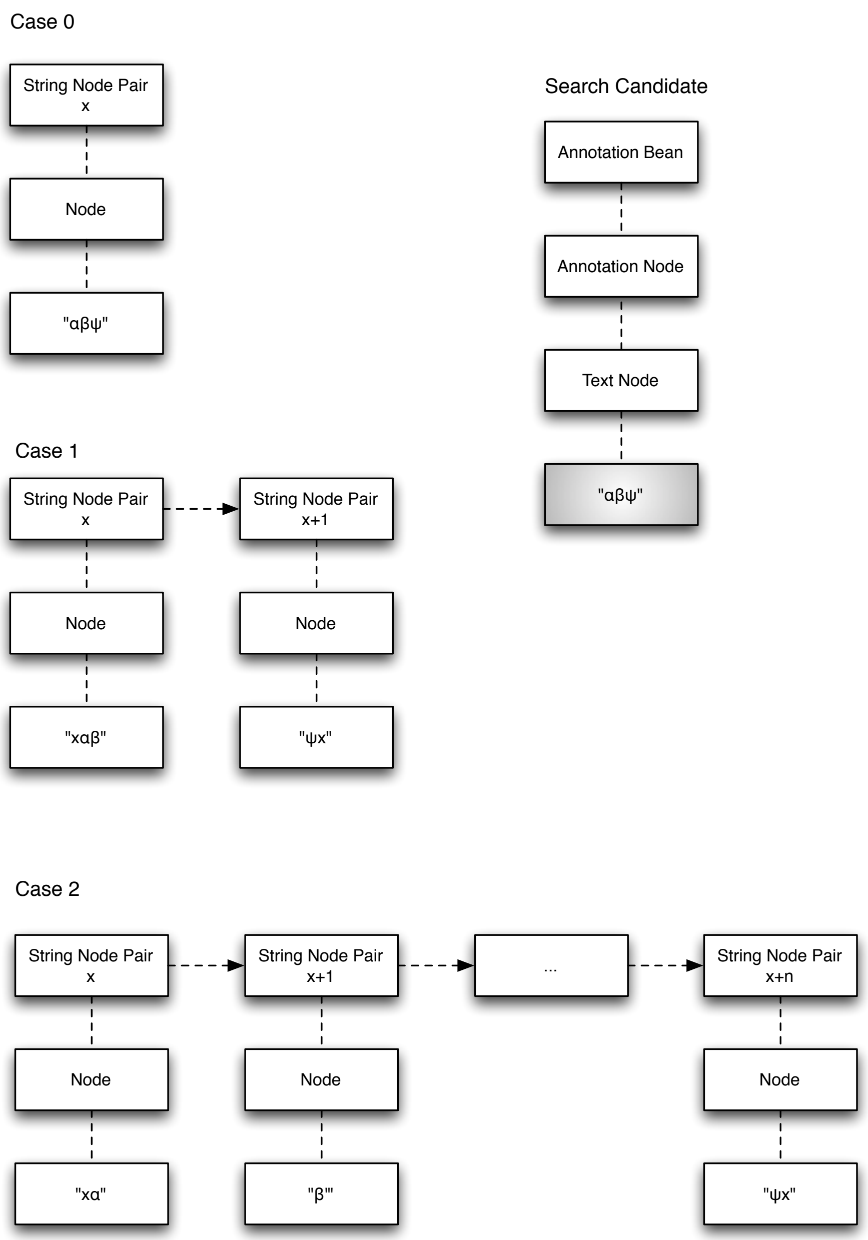


Figure 35: Possible cases when searching for text

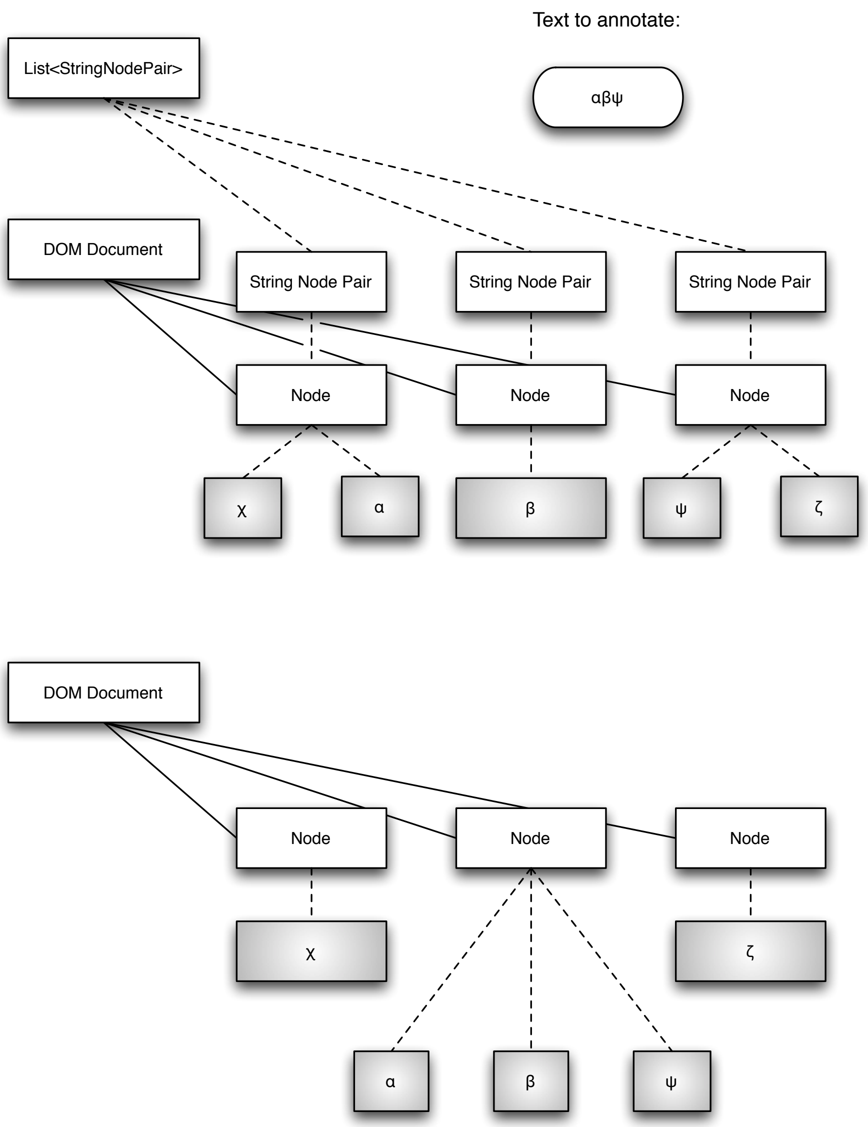


Figure 36: One case of node restructuring upon annotation insertion

Figure 37: Use Cases

ID	1	Milestone	3.1
Name	Start Compano	Component	GUI, CORE
Responsible	BP	Accepted	
Description	Start the program by double-clicking on the executable .jar file.		
Trigger	Execute .jar file		
Pre-Conditions	Compano is not running		
Post-Conditions	Program up and running		
Events	Pop up screen, then program shown		
Exceptions	None		

ID	2	Milestone	3.1
Name	Select Working Directory	Component	GUI
Responsible	BP	Accepted	
Description	Select Working Directory to show the Files in the corresponding frame		
Trigger	Menu Bar -> Select Working Directory X		
Pre-Conditions	UC1		
Post-Conditions	Show files of selected working directory in the corresponding frame		
Events	Show directory selection window		
Exceptions	Selected file not a directory		

ID	3	Milestone	3.3
Name	Save document	Component	GUI, CORE
Responsible	BP	Accepted	
Description	Save changes to target		
Trigger	Menu Bar -> Save		
Pre-Conditions	Target is open		
Post-Conditions	Saved new version		
Events	Show file selection window		
Exceptions	File already exists: Show Popup "File already exists, are you sure you want to override?"		

ID	4	Milestone	3.3
Name	Restore	Component	GUI
Responsible	BP	Accepted	
Description	Reset the workbench layout to the default layout		
Trigger	Menu Bar -> Reset Workbench		
Pre-Conditions	UC1		
Post-Conditions	Default layout of the workbench is restored		
Events	“Are you sure” – Box		
Exceptions	None		

ID	5	Milestone	3.3
Name	Show Help	Component	GUI
Responsible	BP	Accepted	
Description	Access the tool's help documentation		
Trigger	Menu Bar -> Help		

Pre-Conditions	UC1
Post-Conditions	Opened Thesis PDF file in the default PDF reader
Events	None
Exceptions	File not found.

ID	6	Milestone	3.3
Name	Quit program	Component	GUI
Responsible	BP	Accepted	
Description	Quit the program		
Trigger	Menu Bar -> Quit		
Pre-Conditions	UC1, Target is saved		
Post-Conditions	Closed program		
Events	"Do you want to exit" – Box		
Exceptions	None		

ID	7	Milestone	3.1
Name	Compare Docs	Component	CORE
Responsible	BP / PW	Accepted	
Description	Perform the diff algorithm		
Trigger	Action Bar -> Compare documents		
Pre-Conditions	Both docs must be open		
Post-Conditions	Mark the differences in target		
Events			
Exceptions	No or only one file is open: Show error in a Popup		

ID	8	Milestone	3.2
Name	Manual Add Annotation	Component	CORE, GUI
Responsible	BP / PW	Accepted	
Description	Add selected annotation at the desired location in new version		
Trigger	Action Bar -> Add selected annotation		
Pre-Conditions	An unused annotation is selected and a position, with no other annotation, is highlighted in the target		
Post-Conditions	Selected annotation is in the target and is shown in the frame.		
Events			
Exceptions	• No annotation is selected • No position in the second doc is selected • On the selected position in the second doc is already an annotation		

ID	9	Milestone	3.3
Name	Automatic Add Annotation	Component	CORE, GUI
Responsible	PW	Accepted	
Description	Add all unplaced annotations automatically.		
Trigger	Action Bar -> automatically add annotations		
Pre-Conditions	Target and source are opened.		
Post-Conditions	All unplaced annotations are placed in the target and are shown in the frame		
Events			

Exceptions	• The corresponding place for a annotation could not be found • Another annotation is already on the place where the new annotation should be set

ID	10	Milestone	3.1
Name	Select document	Component	GUI
Responsible	PW	Accepted	
Description	Select the document to be examined from a typical tree view.		
Trigger	Select document in working directory frame		
Pre-Conditions	Files of the working directory are shown in frame (UC2)		
Post-Conditions	Show the selected file in the file frame		
Events			
Exceptions	Selected file is not in the correct format		

ID	11	Milestone	3.1
Name	Annotation List	Component	CORE, GUI
Responsible	PW	Accepted	
Description	See all annotations of source in list view in the annotation frame		
Trigger	Open source in the working directory frame		
Pre-Conditions	Original file is loaded		
Post-Conditions	All annotations of the original doc are listed in the corresponding frame		
Events			
Exceptions			

ID	12	Milestone	3.2
Name	See if annotation is placed in new version	Component	CORE, GUI
Responsible	PW	Accepted	
Description	Annotation View: Place a checkmark next to anno's placed in new version		
Trigger	If a Annotation was added manually or automatically		
Pre-Conditions	Annotation is unchecked in the annotation view		
Post-Conditions	Annotation is checked in the annotation view		
Events			
Exceptions			

ID	13	Milestone	3.2
Name	Sort	Component	CORE, GUI
Responsible	PW	Accepted	
Description	Annotation View: Sort the list by number, group by the fact that they are placed or not.		
Trigger	Sort button is pressed		
Pre-Conditions	Annotation list is filled		
Post-Conditions	Show sorted list		
Events			
Exceptions	None		

ID	14	Milestone	3.1
Name	Strip HTML of Delta	Component	CORE
Responsible	BP / PW	Accepted	
Description	Remove all instances of <delta>, obtain clean HTML		
Trigger	Open original doc in the working directory frame		
Pre-Conditions	HTML file with all annotations		
Post-Conditions	HTML file with no annotations		
Events			
Exceptions	• Only an open <delta> Tag was found and no close Tag		

ID	15	Milestone	3.1
Name	Test well formed	Component	CORE
Responsible	BP / PW	Accepted	
Description	After removing the Delta Tags, the doc should be well formed		
Trigger	After strip THML of delta (see UC 15) the test will be executed		
Pre-Conditions	HTML file without the annotations		
Post-Conditions	True if the HTML file is well formed, false if it is not well formed		
Events			
Exceptions	• The HTML file is not well formed		

ID	16	Milestone	3.1
Name	Pure text from HTML	Component	CORE
Responsible	BP / PW	Accepted	
Description	Gather pure text string for diff from HTML. Know String's positions		
Trigger	After the doc was tested if it is well formed		
Pre-Conditions	HTML file with all HTML Tags		
Post-Conditions	Only the pure text strings from the HTML file without the Tags, inclusive the position of the pure text strings in the HTML file		
Events			
Exceptions			

ID	17	Milestone	3.1
Name	Diff pure text	Component	CORE
Responsible	BP / PW	Accepted	
Description	Compare the pure text strings with diff algorithm		
Trigger	After the pure text strings of both docs are collected		
Pre-Conditions	The pure text strings from both docs are available		
Post-Conditions	All differences of the docs are shown in the second doc		
Events			
Exceptions	• No or only one doc is given		

Figure 38: Test Cases

ID	1
Description	Load a not well-formed HTML file. The file has a missing end tag somewhere in the middle of the file.
Pre-Conditions	Program up and running and no HTML file is loaded
Steps	1. Open Directory with corrupted file 2. Load corrupted file
Expected Result	File should be loaded and the parser should set the missing tag.
Actual Result	File loaded and the missing tag is set
Pass/Fail	Passed

ID	2
Description	Load a file with another typ.
Pre-Conditions	Program up and running
Steps	1. Open Directory with an TXT file in it 2. Load the TXT file
Expected Result	The parser should generate a HTML file out of the file.
Actual Result	The <HTML> and <BODY> tag are inserted in the file.
Pass/Fail	Passed

ID	3
Description	Load a HTML file with an incorrect end tag.
Pre-Conditions	Program up and running and no HTML file is loaded
Steps	1. Open Directory with corrupted file 2. Load the file with the incorrect end tag
Expected Result	File should be loaded and the parser should replace the incorrect end tag with the right one.
Actual Result	File loaded and the end tag is replaced.
Pass/Fail	Passed

ID	4
Description	Add an annotation automatically. This annotation has to enclose exactly one tag.
Pre-Conditions	1. Program up and running 2. A loaded source file with one annotation that enclose exactly one tag 3. A loaded destination file with the same text like the source file but without any annotation
Steps	1. Start the add annotation automatically process 2. After the process save the file
Expected Result	The destination file should have the annotation at the correct position and the annotation had to be in the correct form. The destination view should display the destination file with the new annotation and the checkbox of the inserted annotation in the annotation view should be checked.
Actual Result	The annotation is on the right position and the checkbox is checked.
Pass/Fail	Passed

ID	5
Description	Add an annotation automatically. This annotation has to enclose more than 3 tags.
Pre-Conditions	1. Program up and running 2. A loaded source file with one annotation that enclose

	more than 3 tags 3. A loaded destination file with the same text like the source file but without any annotation.
Steps	1. Start the add annotation automatically process 2. After the process save the file
Expected Result	The destination file should have all annotations at the correct position and all annotations had to be in the correct form. The destination view should display the destination file with the new annotations and the checkbox of the inserted annotations in the annotation view should be checked.
Actual Result	The annotation is on the right position and the checkbox is checked.
Pass/Fail	Passed

ID	6
Description	Add multiple annotations automatically.
Pre-Conditions	1. Program up and running 2. A loaded source file with annotations 3. A destination file with the same text like the source file but without the annotations
Steps	1. Start the add annotation automatically process 2. After the process save the file
Expected Result	The destination file should have all annotations at the correct position and all annotations have to be in the correct form. The destination view should display the destination file with the new annotations and the checkbox of the inserted annotations in the annotation view should be checked.
Actual Result	All annotations are on the correct position and the checkbox's are checked.
Pass/Fail	Passed

ID	7
Description	Add multiple annotations automatically but with some annotation that could not be placed in the destination file, because the enclosing text could not be found.
Pre-Conditions	1. Program up and running. 2. A loaded source file with annotations and one annotation that could not be placed. 3. A loaded destination file without annotations and the text from the source file so all annotations up to one could be placed.
Steps	1. Start the add annotation automatically process 2. After the process save the file
Expected Result	The destination file should have all founded annotations at the correct position and all founded annotations have to be in the correct form. The destination view should display the destination file with the new founded annotations and the checkbox of the inserted annotations in the annotation view should be checked.
Actual Result	All founded annotations are at the correct position. The position of the other annotations could not be found because of the character set problem.
Pass/Fail	Passed

ID	8
Description	Add an annotation manually. This annotation has to enclose exactly one tag.
Pre-Conditions	1. Program up and running. 2. A loaded source file with one annotation that enclose exactly one tag. 3. A loaded destination file with the same text like the source file but without any annotation.
Steps	1. Select a position in the destination file 2. Select a annotation in the annotation view
Expected Result	The destination file should have the annotation at the correct position and the annotation had to be in the correct form. The destination view should display the destination file with the new annotation and the checkbox of the inserted annotation in the annotation view should be checked.
Actual Result	The annotation is on the right position and the checkbox is checked.
Pass/Fail	Passed

ID	9
Description	Add an annotation manually. This annotation has to enclose more than 3 tags.
Pre-Conditions	1. Program up and running 2. A loaded source file with one annotation that enclose exactly one tag 3. A loaded destination file with the same text like the source file but without any annotation
Steps	1. Select a position in the destination file 2. Select a annotation in the annotation view
Expected Result	The destination file should have all annotations at the correct position and all annotations had to be in the correct form. The destination view should display the destination file with the new annotations and the checkbox of the inserted annotations in the annotation view should be checked.
Actual Result	The annotation is on the right position and the checkbox is checked.
Pass/Fail	Passed

ID	10
Description	Delete Annotation
Pre-Conditions	1. Source file with annotations is loaded 2. Destination file is loaded 3. One of the annotations from the source file is already add to the destination file
Steps	Select the add annotation in the annotation view and uncheck the checkbox.
Expected Result	The corresponding annotation should be deleted in the destination file. This should be shown in the destination view.
Actual Result	Annotation is deleted
Pass/Fail	Passed

ID	11
Description	No file loaded and the add automatically button is pressed

Pre-Conditions	No loaded source and destination file
Steps	Press the add automatically button
Expected Result	Information for the user that the source file or destination file is missing.
Actual Result	A message box with the information is shown.
Pass/Fail	Passed

ID	12
Description	No file loaded and the add manually button is pressed
Pre-Conditions	No loaded source and destination file
Steps	Press the add manually button
Expected Result	Information for the user that the source file or destination file is missing.
Actual Result	A message box with the information is shown.
Pass/Fail	Passed

ID	13
Description	Add annotation manually but no annotation is selected.
Pre-Conditions	1. Source file with annotations is loaded 2. Destination file is loaded
Steps	1. Select a position in the destination file 2. Push the add manually button
Expected Result	Information for the user that no annotation is selected.
Actual Result	A message box with the information is shown.
Pass/Fail	Passed

ID	14
Description	Add annotation manually but no place in the destination file is selected.
Pre-Conditions	1. Source file with annotations is loaded 2. Destination file is loaded
Steps	1. Select an annotation in the annotation view 2. Push the add manually button
Expected Result	Information for the user that no place for the annotation is selected.
Actual Result	A message box with the information is shown.
Pass/Fail	Passed

Figure 39: readme.txt

```
Compano 1.0 (build from 5.10.2011)
Textual Comparison of XML Documents with Annotations

Authors:
-----------
Bruno Pfeiffer 0717311
Philipp Waibel 0716754

What is Compano:
----------------------
Several research institutions provide researchers and interested persons with
     documents presenting the current state of research. These electronic documents
     are often annotated with additional information. When a new version of a text
     is released, one wants to automatically integrate the exisiting annotations and
      see the difference between the original and the new document.
Compano is a tool to automatically integrate annotations that are created with the
     tool DELT/A into a newer document. Furthermore with the help of Compano it is
     possible to insert the annotation manually as well.

Installation and Environments:
----------------------------------------
Compano is programmed using the RCP-framework for the Eclipse Environment:
Eclipse for RCP and RAP Developers
Version: Helios Service Release 2
Build id: 20110218-0911
Java ver. 1.6

The activator class is compano.Activator.

Configuration files:
-----------------------
The configuration file is the file conf.properties. This file contains the regular
     expressions for the cleaning of the HTML file. Each line is one regular
     expression for the cleaning process.

Default configuration:
<SPAN[a-zA-Z="0-9,:#;'\n\.\-\_ ]*>
</SPAN\n?>
<span[a-zA-Z="0-9,:#;'\n\.\-\_ ]*>
</span\n?>
<TD[a-zA-Z="0-9,:#;'\n\.\-\_ ]*/>
<TR>\n*</TR>
<TABLE>\n*</TABLE>
\n+
<a id="delta\:[0-9]+"*>\n*</a>

Usage:
--------
First you have to load the source file (file with the annotations) and the
     destination file (file where the annotations should be placed)
To load the source file, first you have to open the directory where the file is
     stored. This can be done via the menu (File -> Source Directory) or by clicking
      the corresponding icon at the toolbar. After that, the file can be opened in
     the source directory view. The file should then appear in the view and the
     annotations should be shown in the annotation view.
The destination view can be opened the same way.

When both files are opened, the following actions can be performed:
Add the annotations automatically:
Use the menu (action -> add automatically) or the 'arrow' button in the toolbar.
     This will add all possible annotations from the source file to the destination
     file.

Add the annotations manually:
1. Select an annotation in the annotation view.
2. Select a part of the text in the destination file. The annotation will be placed
     at this position.
3. Use the item in the menu (action -> add manually) or the '+' button in the
     toolbar.

Delete an annotation:
After adding an annotation to the destination file a checkbox in the annotation view
      is set. By unchecking this checkbox the annotation will be deleted from the
     destination file.

Limitations and known Bugs:
-------------------------------------
- Sometimes DELTA/A change the character set of the document so it is no longer
     comparable to the original file. For example is the char "bullet" after DELTA/A
      another character and so it is no longer comparable.
- Text selection for the manuall placement can only be done in the text view and not
      in the browser view.
- The algorithm use only the first appearance of the annotation text in the
     destination document, so if the annotation text is not unique enough, for
     example "and", it may find the wrong place for the annotation.
```

List of Figures

References

[1] Asprise java pdf library. `http://asprise.com/product/javapdf/`, 08 2011.

[2] Character encoding. `http://de.wikipedia.org/wiki/Zeichenkodierung`, 08 2011.

[3] Dom xml parser. `http://www.w3.org/DOM/`, 10 2011.

[4] itext. `http://itextpdf.com/`, 08 2011.

[5] Jtidy. `http://jtidy.sourceforge.net/`, 08 2011.

[6] Neko html. `http://nekohtml.sourceforge.net/index.html`, 08 2011.

[7] Pair programming. `http://en.wikipedia.org/wiki/Pair_programming`, 08 2011.

[8] Sax xml parser. `http://www.saxproject.org`, 10 2011.

[9] Stax xml parser. `http://stax.codehaus.org/Home`, 10 2011.

[10] Test-driven development. `http://en.wikipedia.org/wiki/Test-driven_development`, 08 2011.

[11] Unix diff utility. `http://en.wikipedia.org/wiki/Diff`, 08 2011.

[12] Xml w3c recommendation 26 november 2008. `http://www.w3.org/TR/xml/#sec-well-formed`, 09 2011.

[13] Michael Gilleland. Levenshtein distance, in three flavors. `http://www.merriampark.com/ld.htm`, 08 2011.

[14] Eugene W. Myers. An o(nd) difference algorithm and its variations. Technical report, Department of Computer Science, University of Arizona, Year Not Stated.

[15] Peter Votruba, Silvia Miksch, and Robert Kosara. Facilitating knowledge maintenance of clinical guidelines and protocols. pages 57–61. AMIA, IOS Press, 2004.