Document Engineering of Complex Software Specifications

Mehrdad Nojoumian

Thesis submitted to the
Faculty of Graduate and Postdoctoral Studies
In partial fulfillment of the requirements
For the **Master of Science** degree in **Computer Science**

Ottawa-Carleton Institute for Computer Science
School of Information Technology and Engineering
University of Ottawa

© Mehrdad Nojoumian, Ottawa, Canada, 2007

Abstract

The research presented in this thesis aims at document engineering of complex specifications, of which the UML Superstructure Specification (version 2.1) is our initial target. *Document engineering* deals with principles, tools and processes that improve our ability to create, manage, and maintain documents [40].

Our motivation is that such specifications are dense and intricate to use, and tend to have complicated structures with lots of repetitive, or 'boilerplate' material. End users cannot use them efficiently because of the general complexity of the document.

Our objective and main contribution in this thesis is therefore to create an approach that allowed us to re-engineer PDF-based documents, and to illustrate how to make more usable versions of electronic documents such as specifications, conference proceedings, technical books, etc so that end-users to have a better experience with them.

The first step was to extract the logical structure of the document. Our initial assumption was that, many key concepts of a document are expressed in this structure, which includes the headings of the chapters, sections, subsections, etc. We demonstrated this by analyzing some data, and created a special-purpose parser to generate a well-formed XML document with various types of tags.

In the next phase, we created a user interface for end users by generating a multi-layer HTML version of the document to facilitate document browsing, navigating, and concept exploration.

Although our targeted document was the UML Superstructure Specification, we chose a general approach for most phases of our work including format conversions, logical structure extraction, text extraction, hypertext generation, etc. Therefore, by minor adjustments we can process other complex documents to gain our mentioned goals. We also established the major infrastructure for a new document engineering framework.

To my wife, **Sareh**, my mother, **Fatemeh** and my father, **Mehdi**, with deepest love

Acknowledgements

My most sincere appreciation goes to my supervisor Professor Timothy C. Lethbridge for all the knowledge and experience I gained from him. His valuable support, patience and kindness were my motivations for research progress. Also, many thanks given to Professor Bran Selic for his positive feedback.

I would also like to express my gratefulness to my family; my dear wife, mother and father who were always there for me at worst times, my sister and brother, Mahboubeh and Peyman, whom their kindness has always been cheering.

This work was supported by the **IBM Ottawa Software Lab**. I would like to express my thanks to them as well.

Table of Contents

Abstract ii
Acknowledgements iv
Figures vii
Tables viii
1 Introduction 1
1.1 Definitions and technologies 2
1.2 Motivation and research questions 4
1.3 Contributions 5
2 Literature Review 7
2.1 Document structure analysis 7
2.2 Existing document analysis systems 8
2.3 Leveraging tables of contents 10
2.4 Knowledge extraction 11
2.5 Ongoing research on XPath 12
3 Document Transformation 14
3.1 Conversions 16
3.2 Criteria 17
3.3 First stage of evaluation 18
3.4 Second stage of evaluation 19
4 Logical Structure Extraction 22
4.1 Data analyses to provide evidence for our initial assumption 22
4.2 First refinement approach: Stack-based parser 30
4.3 Second implementation approach: Bookmarks 31
5 Text Extraction to Create Initial Hypertext Pages 37
5.1 Checking well-formedness 38
5.2 Generating a valid schema 41
5.3 Producing multiple outputs 43

5.4 Connecting generated outputs sequentially 48
5.4.1 Connecting pages using XPath expressions 48
5.4.2 Connecting pages using a programming approach 50
5.5 Forming major document elements 53
5.5.1 Anchors in long pages 53
5.5.2 Figures 55
5.5.3 Tables 57
5.5.4 Lists 59
6 Concept Extraction and Cross Referencing 61
6.1 Concepts extraction 61
6.1.1 UML class hierarchy extraction 63
6.1.2 UML package hierarchy extraction 65
6.2 Cross referencing 67
7 Experimental Result and Architecture of the Framework 70
7.1 Re-engineering of various OMG specifications 70
7.2 Assessment of generated HTML Interfaces 72
7.3 Initial architecture of the proposed framework 74
8 Conclusions and Future Work 76
References 79
Appendix A: List of Acronyms 84
Appendix B: Logical Structure Extractor 85
Appendix C: First Java Parser 88
Appendix D: XSLT Codes 91

Figures

Figure 1. Bookmarks of the UML superstructure specification 15
Figure 2. A sample figure in XML format 20
Figure 3. A sample table in XML format 20
Figure 4. A sample nested list in XML format 20
Figure 5. A sample figure in HTML format 21
Figure 6. A sample table in HTML format 21
Figure 7. A sample nested list in HTML format 21
Figure 8. The 50 most frequent words in the document headings 28
Figure 9. Frequency of the 50 heading words as found in the words extracted from the entire document 28
Figure 10. The 24 most frequent words in the document index 28
Figure 11. First data collection (smaller surface) and their frequency in the document as a whole (bigger surface) 29
Figure 12. First data collection (bigger surface) and their frequency in the document index (smaller surface) 29
Figure 13. State machine for sample headings 32
Figure 14. Logical structure extracted in XML format 35
Figure 15. Logical structure model in the protégé 3.2.1 36
Figure 16. A nested list spread over two pages 5-6 (UML Spec. v2.1) 39
Figure 17. A complex table spread over two pages 163-164 (UML Spec. v2.1) 39
Figure 18. A list with two columns spread over two pages 205-206 (UML Spec. v2.1) 40
Figure 19. Schema component representations: “Book” & “Chapter” 41
Figure 20. Schema component representation: “Figure” 42
Figure 21. Schema component representation: “Table” 42
Figure 22. Schema component representation: “List” 42
Figure 23. Table of contents showing distinct types of navigation paths 48
Figure 24. Top of a long page in our first design, showing links to internal anchors 53
Figure 25. Heading tags structure in the XML document 54
Figure 26. Figure tag structure in the XML document 55
Figure 27. Screenshot of the Altova StyleVision for importing figures 56
Figure 28. Table tag structure in the XML document 57
Figure 29. Screenshot of the Altova StyleVision for importing tables 58
Figure 30. List tag structure in the XML document 59
Figure 31. Screenshot of the Altova StyleVision for importing lists 60
Figure 32. Headings of the UML specification (v2.1), containing UML concepts 62
Figure 33. Part of tagging structures in the XML document 63
Figure 34. Headings are among the most frequent words in the entire document 71
Figure 35. Initial architecture of the proposed document engineering framework 75

Tables

Table 1. Different conversions of Chapter 7 of the UML 2.1 specification 16
Table 2. Statistical summary related to the heading and index words 23
Table 3. The 50 most frequent words in the document headings 25
Table 4. Frequency of the 50 heading words from Table 3 as found in the words extracted from the entire document 26
Table 5. The 24 most frequent words in the document index 27
Table 6. Different kinds of headings 31
Table 7. Sample XML tags in the UML superstructure specification 36
Table 8. Some document navigation paths related to the figure 18 49
Table 9. Re-engineering of ten OMG specifications 71

1 Introduction

Published electronic documents, such as specifications, are often rich in knowledge, but that knowledge is often complex and only partially structured. This makes it difficult for human beings to make maximum use of the documents.

The objective of this thesis is to develop an approach by which a typical published specification can be made more usable to the end user. We achieve this by reverse engineering the document, and then generating a new hypertext document that makes the knowledge more explicit, and facilitates searching, browsing, comparison and other operations needed by end users.

As a case study, we applied our approach to version 2.1 of the UML Superstructure Specification, as published in PDF format. However, we ensured that all aspects of our work are as general as possible, so the same approach can be applied to other specification documents. We chose the UML specification because it is important to software engineering, and since members of our research group have studied it in depth and have experienced frustrations with it.

Our overall approach is an example of *document engineering*, and is divided into two distinct phases:

The first phase of our approach is to extract the document's *logical structure* and *core knowledge*, representing the result in XML (Extensible Markup Language). This result consists only of content information and excludes irrelevant details of the original document's presentation. Capturing the content in XML allows for easy exploration and

editing of data by XML editors and other tools, and allows generation of the new presentation to be a separate responsibility (achieved in Phase 2). The first task in phase 1 is to use a Commercial Off-The-Shelf (COTS) tool to convert the input PDF file into a format we can more readily work with. We conducted an experiment to see which one would generate the best XML the next task was to parse the output of the COTS tool to clean up the XML and tag knowledge that was only simple semi-structured plain text in the original document.

The second phase of our approach is to produce a usable new document presentation that includes facilities for navigating the important relationships in the data. In our case study these include an ability to navigate metamodel class diagram and package diagram relationships. We achieved this by using XPath and XQuery technologies, discussed below.

We anticipate that if the developers of specifications published their documents in the format we developed, it would greatly assist end users of the specifications. In the next subsection, we present some major definitions and technologies that are important to our research.

1.1 Definitions and technologies

In general, document processing can be divided into two phases: document analysis and document understanding. A document has several layers of structure. Extraction of the *geometric structure* (including entitles such as pages, blocks, lines, and words) is referred to *document analysis.* Mapping this structure into a *logical structure* (including titles, headings, abstract, sections, subsections, footnotes, tables, lists, explicit cross-references, etc.) is referred to *document understanding* [1]. Extracting concepts embedded in the document structure, such as realizing that the names of some sections represent concept names, and the cross-references represent relationships among the concepts, is a form of *knowledge acquisition*.

From the structural point of view, a document can be unstructured, semi-structured or structured. A plain text document with nothing marked other than the normal

conventions of natural language (e.g. a period at the end of a sentence) would be considered *unstructured*. A document with tags dividing it into paragraphs, headings, and sections would be considered *semi-structured*; most web pages are of this type. A document in which all the elements are marked with meta-tags, typically using XML, would be considered *structured*. A structured document can be represented as a tree, with leaf nodes representing very small snippets of textual content. In practice, software specification documents fall somewhere on the continuum between semi-structured and structured. However, the markup is usually noisy.

When more structure is imposed on a document, the resulting richer representation allows computers to make use of the knowledge directly. Unstructured documents or sections have to rely on natural language understanding technology before the knowledge can be used. One of the major advantages of electronic documents is that we can partition them into a hierarchy of *physical components*, such as pages, columns, paragraphs, lines, words, tables, figures, etc or a hierarchy of *logical components*, such as titles, authors, affiliations, sections, subsection, etc. This structural information can be very useful in information extraction and knowledge acquisition.

In the next few paragraphs, we describe some of the W3C (World Wide Web Consortium) definitions and technologies that we use throughout the thesis.

XML is a general-purpose markup language which supports a wide variety of applications and its major purpose is to facilitate the sharing of data across different information systems, especially systems connected via the internet. In XML, tags are not predefined and everyone has to define his/her own tags. It is a human and machine-readable format and can present the most general data structures [35].

An *XML Schema* describes the structure of an XML document. A document written in XML Schema language is also referred to as an *XSD* (XML Schema Definition). XML Schema is a new and more powerful schema language that is the successor of DTD (Document Type Definition) [36].

Since XML is a content-driven language, it does not carry any information about how to display the data; therefore as a solution, *XSL* (Extensible Stylesheet Language) can be used to manipulate the XML data, typically extracting information from it or converting it into HTML (Hyper Text Markup Language) or other formats such as PDF

(Portable Document Format), RTF (Rich Text Format), etc [37].

For access to the divisions of an XML document we can apply *XPath* (XML Path language) technology which makes it possible to extract every part of an XML file. XPath expressions can refer to all or part of the text, data and values in XML elements, attributes, etc [38]. *XQuery* (XML Query Language) is a language which has some programming features and is designed to query collections of XML data. It is semantically similar to SQL (Structured Query Language) [39]. In the next section, we present our major motivation and research questions.

1.2 Motivation and research questions

The motivation for our work is that complex documents such as software specifications are not as usable as we believe they should be. The following are some of our observations:

They are large, dense and intricate to use. They are too large for all but the most dedicated to read from end-to end, so most users will skim them or look things up when needed. However, readers will often have to jump backwards and forwards many times to follow cross references. For example, in the UML specifications, there are definitions of metaclasses. Each of these has inherited properties that come from metaclasses that may be in other ‘packages’. It is hard for people to understand the context of one of the metaclasses.

They tend to have lots of repetitive, or ‘boilerplate’ material: If a user is interested in one type of information, then he or she nevertheless has to wade through lots of other information. Many headings are repeated over and over again obscuring subtle details.

Numerous concepts tend to be connected only implicitly: It is not easy to follow references to the place where the reference points.

The documents are published using a format that mimics legacy paper documents: Although PDF is an excellent way of rendering a paper document faithfully in electronic form, and has some built-in navigation capability; the use of PDF does not take advantage of modern computational capabilities. In particular, it is now best practice to

separate presentation from content, and PDF publication formats do not allow for this.

The above issues led are to formulate the following research questions:

Research question 1: How can we re-engineer a PDF-based specification in as general and straight-forward way as possible?

Research question 2: What facilities are needed for end-users to have a better experience with a specification?

1.3 Contributions

The following is a list and brief description of the key contributions of our work for document engineering. The first three contributions are related to the value in the process of doing document engineering and the last one presents the value in the final result.

Contribution 1: A technique for capturing document structure and knowledge effectively from a PDF file. We experimented with conversions using different COTS tools to select the best file transformation, extracted document's logical structure in XML format, and proved our key reasons for the logical structure extraction. We further processed this using a parser written in Java. We encountered problems such as mis-tagging related to the conversion phase and lack of well-formed characteristic of our XML file. We overcame these problems and generate a well-formed XML document with various types of meaningful tags.

Contribution 2: Various techniques for text extraction. We experimented with numerous methods to create hypertext pages and produce the initial HTML user interface for end users. We also applied the latest W3C technologies for concept extraction and cross referencing to improve the usability of the user interface.

Contribution 3: A general approach for document engineering. Although our targeted document was the UML Superstructure Specification, we chose a generic approach for most phases of our work including format conversions, logical structure extraction, text extraction, hypertext generation, etc. Therefore, by minor adjustments we can process other complex documents. We also established the major infrastructure for a document engineering framework.

Contribution 4: Significant values in the final result. After showing how to create a more useful format of a document, we demonstrate the usability of our final outcome such as better navigating and scrolling structure, efficient learning, faster downloading, easier printing, monitoring, coloring, and cross referencing among various documents.

The rest of the thesis is organized as follows. Chapter 2 reviews the existing literature on document analysis and document engineering. Chapter 3 presents the properties of our targeted document and focuses on various document transformations. Chapter 4 demonstrates two experimental results for the document's logical structure extraction and our reasons for such an extraction. Chapter 5 focuses on text extraction to create multi-layer hypertext pages. Chapter 6 improves the created user interface by concept extraction and cross referencing. Chapter 7 provides some experimental results and the architecture of the proposed document engineering framework. Chapter 8 gives our final conclusions and suggested future work. Finally, in the last part Appendices have been included. For full access to all implementations you can visit the following homepage: **http://www.site.uottawa.ca/~tcl/gradtheses/mnojoumian/**

2 Literature Review

In this section, we review document structure analysis, relevant existing systems, leveraging Table of Contents (ToC), knowledge extraction, and ongoing researches with respect to XPath technology in order to form a clear vision of these areas.

2.1 Document structure analysis

Klink et al. [1] present a hybrid and comprehensive approach to document structure analysis. Their approach is hybrid in the sense that it makes use of layout (geometrical) as well as textual features (logical) of a given document.

In [2], Mao et al propose numerous algorithms to analyze the physical layout and logical structure of document images (images of paper documents) in many different domains. The authors provide a detailed survey of diverse algorithms in the following three aspects: physical layout representation, logical structure representation, and performance evaluation.

Summers [3] explains an approach for finding a logical hierarchy in a generic text document based on layout information. The logical structure detection has two problems, segmentation and classification. The first one separates the text into logical pieces and its algorithm relies totally on layout-based cues, while the second one labels the pieces with structure types and its algorithm uses word-based information.

Tsujimoto and Asada [4] represent the document physical layout and logical

structure as trees. They characterize document understanding as transformation of a physical tree into a logical one. Blocks in the physical tree are classified into head and body and in the logical tree are categorized into title, abstract, sub-title, paragraph, header, footer, page number, and caption. They tested their algorithm on 106 pages from various sources and reported “94 out of 106” logical structure recognition accuracy.

Lee et al. [5] provide a syntactic method for sophisticated logical structure analysis which transforms multiple page document images with hierarchical structure into an electronic document based on XML format. Their proposed parsing method takes text regions with hierarchical structure as input.

Liang [6] presents a unified document structure extraction algorithm that is probability-based for scanned document image pages. He also developed a system that detects and recognizes special symbols (Greek letters, mathematical symbols, etc.) on technical document pages that are not handled by the current systems.

Conway [7] uses page grammars and page parsing techniques to recognize document logical structure from physical layout. The physical layout is described by a set of grammar rules. Each of these rules is a string of elements specified by a neighbor relationship such as above, left-of, over, left-side, and close-to. For describing the logical structure a context-free string grammars are used.

In [8], Aiello et al. provide a framework for analyzing color documents of complex layout. In this framework, no assumption is made on the layout. The proposed structure combines two major sources of information: textual and spatial. It also uses shallow natural language processing tools, such as partial parsers, to analyze the text.

In the next subsection, we demonstrate many remarkable document analysis and understanding systems.

2.2 Existing document analysis systems

In [9] a document analysis system named WISDOM++ (Wise System for Document Management) is presented. This processing system operates in five steps: document analysis, document classification, document understanding, text recognition with optical

character recognition, and text transformation into HTML/XML format.

Ishitani [10] proposes a new system for document transformation using OCR (Optical Character Recognition) to produce various XML files from printed documents. In the first step, document elements such as title, authors, abstract, headings, paragraphs, lists, captions, tables and figures are extracted from document images and in the second step, the structure of document elements is extracted and described by a DOM (Document Object Model) tree, which is an ordered tree where each node is either an element or a text node.

In [11] Ishitani also presents a document logical structure analysis system based on emergent computation which is a key concept of artificial life. The system includes five basic modules: typography analysis, object recognition, object segmentation, object grouping, and object modification. The document image is first segmented into text lines. After that, they are classified into various types. The classified text lines are then grouped and classified into logical components.

Dengel and Dubiel [12] describe a system named DAVOS which is capable of both learning and extracting document logical structure. This system can learn document structure concepts. The structural concepts are represented by relation patterns and a geometric tree is used to represent the concept language.

Niyogi and Srihari [13] present a system called DeLoS for document logical structure derivation. They develop a computational model according to a rule-based control structure. In this system, knowledge about the physical and logical structures of various types of documents is encoded into a knowledge base. The system has three types of rules: knowledge rules, control rules, and strategy rules. The control rules manage the application of knowledge rules and the strategy rules define the usage of control rules. First of all, the document is segmented then segmented blocks are classified. Finally, the classified blocks are input into the DeLoS system and a logical tree structure is derived.

Kreich et al. [14] provide an environment called SODA (System for Office Document Analysis) for document analysis. They use a bottom-up approach to group connected components into text blocks, then find lines within each text block and words within each line. The physical layout and logical structure knowledge are also stored in a

knowledge base. Finally, document objects are matched to the layout and logical information in the knowledge base. A match is considered successful if its confidence measure is greater than a particular threshold.

In [15], Nakagawa et al. proposes a mathematical knowledge browser which helps people to read mathematical documents. Using this browser, printed mathematical documents can be scanned and recognized. Then the meta-information (e.g. title, author) and the logical structure (e.g. section, theorem) of the documents are extracted.

In the next subsection, we explain the analysis of table of contents for document understanding and logical structure extraction.

2.3 Leveraging tables of contents

Déjean and Meunier [16] describe a technique for structuring documents according to the information in their tables of contents. In fact, the detection of the ToC as well as the determination of the parts it refers to in the document body rely on a series of properties that characterize any ToC.

He et al. [17] propose a new technique for extracting the logical structure of documents by combining spatial and semantic information of the table of contents. They exploited page numbers and numbering scheme to compute the logical structure of a book. Their method is not a general approach because of the observed diversity of page or section numbering and ToC layout.

Lin et al. [18] propose a method for analyzing the logical structure of books based on their tables of contents by layout modeling and headline matching. In general, the contents page holds accurate logical structure descriptions of the whole book. In this approach, text lines are first extracted from the contents page, and OCR is then performed for each text line. The structures of the page number, head, foot, headline, chart and main text of the text page are analyzed and matched with information obtained from the contents page.

Lin and Xiong [19] introduce a new approach to explore and analyze ToC based on content association. Their method leverages the text information in the whole document

and can be applied to a wide variety of documents without the need for analyzing the models of individual documents. NLP (Natural Language Processing) and layout analysis are integrated to improve the ToC tagging.

Satoh et al. [20] propose a system where ToC pages of academic journals were converted into bibliographic database by image segmentation and understanding techniques. They use training data to learn decision trees for various kinds of journals.

Bourgeois et al. [21] describe a statistical model for document understanding which uses both text attributes and document layout. In this model, probabilistic relaxation, which is a general method to classify objects and to repetitively adjust the classification, is used as a recognition method for understanding the table of contents and discovering the logical structure.

In the next subsection, we provide a quick literature review on knowledge extraction and relevant tools and approaches.

2.4 Knowledge extraction

Crowder and Sim's [22] goal is to capture relevant knowledge from legacy documents. Firstly, they converted the legacy documents to XML documents where the output is semantically tagged. Once in an XML form, the data can be easily transformed. They describe the development of tools to automate the process of converting legacy documents to XML documents. They also show that XML versions of legacy documents provide better results than a basic text search over the identical documents.

In the past decade, most work on extraction has been focused primarily on factual information. Only in recent years have we witnessed a growing interest in subjective texts such as evaluative ones. The general problem that Carenini et al consider in [23] is how to effectively extract useful information from large corpora of evaluative text.

Cyre [24] developed a tool for knowledge extraction. The process is to begin with a basic ontology and extract Conceptual Graphs from text in the domain of interest. During this process, the ontology is augmented by the knowledge engineer. In this approach, the user scans the text and creates conceptual graphs from sentences or other

expressions, and joins the individual graphs into a knowledge-base.

In [25], Cohen and Jensen assume that structured documents are represented with the document object model. Their approach to information extraction is based on a DOM tree. An element node has an ordered list of zero or more child nodes, and contains a tag (such as “table”, “h1”, or “li”) and attributes (such as “href” or “src”). A text node is normally defined to contain a single text string.

Sakamoto et al. [26] show their recent results in knowledge discovery from semi-structured texts which contain heterogeneous structures represented by labeled trees. The aim of their study is to extract useful information from documents on the Web.

The approach presented by Vargas-Vera et al [27] describes a semantic annotation tool for extraction of knowledge structures from web pages through the use of simple user-defined knowledge extraction patterns.

IKRAFT (Interactive Knowledge Representation and Acquisition From Text) [28] is an interactive tool to elicit from users the rationale for choices and decisions as they analyze information used in building a knowledge base. Starting from raw information sources, most of them originating on the Web, users are able to specify connections between selected portions of those sources.

In [29], Henzinger and Lawrence discuss methods for extracting knowledge from the web by randomly sampling and analyzing hosts and pages, and by analyzing the link structure of the web. By this approach, much interesting information can be extracted, such as the distribution of interest in different areas, the nature of competition in different categories of sites, and the degree of communications among countries.

In the last part of this section, we present a quick literature review with respect to the latest researches on XPath technology.

2.5 Ongoing research on XPath

In the context of query systems, change detection in XML documents, filtering them, and XPath decision problems are the most significant issues. In [30], a customizable change detection approach for XML documents is presented. This method performs

change detection and XPath based filtering of XML document; filtering means the extraction of those elements that we are interested in.

Qeli and Freisleben [31] also present the design and implementation of a system for filtering XML documents. Their method works based on XPath expressions and Aspect Oriented Programming (AOP) which is a dynamic programming approach for embedding of simple paths into XPath expressions.

Geneves and Layaida [32] present a sound and complete decision procedure for XPath decision problems such as equivalence (whether two queries return the same result), overlap (whether the intersection of two expressions is non-empty), containment (whether the result of a query is included in the result of another one), and coverage (whether an expression contained in the union of several expressions). They propose a unifying logic for XML, illustrate how to translate major XML concepts such as XPath into this logic, and show how XPath decision problems can be solved.

In [33], a logic-based structure for the static analysis of XPath is proposed. In particular, they propose an appropriate logic for effectively solving XPath decision problems. They also demonstrate a translation of a large XPath fragments into the mentioned logic.

What is missing now is a clear characterization of the expressive power of XPath. In fact, Core XPath cannot express queries with conditional paths such as: “does a child step, while test is true at the resulting node”. Marx [34] adds conditional axis relations to Core XPath and illustrates that the resulting language, called conditional XPath, is as expressive as FOL (First Order Logic).

As we have seen, there is remarkable ongoing research related to XPath technology. We will take advantage of XPath expressions and the XQuery language to extract hidden concepts from UML Superstructure Specification.

In the next chapter, we present the experimental results with respect to the first phase of our research (document transformation).

3 Document Transformation

The document we targeted, the UML superstructure specification (version 2.1), is a large specification in PDF format with 771 pages. It has almost 2200 headings with a lot of nested lists, hyperlinks, figures, tables, etc.

What was our motivation for analyzing PDF documents? First of all, people do not have access to the original word-processor formats of the documents much of the time. When documents are published to the web, an explicit choice is usually made to render the result as PDF or HTML to guarantee that everyone can read it (without having to have Microsoft Word, FrameMaker, etc.) and so that people can not so easily create a new version of the document that appears to be an official version. Moreover, PDF format has some useful features that make it semi-structured; for example it often contains “bookmarks” created from headings to enable a user to navigate a document. However, a computer can also easily use this information to extract the structure.

Figure 1, shows sample bookmarks of the UML specification. The general structure of this document consists of parts, chapters, sections, subsections and keyword-headed sub-subsections. The names of some of these correspond to concepts such as ‘Abstraction’ and ‘Associations’.

One of our major goals is to extract the document’s logical structure, as discussed in Chapter 1. As we mentioned, many key concepts of the targeted specification are expressed in this structure. By extracting the structure and representing it as XML, we can form a good infrastructure for our subsequent objectives.

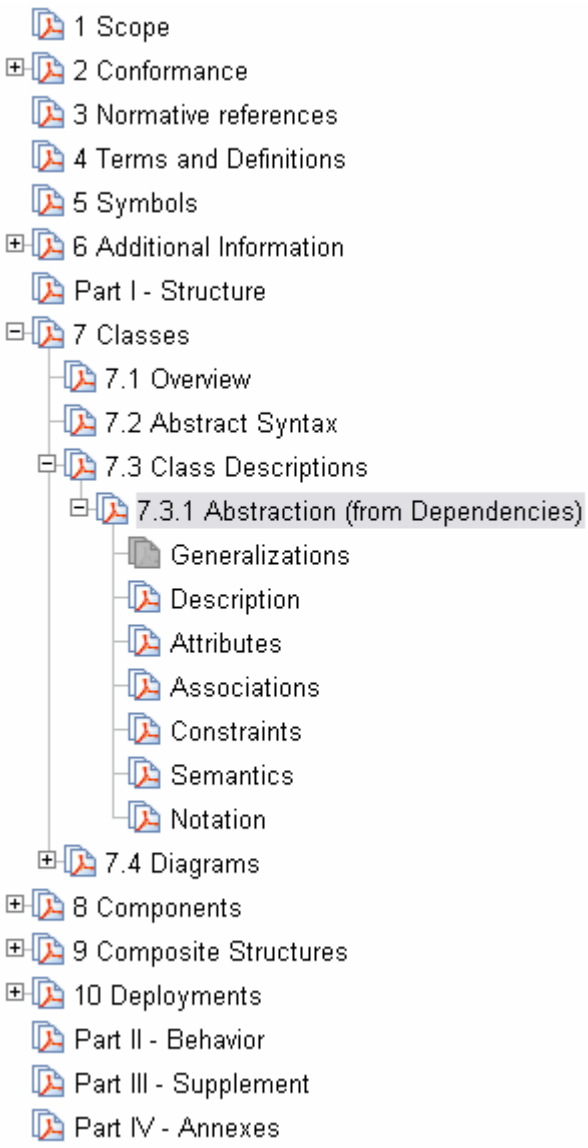


Figure 1. Bookmarks of the UML superstructure specification

We approached the structure extraction problem as a two-stage problem. In this chapter we describe the first step: Transforming the raw input into a format more amenable to analysis. The second step, extracting and refining the structure, is the topic of the next chapter.

To extract the logical structure of the document, we experimented with transformations using various existing tools to see to what extent each could facilitate the extraction process. Since our targeted document is a large specification, we started

with much smaller documents. Firstly we performed various conversions using a simplified sample file which had similar properties to the target document. Then we analyzed a single chapter, Chapter 7, before moving on to process the first 219 pages, which covered the first 9 chapters.

3.1 Conversions

Table 1 shows the tools we used for conversion and the formats we experimented with. We just used some popular input formats (DOC, PDF, etc) and applied different tools in this respect such as "Adobe Acrobat Professional 7.8", "Microsoft Word 2003", "Stylus Studio® 2007 XML Enterprise Suite", and "ABBYY PDF Transformer 1.0". In this table, we excluded transformations with similar results.

Table 1. Different conversions of Chapter 7 of the UML 2.1 specification

Input Format (Size)	Tools for Conversions	Output Format (Size)
DOC (34.5)	Microsoft Office Word 2003	**TXT** (2.81)
DOC (34.5)	Microsoft Office Word 2003	**RTF** (55)
DOC (34.5)	Microsoft Office Word 2003	**HTML** (40.7)
DOC (34.5)	Microsoft Office Word 2003	**XML** (55)
DOC (34.5)	Adobe Acrobat Professional 7.8	**PDF** (19) with Bookmarks
DOC (34.5)	Adobe Acrobat Professional 7.8	**PDF** (15.9) without Bookmarks
PDF (19) with Bookmarks	Adobe Acrobat Professional 7.8	**HTML** (6.38)
PDF (15.9) without Bookmarks	Adobe Acrobat Professional 7.8	**HTML** (5.15)
PDF (19) with Bookmarks	Adobe Acrobat Professional 7.8	**XML** (9.92)
PDF (15.9) without Bookmarks	Adobe Acrobat Professional 7.8	**XML** (8.30)
PDF (19) with Bookmarks	ABBYY PDF Transformer 1.0	**HTML** (19.2)
PDF (19) with Bookmarks	ABBYY PDF Transformer 1.0	**TXT** (2.82)

In the next section, we define five major criteria for choosing the best transformation; subsequently we evaluate these conversions according to these criteria.

3.2 Criteria

Since we want to extract the document's logical structure and convert it to XML, we are most interested in an output format listed in Table 1 which can most facilitate this. To select the best conversion, we defined a set of criteria based on the experiences we gained during our experiments. These criteria are as follows:

(a) Generality: A format should enable the design of a general extraction algorithm for processing other electronic documents.

(b) Low volume: We should avoid a format which contains of a lot of extra unneeded material that is not related to the document content. This includes information related to the presentation format, for instance, the position of elements such as words, lists and paragraphs.

(c) Clean and understandable: Even if a format results in small files, it still might not be adequate; it should also be clean and understandable. For instance, formats which cleanly mark constructs such as paragraphs with a single marker and that use carriage returns judiciously are easier to work with than formats that don't do this. For instance, some formats marked constructs with multiple markers and were not even consistent about this.

(d) Similarity to XML: We prefer a format which has a similar structure to XML, such as XML itself or HTML, because we want our final output of this step to be in XML format.

(e) Having good Clues: A format should use markers which provide accurate and good clues for processing and finding the logical structure, such as meaningful keywords with respect to the headings: "LinkTarget", "DIV", "Sect", "Part", etc.

Sometimes, formats that contain a lot of extra data such as font, size, style and position are more useful, while in other cases documents that are mostly text without any much extra detail would be more useful. For example the extra data would be useful for algorithms which detect headings of a document based on this information, whereas style and font tags are of little use to our algorithm in Section 4.3. Hence, we would like

to compromise among different kinds of formats to satisfy our mentioned criteria. In the next section, we evaluate the presented transformations to define the best candidate.

3.3 First stage of evaluation

To narrow down the list of possible transformations to use, we evaluated every transformation in Table 1 according to how they satisfy the above criteria. We performed all the presented conversions on the UML superstructure specification. Our observations are as follows:

DOC and RTF formats are messy, for example, they code figures among the contents of the document while some formats such as HTML or XML put all the figures in a separate folder in an image format. In addition, they store information related to the font, size, style, etc of each heading, paragraph, sentence and even words beside them. This information is not useful for us because they vary from document to document, contradicting the generality property and increasing the potential for noise during processing. In addition, if we extract HTML or XML formats from DOC/RTF, the results also tend to have the same unneeded properties.

TXT format is very simple but does not give us any clues for processing and you may not even find the beginning of the chapters, headings, tables, etc. Therefore, it does not have a suitable structure for analysis.

PDF is complex itself, but after a conversion into HTML or XML by Adobe Acrobat Professional 7.8, the result is very nice, especially in the case of PDF files which have bookmarks. They are clean, low sized, with tagging structure and useful clues for processing. They can even satisfy the generality property.

Therefore, our finalist candidates are HTML and XML formats extracted by Adobe Acrobat professional 7.8 from the PDF file with bookmarks. In the next section, we compare these two options.

3.4 Second stage of evaluation

To further narrow our choice of transformation, we analyzed the following sample parts of our target document using the two finalist candidates. These cover an array of possible structures that appear repeatedly in the UML superstructure specification:

1) Sample paragraphs
2) Sample figures (e.g. figure 7.26)
3) Sample tables (e.g. table 2.1)
4) Complex tables which have figures and hyperlinks in their cells (e.g. table 12.1)
5) Complex nested lists which have complicated hierarchy structures (e.g. part 2.3)

To conclude, we found out the XML format is the best candidate for processing. Our many assessments revealed that this format is more understandable and simple for analysis. Moreover, in the XML style, each tag is in a line, so we can analyze and parse the document line by line which is easier in compare to the HTML format in which we have to explore the document character by character. In the next page, some of these parts from two mentioned formats are presented.

In the next chapter, our experimental outcomes related to the logical structure extraction are presented.

```
- <Figure>
    <ImageData src="images/UML_img_67.jpg" />
  - <Caption>
      <P>Figure 7.26 - Composite aggregation...</P>
    </Caption>
  </Figure>
```

Figure 2. A sample figure in XML format

```
- <Table>
  - <Caption>
      <P>Table 2.1 Example compliance statement</P>
    </Caption>
  - <TR>
      <TH>Compliance Summary</TH>
    </TR>
  - <TR>
      <TD>Compliance level</TD>
    </TR>
  </Table>
```

Figure 3. A sample table in XML format

```
- <L>
  - <LI>
      <LI_Label>*</LI_Label>
      <LI_Title>Abstract syntax compliance:</LI_Title>
    </LI>
  - <L>
    - <LI>
        <LI_Label>**</LI_Label>
        <LI_Title>Compliance with the metaclasses...</LI_Title>
      </LI>
    - <LI>
        <LI_Label>**</LI_Label>
        <LI_Title>The ability to output models...</LI_Title>
      </LI>
    </L>
  </L>
```

Figure 4. A sample nested list in XML format

```
<IMG align="" width="387" height="173"
 src="images/UML-pdf Acrobat-htm3.2_img_67.jpg" >
 *** Caption Missing ***
```

Figure 5. A sample figure in HTML format

```
<TABLE
 align="center" border=0 cellspacing=0 cellpadding=2
><CAPTION
><P
><FONT size="+1">Table 2.1 Example compliance statement </P
></CAPTION
><TR
><TH
 colspan=2 align="left" width="239"  valign="middle" height="25"
><FONT size="+1">Compliance Summary </TH
></TR
><TR
><TD
 align="left" width="166"  valign="top" height="38"
>Compliance level </TD
></TR
></TABLE
>
```

Figure 6. A sample table in HTML format

```
<UL
><LI
>Abstract syntax compliance: </P
></LI
><UL
 type="disc"
><LI
>compliance with the metaclasses... </P
></LI
><LI
>the ability to output models... </P
></LI
></UL
></UL
>
```

Figure 7. A sample nested list in HTML format

4 Logical Structure Extraction

After following the step described in previous chapter, we have our initial XML document. However, aspects of the document structure (headings) still need to be extracted. First of all, we would like to give some evidence for our reasons for the logical structure extraction by some data analyses. Then, we discuss two implementation approaches to finalizing our extraction of structure and evaluate our methods and the reasons for failure in the first technique. Finally, we present our successful practice for the logical structure extraction.

4.1 Data analyses to provide evidence for our initial assumption

The main motivation to extract the logical configuration is the results which we observed after analyzing the terminology found in the document headings, document body, and document index. Our first assumption was that document headings, i.e. those that appear in the table of contents, carry the most important concepts with respect to a targeted document. This assumption seems particularly reasonable when we have a large document with numerous headings. That is why people usually explore the table of contents when they start working with a new document.

In the data analysis phase, first we created the following text files from the original PDF document in order to isolate headings, document body, and document index:

1. A text file consisting of document headings (almost 2200 headings)
2. A plain text file for the whole document excluding headings and index part
3. A text file for the document index

Afterward, we used some simple Unix commands to count and sort all words in the mentioned text files, in a case-insensitive manner and considering plural words as singular ones. We also ignored some kinds of words such as verbs, prepositions, numbers, etc. in our samples shown in Table 3 and Table 5.

In the initial stage, for the simplification of our data analysis, we just collected the 50 most frequent words in document headings, as shown in Table 3. After that, we calculated the overall frequency of these words in the whole document, Table 4, and then gathered the 24 most frequent words in the document index, Table 5. The reason for selecting just 50 and 24 words was that, after these words the "Number of Occurrence" parameter became 2 and consequently 1 for the rest of words which was a relatively long list. We also did some statistical calculations on these data collections, which is presented in the following table:

Table 2. Statistical summary related to the heading and index words

	Heading words: Table 3	**Occurrences of frequent header words in the main text: Table 4**	**Index words: Table 5**
Number of Data	50	50	24
Minimum	3.000	18.00	3.0000
Maximum	296.000	1063.00	45.0000
Median	11.000	168.00	6.0000
Mean	36.880	304.36	8.9583
Standard Deviation	68.846	311.35	9.1008
Variance	4739.781	96936.77	82.8243

As you can see in Table 2, the *mean* of the heading words (Table 3) is 36.88 while the occurrence of the 7 most frequent words in the document headings are in the range of [296,171], which shows a huge gap between the frequencies of these 7 words in comparison to the other 43 words. This issue can be seen in the *standard deviation* (68.846) of this data collection as well, which has a considerable difference with the arithmetic mean of the mentioned data set. These seven words are among the most important UML keywords.

The other interesting observation is, when we gathered the second data collection which is the overall frequency of the first data set, these 7 words were still among the most frequent words (Table 4), all found in the first 15 positions.

To go further beyond these seven words, and to better understand the distribution of heading words in the document body, and to investigate our earlier assumption with respect to the table of contents, we compared these 50 words with the most frequent words of the whole document. In total, we collected almost 10,000 unique tokens in the entire document for counting, and observed that all the words in the first data collection (Table 3) were among the first 1,000 words of these unique tokens. This means that if we rank the set of words in the whole document based on their frequency, the heading words would be in the top level of this ranking scheme.

The last examination was about the document index. We collected the 24 most frequent words from index section and observed that all of them were also found first 50 most frequent heading words. This shows that the document index could be a good source of data collection for document engineering and concept extraction.

To make a solid conclusion for this part of our research, we extended the sample population to more than 50 words for other documents. We also did such statistical analyses on some other similar software specifications and observed the same results, specifically, when the specifications were long documents with various headings. These outcomes were our major motivation for extracting the logical structure of documents based on their headings. We later on show how this extracted structure could be useful for generating multiple hypertext pages and cross referencing all over the document.

Table 3. The 50 most frequent words in the document headings

	#	Frequent words		#	Frequent words
1	**296**	**Generalization**	26	**11**	**BehaviorStateMachines**
2	**213**	**Description**	27	**10**	**Component**
3	**201**	**Semantic**	28	**8**	**StructuredActions**
4	**178**	**Association**	29	**7**	**Interface**
5	**172**	**Constraint**	30	**7**	**Dependency**
6	**171**	**Notation**	31	**7**	**CompleteStructuredActivities**
7	**171**	**Attribute**	32	**6**	**UseCase**
8	**45**	**Kernel**	33	**6**	**ProtocolStateMachines**
9	**22**	**IntermediateActions**	34	**6**	**ExtraStructuredActivities**
10	**21**	**Template**	35	**6**	**BasicBehaviors**
11	**21**	**BasicInteractions**	36	**5**	**Structure**
12	**18**	**Variation**	37	**5**	**InternalStructures**
13	**18**	**CompleteActivities**	38	**5**	**FundamentalActivities**
14	**17**	**Node**	39	**5**	**Collaboration**
15	**16**	**Profile**	40	**4**	**Property**
16	**16**	**Communication**	41	**4**	**Port**
17	**15**	**Class**	42	**4**	**Operation**
18	**14**	**CompleteActions**	43	**4**	**Enumeration**
19	**13**	**Diagram**	44	**4**	**Concept**
20	**13**	**BasicActivities**	45	**4**	**Classifier**
21	**12**	**SimpleTime**	46	**4**	**Action**
22	**12**	**IntermediateActivities**	47	**3**	**PowerType**
23	**12**	**Fragment**	48	**3**	**Package**
24	**12**	**BasicActions**	49	**3**	**Behavior**
25	**11**	**StructuredActivities**	50	**3**	**Artifact**

Table 4. Frequency of the 50 heading words from Table 3 as found in the words extracted from the entire document

	#	Frequent words		#	Frequent words
1	**1063**	**Classifier**	26	**160**	**Structure**
2	**958**	**Association**	27	**131**	**Concept**
3	**881**	**Node**	28	**130**	**Communication**
4	**850**	**Behavior**	29	**112**	**CompleteActivities**
5	**844**	**Action**	30	**100**	**Fragment**
6	**843**	**Constraint**	31	**92**	**Variation**
7	**766**	**Notation**	32	**76**	**BasicActions**
8	**758**	**Attribute**	33	**69**	**BasicActivities**
9	**700**	**Package**	34	**67**	**Enumeration**
10	**690**	**Class**	35	**60**	**BasicInteractions**
11	**674**	**Semantic**	36	**54**	**IntermediateActions**
12	**518**	**Operation**	37	**50**	**UseCase**
13	**470**	**Diagram**	38	**50**	**StructuredActivities**
14	**427**	**Generalization**	39	**46**	**IntermediateActivities**
15	**391**	**Description**	40	**45**	**BehaviorStateMachines**
16	**390**	**Property**	41	**44**	**CompleteActions**
17	**382**	**Template**	42	**40**	**CompleteStructuredActivities**
18	**369**	**Component**	43	**37**	**PowerType**
19	**359**	**Interface**	44	**31**	**FundamentalActivities**
20	**279**	**Profile**	45	**28**	**BasicBehaviors**
21	**248**	**Kernel**	46	**24**	**StructuredActions**
22	**242**	**Port**	47	**24**	**SimpleTime**
23	**220**	**Dependency**	48	**23**	**InternalStructures**
24	**188**	**Collaboration**	49	**21**	**ExtraStructuredActivities**
25	**176**	**Artifact**	50	**18**	**ProtocolStateMachines**

Table 5. The 24 most frequent words in the document index

	#	Frequent words		#	Frequent words
1	**45**	**Kernel**	13	**6**	**BasicBehaviors**
2	**19**	**Template**	14	**5**	**ProtocolStateMachines**
3	**19**	**BasicInteractions**	15	**5**	**InternalStructures**
4	**17**	**Communication**	16	**4**	**Property**
5	**11**	**BehaviorStateMachines**	17	**4**	**Collaboration**
6	**10**	**Node**	18	**4**	**Classifier**
7	**10**	**Fragment**	19	**4**	**Class**
8	**9**	**SimpleTime**	20	**3**	**Port**
9	**8**	**Profile**	21	**3**	**Package**
10	**7**	**Interface**	22	**3**	**Operation**
11	**7**	**Dependency**	23	**3**	**CompleteActivities**
12	**6**	**UseCase**	24	**3**	**BasicActions**

In the next page, the corresponding visual presentations with respect to the above data collections and tables are demonstrated for superior understanding of these data sets. The first diagram (Figure 8) shows the occurrence of 50 most frequent words in the document headings, the second one (Figure 9), presents the overall frequency of the mentioned 50 words in descending order, and the last one (Figure 10), illustrates the occurrence of the first 24 frequent words in the document index.

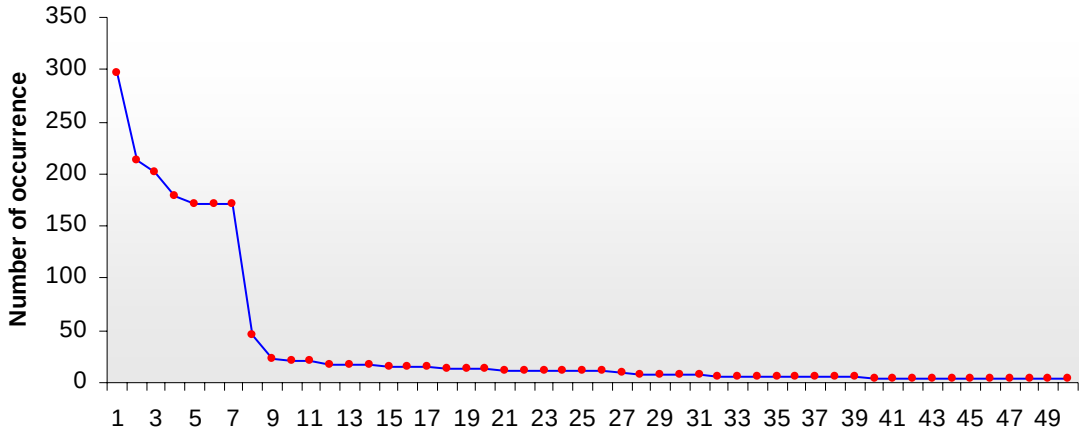


Figure 8. The 50 most frequent words in the document headings

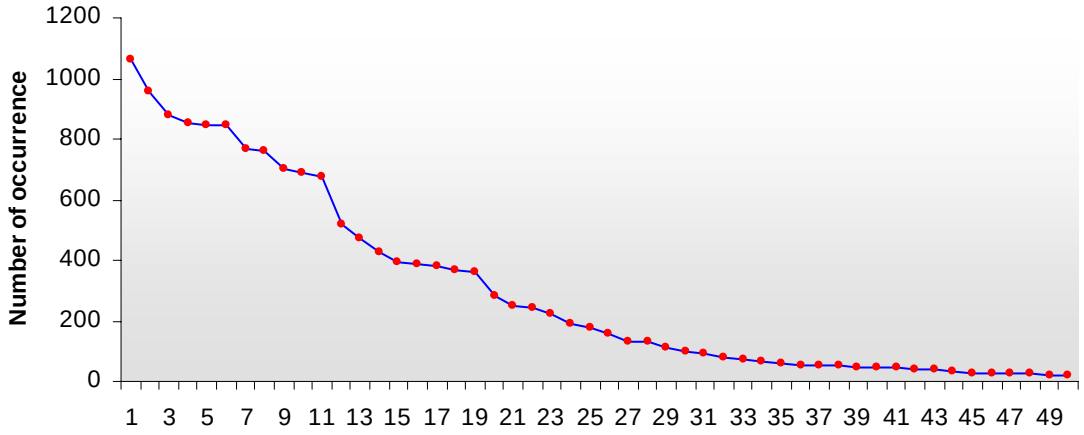


Figure 9. Frequency of the 50 heading words as found in the words extracted from the entire document

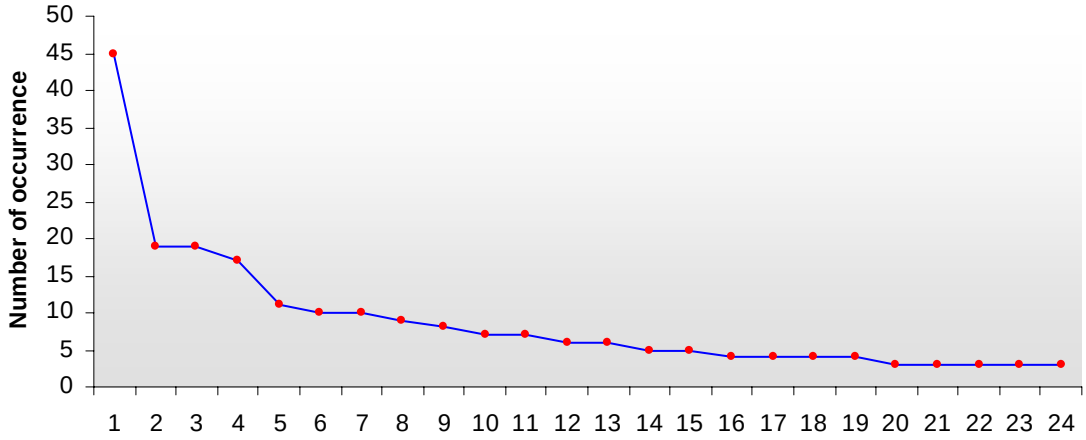


Figure 10. The 24 most frequent words in the document index

The following diagram (Figure 11) shows the occurrence of the first 50 words in the document headings and their corresponding overall frequency at the same time, it is another representation of our earlier discussions with respect to the first data collection and its first 7 frequent words.

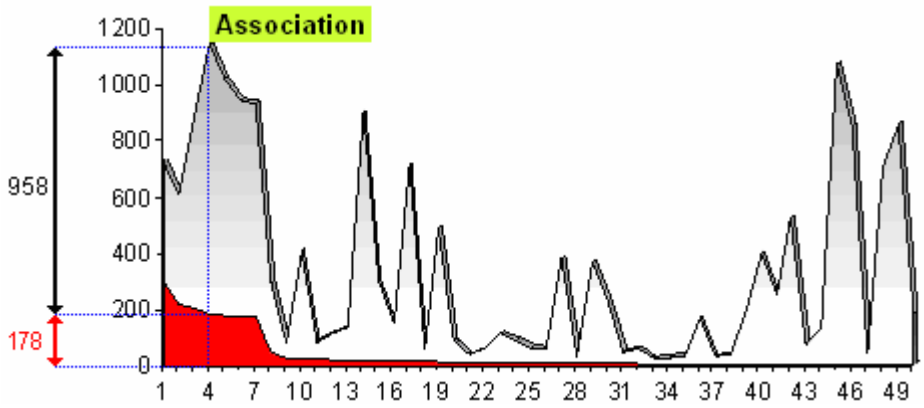


Figure 11. First data collection (smaller surface) and their frequency in the document as a whole (bigger surface)

This diagram (Figure 12) also demonstrates the relationship between the first 24 frequent words in the document index and their distributions in the header words.

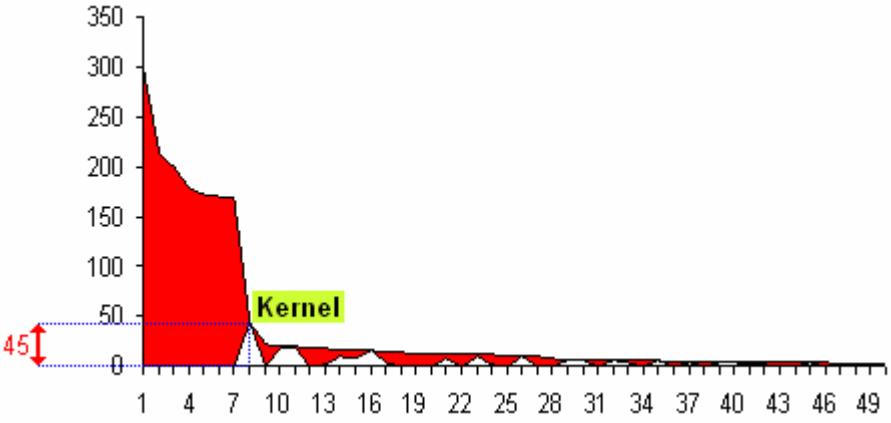


Figure 12. First data collection (bigger surface) and their frequency in the document index (smaller surface)

4.2 First refinement approach: Stack-based parser

In this approach, we turned to writing simple java scanning code, Appendix C, to scan for matching major tags, such as <Part>, <Sect> and <Div>, which Adobe Acrobat Professional 7.8 used to open and close each part, chapter, section, etc of the document. Consider the following simple structure of the document:

```
- <Sect name="Generalization">
  - <Sect name="Class-Ref">
      <Sect name="Name">...</Sect>
      <Sect name="Package-Ref">...</Sect>
    </Sect>
  </Sect>
```

Using a straightforward stack-based parsing approach, we converted this into:

```
- <Generalization>
  - <Class-Ref>
      <Name>...</Name>
      <Package-Ref>...</Package-Ref>
    </Class-Ref>
  </Generalization>
```

Unfortunately, after running the program for the different chapters and the whole document as well, it failed. We found out that there is a considerable amount of incorrect tagging. The tool opened each part, chapter, section, etc by “<Sect>” in a proper place of the document but it closed all of these tags by “</Sect>” in the wrong places. The problem was more crucial when we processed the whole document at once because of the accumulative mis-tagging. Here, a sample of this detection is presented:

```
- <Sect number="7.3">
    <Sect number="7.3.1">...</Sect>
    <Sect number="7.3.2">...</Sect>
  <!-- Correct Place for Closing "7.3"  -->
  <Sect number="7.4">
  </Sect>
  <Sect /> <!--  Wrong Place  -->
```

Therefore, we could not extract the logical structure of the document by this simple approach and decided to develop a new program which is more powerful and capable of detecting such a wrong tagging. In the next part, our successful practice with corresponding results is provided.

4.3 Second implementation approach: Bookmarks

In the second approach, we wrote a java-based parser which focused on a keyword, “LinkTarget”, which corresponds to the bookmark elements created in the previous transformation phase by Adobe Acrobat Professional 7.8. This keyword is attached to each heading in the bookmark such as headers of parts, chapters, sections, etc. Therefore, as a first step, we extracted all the lines containing the named keyword and put them in a queue: “LinkTargetQueue”. We also defined different types of headings in the UML superstructure specification with respect to its logical structure; you can see this classification in Table 6.

Table 6. Different kinds of headings

T	Sample Heading	Type
1	***Part I - Structure***	**Part**
2	***7 Classes***	**Chapter**
3	***7.3 Class Descriptions***	**Section**
4	***7.3.1 Abstraction***	**Subsection**
5	***Generalization, Notation, etc***	**Keyword**
6	***Annex***	**End part**
7	***Index***	**Last Part**

Then, we applied an algorithm which takes the “LinkTargetQueue” as its input; each node of this queue is a line of the input XML file which has “LinkTarget” substring as a keyword: **“<P id="LinkTarget_111914">7 Classes </P>”**. This algorithm extracts

headings (7 Classes) and then defines their type by pattern matching according to Table 6 (Type: Chapter & T=2). Afterward, it applies a stack based approach for opening and closing corresponding tags at the suitable place in the XML file. The implemented algorithm in Java is attached in Appendix B.

By applying this logical analyzer, we extracted 2191 headings from the UML Superstructure Specification (version 2.1) and created a new XML file for this document. We also tested the other documents and specifications such as UML Infrastructure (version 2.0); the extractions were well in all cases with 100% accuracy.

As an alternative solution for the extraction of headings, we can apply various parsing packages as well. We require writing a comprehensive grammar for analyzing the XML document; in particular, we need to parse the internal structure of some of the tag-delimited data. Using the same keyword ("LinkTarget") can significantly facilitate the structure of our grammar for the extraction. The following are some examples of heading texts that needed parsing:

7 Classes	*Number + Whitespace + Word*
7.1 Overview	*Number + Period + Number + Whitespace + Word*
7.2 Abstract Syntax	*Number + Period + Number + Whitespace + Word + Whitespace + Word*
7.3.1 Abstraction	*Number + Period + Number + Period + Number + Whitespace + Word*

In Figure 13, you can see the corresponding state machine with respect to the above sample headings; **"Next Line"** is the acceptance state in this machine.

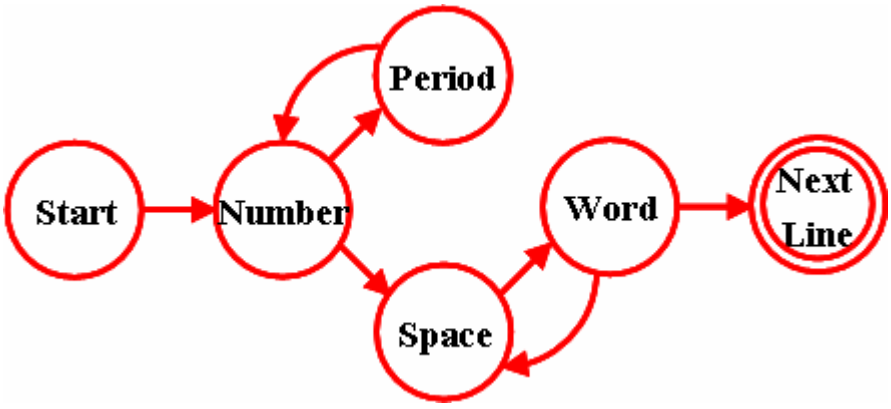


Figure 13. State machine for sample headings

Procedure LogicalStructureExtraction(LinkTargetQueue)

F // a new XML file

L // a line: e.g.: <P id="LinkTarget_111914">7 Classes </P>

H // Heading: e.g.: 7 Classes

T // Type: e.g.: for the Chapters, $T_{Chapter} = 2$

$T_{Last\ member\ of\ the\ HeadingStack} = 0$

HeadingStack = empty

While (LinkTargetQueue != empty) **do**

Get "L" from the LinkTargetQueue

Extract the heading "H" from the "L"

Define heading's type: "T"

While ($T =< T_{Last\ member\ of\ the\ HeadingStack}$) **do**

Pop "H" and "T" from the HeadingStack

Close the suitable tag w.r.t the popped "T"

If (HeadingStack == empty)

Break this while loop

End if

End while

Push the new "H" and "T" in the HeadingStack

Open new tags w.r.t the pushed "H" & "T"

End while

While (HeadingStack != empty) **do**

Pop "H" and "T" from the HeadingStack

Close the suitable tag w.r.t the popped "T"

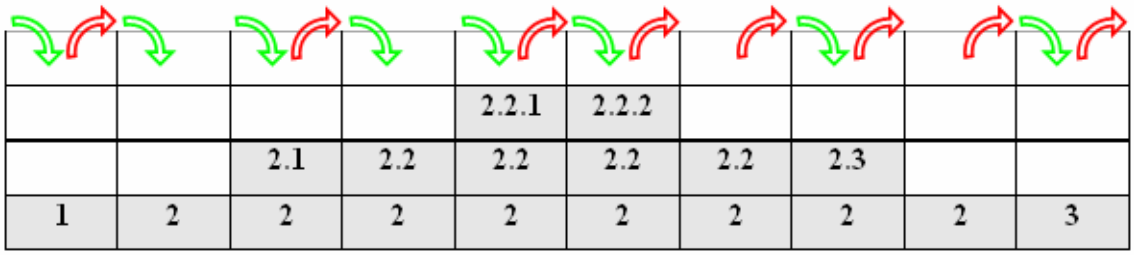
End while

Return "F"

End procedure

To trace the proposed algorithm, assume the following chapter, section and subsection headings in the “LinkTargetQueue”:

1 Heading → Chapter, $T_{Chapter}$ =2
2 Heading
2.1 Heading → Section, $T_{Section}$ = 3
2.2 Heading
2.2.1 Heading → Subsection, $T_{Subsection}$ = 4
2.2.2 Heading
2.3 Heading
3 Heading



The result would be as follows:

<Chapter number=”1”
</Chapter>
<Chapter number=”2”>
<Section number=”2.1”>
</Section>
<Section number=”2.2”>
<Subsection number=”2.2.1”>
</Subsection>
<Subsection number=”2.2.2”>
</Subsection>
</Section>
<Section number=”2.3”>
</Section>
</Chapter>
<Chapter number=”3”>
</Chapter>

The extracted logical structure in XML format with respect to the UML superstructure specification (version 2.1) is presented in the Figure 14. It consists of 4 major parts, 18 chapters and numerous concepts such as generalizations, description, etc. We extracted 71 different tags in three categories (Structures, Blocks and Keywords) which you can see some of them with their number of occurrence in Table 7**.**

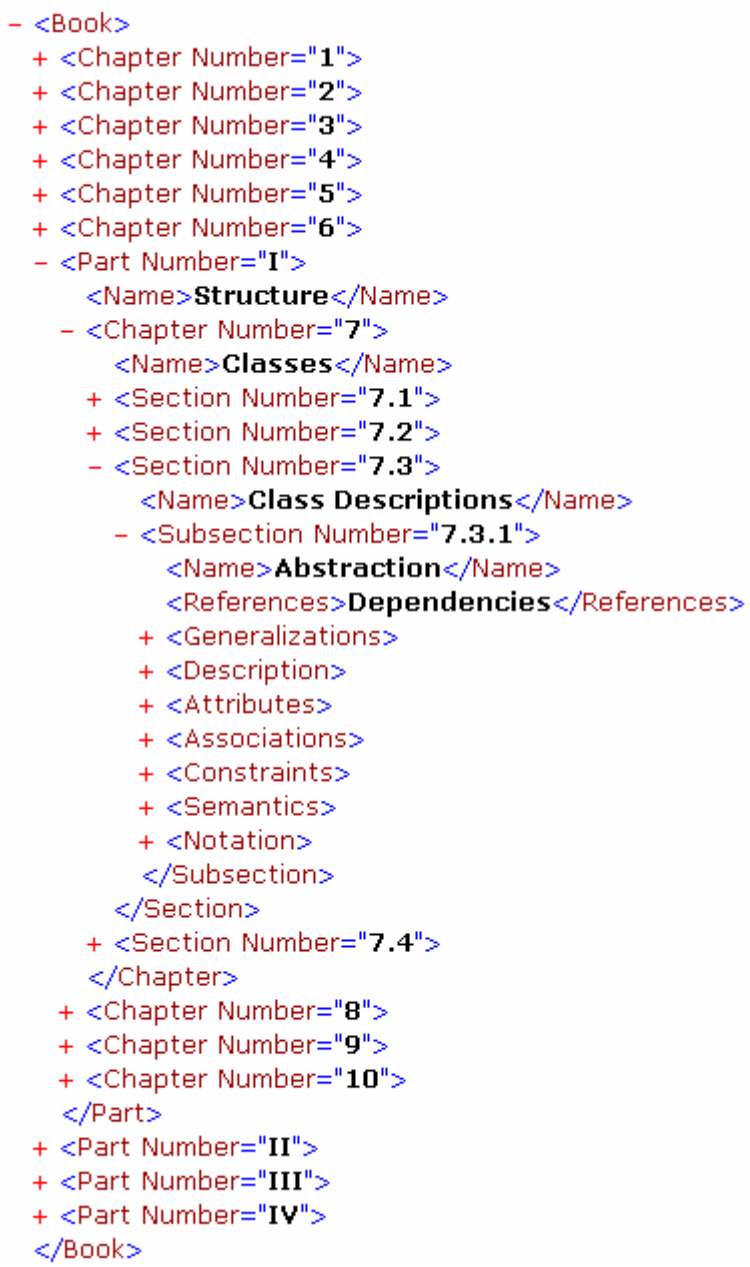


Figure 14. Logical structure extracted in XML format

Table 7. Sample XML tags in the UML superstructure specification

Structures	#	Blocks	#	Keywords	#
<Part>	**4**	<P>: Paragraph	**8228**	<Associations>	**177**
<Chapter>	**18**	<Figure>: Figure	**738**	<Attributes>	**171**
<Section>	**74**	<Table>: Table	**105**	<Constraints>	**172**
<Subsection>	**314**	<TH>: Table Header	**283**	<Description>	**202**
		<TR>: Table Row	**547**	<Generalization>	**296**
		<TD>: Table Data	**1721**	<Notation>	**169**
		<L>: Lists	**245**	<Semantics>	**179**
		<LI>: List Item	**765**	etc	

After extracting such a logical structure and creating the new XML file, we imported our document into "Protégé release version 3.2.1", which is an open source ontology editor and knowledge base framework, in order to visualize our XML structure using the commands available in its XML tab. In Figure 15, part of the logical structure model of the document is presented using the Jambalaya feature of Protégé 3.2.1. In the next chapter, we explain the XML schema generation for our extracted document.

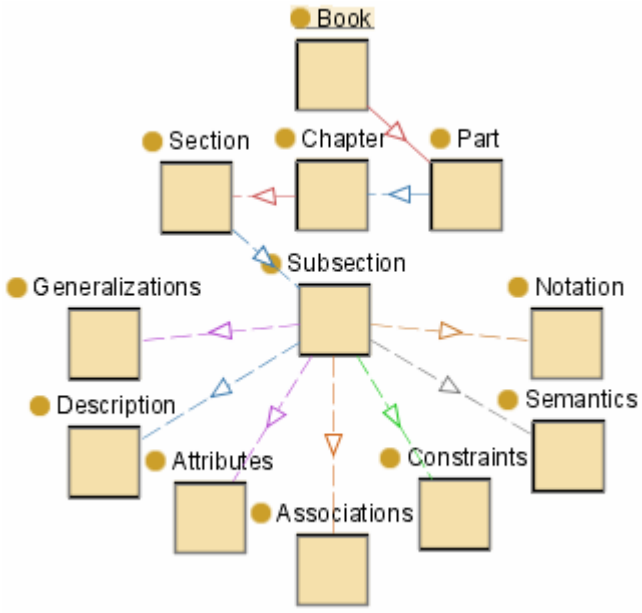


Figure 15. Logical structure model in the protégé 3.2.1

5 Text Extraction to Create Initial Hypertext Pages

In this chapter, firstly we evaluate our document to be well-formed. Afterward, we generate a valid XML schema with some schema component presentations in order to show the configuration of several XML elements. Then, producing multiple outputs along with connecting them together is illustrated. Finally, we address the formation of the document's key elements such as anchor links, figures, tables, and lists. The created user interface consists of the following major elements:

- A page for the table of contents
- A separate page for each major headings (418 HTML pages)
- Hyperlinks for accessing to the table of contents, next and previous pages
- Two separate pages for the package and class hierarchy of the UML (v2.1)
- Various cross references all over the document

To increase the usability of the document and highlight specific classes of information, we used different colors to present each of our XML elements.

5.1 Checking well-formedness

Every XML document must be *well-formed* which means that it properly matches opening and closing tags and abides by logical rules of nesting [36]. For well-formed checking and validating, we used a tool named “Stylus Studio® 2007 XML Enterprise Suite” which is an XML integrated development environment [41].

When we checked the well-formed property of the document we produced in Chapter 4, we discovered two types of errors:

1. The first was related to the forbidden notations among XML tags such as **“>”** (greater than) and **“<”** (less than) in mathematical equations.

2. The second error type, was incorrectly nested opening and closing tags in the generated document in three situations: nested lists (Figure 16), complex tables with figure and hyperlinks in their cells (Figure 17), and lists with two columns (Figure 18) spread over multiple pages. There were few errors in this respect; given the relatively small number in the large document the solution we chose was to fix them by hand: moreover, we did not encounter with such errors when we converted other PDF specifications such as the final version of the current targeted document (which was a draft version) or UML Infrastructure Specification.

Figures 11 to 13 also demonstrate the lack of usability, difficulty in browsing, and general complexity of the original input document for end users.

There are two distinct types of compliance. They are:

- *Abstract syntax compliance.* For a given compliance level, this entails:
  - compliance with the metaclasses, their structural relationships, and any constraints defined as part of the merged

5

UML metamodel for that compliance level and,

- the ability to output models and to read in models based on the XMI schema corresponding to that compliance level.

- *Concrete syntax compliance.* For a given compliance level, this entails
  - Compliance to the notation defined in the "Notation" sections in this specification for those metamodel elements that are defined as part of the merged metamodel for that compliance level and, by implication, the diagram types in which those elements may appear. And, optionally:
  - The ability to output diagrams and to read in diagrams based on the XMI schema defined by the Diagram Interchange specification for notation at that level. This option requires abstract syntax and concrete syntax compliance.

Figure 16. A nested list spread over two pages 5-6 (UML Spec. v2.1)

Component has provided Port (typed by Interface)		See "Port"

163

Table 8.1 - Graphic nodes included in structure diagrams

Node Type	*Notation*	*Reference*
Component uses Interface		See "Interface"

Figure 17. A complex table spread over two pages 163-164 (UML Spec. v2.1)

Associations

Package Artifacts

- nestedArtifact: Artifact [*] — The Artifacts that are defined (nested) within the Artifact. The association is a specialization of the *ownedMember* association from Namespace to NamedElement.

Issue 8132 - rename 'ownedProperty' to 'ownedAttribute'
8137 - reformulated subsets constraints to conform to document conventions

- ownedAttribute : Property [*] — The attributes or association ends defined for the Artifact. {Subsets *Namespace::ownedMember*}.
- ownedOperation : Operation [*] — The Operations defined for the Artifact. {Subsets *Namespace::ownedMember*}

205

- manifestation : Manifestation [*] — The set of model elements that are manifested in the Artifact. That is, these model elements are utilized in the construction (or generation) of the artifact. {Subsets *NamedElement::clientDependency*; Subsets *Element::ownedElement*}

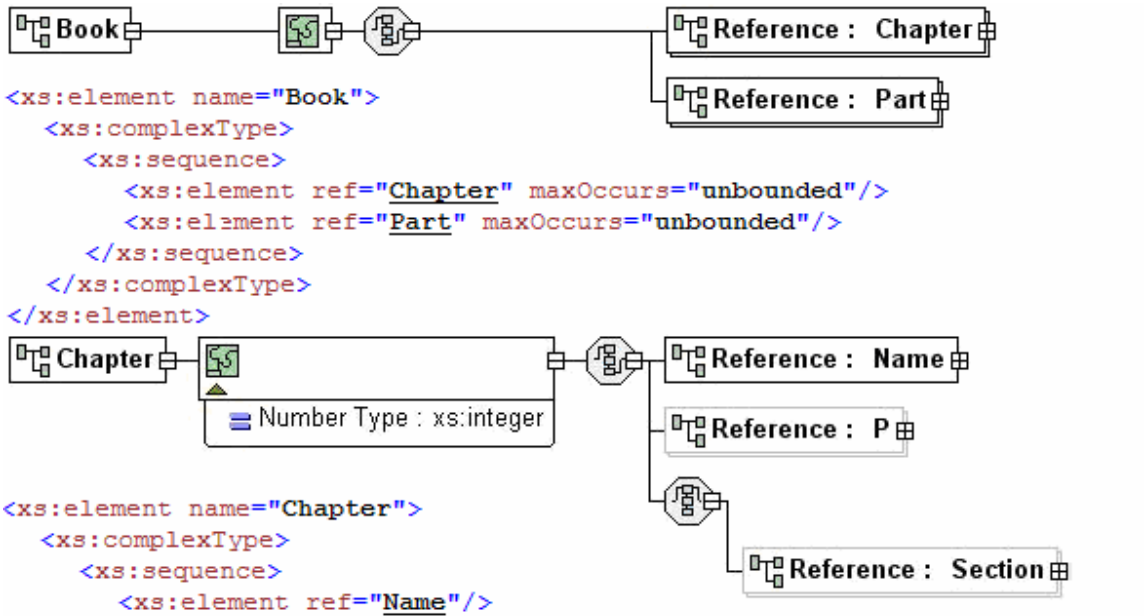
Constraints

No additional constraints

Figure 18. A list with two columns spread over two pages 205-206 (UML Spec. v2.1)

5.2 Generating a valid schema

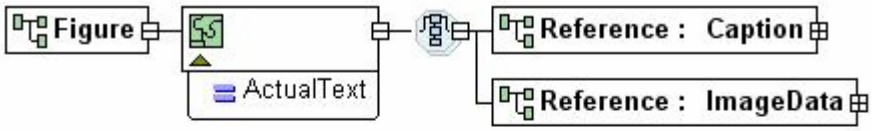
In addition to checking for well-formedness, it is necessary in XML documents to also check for *validity*, i.e. whether a document uses tags in a consistent manner with its schema or not. “A valid document has data that conforms to a particular set of user-defined content rules, or XML Schema, which describe correct data values and locations” [36]. Most of the XML tools support automatic schema generation in addition to the well-formedness checking, they also provide features for error detection during validation procedure which makes it very easy to validate a schema. We firstly generated an XML schema and then validated our extracted document with “Stylus Studio”. In Figure 19 to Figure 22, some schema component representations are provided.




```
<xs:element name="Book">
   <xs:complexType>
      <xs:sequence>
         <xs:element ref="Chapter" maxOccurs="unbounded"/>
         <xs:element ref="Part" maxOccurs="unbounded"/>
      </xs:sequence>
   </xs:complexType>
</xs:element>
```

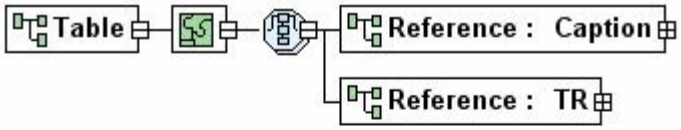
```
<xs:element name="Chapter">
   <xs:complexType>
      <xs:sequence>
         <xs:element ref="Name"/>
         <xs:element ref="P" minOccurs="0" maxOccurs="unbounded"/>
         <xs:choice>
            <xs:element ref="Section" minOccurs="0" maxOccurs="unbounded"/>
         </xs:choice>
      </xs:sequence>
      <xs:attribute name="Number" type="xs:integer" use="required"/>
   </xs:complexType>
</xs:element>
```

Figure 19. Schema component representations: “Book” & “Chapter”



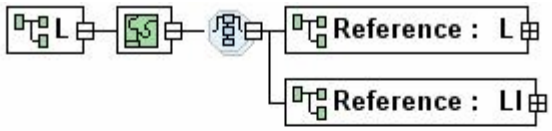

```
<xs:element name="Figure">
  <xs:complexType mixed="true">
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:element ref="Caption"/>
      <xs:element ref="ImageData"/>
    </xs:choice>
    <xs:attribute name="ActualText"/>
  </xs:complexType>
</xs:element>
```

Figure 20. Schema component representation: “Figure”




```
<xs:element name="Table">
  <xs:complexType>
    <xs:choice maxOccurs="unbounded">
      <xs:element ref="Caption"/>
      <xs:element ref="TR"/>
    </xs:choice>
  </xs:complexType>
</xs:element>
```

Figure 21. Schema component representation: “Table”




```
<xs:element name="L">
  <xs:complexType>
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:element ref="L"/>
      <xs:element ref="LI"/>
    </xs:choice>
  </xs:complexType>
</xs:element>
```

Figure 22. Schema component representation: “List”

5.3 Producing multiple outputs

To enhance the user interface efficiency and facilitate document browsing, we decided to produce multiple outputs using the ***<xsl:result-document>*** element, and to generate a small HTML page for each Part, Chapter, Section, and Subsection. This resulted in 418 HTML files in total. The other alternatives were to create a big HTML file, such as existing software specifications on the web, or create eighteen relatively big HTML files for each chapter of our targeted document, such as our fist prototype design.

Our major motivations and objectives for the decision to create many small pages were as follows:

- **A better sense of 'location' when navigating cross-references:** In a large hypertext document one can use anchors (with the syntax <a name="xyz"> and linked to using <a href="#xyz">) to allow jumping from section to section. However, the result of jumping to a section in this manner places you into the middle of a document. If the destination of the jump has a title that is not clear, or is a section with a very small amount of text (i.e. a subsequent section title is also visible in the browser after jumping), or is at the very end of the document, the user can find it confusing to determine exactly where they have arrived at. This issue can be even more confusing when the user has stored such an anchor in a bookmark. On the other hand if the destination of a jump is an entire hypertext page, the above problems go away.
- **Less chance of the user getting lost:** Users are less likely to get lost by scrolling in small pages in comparison to long pages. In a long page, after following a link a user may then move to some other part of the document using a few 'page down' clicks or by searching. But then the user may not know how to go 'back' to where they came from unless they happen to remember the section number or title of the section they came from. Even then they may have to search in order to return. The problem can be

exacerbated if the user leaves a page alone in their browser for a period of time after scrolling. If instead the document is organized as many small hyperlinked pages, it becomes is simply a matter of hitting the ‘back’ button in the browser.

- **A less overwhelming sensation:** A smaller document should help users to manage larger amounts of information and understand the document more efficiently.
- **Faster loading:** Users are not always interested in downloading the whole document at once, especially when the document is fairly big.
- **Easier printing:** Users can print particular topics and pages in accordance to their interests and demands.
- **Statistical analyzing:** It may be useful to calculate the most frequent page-loads and the time which users stay in each page. This information could be used to improve the UI and the specification itself, and to determine what the most significant information is.

Creating small pages has one potential drawback, however: The ‘original order’ of the document may be an asset worth preserving. This might be because the user is used to reading linearly, or they may want to read through the whole document in a systematic manner.

To prevent loss of the original order, we created “Previous” and “Next” hyperlinks in each page and help the user to realize where he or she is, has been, and can go.

It is important to note that there is a logical limit to how finely one wants to break down a large document into small hypertext pages (in the absurd extreme, one could separate each paragraph). What we have done is limit the division to the subsection level; this results in pages that still have several sub-subsection titles inside them. We also created a few special ‘long’ pages. These include a table of contents, a page listing all the UML packages (with the classes they contain), and a page listing all UML classes alphabetically. Links to a package will link to an anchor within the packages page.

We also used color coding to help the user understand what type of information they

are viewing. The following is the scheme:

(1) **Hyperlinks: Blue**

(2) **Headers: Red**

(3) **Paragraphs: Black**

(4) **Table & Figure Captions: Fuchsia**

(5) **Simple Lists: Navy**

(6) **First Part of the Nested Lists: Maroon**

(7) **Second Part of the Nested Lists: Olive**

(8) **Table Headers, Keywords, and Package/Class Names: Lime**

(9) **Table Cells: Green**

In order to generate the separate hypertext pages, we applied "Saxon-B 8.9" [43], which is an open source XSLT and XQuery processor developed by Michael Kay, the editor of the XSLT 2.0 specification. Saxon versions exist for both .Net and Java; we used the Java version with the following command to transform the targeted document (UML.xml) by the XSLT code which we developed (UML.xsl):

java -jar saxon8.jar -t UML.xml UML.xsl.

In our document, each *"Part"* consists of a body as well as Chapters, Sections, and Subsections inside of itself; each *"Chapter"* also consists of Sections and Subsections in addition to its body, and so forth. Therefore, to exclude Chapters, Sections, and Subsections from an HTML file which is just for the body of a *"Part"*, we had to create a global template for each of these entities in our XSLT code, as you can see in the small piece of our code in the next page, a global template is useful if an element occurs within various elements or in various locations of the document.

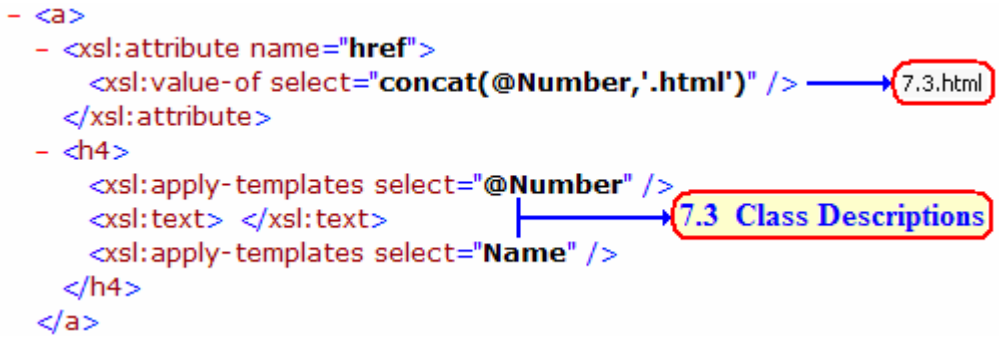
The other significant issue was the naming of these output files. This procedure had more importance when we wanted to link these files together and create the table of contents; therefore, we used the following XPath function to name our outputs:

concat ('UML/', @Number, '.html')

This function concatenates three strings, creates a folder named "UML", and puts each HTML files in this folder. The "*@Number*" refers to the attribute of <Part>, <Chapter>, <Section>, and <Subsection> elements. As a result, we named our HTML outputs as follows: {e.g.: "I.html", "7.html", "7.1.html", "7.2.html", "7.3.html", "7.3.1.html", "7.3.2.html", "7.3.3.html", etc}.

```
- <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="2.0">
    <xsl:output method="html" indent="yes" name="html" />
  - <xsl:template match="/">
    - <xsl:for-each select="//Part">
        <xsl:variable name="filename" select="concat('UML/',@Number,'.html')" />
        <xsl:value-of select="$filename" />
      - <xsl:result-document href="{$filename}" format="html">
        - <html>
          - <body>
              <xsl:apply-templates />
              <!-- XSLT Codes  -->
            </body>
          </html>
        </xsl:result-document>
      </xsl:for-each>
    + <xsl:for-each select="//Chapter">
    + <xsl:for-each select="//Section">
    + <xsl:for-each select="//Subsection">
    </xsl:template>
  - <xsl:template match="Chapter">
      <!-- Global Template  -->
    </xsl:template>
  - <xsl:template match="Section">
      <!-- Global Template  -->
    </xsl:template>
  - <xsl:template match="Subsection">
      <!-- Global Template  -->
    </xsl:template>
</xsl:stylesheet>
```

Since file names were created from the "*@Number*" attribute, we were able to thus facilitate access to each of these files. For instance, by the following piece of XSLT code we generated the related hyperlinks in the table of contents page:



In the next section, we illustrate how to connect these files together by the "*Previous*" and "*Next*" hyperlinks at the top of each page.

5.4 Connecting generated outputs sequentially

In the earlier section, we generated 418 HTML pages for our document. In a later section we will be creating contextual hyperlinks and tables of contents that will allow direct jumping to various pages, however we would still like to link the pages together by creating *"Previous"* and *"Next"* links in each page. This will allow the reader to proceed through the document in its original sequence, should they wish to do that.

We applied two methods in this regard; in the first one we used XPath expressions and in the second one we developed a java program along with a simple XSLT program.

5.4.1 Connecting pages using XPath expressions

To link the generated outputs, firstly we developed some XPath expressions to connect each of these files together. In total, we had 16 navigation paths to go from *{Part, Chapter, Section, Subsection}* to *{Part, Chapter, Section, Subsection}*. Figure 23 and Table 8 show ten unique combinations of these navigation paths.

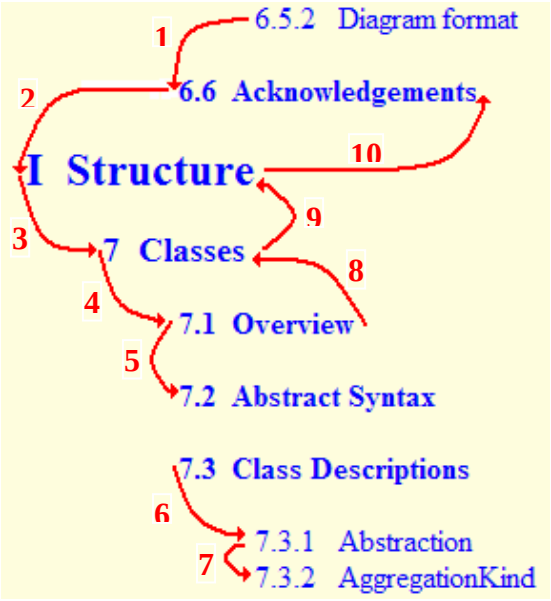


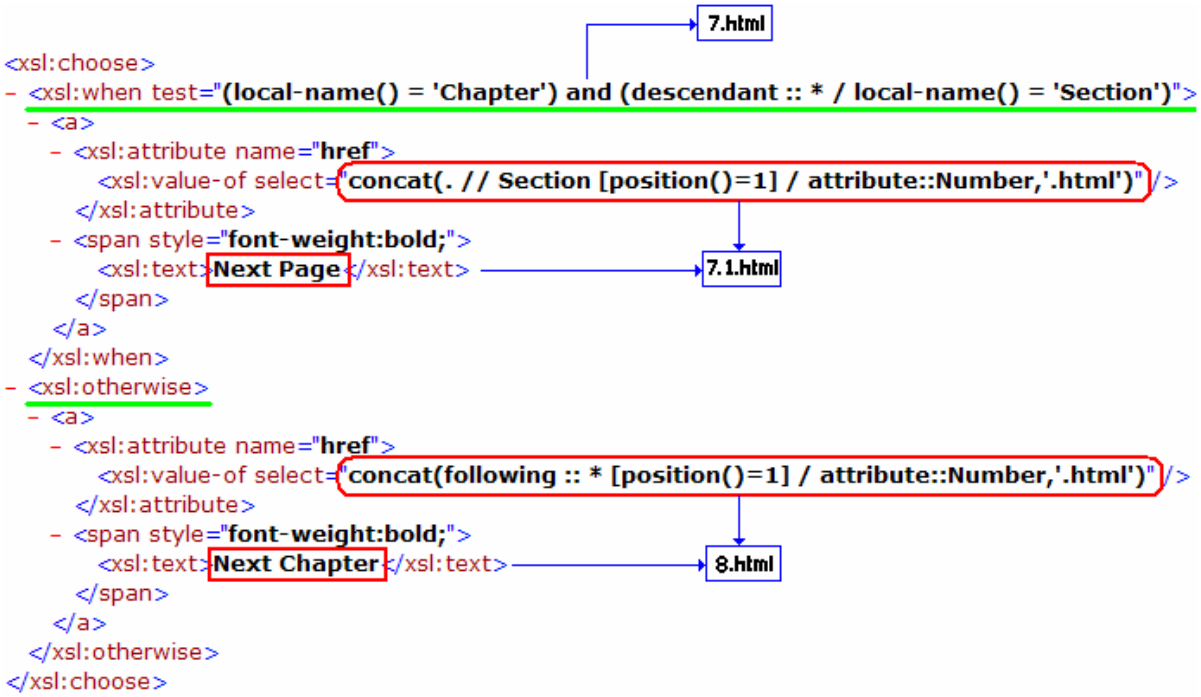
Figure 23. Table of contents showing distinct types of navigation paths

As you can see, we had to write diverse XPath expressions to cover all possible permutations at a suitable place in the document.

Table 8. Some document navigation paths related to the figure 18

	From	To		From	To
1	***Subsection***: 6.5.2	***Section***: 6.6	**6**	***Section***: 7.3	***Subsection***: 7.3.1
2	***Section***: 6.6	***Part***: I	**7**	***Subsection***: 7.3.1	***Subsection***: 7.3.2
3	***Part***: I	***Chapter***: 7	**8**	***Section***: 7.1	***Chapter***: 7
4	***Chapter***: 7	***Section***: 7.1	**9**	***Chapter***: 7	***Part***: I
5	***Section***: 7.1	***Section***: 7.2	**10**	***Part***: I	***Section***: 6.6

Now, we present a sample of our XSLT code and XPath expressions with respect to the following situation. Assume that we would like to go from a *"Chapter"* to its subsequent *"Section"* (e.g. from Chapter 7 to Section 7.1) or from a *"Chapter"* to its next *"Chapter"* (e.g. from Chapter 7 to Chapter 8), the related XSLT code and XPath is presented here:



In this code, first we applied the following XPath condition which means that the current place is a *"Chapter"* and its immediate descendant is a *"Section"* (e.g. 7.html):

(local-name() = 'Chapter') and (descendant :: * / local-name() = 'Section')

Then, we used the following XPath function in the hyperlink's address part to go from a *"Chapter"* to its subsequent *"Section"* (e.g. 7.html to 7.1.html). This function means that selecting the <Section> element, which is in the first position, extracts its attribute named *"Number"*, and then concatenates it with *".html"*:

concat(. // Section [position()=1] / attribute::Number,'.html')

We also used the following XPath function in the hyperlink's address part to go from a *"Chapter"* to its next *"Chapter"* (e.g. 7.html to 8.html). This function means that selects the following sibling, which is in the first position, extracts its attribute named *"Number"*, and then concatenate it with *".html"*:

concat(following :: * [position()=1] / attribute::Number,'.html')

To connect all pages together, we developed various complex conditions and logical expressions to cover all combinations. This method works very well but it does not satisfy the generality property of our project. Diverse documents have different navigation structure; moreover, this method would be more complicated when we have numerous navigation paths. In the next part, we propose an easier method for connecting our HTML outputs.

5.4.2 Connecting pages using a programming approach

In this method, first of all we extracted all elements' attribute named *"Number"* sequentially (1, 2, 2.1, 2.2, 2.3, …, 7, 7.1, 7.2, 7.3, 7.3.1, etc) by the following XSLT code. Then we put them in a text file named *"Num.txt"* and developed a java program to link our files together.

```
- <xsl:for-each select="Part">
    <xsl:apply-templates select="@Number" />
  - <xsl:for-each select="Chapter">
      <xsl:apply-templates select="@Number" />
    - <xsl:for-each select="Section">
        <xsl:apply-templates select="@Number" />
      - <xsl:for-each select="Subsection">
          <xsl:apply-templates select="@Number" />
        </xsl:for-each>
      </xsl:for-each>
    </xsl:for-each>
  </xsl:for-each>
```

We developed a java program for the following algorithm in our second approach:

Procedure Linker()

Num.txt // a text file consisting of all attributes

A1, A2 // variables

A1 = Read the first attributes from “Num.txt” file // (e.g. A1 = 1)

A2 = Read the second attributes from “Num.txt” file // (e.g. A2 = 2)

Call SetupLink (A1, A2) // (e.g. (1, 2))

A1 = A2 // (e.g. A1 = 2)

While (True) do

A2 = Read an attribute from “Num.txt” // (e.g. A2 = 3, A2 = 4, A2 = 5)

If (End of the “Nume.txt”) **Then**

Break this while loop

End If

Call SetupLink (A1, A2) // (e.g. (2, 3), (3, 4), (4, 5))

A1 = A2 // (e.g. A1 = 3, A1 = 4, A1 = 5)

End while

End procedure

Procedure SetupLink(X1, X2)

UMLFolder // a folder which consists of 418 HTML files

X1, X2 // arguments

Extract the **X1.html** and **X2.html** from UMLFolder

// (e.g. 7.3.1.html & 7.3.2.html)

Set *"Next"* Hyperlink in **X1.html** based on the **X2** variable

// (e.g. in 7.3.1.html file *"Next"* hyperlink is equal to 7.3.2.html)

Set *"Previous"* Hyperlink **X2.html** based on the **X1** variable

// (e.g. in 7.3.2.html file *"Previous"* hyperlink is equal to 7.3.1.html)

End procedure

In the next section, we demonstrate our presentation methods for different kinds of document major elements and provide the related XSLT codes for the style sheet design.

5.5 Forming major document elements

To construct the major document elements such as hyperlinks, figures, tables, and lists, we developed various style sheets by XSLT programming and applied some tools such as “Altova StyleVision® 2007 Enterprise Edition” which is a visual style sheet designer for transforming XML and database content into HTML, PDF, and RTF output [42]. In the next parts, a complete discussion with respect to the style sheets design for document elements is demonstrated with relevant XPath expressions and XSLT codes.

5.5.1 Anchors in long pages

As mentioned earlier, we generated several ‘long’ hypertext pages, such a list of all UML packages and a general table of contents.

In our first prototype, which consisted entirely of long pages, we facilitated browsing and navigating by creating anchors for major elements within each page. As Figure 24 shows, we created a mini table of contents at the top of each page.

In the final version, which is divided into many small pages, the anchors are retained, but links to them only are found in references to packages, which link to the ‘long’ page of all packages.

In XSLT, each anchor has a unique name; in the example shown in Figure 25 we named them: “2.1”, “2.2”, “2.3”, and “2.4”.

Figure 24. Top of a long page in our first design, showing links to internal anchors

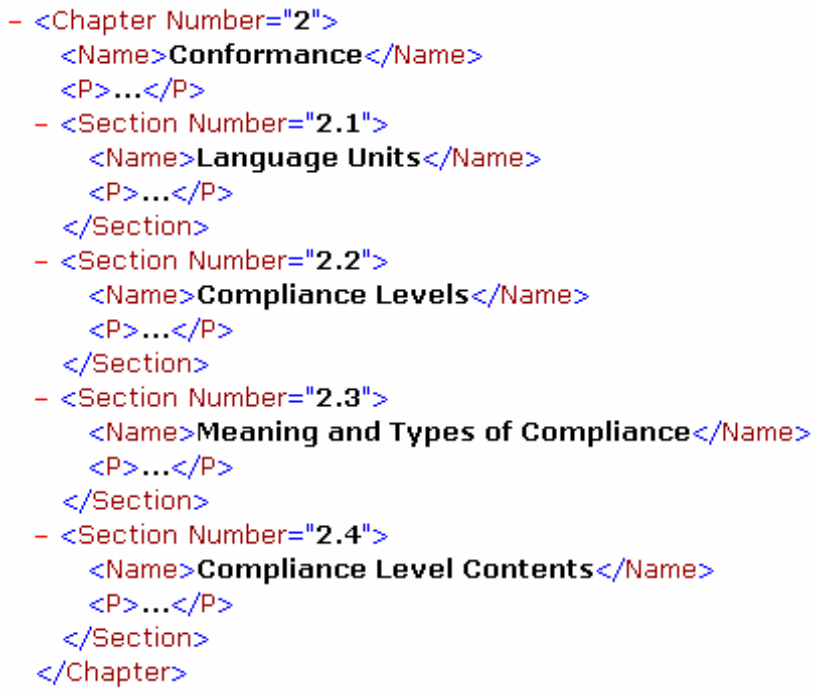


Figure 25. Heading tags structure in the XML document

Now, we show how to create a dynamic hyperlink pattern for the created bookmarks as we explained earlier. XSLT provides different methods for creating hyperlinks: *"Static"* in which you have to refer to a specific page, *"Dynamic"* in which you can refer to a node in an XML document, and *"Static & Dynamic"* which is the combination of both of them. An anchor consists of two parts: number sign: # and the name of the anchor.

To create a dynamic pattern for associating hyperlinks to their corresponding anchors, we applied the following XPath function in the hyperlink's address part:

"concat ('#', @Number)"

This function returns the concatenation of '#' and a dynamic string which refers to the <Section> element's attribute: e.g. "2.1", "2.2", and so forth. Since we deliberately named anchors as the <Section> element's attribute, consequently each hyperlink was connected to its corresponding anchor. In Appendix D, a portion of the XSLT code with respect to the automatic association of dynamic hyperlinks to static bookmarks is demonstrated.

5.5.2 Figures

In this section, we present the automatic importation of figures. As we mentioned before, our document has 738 figures. In the transformation phase, when Adobe Acrobat Professional converted our document into an XML file, it also created a folder named "images", put all figures in this folder, and named them as follows: "UML_img_1.jpg" to "UML_img_738.jpg". In Figure 26, you can see the structure of the <Figure> element which has two children: (1) <ImageData> with its "src" attribute, and (2) <Caption>.

```
- <P>
  - <Figure>
      <ImageData src="images/UML_img_1.jpg" />
    - <Caption>
        <P>Figure 2.1 - Level 0 package diagram</P>
      </Caption>
    </Figure>
  </P>
```

Figure 26. Figure tag structure in the XML document

For the relevant style sheet design, as you can see in Figure 27, first we took out the targeted chapter (Chapter 2: Conformance) and extracted the <Figure> element. Then, we inserted a dynamic hyperlink inside of the *"src"* attribute by the following XSLT code and XPath expression:

<xsl: value-of select="string(.)"/>

This line of code selects the value of the *"string(.)"* which returns the string value of the argument. The argument could be a number, Boolean, or node-set. Here it refers to the current node by *"dot"*; as a consequence, it replaced the values of this attribute ("images/UML_img_**1**.jpg", "images/UML_img_**2**.jpg"…"images/UML_img_**738**.jpg") into the hyperlinks and imported all figures at the right places inside of the targeted document. We also imported the related captions at the end of each figure by presenting its text content.

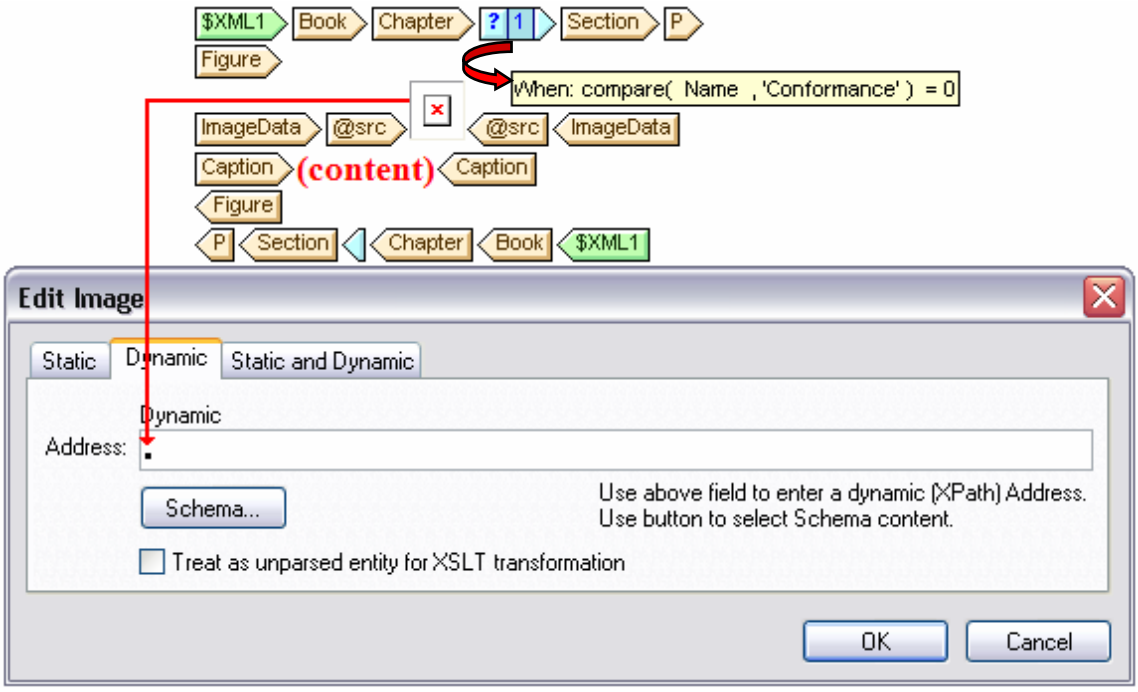


Figure 27. Screenshot of the Altova StyleVision for importing figures

This is the simplified template with respect to the dynamic importation of all figures.

```
- <xsl:template match="Figure">
  - <center>
    - <xsl:for-each select="ImageData">
      - <xsl:for-each select="@src">
        - <img>                                  "images/UML_img_1.jpg"
          - <xsl:attribute name="src">
              <xsl:value-of select="string(.)" />
            </xsl:attribute>
          </img>
        </xsl:for-each>
      </xsl:for-each>
    </center>
  - <center>
    - <xsl:for-each select="Caption">
      - <xsl:for-each select="P">
          <xsl:apply-templates />
        </xsl:for-each>
      </xsl:for-each>
    </center>
  </xsl:template>
```

5.5.3 Tables

In this part, we illustrate how to create a dynamic pattern for importing all tables with different sizes from the XML document. Figure 28 shows the <Table> element structure, it has two children: a <Caption> element which consists of a plain text, and <TR> elements (Table Row) which have two different children: <TH> elements (Table Header) and <TD> element (Table Data).

Figure 28. Table tag structure in the XML document

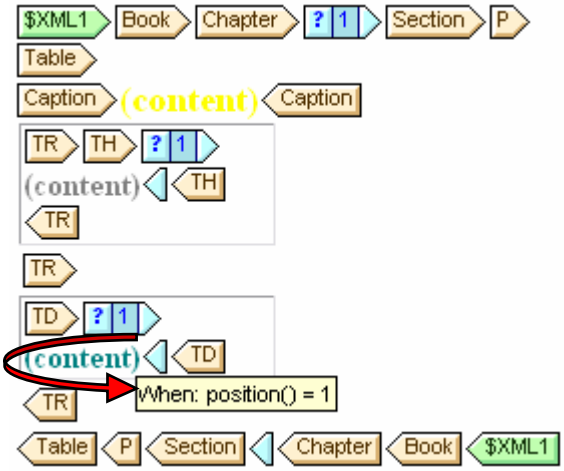
Dynamic table creation is supported by XSLT programming. In these tables one of the dimensions is fixed and the other one is dynamic, for example, the number of columns is fixed but the number of rows is variable.

To create a dynamic pattern for importing our tables, first we created the relevant caption and then selected the <TR> element, as you can see in Figure 29. Afterward, we constructed a dynamic table with six columns and varying number of rows, because tables of our document had at most six columns but much more rows. Figure 29 just

presents the first column; we excluded the other five columns in this figure.

To import table headers (<TH>) and table data (<TD>), we applied the following XPath function: ***"position ()"***. This function returns the index position of the node that is currently being processed. As an example, consider the first <TR> element in the Figure 28, if we apply ***"<TD> When: position () = 1 <TD>"*** then it returns *"Level 1"* string.

We used each of the following expressions in a conditional branch through the first column to the sixth one: ***position () = 1, position () = 2 … position () = 6***. They imported all relevant data at the corresponding table cells. If a table had, for instance, just four columns then the last two columns did not appear.



Compliance Summary			
Compliance level	**Abstract Syntax**	**Concrete Syntax**	**Diagram Interchange Option**
Level 0	**YES**	**YES**	**YES**
Level 1	**YES**	**YES**	**NO**
Level 2	**YES**	**NO**	**NO**

Figure 29. Screenshot of the Altova StyleVision for importing tables

In Appendix D, the relevant XSLT code for dynamic tables is demonstrated.

5.5.4 Lists

In this part, the style sheet design for lists is presented. Figure 30 shows the <L> element structure for a simple list. As you can see, it has two grandchildren named: <LI_Label> element and <LI_Title> element.

Figure 30. List tag structure in the XML document

To present a simple list, we first extracted <LI_Label> and <LI_Title> elements by ***<xsl:for-each select="LI_Label">*** & ***<xsl:for-each select="LI_Title">***, and then presented their contents, as shown in Figure 31. But for the nested lists, after extracting the second <L> element, we applied the following XPath expressions:

child :: * [position()=1] for the first part of the nested lists

child :: * [position()=2] for the second part (nested part) of the nested lists

child :: * means select all children of the current node, and ***child :: * [position()=1]*** means select the child which is in the first place, and so forth.

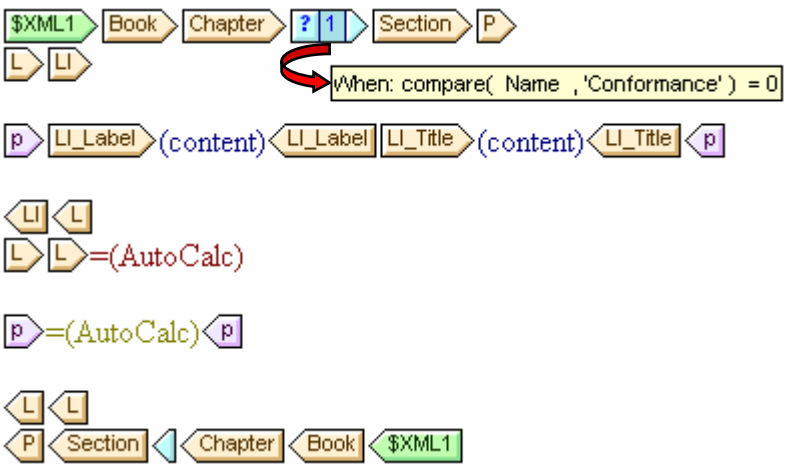


Figure 31. Screenshot of the Altova StyleVision for importing lists

In Appendix D, a portion of the XSLT code with respect to the simple and nested lists importation is demonstrated.

In the next chapter, we focus on development as well as improvement of the designed user interface to increase its usability by concepts extraction such as UML class and package hierarchies, and cross referencing all over the document.

6 Concept Extraction and Cross Referencing

In the first part of this chapter, first we present the related methods for concept extraction from our targeted document, UML Superstructure Specification; these concepts include UML class and package hierarchies. We apply logical expressions using XPath and XSLT to extract such concepts. Although this part has been designed specifically for the UML specification, it can give us a general view of how to perform concept extraction from other documents.

In the second section of the chapter, we present related works for cross referencing all over the document. We illustrate how we extracted almost three hundred keywords to use in the proposed cross referencing algorithm, implemented in Java. These hyperlinks help users to jump from one page to another in order to gather more information as required.

6.1 Concepts extraction

In this section, we present the concept extractions from our XML document. As we mentioned in previous chapters, there are numerous UML concepts in headings, this issue was one of our major reasons for the logical structure extraction of the document. As an example, Figure 32 shows class descriptions with respect to the *"Components"*

and *"Composite Structures"*; it also presents the packages which these classes belong to using *"from"* as a keyword. Since we tagged this information as chapter, section, and subsection headings, therefore using XPath expressions and XSLT code we extracted the class and package hierarchies of the UML Superstructure Specification (v2.1) in two separate pages for our final user interface. In the next part, we illustrate our methods for such extractions.

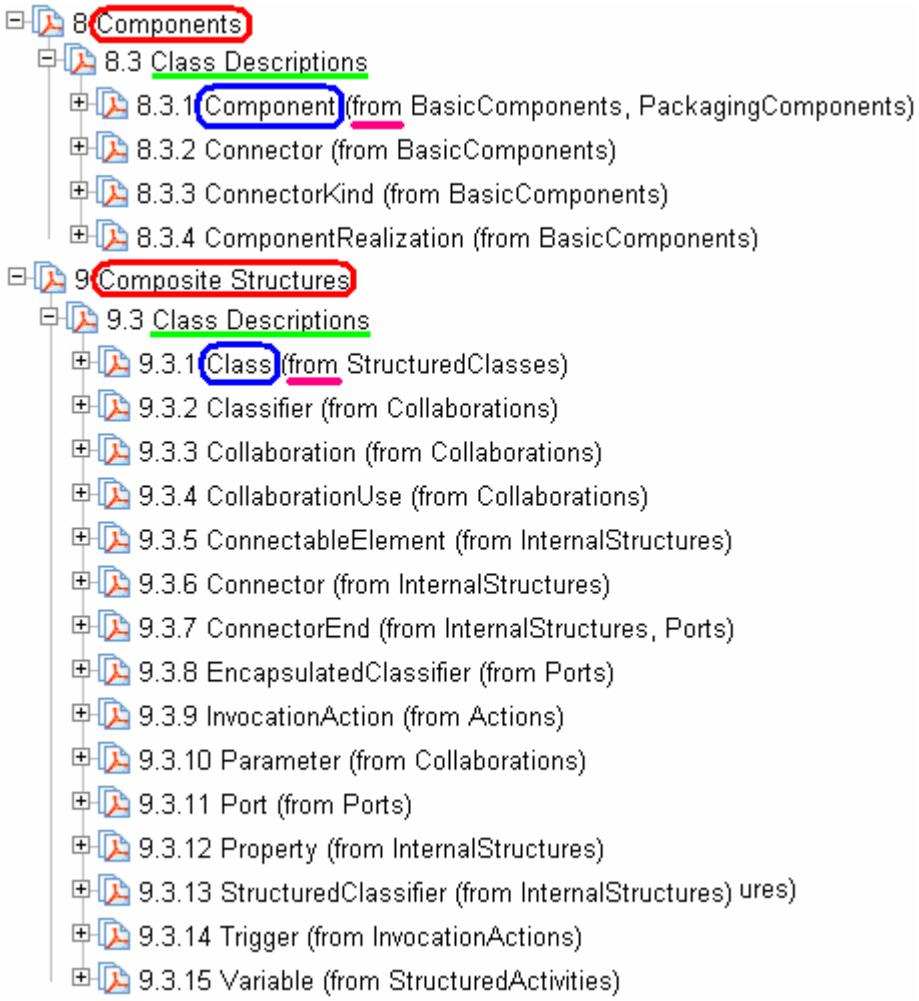


Figure 32. Headings of the UML specification (v2.1), containing UML concepts

6.1.1 UML class hierarchy extraction

In this subsection, we explain how to extract the UML class hierarchy from our XML document. The main clue that we used in our extraction code was the *"Class Descriptions"* which is a keyword string for the UML class hierarchy detection. For this reason, we applied the following XPath expression inside of the <Section> element (Figure 33, arrow-I) to take out all UML classes:

Child :: * [position() = 1] / starts-with(. , 'Class Descriptions')

This expression means select the first child of the <Section> element which its content starts with "Class Description" (Figure 33, arrow-II, <Name> element). By this logical expression we only selected sections that present some descriptions about UML classes. After that, we applied the following expression in order to define the title of a class set:

preceding-sibling :: * [last()]

This expression means select the preceding sibling of the <Section> element which is in the last place (Figure 33, arrow-III, <Name> element). As you can see in Figure 33, <Section 9.3> has three preceding-siblings: <Section 9.2>, <Section 9.1>, and <Name> which is the last one. Finally, we moved to the <Subsection> element (Figure 33 <Subsection 9.3.1>) and extracted contents of the <Name> element (e.g. "Class") and the <Reference> element (e.g. StructuredClasses). We also linked this UML class to its relevant hypertext page by the <Subsection> element's attribute (@Number):

concat(@Number , '.html') e.g. 9.3.1.html

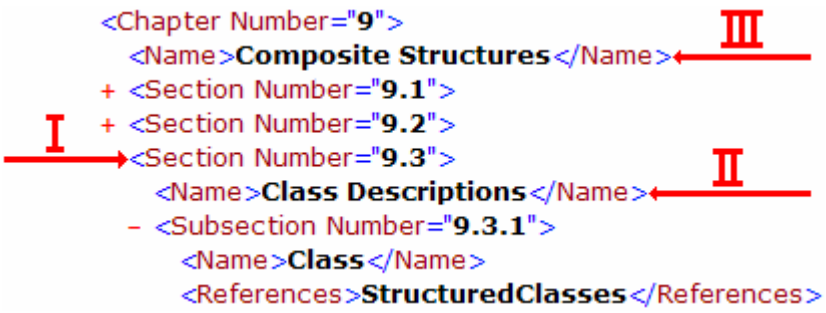
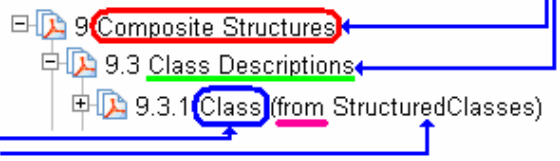


Figure 33. Part of tagging structures in the XML document

The XSLT code with respect to the above explanation for the UML class hierarchy extraction is presented here:

```
<xsl:for-each select="Part">
- <xsl:for-each select="Chapter">
  - <xsl:for-each select="Section">
    - <xsl:if test="child :: * [position()=1] / starts-with( . , 'Class Descriptions' )">
      - <h2>
        - <span>
            <xsl:value-of select="preceding-sibling :: * [last()]" />
          </span>
        </h2>
      - <xsl:for-each select="Subsection">
        - <a>
          - <xsl:choose>
            - <xsl:when test="substring(string(concat(@Number,'.html')), 1, 1) = '#'">
              - <xsl:attribute name="href">
                  <xsl:value-of select="concat(@Number,'.html')" />
                </xsl:attribute>
              </xsl:when>
            </xsl:choose>
          - <span>
              <xsl:value-of select="Name" />
            </span>
          </a>
        - <span style="color:fuchsia;">
            <xsl:value-of select="References" />
          </span>
        </xsl:for-each>
      </xsl:if>
    </xsl:for-each>
  </xsl:for-each>
</xsl:for-each>
```



In the next subsection, we present a portion of the XSLT code for the UML package hierarchy extraction.

6.1.2 UML package hierarchy extraction

To extract the UML packages, we used the <Reference> element which was inside of the <Subsection> element. The <Reference> element was created by the *"from"* as a keyword string during the logical structure extraction in Chapter 4. For instance, in order to extract all classes belong to the *"Actions"* package we applied the following expression inside of the <Subsection> element:

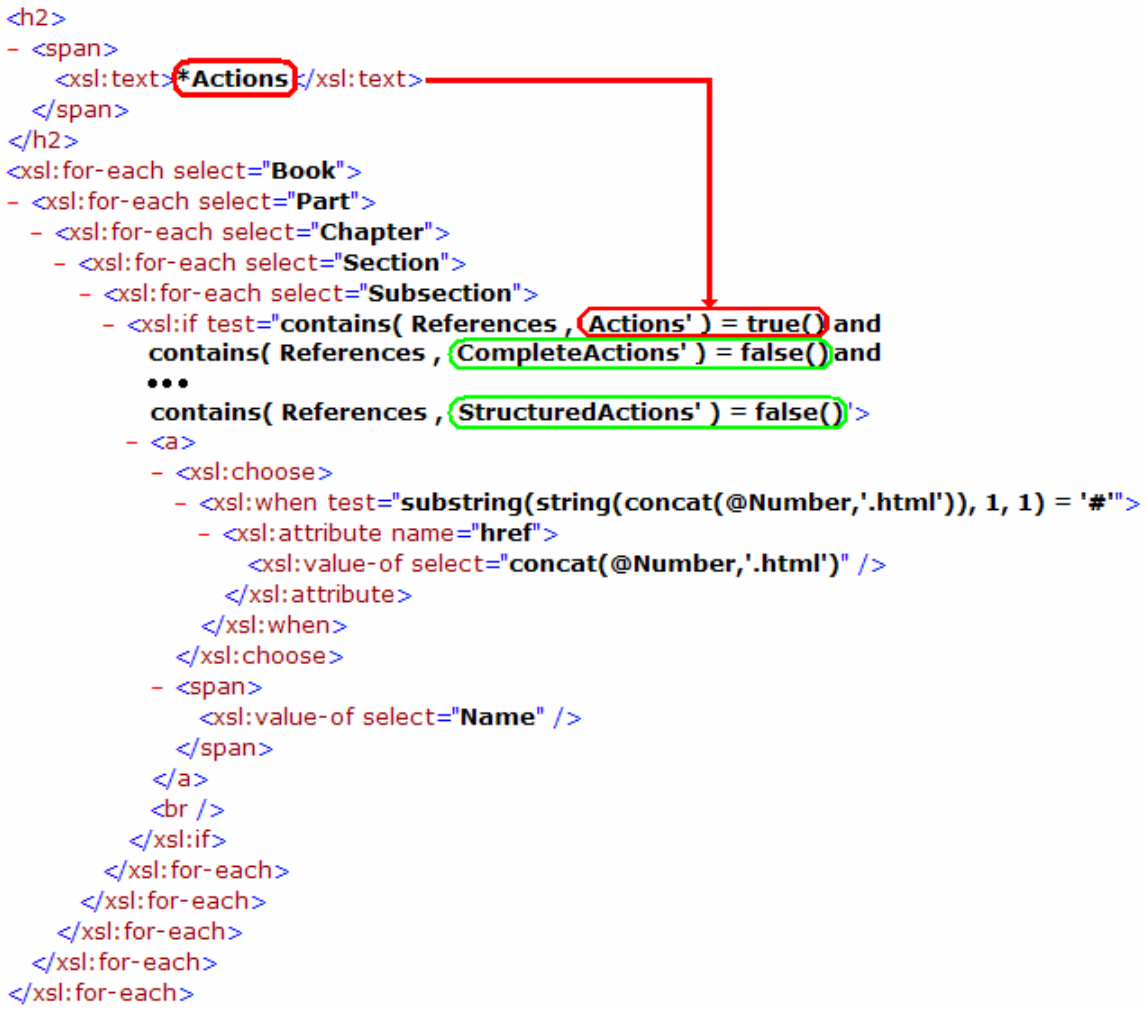
contain(Reference,'Actions') = true() and

contain(Reference,'CompleteActions') = false() and

…

contain(Reference,'StructuredActions') = false()

The *"contain(string-1 , string-2)"* function, returns true if string-1 contains string-2, otherwise it returns false. Therefore, the above XPath expressions mean select subsections whose <Reference> element contains "Actions" but are not "CompleteActions" or "StructuredActions", etc. As you can see, we excluded other packages whose names overlapped with "Actions" package (we had such a string similarity just for two packages: "Actions" and "StructuredActivities", for the rest of packages we simply used just one contain function). The other alternative was to use the *"starts-with(string-1,string-2)"* function which returns true if string-1 starts with string-2, otherwise it returns false. Finally, we extracted the <Name> element which carried the class names of the "Actions" package, and then linked each of these classes to its relevant hypertext page. A portion of the XSLT code with respect to the above description for the UML package hierarchy extraction is presented here:

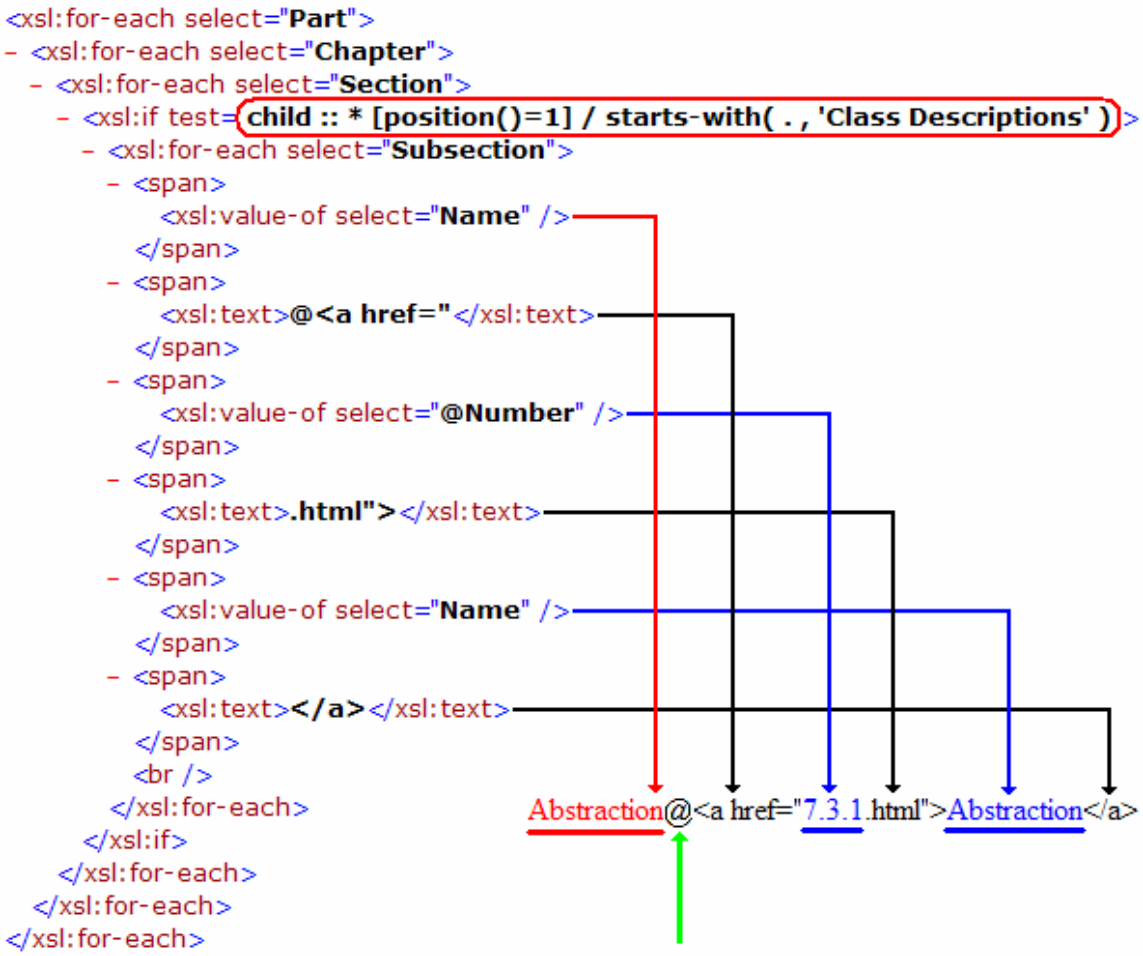


We generated a simple script that would execute the above code repeatedly, plugging in each of the package names where ‘Actions’ appears.

In the next section, the related approach for cross referencing all over the document is presented to improve the usability of the final user interface.

6.2 Cross referencing

To facilitate document browsing for end users, we developed some XSLT and Java code to generate hyperlinks for major document keywords all over the created user interface. These words consist of class names as well as package names. As we mentioned previously, since these keywords were among document headings each of them had an independent hypertext page (such as 285 class names) or anchor link (such as 36 package names) in the final user interface. We generated the following XSLT code to produce the related strings for UML class names, used in cross referencing algorithm:



This code selects sections that consist of UML class descriptions and then generates a string, which is made from the following six substrings, for every UML class:

Name + **@** + **Name** + ****

For instance, for *“Abstraction”* is a class name, so it generated the following string:

Abstraction@Abstraction

We applied a similar approach to generate related strings for package names, for example, the following string is generated for the *“Actions”* as a package name:

Actions@Actions

As you can see, we isolated keywords from their corresponding hyperlinks by the “@” character. We also listed all of these strings in a text file named “UniqueKeywords.txt”, and then applied the algorithm in the next page (CrossRef), implemented by Java, for cross referencing.

To generalize this cross referencing approach for other documents, one can simply extract all headers (which have an independent hypertext page or anchor link) with their corresponding hyperlinks in the “UniqueKeywords.txt” file, and then use the “CrossRef” procedure.

In the next chapter, the experimental results and the initial architecture of a proposed document engineering framework are illustrated.

```
Procedure CrossRef()
    UML // a folder consisting of 418 hypertext pages
    F // a hypertext file
    UniqueKeywords.txt // a text file consisting of the mentioned strings
    L // a line: e.g.: Abstraction@<a href="7.3.1.html">Abstraction</a>
    S1, S2 // string variables

    While (True) do
        F = Extract a new hypertext page from UML folder
        If (all 418 hypertext pages are extracted) Then
            Break this while loop
        Else
            While (end of the "UniqueKeywords.txt" file) do
                Get a new "L" from the text file // read a new line
                Split "L" into two strings from "@" character
                S1 = first part of the "L" // Abstraction
                S2 = second part of the "L" // corresponding links
                If (find S1 in F in one place or many places) Then
                    Replace All (S1, S2)
                    // replace all S1 strings with S2
                End If
            End while
        End If-Else
    End while
End procedure
```

7 Experimental Result and Architecture of the Framework

In this chapter, we present experimental results on various specifications and address usability of generated hypertext pages by comparing them to the original PDF documents. We also illustrate the initial architecture of a document engineering framework with the re-engineering capability of PDF based documents.

7.1 Re-engineering of various OMG specifications

As we mentioned, our first targeted document for processing was the UML Superstructure Specification. For further examination, we selected some other software specifications from Object Management Group (OMG) homepage with different number of pages and headings, the final result of this assessment is demonstrated in Table 9.

In this evaluation, for each of these documents we generated a separate hypertext page for its headings in addition to a page for the table of contents. To increase the usability of outcomes, we did cross referencing all over hypertext pages by detecting headings in each of these pages and connecting them to their corresponding entries. For instance, if the “AssociationClass” is among headings, certainly we have an independent hypertext page for that, and consequently hyperlinks for this phrase in all other pages where it appears. To avoid ambiguity, we filtered some phrases with common substrings

such as “Association” & “AssociationClass” and removed phrases which had many independent pages.

Table 9. Re-engineering of ten OMG specifications

Original OMG Spec.	# of PDF Pages	# of Headings	Headings Used in Cross-Ref	# of Tokens in Doc Body	# of Tokens in Headings	Data Analysis Results	# of HTML Pages
CORBA	1152	787	662	13179	702	15.1%	788
UML Sup.	771	418	202	10204	378	12.2%	421
CWM	576	550	471	6434	463	13.2%	551
MOF	292	61	52	6065	92	8.0%	62
UML Inf.	218	200	122	4329	176	9.3%	201
DAIS	188	135	102	3051	151	12.6%	136
XTCE	90	18	18	3075	26	2.6%	19
UMS	78	69	59	1937	94	22.7%	70
HUTN	74	88	83	2264	144	9.8%	89
WSDL	38	17	17	1106	36	16.3%	18

Furthermore, for each of these specifications we sorted document and heading tokens based on their frequency in two separate lists, defined positions of heading tokens among document tokens: $[P_1 \ldots P_N]$, and determined how important headings are:

M_P: Mean of $[P_1 \ldots P_N]$

N_{DT}: Total number of document tokens

Percentage = $(M_P * 100) / N_{DT}$

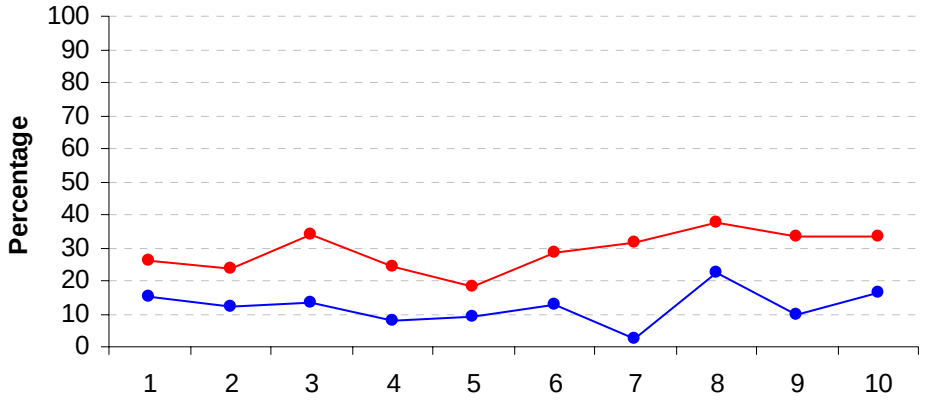


Figure 34. Headings are among the most frequent words in the entire document

Figure 34 shows that headings are among the most frequent words in the entire document. In the lower diagram we just evaluated headings which their number of occurrence were bigger that two but in the higher diagram we assessed entire headings. As we mentioned in previous chapters, this conclusion was our major motivation for:

- Extracting the logical structure of documents based on their headings
- Generating a separate hypertext page for each heading
- Detecting major concepts among document headings
- Cross-referencing by detection of headings in the entire hypertext pages

All mentioned experiments verify that our approach is repeatable for all documents and is not overly labour intensive. We just spent few minutes on each of these documents after transformation phase to deal with some mis-tagging problems in order to generate the initial well-formed XML format, the rest of our engineering procedures and software modules are totally automatic.

Our re-engineering method also has some limitations. Although most created software components apply a general approach but we can just process PDF based documents with bookmarks because of the transformation step. If we use an alternative solution to generate the preliminary XML file then we can process other formats such as DOC, RTF, etc. Moreover, the concept extraction module does not apply a generic approach; we used this component only for some specifications and would like to address this issue as one of our major future works. In the next part, we investigate the usability of our final results.

7.2 Assessment of generated HTML Interfaces

In this part, we address the usability of generated HTML interfaces.. We also compare the HTML interfaces generated by our framework with the HTML format of the specifications that can be provided directly by Adobe Acrobat Professional. This tool made a long hypertext page for each of those specifications along with anchors for headings at the top of each output.

After some evaluations, we detected the following benefits in our outcomes which did not exist in the original PDF formats or Adobe-Generated HTML formats:

- ✓ **Navigating:** To be able to define previous, current, and next locations and go forward and backward by sequential browsing of headings.
- ✓ **Scrolling:** It would be confusing to scroll a long hypertext page containing hundreds of topics, headings, and cross references. Moreover, page boundary in the PDF version makes it difficult to follow up related materials spread over various pages such as a big table or a programming code.
- ✓ **Learning:** Humans can better handle small amount of information presented in a single hypertext page which are related to a unique topic.
- ✓ **Monitoring:** To define a set of hypertext pages which have been downloaded several times and are probably more interesting (They also get high ranking in popular search engines such as Google, Yahoo, etc), and distinguish professional users from regular ones who browse the document randomly.
- ✓ **Downloading:** Specifications, conference proceedings, technical books, etc are not like novels. In the other word, we do not need to provide the whole document at once, the better idea is to provide the table of contents as a menu for users and then let them to select whatever they require. In a large scale assessment this issue decreases the Internet traffic to a considerable amount.
- ✓ **Printing:** To be able to print all materials related to a single topic or heading easily without having the entire document.
- ✓ **Referencing:** Cross-referencing among various specifications or documents which carry common concepts, definitions, headings, and materials. For example, connecting UML Superstructure Specification to the UML Infrastructure Specification wherever it is necessary.
- ✓ **Coloring:** To be able to use different colors to present various classes of information and highlight some significant parts of the document automatically.

To address deficiency of the final result, we should say that our final user interfaces are not totally clean like the original PDF formats since we had a few wrong tag-

recognitions during the conversion phase by Adobe Acrobat, for example, some lists have been recognized as table cells or many extracted pictures are not clean. Moreover, if the targeted document has small number of headings in comparison to the total number of document pages, such as MOF in Table 9, then generated hypertext pages would be very long and some of the mentioned usability will not be satisfied.

In the next part, the preliminary structure of our proposed document engineering framework is demonstrated.

7.3 Initial architecture of the proposed framework

As we went further by re-engineering of more software specifications and technical documents, we modified our software components and ended up with the initial architecture of a specific document engineering framework which takes a PDF document with bookmarks and generates corresponding XML and HTML versions of the document. We do believe that Adobe can use our approach in order to generate a more useful HTML version of a document. This architecture is demonstrated in Figure 35. We also would like to address engineering lessons we learned during this framework design:

- Generating a clean XML file from PDF images requires complicated features to recognize each document element correctly and deal with mis-tagging, page boundary and that sort of things.
- Remarkable role of latest technologies in engineering tasks, for instance, we applied XPath 2.0 vs. parsing packages which was a high level interaction close to human's language for concept extraction.
- Data analysis can facilitate the document engineering process, form a better understanding, and construct robust rules for such a processing.

During the entire development phase, we encountered with some low level challenges such as generating multiple hypertext pages by Saxon, detecting errors in XSTL programming, creating complicated XPath expressions, and so on. In the last chapter, final conclusions and proposed future work are presented.

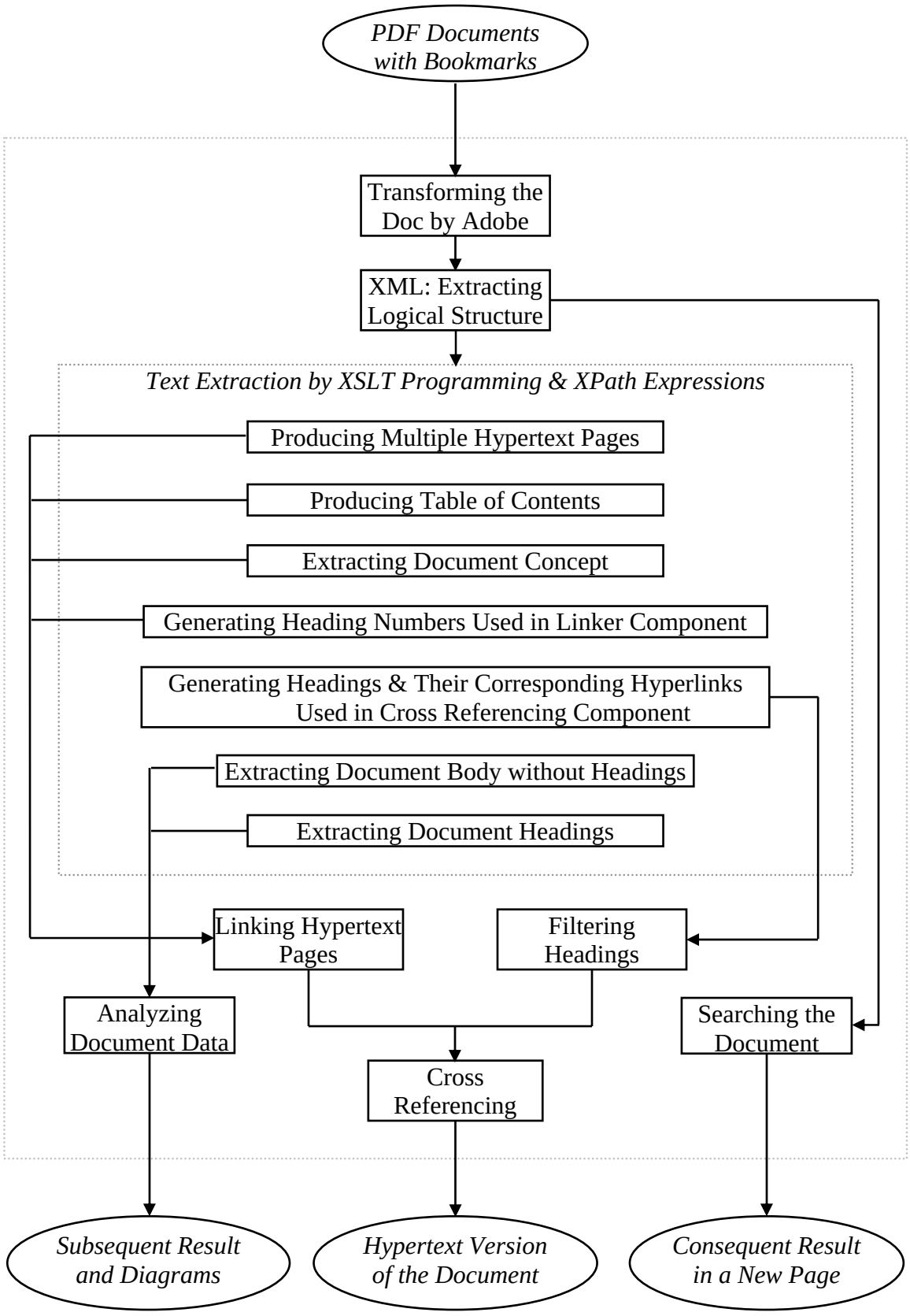


Figure 35. Initial architecture of the proposed document engineering framework

8 Conclusions and Future Work

In this thesis we have described an approach to taking a raw PDF version of a published specification, and converting this into a hypertext document that will be much more useful to end users. As an intermediate step, we generated a clean XML document with meaningful tags, and then generated from this a series of html documents constituting the final system.

The key contributions of the thesis are 1) to illustrate methods for re-engineering a PDF-based specification in a general way, and 2) to demonstrate how to make a more usable HTML version of a document so that end-users to have a better experience with software specifications. Our major goals were to make a complex document more usable by allowing navigation of both its structure, and also of semantics described by the document (e.g. in this case the UML class diagram relationships and package diagram relationships).

The first phase involved document analysis to better understand the structure of the document and establish a good infrastructure for our later objectives. We experimented with processing using a variety of tools and formats for transformation and extracted the logical structure of our document in the XML format; we also illustrated our reasons for such an extraction by some statistical analyses.

In the second phase, we generated multiple hypertext pages for end users to facilitate document browsing, navigating, and concept exploration. We applied the latest W3C technologies such as XSLT and XPath expressions and learned that although using these

technologies we can parse every XML document, it would be more usable if the created XML document has strong logical relationships among its elements and attributes similar to the XML document we produced (e.g. if the first child of a <Section> element contains the 'Class Descriptions' string then you can detect UML classes & packages in grandchildren of that <Section> element and so on, as you can see in Figure 33). That is because of the excellent logical capabilities that XPath and XQuery expressions provide for processing XML documents while other kinds of parsers may not supply such a valuable ability. As a consequence, we do believe that people in charge of software specifications can enrich their documents by embedding such a mentioned logical relationship and meaningful keywords inside headings, paragraphs, etc.

To conclude, since we considered the generality of every module that was generated, except the concept extractions which needs further research, we ended up with the idea of creating a new *"Document Engineering Framework"* for complex specifications and documents. Therefore, we are looking forward to do research in the following areas as future work:

- To produce the mentioned tool we need to extract the initial XML document independently from Adobe Acrobat which also generated some incorrectly nested opening and closing tags. We are also interested to extract such an XML document from other formats such as DOC, RTF, HTML, etc.

- To automate the concept extractions or at least create some Human Computer Interaction (HCI) features for the detection of the logical relationships among headings (as you can see in Figure 32) and creation of the corresponding XPath expressions or simplified logical expressions by humans. This should be done to extend the generality of the project to the concept extractions module for other software specifications and documents. We intend to focus on the hidden concepts found in the remaining natural language elements, and consequently perform knowledge acquisition from software specifications. For instance, we would like to capture lists of all bi-grams, tri-

grams and quad-grams with their frequency of occurrence. The most frequent of these, after excluding those that are simply stop words, will give us a sense of the terminology and concepts in the document as a whole and present a sense of the key topics in each chapter, section and subsection. We would like to do related-phrases analysis for relationships between the concepts identified in the terminological analysis. For example, patterns such as "X is a kind of Y", "X has a Y", etc.

- To apply our framework to numerous other software specifications and complex documents for exploring potential problems and research questions that may arise.

- To extend our current statistical and data analysis to hundreds of software specifications by an automatic document analyzer. We believe that leveraging mathematical analysis can facilitate the document engineering process, give us a better understanding of the document structures, and forming robust rules and regulations for such a processing.

- To do usability studies for improving our current methods and discovering users' demands. Only by such an investigation we can have a deep understanding of users' difficulties; moreover, this exploration can enhance the quality of the final user interfaces that we generate. For instance, we can add a Frame-like interface with a tree control on the left that shows the overall structure of a document in order to improve the navigability of the final hypertext pages or create features that allow a user to add values to a document and share it with others such as annotations, cross references, links to related documentations, and so on.

Using the above approaches, we can establish a better infrastructure to increase the understanding of complex specifications and make them more usable for end users.

References

[1] S. Klink, A. Dengel, and T. Kieninger, “Document structure analysis based on layout and textual Features”, *in Proceedings of International Workshop on Document Analysis systems*, Brazil, 2000, pp. 99-111.

[2] S. Mao, A. Rosenfeld and T. Kanungo, “Document structure analysis algorithms: A literature survey”, *in Proceedings of SPIE Electronic Imaging*, Vol. 5010, USA, 2003, pp. 197-207.

[3] K. Summers, “Automatic discovery of logical document structure”, *PhD Thesis, Cornell University*, 1998.

[4] S. Tsujimoto and H. Asada, “Understanding multi-articled documents”, *in Proceedings of 10th International Conference on Pattern Recognition*, USA, 1990, pp. 551–556.

[5] K. Lee, Y. Choy and S. Cho, “Logical structure analysis and generation for structured documents: A syntactic approach”, *IEEE Transactions on Knowledge and Data Engineering*, Vol. 15, No. 5, 2003, pp. 1277-1294.

[6] J. Liang, “Document structure analysis and performance evaluation”, *PhD Thesis, University of Washington*, USA, 1999.

[7] A. Conway, “Page grammars and page parsing: A syntactic approach to document layout recognition”, *in Proceedings of International Conference on Document Analysis and Recognition*, Japan, 1993, pp. 761–764.

[8] M. Aiello, C. Monz and L. Todoran, “Combining linguistic and spatial

information for document analysis", *in Proceedings of RIAO Content-Based Multimedia Information Access*, France, 2000, pp. 266-275.

[9] O. Altamura, F. Esposito and D. Malerba, "Transforming paper documents into XML format with WISDOM++", *International Journal on Document Analysis and Recognition*, vol. 4, 2001, pp. 2-17.

[10] Y. Ishitani, "Document transformation system from papers to XML data based on pivot XML document method", *in Proceedings of the Seventh International Conference on Document Analysis and Recognition*, Vol.1, Scotland, 2003, pp. 250-255.

[11] Y. Ishitani, "Logical structure analysis of document images based on emergent computation", *in Proceedings of the Fifth International Conference on Document Analysis and Recognition*, India, 1999, pp. 189-192.

[12] A. Dengel and F. Dubiel, "Computer understanding of document structure", *International Journal of Imaging Systems and Technology*, Vol. 7, 1996, pp. 271–278.

[13] D. Niyogi and S. N. Srihari, "Knowledge-based derivation of document logical structure," *in Proceedings of International Conference on Document Analysis and Recognition*, Canada, 1995, pp. 472–475.

[14] W. Cohen and L. Jensen, "A structured wrapper induction system for extracting information from semi-structured documents", *in Workshop on Adaptive Text Extraction and Mining*, USA, 2001.

[15] K.Nakagawa, A.Nomura, and M.Suzuki, "Extraction of logical structure from articles in mathematics", *3rd International Conference on Mathematical Knowledge Management*, Poland, 2004, pp. 276-289.

[16] H. Déjean and J. Meunier, "Structuring documents according to their table of

contents”, *in Proceedings of the ACM Symposium on Document Engineering*, United Kingdom, 2005, pp. 2-9.

[17] F. He, X. Ding and L. Peng, “Hierarchical logical structure extraction of book documents by analyzing tables of contents”, *Document Recognition and Retrieval XI, in Proceedings of SPIE-IS&T Electronic Imaging*, SPIE Vol. 5296, 2004, pp. 6-13.

[18] C. Lin, Y. Niwa, and S. Narita, “Logical structure analysis of book document images using contents information”, *in Proceedings of International Conference on Document Analysis and Recognition*, Germany, 1997, pp. 1048–1054.

[19] X. Lin and Y. Xiong, “Detection and analysis of table of contents based on content association”, *Journal on Document Analysis & Recognition*, 2005, pp. 132-143.

[20] S. Satoh, A. Takasu, and E. Katsura, “An automated generation of an electronic library based on document image understanding”, *in Proceedings of 3rd International Conference on Document Analysis and Recognition*, Vol. 1, Japan, 1995, pp. 163-166.

[21] F. Bourgeois, H. Emptoz and S. Bensafi, “Document understanding using probabilistic relaxation: Application on tables of contents”, *Sixth International Conference on Document Analysis and Recognition (ICDAR)*, USA, 2001, pp. 508-512.

[22] R. Crowder and Y. Sim, “An approach to extracting knowledge from legacy documents”, in Proceedings of ASME International Design Engineering Technical Conferences and Computers & Information Engineering Conference, USA, 2004, 7 pp.

[23] G. Carenini, R. T. Ng, and E. Zwart, “Extracting knowledge from evaluative text”, *in Proceedings of the 3rd International Conference on Knowledge Capture*, Canada, 2005, pp. 11-18.

[24] M. Henzinger and S. Lawrence, “Extracting knowledge from the WWW”, *in*

Proceedings of the National Academy of Science, Vol. 101, USA, 2004, pp. 5186-5191.

[25] J. Kreich, A. Luhn, and G. Maderlechner, "An experimental environment for model based document analysis", *in Proceedings of 1st International Conference on Document Analysis and Recognition*, France, 1991, pp. 50–58.

[26] H. Sakamoto, H. Arimura, and S. Arikawa, "Knowledge discovery from semi-structured texts", *Progress in Discovery Science*, Germany, 2002, pp. 586-599.

[27] M. Vargas-Vera, E. Motta, J. Domingue, S. Shum, and M. Lanzoni, "Knowledge extraction by using an ontology-based annotation tool", *in Proceedings of the Knowledge Markup and Semantic Annotation Workshop*, Canada, 2001, pp. 5-12.

[28] Y. Gil and V. Ratnakar, "IKRAFT: Interactive Knowledge Representation and Acquisition from Text", *in Proceedings of the 13th International Conference on Knowledge Engineering & Knowledge Management*, Spain, 2002, pp. 27-36.

[29] W. R. Cyre, "Knowledge Extractor: A tool for extracting knowledge from text", *in Proceedings of Fifth International Conference on Conceptual Structures (ICCS)*, USA, 1997, pp. 607-610.

[30] E. Qeli, J. Gllavata and B. Freisleben, "Customizable detection of changes for XML documents using XPath expressions", *in Proceedings of the ACM Symposium on Document Engineering*, Nethelands, 2006, pp. 88-90.

[31] E. Qeli and B. Freisleben, "Filtering XML documents using XPath expressions and Aspect-Oriented Programming", *in Proceedings of the ACM Symposium on Document Engineering*, Netherlands, 2006, pp. 85-87.

[32] P. Geneves and N. Layaida, "Comparing XPath Expressions", *in Proceedings of the ACM symposium on Document engineering*, Netherlands, 2006, pp. 65-74.

[33] P. Geneves and N. Layaida, “A System for the static analysis of XPath”, *ACM Transactions on Information Systems*, Vol. 24, No. 4, 2006, pp. 475–502.

[34] M. Marx, “Conditional XPath: The first order complete XPath dialect”, *in Proceedings of the twenty-third ACM Symposium on Principles of Database Systems*, France, 2004, pp. 13-22.

[35] Extensible Markup Language (XML): Version 1.1 (Second Edition), W3C Recommendation, 16 August 2006, http://www.w3.org/TR/xml11/

[36] XML Schema Part: 0-Primer, 1-Structures, and 2-Datatypes (Second Edition), W3C Recommendation, 28 October 2004, http://www.w3.org/TR/xmlschema-1/

[37] XSL Transformations (XSLT): Version 2.0, W3C Recommendation, 23 January 2007, http://www.w3.org/TR/xslt20/

[38] XML Path Language (XPath): Version 2.0: W3C Recommendation, 23 January 2007, http://www.w3.org/TR/xpath20/

[39] XML Query Language (XQuery): Version 1.0, W3C Recommendation, 23 January 2007, http://www.w3c.org/TR/xquery/

[40] ACM Symposium on Doc Engineering, http://www.documentengineering.org/

[41] Stylus Studio® Corporate: http://www.stylusstudio.com/

[42] Altova Company: http://www.altova.com/

[43] XSLT and XQuery Processing: http://www.saxonica.com/

Appendix A: List of Acronyms

AOP	Aspect Oriented Programming
COTS	Commercial Off The Shelf
CSS	Cascading Style Sheets
DOM	Document Object Model
DTD	Document Type Definition
FOL	First Order Logic
HTML	Hyper Text Markup Language
IKRAFT	Interactive Knowledge Representation and Acquisition From Text
NLP	Natural Language Processing
OCR	Optical Character Recognition
PDF	Portable Document Format
RTF	Rich Text Format
SODA	System for Office Document Analysis
SQL	Structured Query Language
ToC	Table of Contents
UI	User Interface
UML	Unified Modelling Language
W3C	World Wide Web Consortium
WISDOM	Wise System for Document Management
XML	Extensible Markup Language
XPath	XML Path Language
XQuery	XML Query Language
XSD	XML Schema Definition
XSL	Extensible Stylesheet Language
XSLT	Extensible Stylesheet Language Transformations
HCI	Human Computer Interaction

Appendix B: Logical Structure Extractor

```
public static void LogicalStructureExtraction(String[] inputStrings)
{
      boolean startAnalysis=false;
      int indexOfLinkTarget; // "LinkTarger" is a key word
      int indexOfFirstHeader;
      int indexOfBeginingOfHeaders;
      int indexOfEndOfHeaders;
      int stackPointer=-1;
      int[] tempStack = new int[1000];
      String lineSt="";
      String tempStHeaders="";
      String tempHeaderNumberFive="";
      String [] analyzedHeader={"","","",""};
try{ // Read the INPUT FILE
FileReader inputFileReader = new FileReader(inputStrings[0]);
BufferedReader inputFileBuffer = new BufferedReader(inputFileReader);
while(true){ // Until the end of the input file
      lineSt=inputFileBuffer.readLine();
      if(lineSt==null){ // End of the input file
            break;
      }
      indexOfLinkTarget=lineSt.indexOf("LinkTarget");
      if(indexOfLinkTarget!=-1){
      if(startAnalysis!=true){
      indexOfFirstHeader=lineSt.indexOf(inputStrings[1]);
      if(indexOfFirstHeader!=-1){
      startAnalysis=true;
      System.out.println("\n"+inputStrings[1]);
      analyzedHeader=headersAnalysis(inputStrings[1]);
      stackPointer ++;
      tempStack[stackPointer]=Integer.parseInt(analyzedHeader[0]);
      writeUMLFile("<Chapter
      Number="+(char)34+analyzedHeader[1]+(char)34+">"+
      (char)10+"<Name>"+analyzedHeader[2]+"</Name>"+(char)10);
      }
      }
      else{
      indexOfLinkTarget=indexOfLinkTarget+10;
      indexOfBeginingOfHeaders=lineSt.indexOf(">",indexOfLinkTarget);
      indexOfEndOfHeaders=lineSt.indexOf("<",indexOfBeginingOfHeaders);
      indexOfBeginingOfHeaders ++;
      indexOfEndOfHeaders --;
      tempStHeaders=lineSt.substring(indexOfBeginingOfHeaders,indexOfEn
      dOfHeaders);
      if(tempStHeaders.compareTo("Index")==0){ // End of the processing
            break;
      }
      System.out.println("\n"+tempStHeaders);
      analyzedHeader=headersAnalysis(tempStHeaders);
      while
```

```
((Integer.parseInt(analyzedHeader[0])<=tempStack[stackPointer])||
(Integer.parseInt(analyzedHeader[0])==6 &&
tempStack[stackPointer]==3))
{
if(tempStack[stackPointer]==1){
writeUMLFile("</Part>"+(char)10);
}
else if(tempStack[stackPointer]==2){
writeUMLFile("</Chapter>"+(char)10);
}
else if(tempStack[stackPointer]==3){
writeUMLFile("</Section>"+(char)10);
}
else if(tempStack[stackPointer]==4){
writeUMLFile("</Subsection>"+(char)10);
}
else if(tempStack[stackPointer]==5){
tempHeaderNumberFive=tempHeaderNumberFive.replace((char)32,(char)
95);
writeUMLFile("</"+tempHeaderNumberFive+">"+(char)10);
}
else if(tempStack[stackPointer]==6){
writeUMLFile("</Annex>"+(char)10);
}
stackPointer --;
if(stackPointer==-1){ // Stack is empty
      break;
}
}
stackPointer ++;
tempStack[stackPointer]=Integer.parseInt(analyzedHeader[0]);
if(Integer.parseInt(analyzedHeader[0])==1){
writeUMLFile("<Part Number="+(char)34+analyzedHeader[2]
+(char)34+">"+(char)10+"<Name>"+analyzedHeader[3]+"</Name>"
+(char)10);
}
else if(Integer.parseInt(analyzedHeader[0])==2){
writeUMLFile("<Chapter Number="+(char)34+analyzedHeader[1]
+(char)34+">"+(char)10+"<Name>"+analyzedHeader[2]+"</Name>"
+(char)10);
}
else if(Integer.parseInt(analyzedHeader[0])==3){
writeUMLFile("<Section Number="+(char)34+analyzedHeader[1]
+(char)34+">"+(char)10+"<Name>"+analyzedHeader[2]+"</Name>"
+(char)10);
}
else if(Integer.parseInt(analyzedHeader[0])==4){
writeUMLFile("<Subsection Number="+(char)34+analyzedHeader[1]
+(char)34+">"+(char)10+"<Name>"+analyzedHeader[2]+"</Name>"
+(char)10+"<References>"+analyzedHeader[3]+"</References>"
+(char)10);
}
else if(Integer.parseInt(analyzedHeader[0])==5){
analyzedHeader[1]=analyzedHeader[1].replace((char)32,(char)95);
writeUMLFile("<"+analyzedHeader[1]+">"+(char)10);
tempHeaderNumberFive=analyzedHeader[1];
```

```
        }
        else if(Integer.parseInt(analyzedHeader[0])==6){
        writeUMLFile("<Annex Number="+(char)34+analyzedHeader[2]
        +(char)34+">"+(char)10+"<Name>"+analyzedHeader[3]+"</Name>"
        +(char)10);
        }
        }
        }
        else{
        if(startAnalysis==true &&
        lineSt.compareTo("<Part>")!=0&&lineSt.compareTo("</Part>")!=0 &&
        lineSt.compareTo("<Sect>")!=0&&lineSt.compareTo("</Sect>")!=0 &&
        lineSt.compareTo("<Div>")!=0 && lineSt.compareTo("</Div>")!=0 &&
        lineSt.compareTo("")!=0 &&
        lineSt.compareTo("</TaggedPDF-doc>")!=0 &&
        lineSt.compareTo("<P>UML Superstructure Specification</P>")!=0){
        writeUMLFile(lineSt+(char)10);
        }
        }
} // End of the WHILE loop (End of the file)
while(stackPointer!=-1){ // We have to empty the "stack"
if(tempStack[stackPointer]==1){
writeUMLFile("</Part>"+(char)10);
}
else if(tempStack[stackPointer]==2){
writeUMLFile("</Chapter>"+(char)10);
}
else if(tempStack[stackPointer]==3){
writeUMLFile("</Section>"+(char)10);
}
else if(tempStack[stackPointer]==4){
writeUMLFile("</Subsection>"+(char)10);
}
else if(tempStack[stackPointer]==5){
tempHeaderNumberFive=tempHeaderNumberFive.replace((char)32,(char)95);
writeUMLFile("</"+tempHeaderNumberFive+">"+(char)10);
}
else if(tempStack[stackPointer]==6){
writeUMLFile("</Annex>"+(char)10);
}
stackPointer --;
}
inputFileReader.close();
} // End of the TRY
catch(FileNotFoundException e){
System.out.println("Unable to Open INPUT File");
}
catch(IOException e){
System.out.println("Unable to Close INPUT File");
}
}
```

Appendix C: First Java Parser

```
public static void readUMLFile(String f_Name,int numChar)
{
	int fileChar;
	int tempInt=0;
	int stackPointer=0;
	boolean flagSpaceChar=false;
	String firstTags="<html>";
	String lastTags="</html>";
	String tempStDiv="";
	String tempStHeader="";
	int [] tempStack = new int[100];

writeUMLFile(firstTags+(char)10);
try{ // Read the INPUT FILE
	FileReader inputReader = new FileReader(f_Name);
	fileChar=inputReader.read();
	while(fileChar!=-1){ // It is not end of the FILE
	if(fileChar==60){ // 60 = "<"
	for(int i=0;i<4;i++){
	fileChar=inputReader.read();
	tempStDiv=tempStDiv+(char)fileChar;
	}
	if(tempStDiv.equals("div ")){
	while(true){
	fileChar=inputReader.read();
	if (fileChar==62){ // 62 = ">"
	fileChar=inputReader.read();
	if(fileChar >= 49 && fileChar <= 57){ // 1 .. 9
	do{
	if(fileChar!=13 && fileChar!=10 && fileChar!=555){
	tempStHeader=tempStHeader+(char)fileChar;
	if(fileChar==32){ // 32 = Space
	flagSpaceChar=true; // I saw the first SPACE
	}
	}
	fileChar=inputReader.read();
	if(flagSpaceChar==true && fileChar==32){ //I saw the second SPACE
	fileChar=555; // To skip first IF in the DO loop
	}
	else // (flagSpaceChar == False || fileChar!=32)
	if(flagSpaceChar==true){ // Just one SPACE or last SPACE
	flagSpaceChar=false;
	}
	}while(fileChar!=60); // 60 = "<"
	tempInt=headersAnalysis(tempStHeader);
	tempStack[stackPointer]=tempInt;
	stackPointer++;
	tempStHeader="";
	break; // Exit from WHILE(true)
	} // End of the IF ("1 .. 9")
```

```
        } // End of the IF ("62")
        } // End of the WHILE(true)
        }else // End of the IF ("div ")
        if(tempStDiv.equals("/div")){
        stackPointer --;
        if(tempStack[stackPointer]==0){
        writeUMLFile("</Chapter>"+(char)10);
        }
        else if (tempStack[stackPointer]==1){
        writeUMLFile("</Section>"+(char)10);
        }
        else{ // periodCounter==2 like 1.2.3
        writeUMLFile("</Subsection>"+(char)10);
        }
        } // End of the IF ("/div")
        tempStDiv="";
        } // End of the IF ("<")
        fileChar = inputReader.read();
        } // End of the WHILE loop (End of the file)
        inputReader.close();
} // End of the TRY
catch(FileNotFoundException e){
System.out.println("Unable to Open INPUT File");
}
catch(IOException e){
System.out.println("Unable to Close INPUT File");
}
writeUMLFile(lastTags);
}
// ************************************************************
public static int headersAnalysis(String tempStHeader)
{
        int lengthHeader;
        int periodCounter=0;
        int firstIndex=0, secondIndex=0;
        char tempChar=' ';
        String firstPartHeader="";
        String secondPartHeader="";
        String thirdPartHeader="";
        lengthHeader=tempStHeader.length();
        firstIndex=tempStHeader.indexOf(" ");
        for(int i=0;i<firstIndex;i++){
        tempChar=tempStHeader.charAt(i);
        if(tempChar==46){ // Period "."
        periodCounter ++;
        }
        firstPartHeader=firstPartHeader+tempChar;
        }
        firstIndex ++; // To skip the "space" for the next while loop
        while(firstIndex<lengthHeader){
        tempChar=tempStHeader.charAt(firstIndex);
        if(tempChar==40){
        break;
        }
        secondPartHeader=secondPartHeader+tempChar;
        firstIndex ++;
```

```
        }
        secondIndex=tempStHeader.indexOf(" (");
        if(secondIndex!=-1){ // Found the " ("
        secondIndex=secondIndex+7; // To skip the " (from "
        while(secondIndex<(lengthHeader-1)){
        tempChar=tempStHeader.charAt(secondIndex);
        thirdPartHeader=thirdPartHeader+tempChar;
        secondIndex ++;
        }
        }
        if(periodCounter==0){
        writeUMLFile("<Chapter Number="+(char)34+firstPartHeader+
        (char)34+">"+(char)10+"<Name>"+secondPartHeader+"</Name>"
        +(char)10);
        }
        else if (periodCounter==1){
        writeUMLFile("<Section Number="+(char)34+firstPartHeader+
        (char)34+">"+(char)10+"<Name>"+secondPartHeader+"</Name>"
        +(char)10);
        }
        else{ // periodCounter==2 like 1.2.3
        writeUMLFile("<Subsection Number="+(char)34+firstPartHeader+
        (char)34+">"+(char)10+"<Name>"+secondPartHeader+"</Name>"+
        (char)10+"<References>"+thirdPartHeader+"</References>"
        +(char)10);
        }
        return periodCounter;
}
```

Appendix D: XSLT Codes

Hyperlinks: Automatic association of dynamic hyperlinks to static bookmarks

```
- <xsl:for-each select="Chapter">
  - <xsl:if test="compare( Name , 'Conformance' ) = 0">
    - <xsl:for-each select="Section">
      - <h3 align="justify">
        - <a>
          - <xsl:choose>
            - <xsl:when test="substring(string(concat('#',@Number)), 1, 1) = '#'">
              - <xsl:attribute name="href">
                  <xsl:value-of select="concat('#',@Number)" />
                </xsl:attribute>
              </xsl:when>
            - <xsl:otherwise>
              - <xsl:attribute name="href">
                - <xsl:if test="substring(string(concat('#',@Number)), 2, 1) = ':'">
                    <xsl:text>file:///</xsl:text>
                  </xsl:if>
                  <xsl:value-of select="translate(string(concat('#',@Number)), '\', '/')" />
                </xsl:attribute>
              </xsl:otherwise>
            </xsl:choose>
          - <xsl:for-each select="@Number">
            - <span style="color:blue; font-weight:bold;">
                <xsl:value-of select="string(.)" />
              </span>
            </xsl:for-each>
          - <xsl:for-each select="Name">
            - <span style="color:blue; font-weight:bold;">
                <xsl:apply-templates />
              </span>
            </xsl:for-each>
          </a>
        </h3>
      </xsl:for-each>
    </xsl:if>
  </xsl:for-each>
```

Tables: Dynamic table structure

```
- <xsl:template match="Table">
  - <xsl:for-each select="Caption">
    - <xsl:for-each select="P">
        <xsl:apply-templates />
      </xsl:for-each>
    </xsl:for-each>
  - <table border="1">
    - <thead>
      - <tr>
        - <td style="border-bottom-color:black; border-color:black; color:black;">
          - <xsl:for-each select="TR">
            - <xsl:for-each select="TH">
              - <xsl:if test="position() = 1">
                  <xsl:apply-templates />
                </xsl:if>
              </xsl:for-each>
            </xsl:for-each>
          </td>
        </tr>
      </thead>
    - <tbody>
      - <xsl:for-each select="TR">
        - <tr>
          - <td style="border-bottom-color:black; border-color:black;" width="124">
            - <xsl:for-each select="TD">
              - <xsl:if test="position() = 1">
                  <xsl:apply-templates />
                </xsl:if>
              </xsl:for-each>
            </td>
          </tr>
        </xsl:for-each>
      </tbody>
    </table>
  </xsl:template>
```

Simple and Nested Lists: List Structure

```
- <xsl:template match="L">
  - <xsl:for-each select="LI">
    - <ul>
      - <p style="text-align:justify;">
        - <xsl:for-each select="LI_Label">
          - <span style="color:navy; text-align:justify;">
              <xsl:apply-templates />
            </span>
          </xsl:for-each>
        - <xsl:for-each select="LI_Title">
          - <span style="color:navy; text-align:justify;">
              <xsl:apply-templates />
            </span>
          </xsl:for-each>
        </p>
      </ul>
    </xsl:for-each>
  - <xsl:for-each select="L">
    - <ul>
      - <p align="justify">
        - <span style="color:maroon;">
            <xsl:value-of select="child :: * [position()=1]" />
          </span>
        </p>
      - <ul>
        - <p align="justify">
          - <span style="color:olive;">
              <xsl:value-of select="child :: * [position()=2]" />
            </span>
          </p>
        </ul>
      </ul>
    </xsl:for-each>
  </xsl:template>
```