A Formatted Document Model (FDM) for Content Management Systems with Multiple Document Representations and an Implementation of an XML Converter

Project Work

Submitted by:
Sheng Wei
Matriculation Number: 16508
Information and Communication Systems
Master Program
sheng.wei@tuhh.de

Supervisor:
Prof. Dr. Florian Matthes
Arbeitsbereich Softwaresysteme

Technical University Hamburg Harburg

Germany, July 2002

I declare that:

This work has been prepared and done completely by myself.
All literal or content-related quotations from other sources are pointed out in paper.

Sheng Wei

Hamburg, Germany
2002.07.06

Table of Contents

Preface 6

Software and Versions 6
Organization 7
Conventions Used in This Report 9
Important Note 9
Acknowledgements 10

Chapter 1 11

Design Objectives 11

1.1 Market Demand and Challenges 11
1.2 XML Is the Choice. 12
1.3 XML Advantages 13
1.4 XML Converters 13
1.5 Why Not Use them, But Develop My New One 14
1.6 Design Objectives 15
1.7 What is Next 16

Chapter 2 18

Document Formatting Information 18

2.1 Whether We Need Formats 18
2.2 Formatting Information in Microsoft Word 19
2.3 Fonts and Styles 21
2.3 Implementation Strategies to Formats 23
2.5 My Consideration to Support Font 25
2.6 Formatting Table File 26

Chapter 3 28

Architecture 28

3.1 Task Description 28
3.2 Converter System Architecture 30

Chapter 4 33

XML Target File Consideration 33

4.1 XML DTD and Schema 33
4.2 DocBook DTD 34

4.4 My Extension to DocBook DTD....35
4.4 How to Process XML....37
4.5 How to Create XML Output Representation....38
Chapter 5....40
Formatted Document Model....40
5.1 Reference Foundations....40
5.2 Detail Description....41
Chapter 6....47
Implementation of an FDM Word XML Converter....47
6.1 Contents in the Package....47
6.2 Use Case Diagram....49
6.3 How Converter System Works?....51
6.4 Detail Usages Explanation....51
6.6 Design Patterns....53
6.7 Step by Step....57
6.7.1 Check Source Document....57
6.7.2 Parse Source Document....57
6.7.3 Export Format Library to External File....58
6.7.4 Verify Formatting Table File....59
6.7.5 Import Formatting Table File....59
6.7.6 Update XML File....59
6.7.7 Check the Result XML File....60
6.7.8 Before Transforming to XHTML....60
6.7.9 Transform XML into XHTML....60
Chapter 7....62
Programming Notes to Using Microsoft Word Object Model..62
7.1 Document Types....62
7.2 RTF Format....62
7.3 Type of Tools....63
7.4 Library....63
7.5 Some Important Objects in Word Object Model....64
7.5.1 Application Object....64
7.5.2 Document Object....64
7.5.3 Range Object....64
7.5.4 Find Object....65
6.5.5 Selection Object....65
7.6 Different Mechanisms Parsing Word Documents....65
7.6.1 Style....65

7.6.2 Structured-Based Elements 66
Chapter 8 72
Discussion and Open Issues 72
7.1 Current Implementation Limitations 72
Appendix A 73
API Reference 73
Package com.model 73
Package com.model.util 73
Package com.model.FDM 74
Bibliography 76

Preface

The task of this project is to develop an ***XML Converter*** that can transform Microsoft Word documents into a ***Java Intermediate Representation***, which is named ***Formatted Document Model (FDM)***, which contains both content information and formatting information inside the documents, which is suitable for algorithmic processing of these formatted documents in architecture level independent of the peculiarities of the processing APIs, such as Microsoft Word API, and which will be mapped into the ***XML target*** file. This XML Converter and FDM model library can be integrated in ***Content Management Systems (CMS)*** and can be extended to support multiple document representations.

The project takes into account standardized XML-based document standards like the ***DocBook DTD*** [WuMu99].

In the end of this paper, I will also discuss other different techniques that can be put to use in this task and some open issues and suggestions that can be considered in further versions of implementation if necessary.

Software and Versions

Keeping up with the latest technologies is always a challenge, particularly when using Java and XML-related tools. The set of tools listed in following table is sufficient to run the XML Converter implementation program.

Tool	URL	Description
JDK 1.2.x	http://java.sun.com	Any Java 2 Standard Edition SDK
Bridge2Java	http://www.alphaworks.ibm.com/tech/bridge2java	A bridging tool for Java to call COM Automation libraries, such as Microsoft Word COM library

JDOM 8	http://jdom.org	A easy to use open source Java API for XML
JAXP 1.1	http://java.sun.com/xml	A Java API for XML Processing from Sun
Crimson	Included with JAXP 1.1 http://xml.apache.org	An Java-based XML parser from Apache, used by JAXP
Xerces 2	http://xml.apache.org	An Java-based XML parser from Apache
Xalan 2	http://xml.apache.org	An Java-based XSLT processor from Apache

The following set of tools is recommended for helping.

Tool	URL	Description
JBuilder 7	http://www.borland.com	A Java IDE programming tool
XMLSpy 4.4	http://www.xmlspy.com	An XML Tools Suite including XML Editor. The XML editor can be integrated with latest Xalan XSLT processor, MSXML XSLT processor and Apache FOP XSLT processor.
XML Authority 1.2	http://www.extensibility.com	A very good XML development tool for DTD/Schema, with a visual way to show relationships between elements.
Together Control Center 6	http://www.togethersoft.com	A Java-based UML modeling tool
Ant 1.5	http://jakarta.apache.org	A Java-based building tool from Apache

Organization

This book consists eight chapters and one appendix, as following:

Chapter 1, Design Objectives

This chapter firstly introduces the requirements of the storage-formats and exchange-formats for current *Content Management Systems (CMS)* and *Digital Library Systems*. And then I will explain why XML can fulfill such requirements and what are the advantages of XML. In this project I want to implement a new XML Converter. So, here I will explain here why I need a new Converter,

although a lot of commercial products of XML Converter are available. In the end of this chapter, I will summarize the detail design objectives of my XML Converter implementation.

Chapter 2, Document Formatting Information

This chapter discusses which types of formatting information in document need to be considered when we need to store contents of different types of documents in *CMSs*, especially when we are talking about specific formatting information in Microsoft Word documents. I will put emphasis on discussing two format types - font and style. I will also illustrate my considerations and processing strategies about how to extract the formatting information from documents such as Microsoft Word in my implementation.

Chapter 3, Software Architecture

This chapter summarizes the viewpoints of former two chapters and lists detail sub-tasks of this project. Then I will propose the software architecture for the Converter System that can fit in with the desired goals.

Chapter 4, XML Target File Consideration

This chapter explains how to define the XML target file as the results of Converter System and how XML target file maps with the intermediate document model with formatting information. Which XML standards I need to take into account? What are the key requirements? What is my extension? In the end, a brief introduction related to XML programming in Java will be given.

Chapter 5, Formatted Document Model

This chapter explains the detail about the intermediate document model, named *Formatted Document Model (FDM)* of Converter System. At the beginning of this chapter, I will first discuss the two reference models used when I create FDM. They are Microsoft Word object model and DocBook object model (with DTD). Then I will describe different important parts of FDM and use class diagrams to show the relationships between different elements.

Chapter 6, Implementation of an FDM Word XML Converter

This chapter explains how the FDM Word XML Converter implementation works. First, I will introduce the contents and file structures in the package. Then, I will describe the use case scenarios and collaborations between different roles to help readers to understand how the whole system runs. Then I will explain the commands I use and the algorithms of parsing document. In the end, I will show step by step how to run the program. For detail about the process, please take a look at the video demo of the implementation to get more impression.

Chapter 7, Programming Notes to Using Microsoft Word Object Model

This chapter explains some important points when writing programs in Java to access Microsoft Word Object Model COM library. Since it seems that there are too much need to say and it is hard to decide how detail and what level I should mention, here is only a little my personal experience, and not all. This chapter gives an introduction to the programming and some possible difficulties programmer will meet.

Chapter 8, Discussion and Open Issues

This chapter lists some unsolved problems as open issues in my limited-time implementation and some other suggestions for further implementation version.

Appendix A, API Reference

This appendix lists all the classes and packages used in XML Converter implementation.

Conventions Used in This Report

Italic is used for:

- Pathnames, filenames, and program names
- New terms where they are defined
- Internet addresses, such as domain names and URLs
- Keywords, proper nouns, important names

Bold is used for:

- Emphases

Color is used for:

- Special effects, meanings

Important Note

At the beginning, I want to write a simple separate tutorial about this XML Converter implementation demo program and distribute it with the package. And actually I have done the main part of tutorial. The tutorial is written completely in XML and uses my Tutorial Builder automatically generate HTML version and PDF version tutorials by using XSLT (*Apache Xalan* and *Apache FOP*, very interesting!). But finally, I found that I am in a dilemma and I feel that readers are not easy to understand what I mean without my explanation in the former several chapters, since I have a lot of assumptions before implementation. So, I decide to move the whole tutorial in Chapter 6 of this paper and will distribute this paper with the package. **Anyone who is experienced and has enough knowledge can skip all the other chapters and only read this chapter directly**. Thanks! For questions and further information, please feel free to contact and ask me.

Acknowledgements

My special thanks go to Prof. Dr. Florian Matthes for his intensive guidance to this work. In addition, I would like to thank Dipl. Inform. Rainer Mueller, Dipl. Inform. Patrick Hupe for providing their helps in my project.

Chapter 1

Design Objectives

This chapter firstly introduces the requirements of the storage-formats and exchange-formats for current *Content Management Systems (CMS)* and *Digital Library Systems*. And then I will explain why XML can fulfill such requirements and what are the advantages of XML in such scenario. In this project I want to implement a new XML Converter. So, I will explain here why I need a new one, although a lot of commercial products of XML Converters are available in market. In the end of this chapter, I will summarize the detail design objectives of my XML Converter implementation.

1.1 Market Demand and Challenges

In *Content Management Systems* or *Digital Library Systems*, a large deposit of documents needs to be stored and, more importantly, need to be deeply analyzed or searched or summarized. But currently all these digital version documents are using different document format types like *Microsoft Word, Microsoft PowerPoint, QuarkXpress, Adobe PDF, Adobe PostScript* and etc. Such format types are usually created for human reading and accessing, not for machine reading, like automatically parsed by software systems. If people don't know the internal mechanisms or rules of how file constructs in these formats, and if they see a mess of strange characters if they open the file as simple text file, we can say that these format types are proprietary binary-based document format types, which normally must only be opened or authored by specific word processors or parsed by specific programming APIs.

Two examples of such programming APIs:

- Microsoft Word and PowerPoint applications provide OLE Automation library to developers and give them the ability to access and parse respective documents, by developing their own systems with different programming languages, such as VB, Delphi, C++, Java, and so on.
- Adobe Acrobat provides proprietary ActiveX library for creating and parsing PDF documents.

What is more, by using proprietary document format types, of course we can define a rich set of elements to represent different content. But information is usually stored in some unstructured, binary-based formats. This may disturb users from extracting useful information from document with these formats directly, not to say extracting contents from a large deposit of documents.

In the other way, if we use simple text file (*.txt) to store contents, we are easy to open the file with Notepad and read the content and we are easy to do programming to extract content and save them into systems. But we will face the bigger difficulties of how to describe complex content information in such format.

For example, Java property file is simple and easy to use. But I doubt that, in property file, whether such name-value sets are sufficient to represent abundant information. That is why in EJB 1.0 deployment descriptor is replace by XML format type so quickly in EJB1.1.

So we face two challenges:

- I need to find out a machine-readable document format type to store content instead of using above proprietary binary-based document formats. It should be structured, text or stream-based, and we need to have a standard and uniform way to handle such documents. It should be easily stored in systems, transmitted between systems, transformed, and interpreted by entities that understand the structure.
- I need to find out a flexible way to convert documents with above-mentioned formats into this format. This converter should not be limited with only Word to XML Conversion and unidirectional conversion.

1.2 XML Is the Choice.

XML (eXtensible Market Language) is a set of syntax rules and guidelines for defining text-based markup language. It is rapidly becoming a cornerstone of the Internet and e-business industry. Under the leadership of the World Wide Web Consortium (W3C) [W3C02] all the major players in the software and applications

industries have developed a bundle of open standards for XML and related tools that will revolutionize how we exchange information not just via the internet but in all IT areas. XML is gaining ground in many industries that are web-based or are moving towards the Internet and its technologies.

1.3 XML Advantages

In contrast to other format-based markup languages, XML (and similar languages) has several important advantages [Tess00]. XML

- is stream-based format,
- provides "meaningful" markup,
- is extensible,
- is flexible,
- has precise and deep structures,
- is well-suited for object-oriented development,
- is relatively easy to integrate with legacy environments and databases,
- separates content from presentation,
- allows reuse of information,
- processing API and programming models available

"Meaningful" markup allows people to annotate a XML document with useful information. The ability to search for (structured) information is a very basic and fundamental requirement for any form of information management. XML with its clear tree-based structure provides the right structure to enable rule-based processing of information.

So, from above saying, XML is innate best format for storing documents in Content Management Systems and Digital Library Systems. As I say before, now we need to find flexible solutions to convert different document formats, like Microsoft Word to XML, by using XML Converter.

The objective of this project is to implement a ***Word-XML Converter*****.**

1.4 XML Converters

Currently mainly there are three types of tools for conversion between Word and XML, if from the view of how to run the tools: [WaDa01]

- Standalone Word to XML Converters (including batch process programs). Standalone converters are useful when there is a large repository of documents

needs to be converted. Batch-oriented approach is very efficient by using this type of Converter. But the drawback of it is that the conversion process is not integrated with authoring environment and is not so convenient to use.

- Integrated Word to XML Editing and Conversion. It is a type of tool like using Microsoft Word macros and using Microsoft Word as one type of XML Editor.
- Microsoft Word extension. This type of tool is like VBA plug-ins to analyze documents and generate from them into XML files.

From the view of how to access or parse document, there are two categories of tools:

- Word-RTF-XML Converter. This type of converter takes into account that RTF is a stream-based format (while Microsoft Word is a binary-based proprietary format). The converter will first convert the Word document to RTF file, the read the stream-based RTF file as plain text file byte by byte to extract contents inside the document.
- Using provided Microsoft Office API (COM/OLE Automation library) to make method calls to extract content is another kind of tools. Programming languages like Visual Basic, Visual C++, Delphi can directly call COM library. But for Java, it cannot directly call COM libraries. So some open source organizations, persons or ISV vendors provide such COM-to-Java bridging tools to call COM libraries from Java programs.

1.5 Why Not Use them, But Develop My New One

Currently, there are a lot of commercial products of Word-XML Converters available. But after my evaluating them, I still cannot find satisfactory tool. To summarize, there are several problems:

- Word-RTF-XML Converter is not a perfect solution. Because Word-RTF conversion will lose some information in Microsoft Word, because Word file format has a much richer set of elements.
- Plug-ins or Integrated converters cannot fit in with requirement of large deposit of documents batch processing. For Content Management Systems or Digital Library Systems, we need the ability of auto-processing without too much intervention of administrator.
- The conversion mechanisms of those tools are mainly base on built-in or customized "Style" format in Word documents. This kind of XML Converter works well with documents that created with specific document template ("Style"s are already defined by controller, and document authors and writers just need to directly use them.)
- In document, formatting information are mixed with content. But for systems like Digital Library Systems are interested more in document content, not in

Microsoft Word's flaring formats. At the same time, they are interested in reconstructing documents from XML that still need to retrieve formatting information. So, in my implementation, it needs a flexible and extensible way to handle formatting information and text content.

- Those converter tools are usually using one time conversion. Firstly, user specifies the applied DTD, and then asks the converter to do the processing. If some modifications are necessary, user needs to do the whole once again completely. I doubt if it is necessary and it is too low efficient?
- What should I do if clients want to change the output results, like markup, tag information and etc? Normally these tools use different DTDs before conversion and do conversion again completely.

1.6 Design Objectives

To solve the above-mentioned problems, here I list out some important points I need to consider when I implement my XML Converter. I categorize them into separate objective groups.

Software Architecture

- The XML Converter should be a standalone application, which supports batching processing large deposit of source documents, such as Microsoft Word documents.
- The XML Converter only supports Microsoft Word document? Can it support some format types like PDF, PowerPoint, even formats not available now?
- Microsoft Word documents have a lot of versions. Which version the implementation can support, Word 97, 2000, XP? How to flexibly support all of them in actual implementation?
- Multilingual problem. How to recognize which language the contents of document are in, English, German, Chinese, even mixture of different language and how the conversion supports multilingual content?
- Considering above points, do I need to recompile whole or part of the programs? Can I use good software architecture and design patterns to have plug and play effects to support new functional plug-ins or upgrade?

How to Handle Formatting Information

- How many format types in Microsoft Word can be supported in conversion process? I obey the “80/20” principle, that is I would like to support the 20% format types that are used 80% of whole time.
- I need to find a way to flexibly separate formatting information in document with content and this way should be able to be easily extended.
- In implementation, converter should not just focus on "Style" format, but also other different formats like fonts formats.

- I must find out the formats difference in different level, such as sentences, words, even characters level.

Can Customer Intervene with Processing and Affect the Result Products

- Such intervention should be easy, visual, and customers don't need to have good knowledge of computer and software.
- Such intervention should be able to happen in the middle phase of conversion process, not just in early phase like some Word-XML Converter products I have seen by only changing XML DTD.
- I don't want to re-parse saga novel completely to update the result file just because I need a smaller header font.

How XML Target File is Constructed

- Do I need to use my proprietary XML DTD/Schema or use widely support industry standards?
- If industry standards (DTD/Schema) are not enough to fit in with my scenario, I must add modifications or extensions. How to do that without hurting the compatibility with original standards and how such extension combines both formatting information and document content?

Further Processing and Representation of Result Files

- How do current Internet browsers support of XML, because not all Internet browsers can show XML file.
- How to present result? If I can reconstruct original document with formats and content from result files, that will be very helpful.
- Can I support rule-based tagging?
- I need to use XHTML and XSL/XSLT to do the tasks.

Other Non-Trivial Problems

- Memory problems. The conversion process is like XML DOM model programming, not SAX. When doing conversion, I will create a FDM model that completely maps with the source document and puts the whole document tree in memory. So, memory is an important effect to performance.
- How about the speed and efficiency of the conversion process? What algorithms can be used to accelerate the process?
- How to choose XML Parsers?

1.7 What is Next

In this chapter, I have explained why I choose XML to fit in with the internal storage format type requirement of CMS systems and Digital Library System and I have also listed some of the objectives of my implementation of XML Converter. In the

following chapters, I will talk about some of my ideas and how I plan to fulfill these objectives. Chapter 2 will discuss how to decide which types of formatting information I will support and the implementation strategies. Chapter 3 will focus on architecture of whole system. Chapter 4 and 5 are based on discussion of how to design XML target and FDM.

Chapter 2

Document Formatting Information

This chapter discusses which types of formatting information in document need to be considered when we need to store contents of different types of documents in CMSs, especially when we are talking about specific formatting information in Microsoft Word documents. I will put emphasis on discussing two format types - font and style. I will also illustrate my considerations and processing strategies about how to extract the formatting information from documents such as Microsoft Word in my implementation.

2.1 Whether We Need Formats

When Converter system parses the source documents, whether we need to extract formatting information inside the document to XML target file depends on the further use of XML.

In Digital Library Systems or Content Management Systems:

- If people only need to search for keywords, systems do not need to consider formats too much.
- If people need to reconstruct original document in future, that means systems need to extract and store most formatting information. Of course systems need to consider formatting information as much as possible.
- If objective is for transformation or information representation, systems need to save most important formats to know how original texts in source document look like.

Current available XML Converter tools usually convert document content into XML with or without formats completely. If yes, formatting information is embedded as

attributes or sub-elements of parent XML elements. That is a dilemma and hard to decide how detail we need. Such solution will create redundancy if the purpose is only storage, or if purpose is transformation, this solution will lose a lot of formatting information due to the difficulty to record all formats using this method. That maybe is insufficient for further XML content representation or reconstructing original document from XML. We need a better solution in that Converter should separate content and formatting information.

2.2 Formatting Information in Microsoft Word

As I have said before, most current document formats are usually used for human reading. For better layout representation and easier to emphasize main points in documents, documents can contain a lot of formatting information, by using different ways (different fonts, symbols, bullet lists, tables …) to present different contents (paragraphs, headers, footer ...).

Especially in Microsoft Word document, it contains formats in different levels, such as font, paragraph, list, picture, line, text field, shape, 3D shape and etc. For example, in Word,

- *Shape* represents an object in the drawing layer, such as an AutoShape, freeform, OLE object, ActiveX control, or picture. It can contain *CalloutFormat, FillFormat, LineFormat, LinkFormat, OLEFormat, PictureFormat, ShadowFormat, TextEffectFormat, WrapFormat, ThreeDFormat* and so on formats properties. Please see figure 2.1;
- *Paragraph*, as we are very familiar with what it uses for in Word, can contain *Borders, Shading, ParagraphFormat, Style, TabStops* formats and so on. Please see figure 2.2.

Each format is accessed as a function or an attribute of parent object in Word COM Automation library. From here we can get some impression how complex different components are constructing a Word document. And some of these format names we have not considered or heard in usual days.

To implement a Word-XML Converter, I need to extract and store such formatting information, though not necessary to reserve all formats, but at least formatting information applying to text content, which are important and used most.

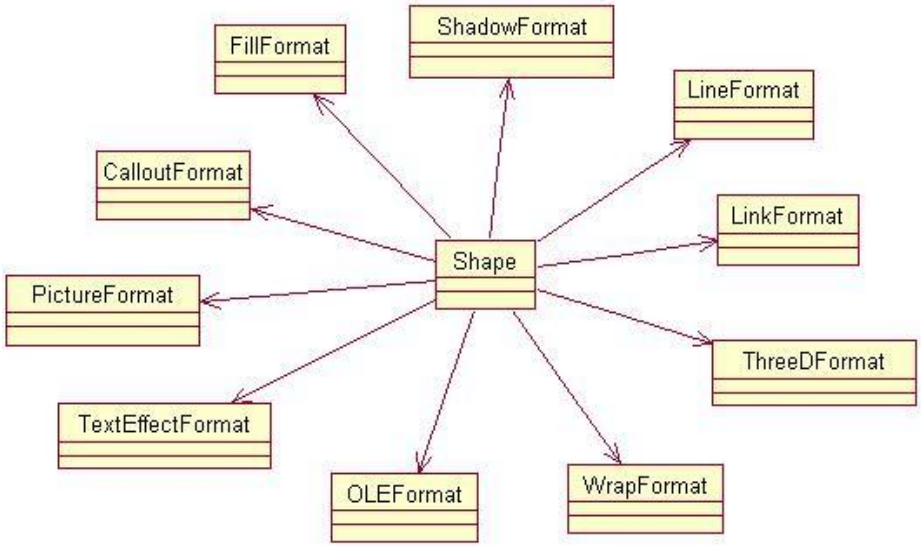


Figure 2.1 Formats contained in Shape object

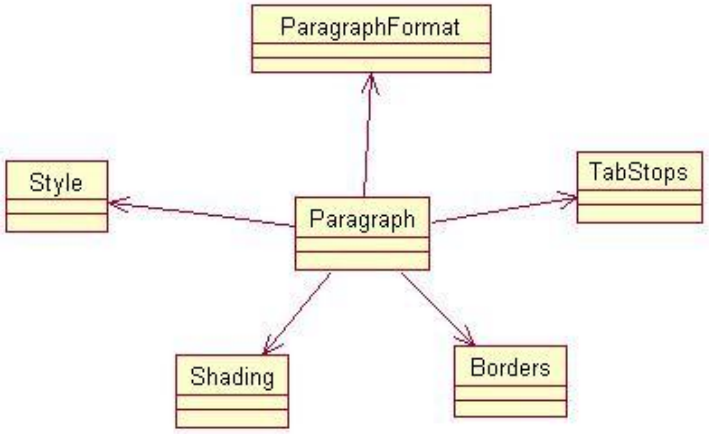


Figure 2.2 Formats contained in Paragraph object

In former chapter I mention the design objectives of this project. I want to create a XML Converter that supports multiple document representations. That means the XML Converter should be able to support other types of documents, like PDF, PostScript and so on. So when I analyze the documents structures, I should consider the common core elements available in all documents, rather than trying to include all elements. What is more, if the implementation has flexible software architecture, it is not difficult to extend and support new formats.

And since my implementation is only demo to show the mechanism of how to realize XML Converter functions, according to the “20/80” principle (that is, I will put emphasis on 20% contents that are used 80% in time), in implementation of Converter,

I can simplify the implementation by only considering most common used formatting information when Converter parses the source document. But the mechanism of adding supports for newer formats is almost the same as how to do with available formatting types.

2.3 Fonts and Styles

As I know, in Word document, Style is most important formatting type, though not all people pay too much attention to it. It represents a collection of formats specific applying to text content. We can say that:

Style = Font + ParagraphFormat + TabStop + ListFormat + Language + ...

In fact, mostly people use it intentionally or unconsciously. Because there are 2 categories of styles, built-in or customized styles, if we do not specify the styles of text content when we create document, the built-in styles will take effect. Microsoft Word has many built-in styles like ***headline 1, headline 2, Hyperlink, formal text*** and etc. When writing documents, users just need to select a range of text and apply appropriate style on it. So, all documents that are created using this mechanism can have uniform format styles. What is more, if built-in styles are not enough, people can create more their own customized styles and save them into a Microsoft Word Template files (*.dot) and deliver them to clients or document authors. People use this template to create new Word document by selecting ranges of text content and applying specific style they like to it. In fact, currently people using Microsoft Word to create document are using a default template (*Normal.dot*) and are using lots of pre-customized styles in this template.

It is a good idea for documents authoring in future. Customizing styles is a promising method for creating uniform predefined fashion document. Same layout, fonts, fashion.... No matter for reader or document writer, or document collector. What is more, for machine reading, like software system parsing the documents, the unified styles will extremely simplify the parsing algorithm and accelerate the speed and efficiency of whole process.

But Problems may still come out. Don’t forget there has been already a large deposit of documents that need to be processed. And most past, current and future authors who are not computer experts, usually do not know or pay much attention to “Styles” function in Word Processors. It still need time to advise them to accept and get accustom to using *Style* to format their documents.

- But even they use *Style* format, people can always customize the specific formats inside texts with same style, such as bold, color. That is, in Microsoft Word, to be flexible, texts with same style can be applied different fonts without changing the

style. It is right, I also don't want to spend so much time to create a new named style just because I simply change my name with *Bold* font. I will only select the fragment of text and just change with *Bold* font. I will not buy that word processor if it only supports *Style*. It is too inconvenient.

So, we can see, only using styles as formatting format is not enough and is a limited solution. So, we need to look inside what information *Style* object contain in Microsoft Word Object Model.

In Word, as I show before, *style* is a collection of most format types which applied to text content, including information like *Font, Paragraph Format, Table Stop, List Format, Language* and so on. But to simplify the problem, I only consider *Font and PargraphFormat* format in *Style* in my implementation.

Ø

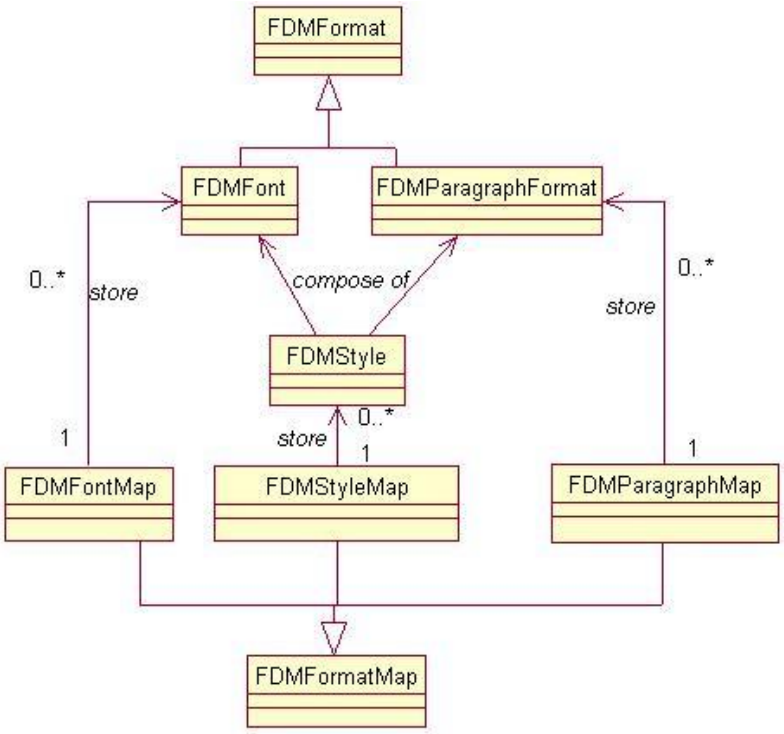


Figure 2.3 Class relationships between font and style

Ø

Figure 2.3 shows the relationship of format classes in my implementation.

- **FDMFormat** is abstract class for all kinds of format in document
- **FDMFormatMap** is abstract class for libraries, which used to store different formats. (But not a good name here)
- **FDMFont, FDMParagraphFormat** are type definition for different format type.
- **FDMStyle** is a composite format type which is composed by other types and use
- **FDMStyleMap** to store all instances of style
- **FDMParagraphMap** to strore all instance of paragraph format

To summary it, *FDMFormat* is the abstract parent class for all format classes in document and *FDMStyle* is an aggregation of different format instance. I use *HashMap* to store each type of format class instance, which named *FDMFontMap, FDMParagraphMap, FDMStyleMap* here.

2.3 Implementation Strategies to Formats

From above diagrams, we can understand the hierarchical relationships among different formats. One point need to emphasis here, *Style* is not the highest abstract definition of all format in Word, just a collection of some format types. For example, we can also consider formats like *header, footer, other layout formats, index, line format, shape format* and etc. in figure 2.4, I give an example of higher-level format relationships. We can see that, the relationship of formats can be very complex and need a lot of analysis and research of different document types. My implementation is only a demo of the XML conversion process and will not cover all of formats.

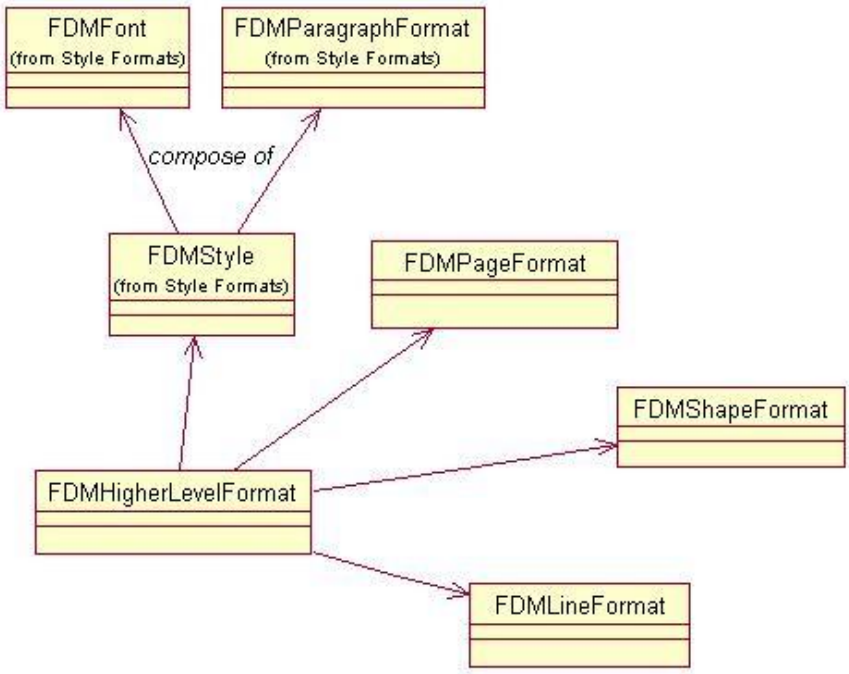


Figure 2.4 Higher-level relationships between format classes

Here I explain some of my considerations about how to implement the formats. From figure 2.3, I will only discuss *Style* and *Font* formats.

Style

- If we only consider *Style* in conversion, we can store all different *Styles* instances (no matter built-in or customized *Styles*) in *FDMStyleMap* (*HashMap* as parent class) instance with names. When the program parses the document, since all text contents (*Range* instance and *Paragraph* instance) in document have *Style* property, I just need to loop through each content and check the *Style* property. To know detail about how this *Style* defines, I can look inside detailly the *Style* instance.
- That only considering *Style* in looping each text content when we parse document can be very fast and efficient. But as we know from above, this is only *Style* level format. Inside this level, a lot of paragraph level, sentence level, character level *font* differences are omitted.

Font

- If we consider *Font* format directly, we can get all most important formatting information when we parse the document according our usual experience.
- I parse the whole document and check the *Font* at the same time. When I find a new *Font* (which has some differences with other *Font* instances), I will create a new *FDMFont* instance and store it to my *Font* library instance named *FDMFontMap* (sorry for this improper name).
- The problem of this solution is the efficiency of algorithm. We can imagine it is unreasonable to parse with all document contents character by character. We need to have better algorithms to solve this problem.

Style + Font + More other formats

- Although *Style* solution has limitation, anyway it fits in with a specific kind of documents that I mention before with very high parsing efficiency. Can we combine the two solutions?
- Idea of solution is simple. When we parse the document, we will first check *Style* property of text content, and then inside the content with same *Style*, we can check detailly *Font* information and use *Style* instance as default *Font* for this range of text content.
- If we consider *page layout*, *header* and *footnote*, we can have a higher-level format over *Style*, we can also use the idea and mechanism here of how to combine *Font* and *Style* to support more formats.
- My implementation only considers *Font* level format solution. So I use *FDMFont* instance to store new *Font* and use *FDMFontMap* instance as library to store all font instances of the documents. That means I only use a list for storing *Font* instances with unique index number. We can use more sophisticate and complicated way to organize the Font instances, like using directory naming services. It depends on the specific requirement of CMSs. This is out of the topics

of my implementation. Just mention this a bit here.

2.5 My Consideration to Support Font

Because the limitation of *Style* level format solution and *Font* level format solution can obtain most required information, my implementation only considers *Font* level format solution.

- I use sub-class of *HashMap* to store different font instance using index number
- Font may have different levels inside a document, such as *Character*, *Word*, *Sentence*, *Paragraph* or even larger block. That means that, in document, between two characters or two paragraphs, we can use different fonts. But how to find out the difference? Because the parsing algorithm should not fix the checking level, because we cannot anticipate which *Font* level a document will be in. For example, if the document is using the same font in most content, I just need to check the font in *Paragraph* level, while checking in *character* level will waste a lot of time and is unwise. But if inside the paragraph the author uses a lot of different fonts, *Paragraph* level checking will lose the important information. In Microsoft Word document, each *Paragraph* instance has a *Font* instance as default *Font* for this paragraph. Because it can make me omit the *Font* differences inside the *paragraph*, this is why in my implementation I do not simply use this default *Font* instance.
- I use **self-adaptive parsing algorithm**. For example, if I find that the whole paragraph uses a same font, there is no need to check detail font of each sentence inside. I can just use the default *Font* instance of the paragraph. If different fonts are found inside the paragraph, I need to lower the checking font level from *Paragraph* to *Sentence*. If still differences are existed in sentence, the rest will be deduced by analogy to *Word*, even *Character* level. Such parsing algorithm can fit with any document and has the learnt-by-doing effect, and has the advantage of performance and efficiency.
- To further improve the efficiency, I use a **cacheTextFont** variable in program to save the former found font instance. Because we know a rule in normal document, font of following unit of text content is same as former text content. I do not need to search *FontMap* and compare with all available *Font* instances. This can increase the efficiency dramatically.
- What is more, in my implementation, I only consider *Font name, size, bold, Italic, Underline, Forecolor, Backcolor* most common properties, according to "20/80" principle. As we know, it is very easy to add more in future.

Note:
One important point must be emphasized: Here in my simple implementation, I only use

MAP/LIST of Java Collection Library to store all font instances for one document. But if want to support multiple document, if want to support keeping trace of all modification to the font instance, using MAP is obviously not a good choice. We need to use some mechanism like Directory Service to support that. To keep simple, I leave these as open issue.

2.6 Formatting Table File

From above, I store all *Font* information in *FDMFontMap* instance after parsing the source file. We can output this information to a Microsoft Word document that I call "Formatting Table File".

Following is the ***Formatting Table***:

Font code	Detail	Sample	Tag
23	Name=Times New Roman; Size=16.0; Bold=-1; Italic=0; Underline=0; forecolor=0; backcolor=-16777216	**Sample Text**	
24	Name=Times New Roman; Size=14.0; Bold=-1; Italic=0; Underline=0; forecolor=0; backcolor=-16777216	**Sample Text**	
25	Name=Times New Roman; Size=12.0; Bold=0;	Sample Text	

Figure 2.5 Example of Formatting Table File

There four columns in this table:

- **Font Code** - Is the index number of this font in the FDMFontMap
- **Detail** - Is the detail description of this font
- **Sample** - Is the sample text with this font. So, user even don't know too much about software can see how text with this font look like
- **Tag** - Tag name which user want to use for this font. In current version implementation, users can apply <H1>, <H2>, <H3> and so on header tags. (Talk about this later)

In this table, it gives a very direct way to let user find out how this *Font* looks like. The detail description of the *Font* in "Detail" table column should be same with sample in "Sample" table column. People can modify the description or the font of sample text if they want. For example, obviously, he know that font with *fontcode=23* is the font for *headline 1*, he want to change all such font with *underline*. He just need to simply change the number of *underline* to 1, like *underline=1;* or in another way, he can select the sample text and press "*Underline*" button of Word Processor and add ***underline*** to the selected text. That is what we do normally in Word document. To do this, customer does not need know knowledge of my Converter system. He does not need to know too much about computer. He just needs to know how to use Word processor. Converter implementation will do all the other works for them.

For detail about the explanation of how Converter works and how to use this *Formatting Table File*, please read the following chapters.

Chapter 3

Architecture

This chapter summarizes the viewpoints of the former two chapters and lists detail sub-tasks of this project. And then I will propose the software architecture for the Converter system that can fit in with the desired goals defined in chapter one.

Due to time limitation, this architecture is not completely implemented in the demo application. But, this report will have a more detail discussion.

3.1 Task Description

Until now, I have mentioned several problems need to be considered and solved when the Converter system is used in Content Management Systems (CMS) or Digital Library Systems for multiple document representations. I summary them here and detail the requirements now.

- Because of the advantages of XML and the industry-recognition of XML, we have formed the agreement of using **XML as document type** for storage or as **intermediate media** inside CMSs. That is why we need such a XML Converter system as a plug-in function module of CMSs.
- More importantly, to ensure least errors and having a robust system, or to ensure compatible with other systems, or to help communicate and easier integrate with other systems, we require that, the target XML files should be validated by XML DTDs or XML Schemas. This **validation** is done by my own DTDs, or better done **by industry-accepted open standard DTDs**, for example DocBook DTD. I will discuss this topic in next chapter.
- The XML Converter system should support **bidirectional conversion among**

multiple document types, even though my implementation is only a Word-XML unidirectional Converter. There are hundred of document types available. For each one of it, providing a specific Converter tool is not a good solution. It is still a big problem how to integrate all these Converter tools, because there is no common interface of doing conversion. So, how to integrate them and how the Converter system ensures automation processing with least and easy human intervention are important topics of this project. Obviously, I need to define a **flexible architecture** for this Converter system to support these objectives. In this architecture, I need to use **XML as internal conversion resulting** document format and design input and output modules for different document types. What is more, I also create a **document model in Java** as the intermediate conversion result (called Formatted Document Model), which is mapped with XML conversion result. By using these two intermediate models (Java and XML) and unified interface for input and output conversion, we can have a flexible architecture that fit in with our target. In the next section we can have a look at the diagram of this software architecture.

- Most current available Converter products are using **one time conversion**. That means my Converter implementation should **avoid** doing all the conversion again if the users only change a few requirements. Since our Converter architecture has a intermediate result for conversion, XML or Java. We can easily avoid the one time conversion.
- The Converter system should flexibly handle the problem of Word Processor's having **multiple versions**. What is more, further processing should be independent of peculiarities of specific Word Processors and programming API. Because Microsoft earns a lot from upgrading his Office application and it is disgusting to face the document format compatibility problems. Such document changing should be hided in the Converter system and it should provide a flexible way to handle the problem, not leaving these problems to clients.
- **Internationalization** problems always are terrible headaches to a not well-design system; especially the document contains multilingual content. We must think and find out solutions about these problems when we design the system.
- I need to find a flexible way to **separate the content and formatting information** from source document and store them into XML target file, in which content and formats are also separately stored. Because this can avoid the redundant information problems for storage. So that, users can easily modify content or formatting information or both as their wish anytime. At the same time, what formatting information can be outputted and how to extract them is a big interest in this project.
- The Converter system should provide flexible techniques to convert the internal representation (FDM and XML) to **multiple** user-friendly handling formats as well as wide-spread presentation formats **(e.g. PDF, (X)HTML)** or **revert conversion back to original formats** (e.g. Word, PowerPoint, Quark Express). We can use programming APIs or XSL/XSLT, or some other tools to obtain these results. In a word, use XML and Converter internal document model (FDM) as

the media for multiple documents formats conversion. The architecture can support these multiple documents formats conversion.

- The Converter system is better to support **rule-based or algorithm-based** conversion processing; that is, we can define rules to control the results.

So, we can define subtasks and steps of this project:

- Analyze the Microsoft Word Object Model accessible through VBA/COM library and understand how to access VBA/COM from Java (by using Bridge2Java)
- Analyze the DocBook DTD standard
- Define an abstract and expandable intermediate document representation (named "FDM", Formatted Document Model) for technical-industrial documents, which could be generated from several format types provided by commercial text & document processing tools Microsoft Word, Microsoft PowerPoint, Quark Xpress, Latex and so on, which permits a formatting information can be flexibly separated from content.
- Implement the intermediate document representation by Java classes and document the semantics of these classes wrt. the DocBook standard
- Map this intermediate document representation to XML, which can be validated by DocBook DTD
- Present the result XML by using different ways, like revert converting to original document or transforming to HTML or other ways
- Design input and output interface module to support converting multiple documents formats with these intermediate document representation
- Think about how to solve the problems like document versions, internationalization, efficiency, controlling and so on.

3.2 Converter System Architecture

Although demo implementation is only Word-XML Converter, its idea actually fits in with multiple document formats conversion. Figure 3.1 is the architecture of the Converter System. I will have a brief explanation here and in the following chapters, I will give detail discussion on several important topics. At that time, I will explain how my implementation realizes some of this architecture.

Finally, I will give another architecture diagram that is related with my implementation. We can compare two diagrams and understand the roles of different modules of my Converter implementation.

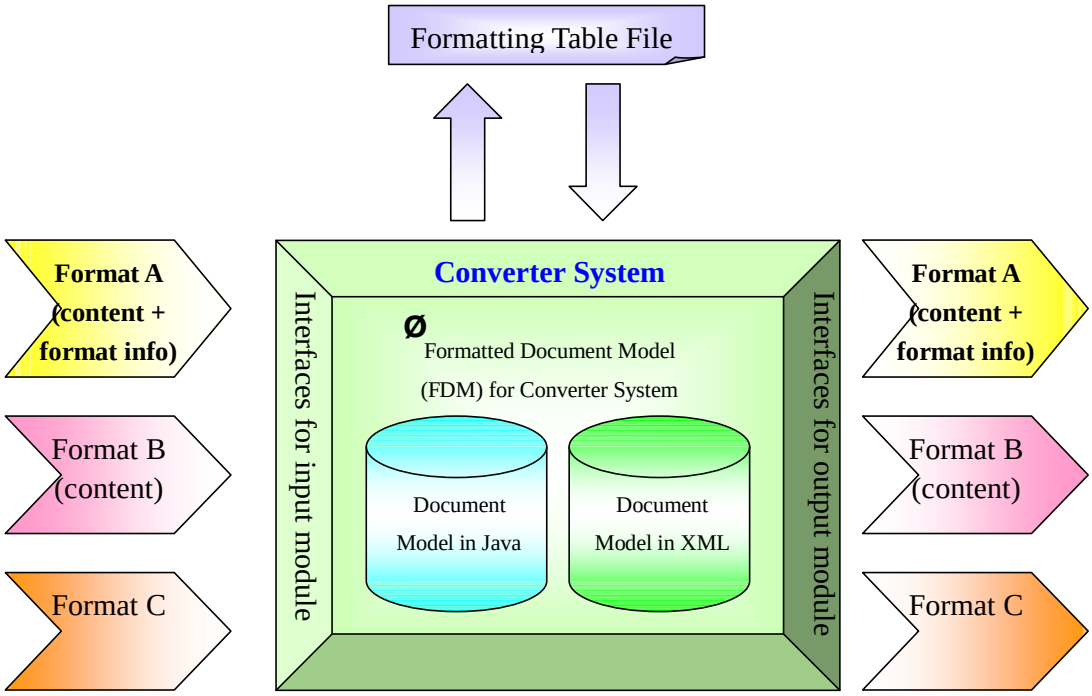


Figure 3.1 Architecture of Converter System

This figure illustrates basic structure of the planned Converter System.

- Document can contain both content and formatting information, or only content information (that means, to omit deliberately formatting information in the document). How to control this depend on requirements and target objectives of system.
- Support bidirectional conversion. Multiple document format types can be used as input or output.
- Design a unified programming API interface as input and output modules to support different document format types.
- For each format type, input and output implementation libraries can be separately installed. So this can provide bigger flexibility. This idea is like using JDBC to access external databases. It is not necessary to provide a specific tool to handle the specific document format. Because they are separate tools and are not easy to integrate with system. By using the common interface, when we need to support new format type, when we need to change the actual implementation, we just need to load related implementation library for this format that implement the interface. We do not need to recompile or modify the Converter System
- Conversion system uses dynamic loading to load the implementation libraries, and changing implementation does not need to restart the system and have the

effect of plug-and-play for each module.

- In the middle of conversion system, there are two ways to represent the conversion results. One is using an intermediate document model in Java, another is using XML file that can be validated by industry-accepted open standard like DocBook. These two are mapped with each other and can be converted to each other inside the system. The effect of these intermediate models is a XML Converter. But now I have extended this simple function to do more jobs.
- Since the available of intermediate document model, we can generate the result XML file or other result formats repeatedly and we can add some other processing to the model before output. Thus, these functions provide flexibility to fit in with the design objectives.
- From the figure, no matter input documents or output documents, we can see that, we have different ways to processing document. We can parse document independently, or accompanying with style format collection. This style format collection means that, customers can use some mechanism to define the rules to handle formatting information first and use them to parse their document, instead of default rules of the conversion system. This provides another flexible way in the conversion processing.
- As we know, one main objective of conversion system is to separate and store the formatting and content of input source file. So, inside the conversion system, no matter use the intermediate Java document model or use XML, we must reserve these information in something like style/format library/collection. And what is more, for convenience and flexibility, conversion system should be able to export the formatting information to outside and let customers modify as they require. In the end, re-import the modified formats library and adjust the internal formats library. This provides a mechanism for customer to define rules for the result representation.

Chapter 4

XML Target Consideration

This chapter explains how to define the XML target file as the results of Converter system and how XML target file maps with the intermediate document model with formatting information. Which XML standards I need to take into account? What are the key requirements? What is my extension? In the end, a brief introduction related to XML programming in Java will be given.

4.1 XML DTD and Schema

As we know from architecture, we need to define a target XML file, as the result of XML Converter conversion processing, as the storage format if the Converter System is integrated with Content Management Systems, as the internal document model in the Converter System, which is at the same time mapped with the Formatted Document Model (FDM) of Java in Converter System.

Firstly, I need to consider how to design this XML file. At the beginning, I design everything including all elements and attributes by myself. But later I found that these methods left too much room for errors and personal interpretation. And I always need to think about how to find a flexible and extensible way to describe document. That spends a lot time and I am not satisfactory with the result. So, I must change the direction of my work. I need to:

- use validation of my XML markups;
- adopt the experience of others and refer to the industry-accepted solutions.

As an important feature of XML, XML can use DTD or Schema to describe all its markups in a process known as validation.

DTD is an acronym of *Document Type Definition.* [Har01] It lists a set of declarations that defines elements, attributes, entities and notations that can be used in a document, as well as their possible relationships to one another. A DTD specifies a set of rules for the structure of the document. If the document matches the constraints listed in the DTD, the document is said to be valid.
DTDs provide a means for applications, organizations, and interest groups to agree upon, document, and enforce adherence to markup standards. At the same time, DTDs also help to ensure that different people and programs can read other's files.

But as a definition method, DTD still has several limitations [Har01]. It:

- is lack of data typing, especially for element content
- uses an unusual non-XML syntax
- has limited extensibility, and it does not scale very well, and it is difficult to combine independent DTDs
- is only marginally compatible with namespaces
- has other problems, like that it cannot enforce the order or the number of child elements in mixed content

XML Schemas are an attempt to solve all those problems by defining a new XML-based syntax [Har01]. It provides:

- powerful data typing including range checking
- namespace-aware validation based on namespace URIs rather than on prefixes.
- extensibility and scalability

For learning detail about XML, I strongly recommend <<XML Bible>> Second Edition [Har01] by Elliotte Rusty Harold, who is an expert and internationally respected writer.

The markup rules are defined in DTDs/Schemas, which define and encapsulate the practice and knowledge of specific user domains, hence providing the necessary flexibility and extensibility. For this purpose, there is no reason not to use robust wide-supported industry standard to fit my design objectives.

In my case, I need a DTD/Schema to define hierarchical tree-structural element relationship to describe the document. This tree structure must accommodate extremely flexible elements and other structures. Thanks to Professor reminding me to pay attention to useful DocBook.

4.2 DocBook DTD

DocBook contains a very popular set of tags for describing books, articles and other

prose documents, particularly technical document, in XML or SGML. DocBook is defined using the native DTD syntaxs of SGML and XML. It is almost 10 years old and since mid 1998, it has become a Technical Committee (TC) of the Organization for the Advancement of Structured Information Standards (OASIS).

To learn more about DocBook and get the latest version and information about it, please visit its official web site of DocBook http://docbook.org [WaMu99] and http://www.oasis-open.org/docbook/ . O'Reilly Publisher has a very good book about DocBook - <<DocBook – The definitive Guide>> by Norman Walsh and Leonard Muellner. We can download this book (chm version or HTML version) of DocBook's official web site.

DocBook contains a very complicated and complete tag library for describing books and papers about computer hardware and software.

The version I am using is the latest *DocBook DTD 4.1.2* version. DocBook is originally using DTD, but now it also has XML Schema version for solving the limitations of DTD. Since the Schema version is still non-official proposal, I still use DTDs.

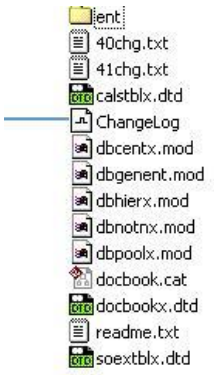


Figure 4.1 Files list of DcoBook v4.1.2

From left figure we can see that, *docbookx.dtd* is the main DTD file. But its contents are only references to other DTD definitions file (*.mod). But this organization makes it too complex.

DocBook also provides a simplified version called Simple DocBook, whose latest version is 4.1.2.5. It mainly defines tags for content in single document. The root element is document using it is <article> and all necessary elements definitions are define in one single DTD file named *sdocbook.dtd.* In my implementation, because only demo to process on a single document, I use Simple DocBook instead of complete DocBook.

4.4 My Extension to DocBook DTD

Is the Simple DocBook DTD enough for describing and validating my target XML file after conversion? No! I analyze the detail of Simple DocBook and found that:

- Simple DocBook 4.1.2.5 does not support *nested tables* that normally are supported in DocBook. I must modify the DTD and add element definitions related with *nested table* from DocBook complete version (DocBook v4.1.2);
- Simple DocBook does not include tags for most formatting information. For

example, it has no tags for *Font*, *ParagraphFormat*, *ListFormat* and so on. I must modify *Simple DocBook DTD* and add elements and attributes that are specifically used for formats.

So, which is mainly modified from *Simple DocBook 4.1.2.5*, I write *extDocBook.dtd* in my implementation for validating the XML conversion target file. In the XML declaration of all XML files that created from my Converter System, we can add this DTD and use XML tool to validate the XML file.

For example, add <!DOCTYPE article SYSTEM "extDocBook.dtd"> after <?xml version="1.0"> declaration.

Ø

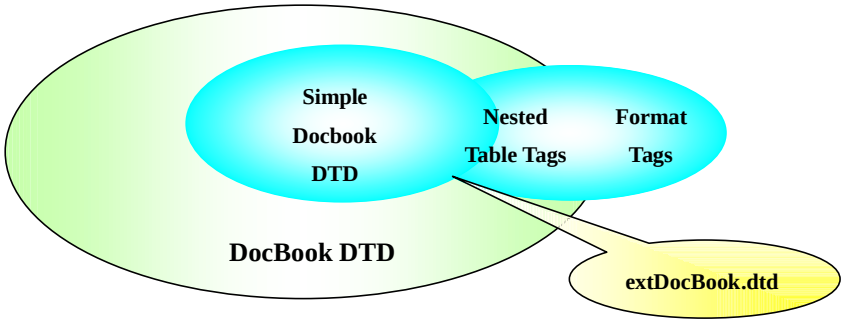


Figure 4.2 Relationship of my DTD with DocBook DTD and Simple DocBook DTD

From figure 4.2, we can see the relationships among different set of DTD definitions. We can say:

extDocBook.dtd
= Simple DocBook DTD + Nested Table Tags Supports + Formats Tags Supports

Here are the design objectives of DTD extension:

- ü Based on simple DocBook DTD
- ü Only small a number of modifications are made to ensure least affects.
- ü Support nested table in DTD, which simple DocBook does not support
- ü Support storing formatting information in result XML file
- ü XML files which can be validated by DocBook are still valid with this extension DTD to ensure compatability.

(Note: In extDocBook.dtd, please use "@@@" to search and look for my modifications to the simple DocBook DTD. Before all my modifications, I have symbols of "@@@" as label and I keep the original statements in the comments. Just compare my modifications and the original definitions, very easy to see what I modify and add.)

What's more, I summarize all the modifications I have made. We can open the *extDocBook.dtd* using XMLSpy and search with *"@@@"* to locate all modifications.

Add nested-table tags:

- add <*entrytbl*> as subelment of <*row*> element
- Ø

Add formats tags:

- Add new element <*fontlist*> with subelement <*font*> as subelement of root element <*article*>
- Add <*paragraphformat*> as subelement of <*para*>
- Add *@empty* attribute to <*para*> element
- Add attributes to element <*pharse*> like *@fontcode and @styletype*
- Add attributes to element <*listitem*> like *@listlevelnumber, @liststring, @listtype, @listvalue*

4.4 How to Process XML

Since I need to create XML file, I need a tool to process, parse and output XML. There are two programming interfaces standards for XML. They are DOM and SAX. There are a lot of such libraries available implementing these two interfaces, no matter in C, C++, Java or Perl. I will not discuss difference of these two interfaces, because this is over my topic. For detail, please read the reference book about XML by Elliotte Rusty Harold [Har01].

I choose to use JDOM [HunMc02] with current latest version 8 for processing XML.

(Note: Please do not use the accompanying JDOM library of JBuilder, even in Jbuilder 6. I don't understand why Borland uses only JDOM 1.0, though this is an open source tool. A lot of APIs in that version are deprecated.)

Here I will explain why I choose to use JDOM. Because I have used different libraries for processing XML and this one is extremely easy to use. What's more, it is developed to fit specially for Java developer. It has been a de facto standard in Java developers. This trend pushes Sun to accept it as JSR project, though with replicate functionality with Sun's own XML processing API - JAXP.

JDOM is a free open source project with Apache license created by Jason Hunter and Brett McLaughlin, two Java experts and writers. It is a programming model to represent XML data, similar to the DOM but not built on DOM or modeled after DOM. Currently it is a Java Specification Request (JSR-102) of Java Community

Process. For detail, please visit its official web site http://jdom.org.

To summary, JDOM was created to … [HunMc02]

- ü be straightforward for programmers using Java technology;
- ü use the power of the Java programming language (method overloading, collections, reflection);
- ü hide the complexities of XML wherever possible;
- ü integrate well with SAX and DOM interfaces.

4.5 How to Create XML Output Representation

One another thing I need to consider in my implementation. How to represent the result XML file to customer? Not all Internet browsers support XML. As I know, Microsoft Internet Explorer 5.0 and newer versions support it.

If people have not red material about DocBook and do not understood DocBook DTD, even they look at the XML, they cannot find out what I have extract from source code. For example, they may be confused by the names of *phrase, tbody, entry* and so on. We can teach them and give an introduction to them. But why not represent XML in other ways.

There are several options according to this Converter System functions and architecture.

1. Directly show as XML file in browser, notepad, or XML tools
2. Show as HTML/XHTML, which can be easily demo in browser
3. Revert convert the XML to original document, like Microsoft Word document
4. Convert XML to another document format, since this Converter System supports more other formats. And actually in point 3 XHTML is another format.
   - Ø

I have talked about 1, and 3 and 4 need to plug in new implementation module to support new formats. To be simple, I prefer to convert XML to HTML for representation in my demo.

One weakness of Java, however, is in its ability to process text. Java may not be the best technology for merely converting XML file to HTML. So, I rely on XSLT, the XSL Transformation, which is a part of XML Stylesheet standard that defines how to transform one XML into other formats. And I am using Apache Xalan application to do this transformation. It is very fast and easy to use.

I need to point out another important thing: The result of transformation of my

implementation is a XHTML file, not just HTML file. XHTML have such differences and advantages over normal HTML [TVDB01]:

- XHTML uses XML-based terminology
- Uses XML syntax rules
  - Close all elements
  - Empty elements must be terminated
  - Quote all attribute values
  - All attributes must have values
  - Case sensitive
  - Tag nesting is important
- Use XML document rules
  - Must require root element
  - Use optional XML declaration
  - Use optional DOCTYPE declaration

For how implementation to do these tasks, please read following chapters.

Chapter 5

Formatted Document Model

This chapter explains the detail about the intermediate document model, named *Formatted Document Model (FDM)* of Converter system. At the beginning of this chapter, I will firstly discuss the two reference models that are used when I create FDM. They are *Microsoft Word Object Model* and *DocBook Element Model (with DTD)*. Then I will describe different important parts of FDM and use class diagrams to show the relationships between different elements.

5.1 Reference Foundations

To design the FDM model, the idea is simple. I use this model to define element for each unit in document. The document model should contain most common elements in most document format types, such as Word, PowerPoint, PDF and etc. I will implement this FDM model in Java and use each tree-structural model instance for one document.

Two materials are needed to be analyzed and are used as my reference foundations.

- *Microsoft Word Object Model*
- *DocBook Element Model*

All contents in Microsoft Word are objects. Why not directly use the Word model as FDM?
Because this Word model is mainly used for accessing content inside the document, it provides too flexible relationship between different objects, and this model is not mainly for storing content as a tree structure. For example:

- For text content in Word, Microsoft Word object model uses *"range"* object. To ensure flexibility, the range of this *"range"* object can cover ***a character, a word,***

a sentence, a paragraph, or bigger of *several paragraph*, even text content of whole document.

- For each item (called *"listitem"*) of *list*, normally we have different nested levels of *listitems*. But in Microsoft Word, no nested *listitems* are available. The nested level number is stored as attribute of *listitem* object.
-

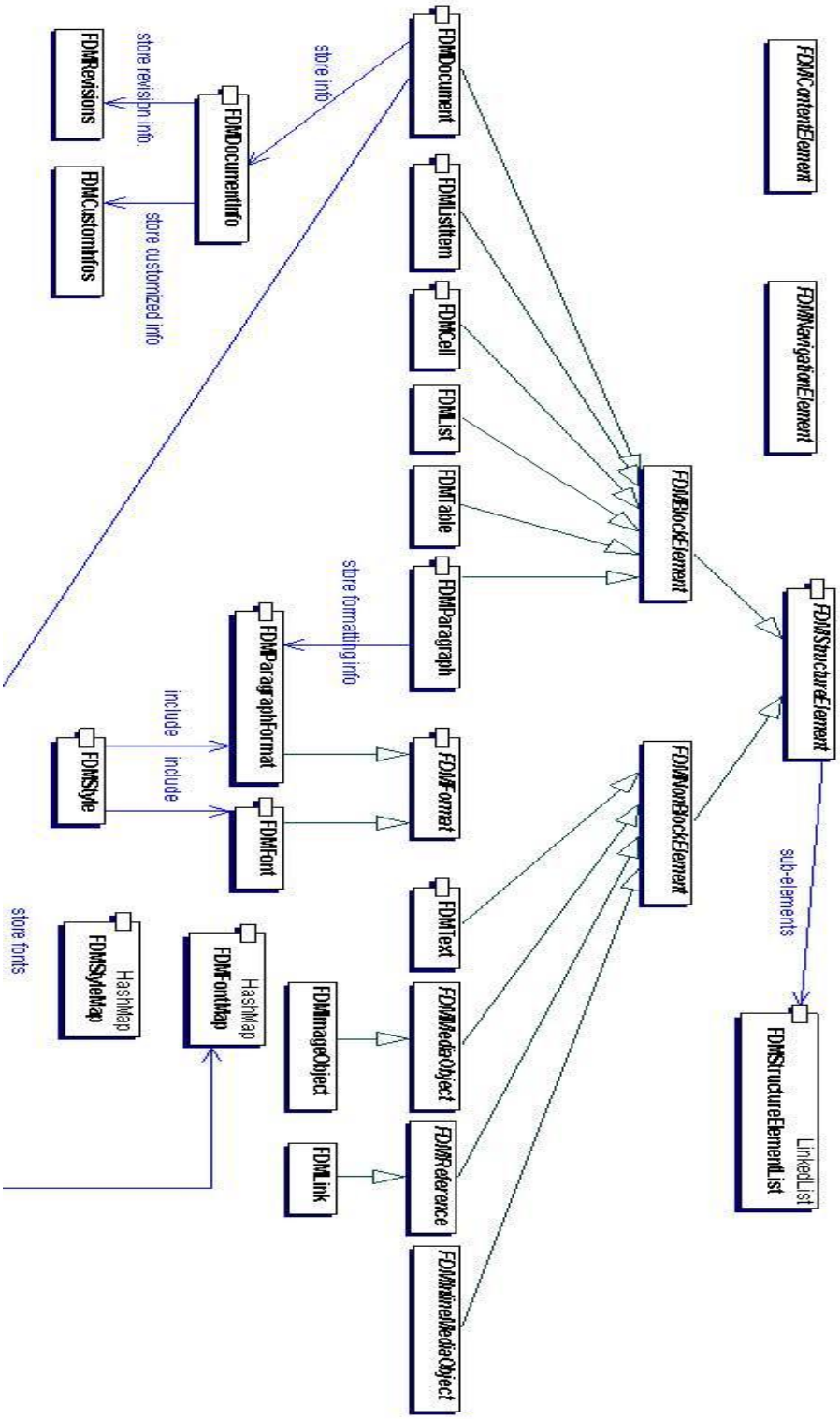
As seen in architecture, the FDM model should be mapped with XML file, which can be validated by my *extended DocBook DTD*. They should have similar tree structure. Since *DocBook* is mature and popular for describing technical documents, a lot of ideas of elements of this proposal are taken from it and simplified from it.

5.2 Detail Description

This is a simplified definition of Formatted Document Model. I say it "simplified" because:

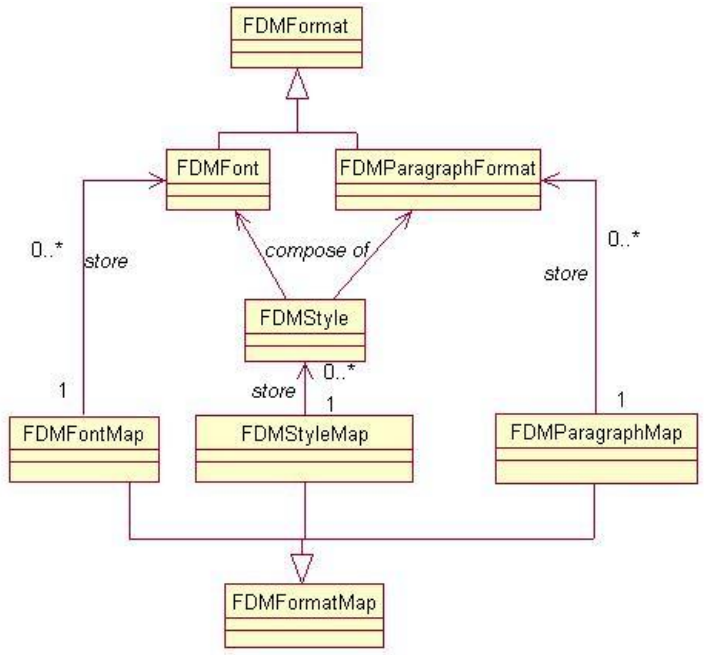
- Since this is an abstract and canonical model which support conversion from current commercial document formats, it should consider all kinds of components in most cases. It needs to probe and analyze deeply into different document formats like Word, PDF, Postscript, Latex, Quark Xpress and extract key common components in them. Here is only a demo and first proposal of such model.
- Base on 20/80 principle, only most used elements in document formats are written out and implemented in demo.
- This model only considers structure of a single document, without library to organize deposit of multiple documents. That is not the discussed topic of this paper.

Above figure the diagram of whole FDM model. But I will explain each part of it.


FDMIContentElement
FDMINavigationElement
FDMDocument
FDMListItem
FDMCell
FDMList
FDMTable
FDMParagraph
FDMBlockElement
FDMStructureElement
FDMNonBlockElement
FDMStructureElementList
LinkedList
sub-elements
store info
FDMRevisions
store revision info.
FDMDocumentInfo
FDMCustomInfos
store customized info
store formatting info
FDMParagraphFormat
include
include
FDMStyle
FDMFont
FDMFormat
store fonts
FDMStyleMap
HashMap
FDMFontMap
HashMap
FDMText
FDMImageObject
FDMLink
FDMMediaObject
FDMReference
FDMMultimediaObject

Detail explanation of model:

1. This FDM model is defined based on Object-Oriented for reusing and inheritance.
2. As having discussed before, for formatting information in my implementation, I only considers “Font” and “Style” (and Mainly only Font) to demo how to put to use this model. By parsing the document, when a new font is found, a new instance of FDMFont is created and stored in FDMFontMap.



3. Main content elements and relationships among content. Usually a document contains tables, lists, paragraphs, images, shapes and so on elements. Table can contain rows, columns and cells as basic unit. Cell can contain other tables and all other elements like lists, paragraphs, images and etc. To understand these relationships, the best way is to refer to “DocBook Element Referentce”.

Following are two figures taken from DocBook in which are too complicated. We need not to pay attention to a lot elements, but the elements with red circle.

Para

Name

Para -- A paragraph

Synopsis

Mixed Content Model

Para ::=
((#PCDATA|FootnoteRef|XRef|Abbrev|Acronym|Citation|CiteRefEntry|
CiteTitle|Emphasis|FirstTerm|ForeignPhrase|GlossTerm|Footnote|
Phrase|Quote|Trademark|WordAsWord|Link|OLink|ULink|Action|
Application|ClassName|Command|ComputerOutput|Database|Email|
EnVar|ErrorCode|ErrorName|ErrorType|Filename|Function|GUIButton|
GUIIcon|GUILabel|GUIMenu|GUIMenuItem|GUISubmenu|Hardware|
Interface|InterfaceDefinition|KeyCap|KeyCode|KeyCombo|KeySym|
Literal|Constant|Markup|MediaLabel|MenuChoice|MouseButton|
MsgText|Option|Optional|Parameter|Prompt|Property|Replaceable|
ReturnValue|SGMLTag|StructField|StructName|Symbol|SystemItem|
Token|Type|UserInput|VarName|Anchor|Author|AuthorInitials|
CorpAuthor|ModeSpec|OtherCredit|ProductName|ProductNumber|
RevHistory|Comment|Subscript|Superscript|InlineGraphic|
InlineMediaObject|InlineEquation|Synopsis|CmdSynopsis|
FuncSynopsis|IndexTerm|CalloutList|GlossList|ItemizedList|
OrderedList|SegmentedList|SimpleList|VariableList|Caution|
Important|Note|Tip|Warning|LiteralLayout|ProgramListing|
ProgramListingCO|Screen|ScreenCO|ScreenShot|Address|BlockQuote|
Graphic|GraphicCO|MediaObject|MediaObjectCO|InformalEquation|
InformalExample|InformalFigure|InformalTable|Equation|Example|
Figure|Table)+)

Phrase

Name

Phrase -- A span of text

Synopsis

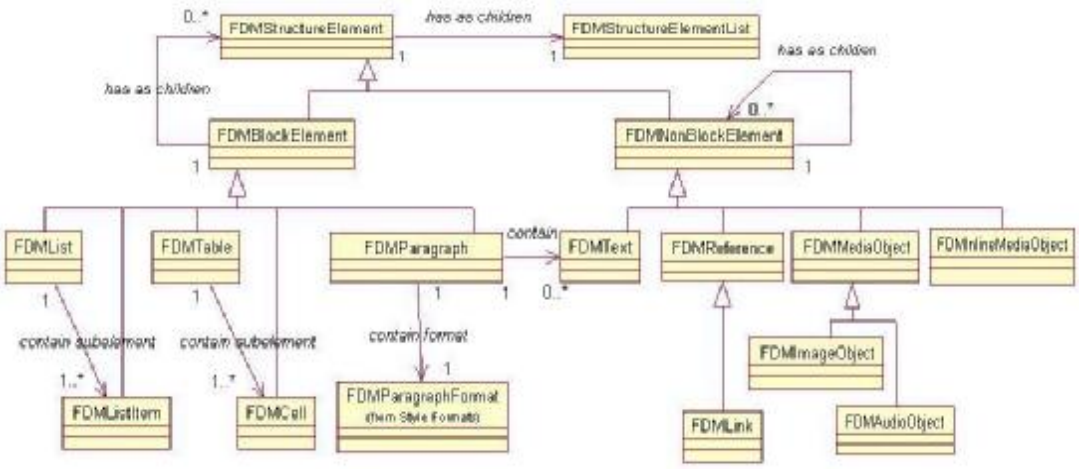
Mixed Content Model

Phrase ::=
((#PCDATA|FootnoteRef|XRef|Abbrev|Acronym|Citation|CiteRefEntry|
CiteTitle|Emphasis|FirstTerm|ForeignPhrase|GlossTerm|Footnote|
Phrase|Quote|Trademark|WordAsWord|Link|OLink|ULink|Action|
Application|ClassName|Command|ComputerOutput|Database|Email|
EnVar|ErrorCode|ErrorName|ErrorType|Filename|Function|GUIButton|
GUIIcon|GUILabel|GUIMenu|GUIMenuItem|GUISubmenu|Hardware|
Interface|InterfaceDefinition|KeyCap|KeyCode|KeyCombo|KeySym|
Literal|Constant|Markup|MediaLabel|MenuChoice|MouseButton|
MsgText|Option|Optional|Parameter|Prompt|Property|Replaceable|
ReturnValue|SGMLTag|StructField|StructName|Symbol|SystemItem|
Token|Type|UserInput|VarName|Anchor|Author|AuthorInitials|
CorpAuthor|ModeSpec|OtherCredit|ProductName|ProductNumber|
RevHistory|Comment|Subscript|Superscript|InlineGraphic|
InlineMediaObject|InlineEquation|Synopsis|CmdSynopsis|
FuncSynopsis|IndexTerm)+)

Here we can analyze two element definitions in DocBook DTD. From them, we can understand two main types of elements in document. “Paragraph” can contain a lot of types of elements, including list, table, mediaobject (image file, audio file, or media file), hyperlink, and etc. But “phrase” as a span of text cannot contain table, list. I define the former element like “paragraph” as block element and the latter element like “phrase” as non-block element. List and table are also block elements, while link,

mediaobject are non-block elements. Block element can contain block and non-block elements as sub-elements, while non-block element can contain only non-block elements like themselves.

So, in FDM, we obtain following diagram of elements.



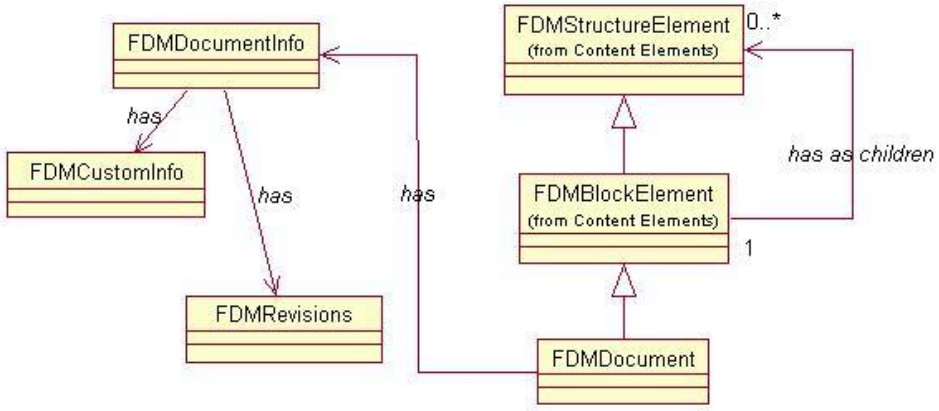
Abstract class FDMStructureElement defines basic attributes fro common document and aggregate a class FDMStructureElementList as sub-element list. Another two abstract classes FDMBlockElement and FDMNonBlockElement inherit from it. Other content element are inherit from them.

Several places should be points in this diagram:

- Children of all element are stored in FDMStructureElementList, since it is aggregate with abstract parent class FDMStructureElement.
- FDMBlockElement contains FDMStructureElement as children. That is why both block element and non-block element can be its sub-element.
- FDMNonBlockElement can only use FDMNonBlockElement as children.
- FDMList can contain any number of FDMListItem as children, while FDMListItem can contain any block elements. That is why they are both inherited from block element class. Same with FDMTable, FDMCell class.
- But since I have said before, that content elements in different document formats are really diverse, subclasses of them also need to classify with class-inheritance relationship, like we can use MediaObject as parent class of ImageObject, VideoObject and AudioObject. If we consider GUI component in document, we can define GUIObject as parent class of GUIButton, GUICheckbox.
- To detail this needs further work to analysis different proprietary document formats and summary most general definition for them. But general rule should be like what I simplify model before.

4. Document Root Element. Target document model should be a tree-structure model. There should exist a Document Root Element (FDMDocument class) which contains information for whole document. Since this element also has feature of containing paragraphs, lists, tables, it also should be subclass of FDMBlockElement.

   But root element should 1:1 aggregates with a FDMDocumentInfo class which saves the built-in and customized information for document. What is more, as an intermediate model for storage, quoting, annotating document sections and new revisions of document should be able to recorded. So, I add a FDMRevision class aggregated with FDMDocumentInfo class. Of course, information to annotate or quote or modify document should not be only stored here. But that is not main topic of this paper.



5. Navigation Element. Normally in document, client can define bookmarks, TOC (Table of Content) for document, as well as Glossary, Index, Bibliography. Such elements are using for navigating in document. I define an abstract class FDMNavigationElement for them and subclass with TOC, Index, and so on.

   Obtaining these elements is different with above content elements. Because above content element can be judged by “element type” when parsing the document content using programming API. We need to define rules for each kind of such navigation element like “text content with xxx font xxx size should be added to TOC class instance for future navigation function”. This should be a promising direction and I will discuss this in open issues.

Chapter 6

Implementation of an FDM Word XML Converter

This chapter explains how the FDM Word XML Converter implementation works. First, I will introduce the contents and file structures in the distributed package. Then, I will describe the use case scenarios and collaborations between different actors in the scenarios to help readers understand how the whole system runs. Then I will explain the commands I use and the algorithms of parsing document. In the end, I will show step by step how to run the program. For detail about the process, please take a look at the 12 minutes video demo of the implementation to get more impression.

Note:
At the beginning, I want to write a simple separate tutorial about this XML Converter implementation demo program and distribute it with the package. And actually I have done the main part of tutorial. The tutorial is written completely in XML and uses my Tutorial Builder automatically generate HTML version and PDF version tutorials by using XSLT (Apache Xalan and Apache FOP, very interesting!). But finally, I found that I am in a dilemma and I feel that readers are not easy to understand what I mean without my explanation in the former several chapters, since I have a lot of assumptions before implementation. So, I decide to move the whole tutorial in this paper and will distribute this paper with the package. Anyone who has enough knowledge can skip all the other chapters and only read this chapter directly. Thanks! For questions and further information, please feel free to contact and ask me.

6.1 Content in the Package

In preface of this paper, I have listed the required and recommend software to run this implementation. I write the whole system using Jbuilder and JDK1.2.x.

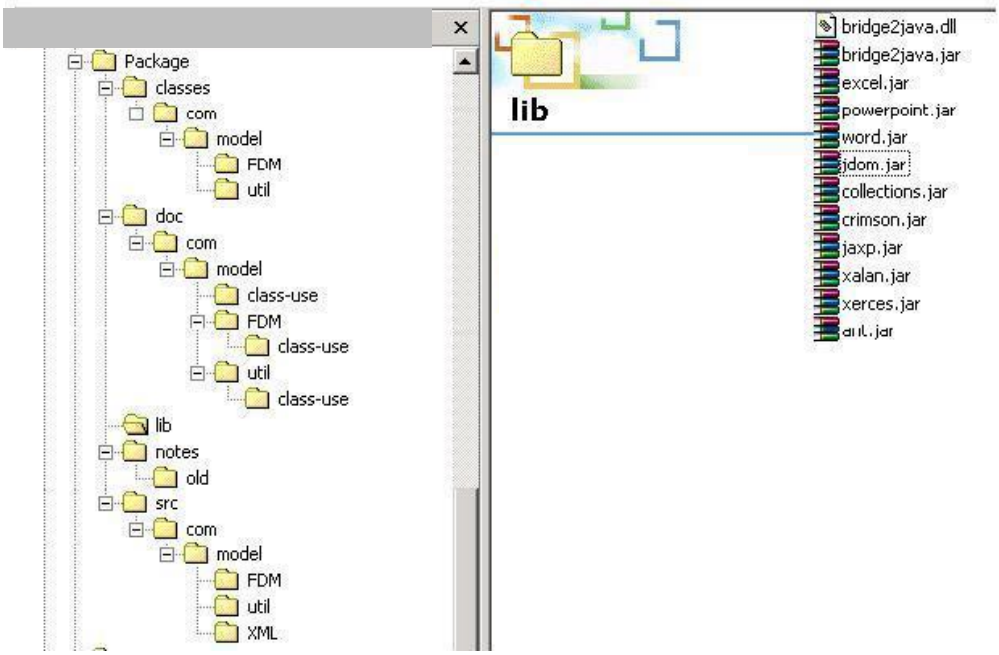


Figure 6.1 Directories of distributed package

Figure 6.1 is directory structure of the package.

- *classes* - Directory for Java binary classes
- *src* - Directory for Java source codes
- *doc* - Directory for Java API help files generated by Javadoc tools
- *notes* - Directory for documents related with this project, including sample test files
- *libs* - Directory for Java libraries used by this project
- *model* - In source code directory, model subdirectory is storing classes which run in client side.
- *util* - In source code directory, util subdirectory is for storing classes which run in server side
- *FDM* (I should use small caps) - In source code directory, FDM subdirectory is for storing FDM document model classes

The files in directory "lib" are also shown above in right frame of explorer. They are the additional Java libraries used by this project.

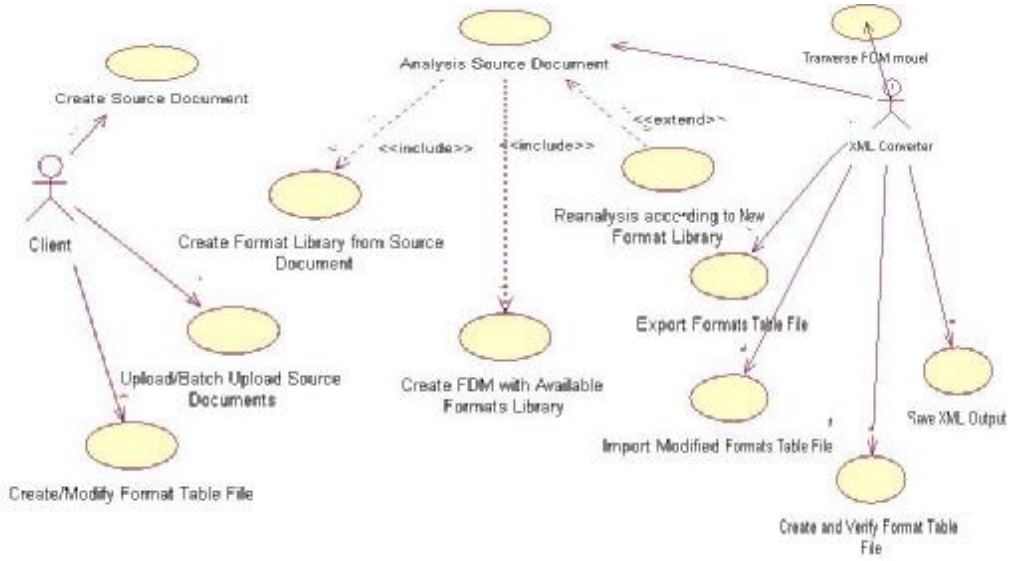
- *bridge2java.dll* - COM-Java Bridge Tools for application to call COM Automation from Java VM, developed by IBM Alphaworks
- *bridge2java.jar* - COM-Java Bridge Tools for application to call COM automation from Java VM
- *word.jar* - Library created by me for Java application to call Microsoft Word

COM library using Bridge2Java.

- *excel.jar* - Library created by me for Java application to call Microsoft Excel COM library using Bridge2Java.
- *powerpoint.jar* - Library created by me for Java application to call Microsoft PowerPoint COM library using Bridge2Java
- *jdom.jar* - JDOM API, an XML processing library specifically for Java developer
- *collections.jar* - Sun JDK1.2 Collection API for JDK1.1 environment, here used by JDOM
- *crimson.jar* - Apache Crimson, supports the Java API for XML Processing (JAXP) by prividing implementation of the following package hierarchies:
- javax.xml.parsers, org.w3c.dom, org.xml.sax.*
- *jaxp.jar* - Sun Java API for XML Processing
- *xalan.jar* - Apache's XSLT Processor, here supports JAXP
- *xcerces.jar* - Apache's XML Parser, here supoorts JAXP
- *ant.jar* - Java package building tool

6.2 Use Case Diagram

In the use case diagram that describes scenario of the converter, there are two actors-client and XML Converter. Converter is running in the server side and receives the requests from client and outputs the target files. (Not consider batch processing now, but it is easy to let the system to support that function.)



Actor “Client” has following use cases:

- Create/Modify source MS Word document

This use case describes the different approaches to get source document that need to be converted to XML.

- Upload/Batch upload the MS Word document to server

  This use case describes how to let XML converter in server side know the source file(s). Client can only send address of source file or upload the file to converter system. What's more, client can only tell converter the directories of source documents or databases where source Word documents store to request for batch processing of conversion. Which method will be used depends on requirement and implementation of real system.

  In my implementation for demo simply, client only sends source document name to converter and converter get the document from default path.

- Create/Correct formatting table MS Word document

  As discussed above, converter can separate the content and formatting information (Font information) of document. It will export Fonts information to a MS Word file named Formatting Table file.

  Client can modify the file and define her parsing rule for conversion in it, like:

  - Specific XML Tag for specific font format
  - Omit some font formats difference
  - Labels that to symbolize Stopping parsing paragraphs
  - More…

Actor "XML Converter" in server side has following use cases:

- Analysis source document

  The use case include:

  - Only create formatting collection (in Java)
  - Create FDM model (in Java) with new/available formatting collection (in Java)

  An extended use case of this is:

  Reanalysis source document according to existing formatting collection, maybe omitting some fonts information because of some rules that client defines in it.

- Export formatting table information as MS Word document

  Export formatting collection (in Java) that created by analysis the source document to MS Word file name "formatting table file" for client to modify and add rule for parsing document.

- Import formatting table from file

  Import formatting table file (MS Word format) to formatting collection (in Java), can clear/update/add to original formatting collection.

- Traverse FDM model
- Check and verify formatting table file

  Client can modify formatting table file and add rules in it. In case error and inconsistent data (like font example and font attribute inconsistent), converter can check such kind of errors. And normally all formatting table files are created from my self-defined MS Template file "format.doc". But if client incautious adds some illegal modifications on it that cannot be accepted, converter can verify

whether the client's formatting table file is legal.

- Output XML file
  According to rule of client, customize and output content and formatting information to XML file. Because DocBook DTD does not contain tag definition for formatting, I create a subset from DocBook DTD and add some definition for formatting in it to ensure XML file can be verified by my self-defined DTD.
- Transform XML to XHTML
  Use XSLT to transform XML file into XHTML to know how original document looks like.

6.3 How converter systems work?

From use case diagrams, this FDM XML converter has 2 actors. One is client and converter is running in server side. FDM_Run class is doing the job of client. After it is run, we can see output "waiting for command … …". Then client can enter the commands to enable XML Converter do relative function. FDM_Run instance will check the syntax of client entries then send valid commands to Converter.
The syntax of commands is as following, or just type "-help" then return key can see following texts.

Usage:

```
[ -export       source_word_file export_format_file ]
[ -createmap source_word_file ]
[ -maptofile export_format_file ]
[ -filetomap import_format_file ]
[ -verify       import_format_file [ sample | detail ] ]
[ -parse        source_word_file ]
[ -updatexml    xml_file ]
[ -list ]
[ -help ]
[ -exit ]
[ -quit ]
[ -test         source_word_file ]
```

6.4 Detail usage explanation

-export source_word_file export_format_file
Parse the source MS Word document and store all the font information in the instance of FDMFontMap of FDM model, then export the formatting table to MS Word file

which name “formatting table file”. (Actually we can write module to export this to external program or database.) Client can modify the file and add rules in this file.

-createmap source_word_file
Parse the source MS Word document and store all the font information in instance of FDMFontMap of FDM model. (Actually we can write module to export this to external program or database.) Client can modify the file and add rules in this file.

-maptofile export_format_file
Export the content in the instance of FDMFontMap of FDM model to MS Word file which name “formatting table file”.

-filetomap import_format_file
After clients modify this formatting table file, reimport all content of file and recreate instance of FDMFontMap class for further reanalysis document. (Function not fiexed. Need reconsider how to use)

-verify import_format_file [sample | detail]
Formatting table contains detail description of fonts and Sample text for fonts. Client can modify them. But cases of inconsistent of sample and detail may occur. Use this method to verify whether client have made error modification to file unintentionally and verify formatting table file using sample or detail description as priority.

-parse source_word_file
Parse source document and create FDM intermediate model and font collection which is stored in instance of FDMFontMap class.

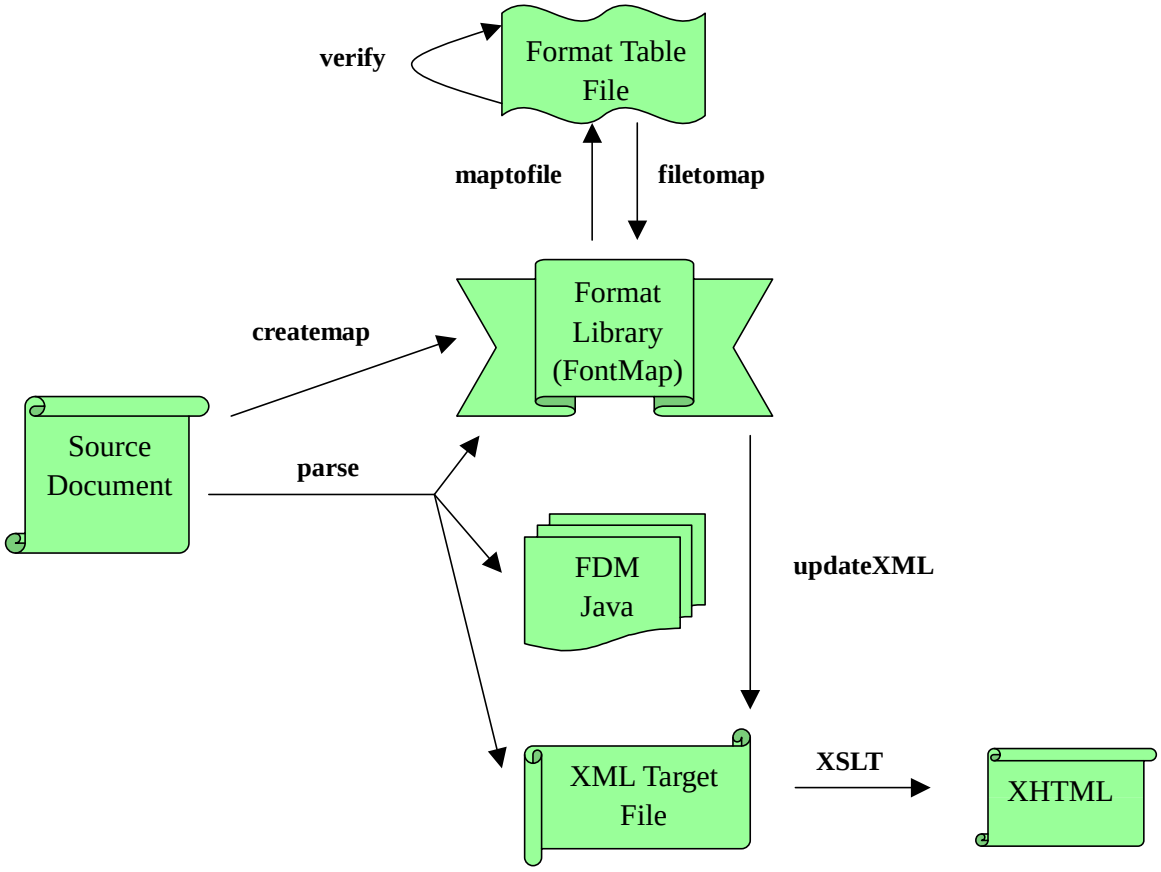
-updatexml xml_file
Update target XML file according to new formatting library.

-reanalysis (not implement)
After re-import formatting table file into FDMFontMap, reanalysis FDM model according to the new font collection and new parsing rules.

-help
Show this usage on output.

-quit/exit
Exit the demo program.

-test source_word_file
Only for testing code.



6.6 Design Patterns

This implementation uses several design patterns:

1. Façade Pattern

As known from above, client only need to run FDM_Run class which will take care of commands syntax checking work. A lot of other classes are for XML Converter and run in server side. It will be a burden for developer to make clear relation between all

classes.
Actually instance of FDM_Run class only need to access Parser class in com.model.util package and call corresponding static methods. It does not need to take care of which concrete parser need to use. See section 4.7 conversion system architecture. Conversion system supports different input and output with different formats. Each format has specific parser for it. Parser class will do the tasks of initialization, judging document LOCALE and language, judging document type (like MS Word or PDF), judging document version (like MS Word 97, Word 2000, Word XP), getting concrete Parser instance from AbstractParser class and doing corresponding task requested by client using concrete parser.

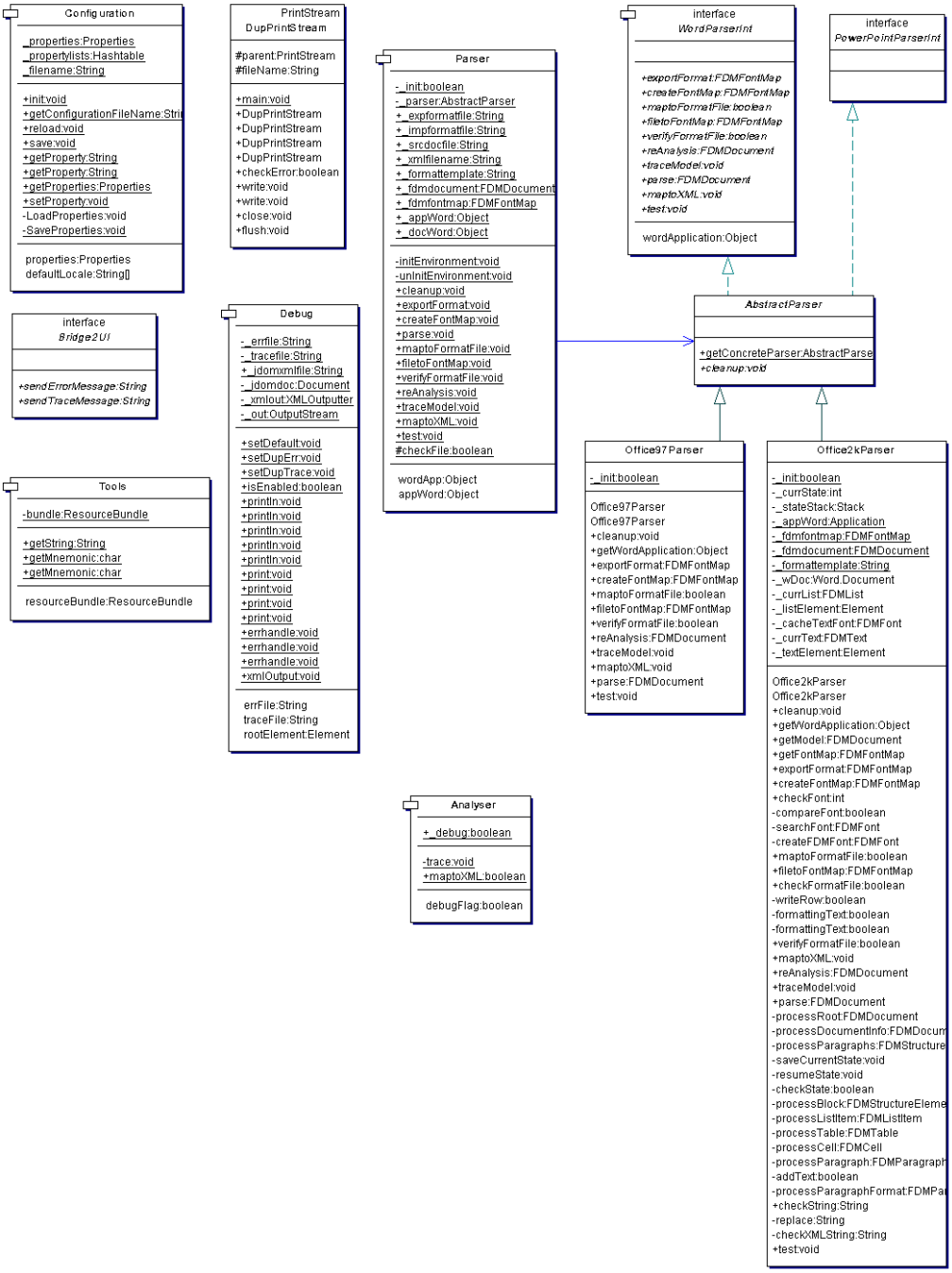
2. Interface Pattern
Now the implementation only supports Word-XML conversion. But in principle the system architecture supports multiple formats conversion, like PowerPoint-XML conversion, PDF-XML conversion. Here uses WordParser and PowerPointParser interfaces to define all methods for corresponding functions. Abstract class AbstractParser implements these interfaces. Different implementation can have uniform processing methods to use.

3. Abstract Factory Pattern
Now the implementation only supports Word-XML conversion. But Microsoft Word has different version like MS Word 97, 2000, XP and etc. Each version need specific programming API library and different implementations. Here defines an abstract class AbstractParser, and all concrete parser implementations (Office97Parser, Office2kParser) are inherited from AbstractParser. AbstractParser has a method getConcreteParser(). By judging the document version and language, return instance of proper concrete parser to Parser class. So client program only need to access Parser class and Parser class can get the proper instance of concrete implementation of parser to do the conversion tasks.

4. Java ResourceBundle class and ListResourceBundle class to support multiple language versions problem.

5. Template method pattern

6. Dynamic linkage pattern to dynamic load new format parsing implementation

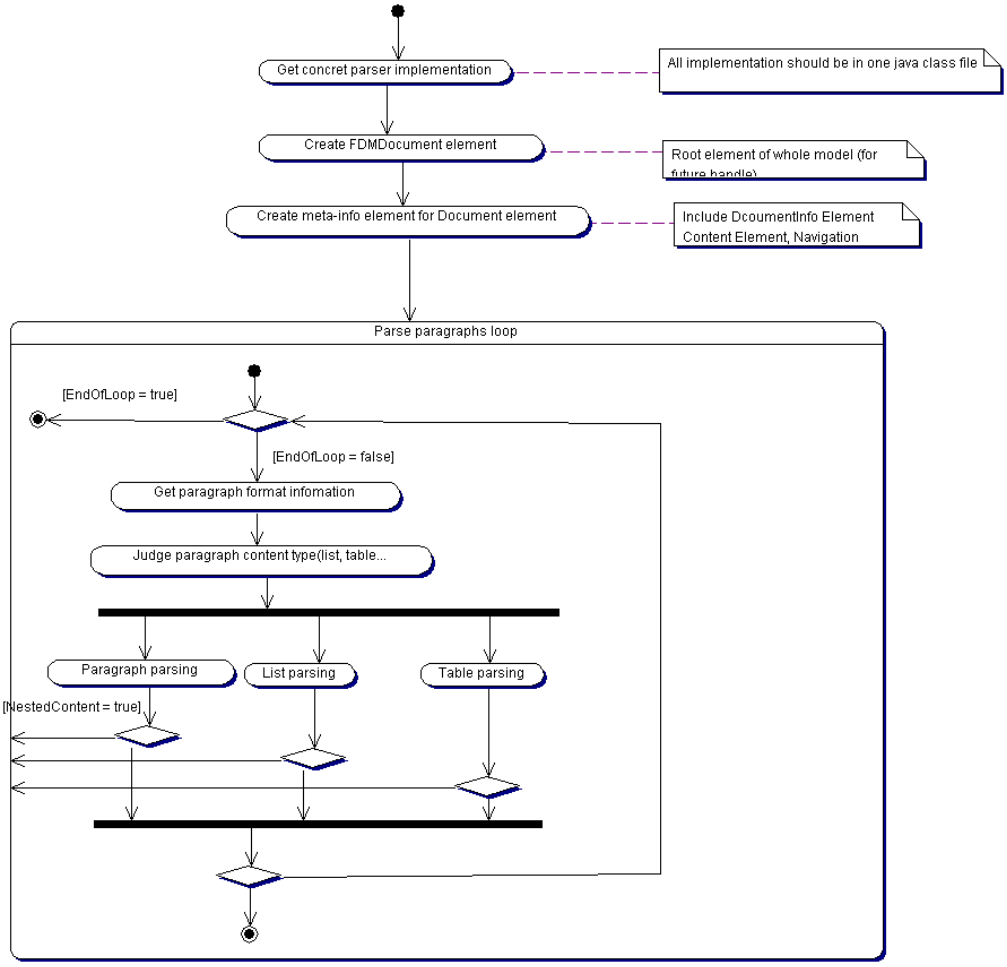


To better understand these design pattern, please refer to reference books I recommend.

6.6 Parsing Algorithm

The intricacies and limitations of the Word Object Model make this project a very interesting programming exercise, more difficult than what I expected when I first

understood the project. Of course, model provides different routines to parse and get contents. But no one is perfect. I must make some trade off.



In this section I will describe the simple parsing algorithm and leave the explanation of roadblocks of Word Object Model to next chapter for people who want to know minutia of programming API.

Idea is simple!

1. Microsoft Word Object Model has a Document Object. All content of document can be obtained from its attributes. In my parsing document process, I get the document object and create relative FDMDocument instance.
2. Content of instance of FDMDocumentInfor can be obtained from document object's BuiltInDocumentPerperties and CustomDocumentProperties attributes.
3. Then from document object I get paragraphs collection. All text contents of document are inside the collection. After that loop with paragraphs collection and check paragraph one by one.
4. When checking paragraph, one important thing is to judge what contain in this paragraph or can say, what type of paragraph it is. Since in Word Object Model, a listItem of list, a cell in table, a nested table, a cell in table or nested table, a

changing row symbol in table, a text content and etc. can be a paragraph. According to the result paragraph type, then jump to relative sub-parsing process module (list, table, text paragraph).

5. In sub-parsing module, if nested content like nested table exists, then call push current processing status in status stack, and enter new level of parsing paragraphs.
6. When parsing text paragraph, need to check whether inlineshapes exist.
7. When parsing text paragraph, if whole paragraph is using same font and style, put all content in same instance FDMText. If using different fonts formats, then parsing paragraph process will go into sentence, word, character level to find out how many font formats are existing inside and create new instance of FDMText for each font format text fragment and at the same time store font format found into font collection of FDM model.

6.7 Step by Step

Then I will demo the converter. Here are the steps. Such steps are used to demo the idea of conversion by implementing this architecure, instead of having a flexible converter on use.

I will try to point out all places needed to pay attention to when running the implementation program which I remember. You can also look at the video demo of all the processing.

6.7.1 Check Source Document

- Open Microsoft Word to see how source document looks like.
- Recommend to close Microsoft Word before running the implementation
- Source document should contain list, listitem, multiple-level listitem, table, nested table, different fonts inside a paragraph, text font with <H1> or <H2>
- In my demo, I use a file name “test.doc” as source document in directory “src”

6.7.2 Parse Source Document

- Use “parse” command to parse source document
  Example: -parse test.doc test.xml
- Parse into FDM Java intermediate model + Format library (FontMap instance) + XML target file
- The first time to load Microsoft Word takes a very long time.
- 128MB memory is not enough.

- Sometimes certain exception of Microsoft Word will happen and a COM exception of type "com.ibm.bridge2java.ComException" will be thrown. Normally this is because memory is not enough and virtual swapping memory cannot free fast enough. When such exception happen, just leave the program, and run it again. Everything should be fine. But before re-run, please open Windows Task Manager by pressing "Ctrl+Alt+Del" and end the process of "word.exe" by hand. Because that is not a Java object instance and just now the exception does not release the instance of Microsoft Word completely, even we leave the program or use garbage collection.
- User can customize and choose the trace information when parsing document. Anyway we need to know program is running, and where it goes by setting Java system property like "-Ddebug.xxxxxx".
- How to see trace information when debugging: (by adding -Ddebug.xxx to VM parameters)
    * err : detail error information
    * trace : trace processing steps
    * content : what content got from processing
    * detail : other detail debugging information
    * tag : simple result tags like XML tags
    * JDOM XML result file just for testing.
- Finish the parsing process, we get the root element instance of FDM model call "_fdmdocument", and an instance of fontmap for format library which is also linked as attribute of "_fdmdocument" and we get XML file at the same time.

6.7.3 Export Format Library to External File

- Use "maptofile" command to export
  Example: -maptofile testformat.doc
- Export format library to Microsoft Word file named "Formatting Table file" for users directly to see which fonts are found in the document without knowledge of converter system.
- Users can modify the font definitions by changing description or sample of font. 0 means disable, 1 means enable.
  Example: bold=1 means enable bold style to this font
- Users can define tags (<h1>, <h2>) for content with font. Please see how I do it in the demo. That means the text content with the font which tags with "H1" represents to users as <h1>, in my implementation, as <h1> in XHTML. This is only a simple demo of ideas. We can define more rules using this way.
- Open Microsoft Word and have a look at the testformat.doc Formatting Table File. After looking at it, I recommend close the file, but be attention, do not close the Microsoft Word. Because this will close the Word instance in the program. If we do that, when user want the system to execute a new command, a "com.ibm.bridge2java.ComException" will be thrown, because the program

founds that the Word instance is lost.

6.7.4 Verify Formatting Table File

- Use "verify" command to validate the Formatting Table file
  Example: -verify testformat.doc detail
  or –verify testformat.doc sample
- To ensure the description and sample of font are consistent with each other. Because as I have said in former step, user can modify the font definitions by changing the description. But if users only change one option, then the description and the example is not the same. So I use this command to let the user to make it consistent easily, and user need have a good knowledge of implementation.

6.7.5 Import Formatting Table File

- Use "filetomap" command to import Formatting Table file (maybe already modified by users)
  Example: -filetomap testformat.doc
- Back to converter system and update the format library (FontMap instance). Content in "Tag" column will update the tag attribute of FDMFont instance.
- Since in my architecture, format library (FontMap) is separated with FDM model tree which only uses font index for each text content. So, we can say, formatting of this document is changed. If reconstruct the original document, it will use the new font format for that part of content.

6.7.6 Update XML File

- Since now format library in the conversion system is changed (maybe), but the former XML target file has not be updated.
- Use "updatexml" command to change the formatting information in former XML result file
  Example: -updatexml test.xml
- Sorry I hard code the result file of this function. The result file is called "update.xml". I think it is better to let the users choose any name they want of the updated file or overwrite the original file, like "test.xml" in my demo.
- The change of this update will be only on the <fontlist> element and its sub-elements <font>, and the tag attribute of FDMFont instance will be assigned to "style" attribute of <font>. Sorry for that, I should use a better name like "tag" instead of "style".
- Now we finish the work of Java program and we can close the JBuilder if we use

it to resume more memory.

6.7.7 Check the Result XML File

- Now we can have a look at the result XML file, I recommend using XML Spy.
- For detail about the requirement of XML file, please read the chapter 4.
- This XML is validated by extDocBook.dtd. We can try this using XML Spy.
  -

6.7.8 Before Transforming to XHTML

- Before transforming XML to XHTML, for XSL transformation efficiency, first does a XSL transformation by using "copy.xsl".
- Objective of this transformation is to copy the value of attribute "style" (sorry! Not a good name for that, I should use tagname for better understanding) in font elements to all phrase element which using this font inside the whole XML file. (simple code, just look at copy.xsl code).
- This reason is that: if when I want to transform XML into HTML, when I meet a phrase element, I need to check the font wh
- Then the result is save as XML file, called as any name you like, in demo, I name it "update2.xml".
- Please open the update2.xml to check what is different.

6.7.9 Transform XML into XHTML

- Former result XML file does the XSL transformation by using "tohtml.xsl" to get final HTML result file and compare with the original file.
- I am using Apache Xalan as tools to do the transformation. But when doing the transformation, a message will popup, saying that I need to:

  use > to replace >
  use < to replace <
  use & " ' instead

- I know these are rules in XML to use these entities to replace the illegal symbols. But I have done several testing and found that message is not because my XML file. Some places in the original DocBook.dtd make the message appear, but I cannot find where in the DocBook makes this conflicts.
- Just comment follow statement in XML file, this message will not appear.
  Make it with XML comment <!-- -->

  <!DOCTYPE article SYSTEM "extDocBook.dtd">

Chapter 7

Programming Notes to Using Microsoft Word Object Model

This chapter explains some important points when writing programs in Java to access Microsoft Word Object Model COM library. Since it seems that there are too much need to say and it is hard to decide how detail and what level I should mention, here is only a little my personal experience, and not all. This chapter gives an introduction to the programming and some possible difficulties programmers may meet.

7.1 Document Types

This following implementation of XML Converter only uses Microsoft proprietary Word document (*.doc) as source document format.
There are many Microsoft Word versions available. The latest one now is Microsoft Word XP (2002). Since Word 97, the document formats have not changed too much and are compatible with the latest Microsoft Word version. So the following Converter implementation has not been tested on other Word document types like Word 6.0/95 for Windows & Macintosh, Word for Windows 5.0, WordPerfect 5.0 and so on.
To support other Word document versions or other document formats need to wait for newer version Converter implementation and further testing.

7.2 RTF Format

Microsoft Word also supports RTF document (*.rtf) to enable exchanging

co-recognized documents with other Word Processors.
The *Rich Text Format* (RTF) provides a format for text and graphics interchange that can be used with different output devices, operating environments, and operating systems. RTF uses the American National Standards Institute (ANSI), PC-8, Macintosh, or IBM PC character set to control the representation and formatting of a document, both on the screen and in print. With the RTF Specification, documents created under different operating systems and with different software applications can be transferred between those operating systems and applications. For detail, please see reference material of RTF Specification 1.6. [Micr99]

My implementation supports RFT format. That is because the Microsoft Word API library can automatically convert *.rtf file to *.doc when opening it by setting parameter of open() method.
Like Majix [Majix00], Some XML converter support RTF-XML conversion. They are using different mechanism. Since RTF is a standardized linear exchange formats (not like Word), those tools read the file from inputStream (in Java program) and trigger relative style processing when meet with different RTF style. They do not need to use Word COM libraries API and do not support Word conversion directly.

7.3 Type of Tools

As discussed above, for the conversion of Word to XML mainly exist three types of tools.

- Independent converters (including batch-programs) developed by different programming language like (VB, Delphi, C++, Java).
- By using macros and uses Word as XML Editor.
- Use Word extensions (VBA Plug-in) to generate XML.

The common place is that these tools need to access and use Microsoft Word Object Model by COM library.

7.4 Library

Microsoft Word Object Library is one part of Microsoft Office Object Library. It provides all program functions and enables object-oriented access to all their medium items in the form of typed objects and classes of DCOM Object model. Other programming language can access them by using ActiveX components like in VB, C++ and Delphi.

Java cannot directly use library API. It must realize that by using bridging tool for Java and COM. There are several such tools available, like Bridge2Java [Phil02] and JACOB. Sun, BEA, Microsoft and some independent developer also provide such

kind of tools.

For using Bridge2Java with Word and PowerPoint, we must create relative Java libraries from relative OLB (Object linking library file) for them. For Word 2000, this file is MSWord9.olb; for PowerPoint 2000, the file is MSPpt9.olb. By importing the library into Java code, Java application can run all functions of Word and PowerPoint.

ü JACOB Project: A Java-COM Bridge

JACOB is another tool that is developed by Dan Adler. It also has been widely used. For detail see http://users.rcn.com/danadler/jacob/

My implementation is using Bridge2Java for current version.

7.5 Some important Objects in Word Object Model

7.5.1 Application Object

To use *Microsoft Word Object Model* to access Word document, the first thing is to get an instance of Word Application and this instance can persist until the application server never needs to do conversion works and calls the *unload* function. So in implementation, I create this instance when client first call any conversion method and release when whole system exits. If client does nothing, to save time, the Word instance will not be created.

7.5.2 Document Object

Word Application can contain a collection of documents by calling *get_Documents()* method of application. This is an empty collection. We may create new document by calling *Add()*, or open a Word document by calling *Open()*. If source document is not Word, application can convert it into Word format. These actions are set by parameter. For detail, please refer to VBA Word Reference [Word00].

7.5.3 Range Object

Range object is most common in Word Object Model. It represents a continuous area in a document. Each Range object is defined by a starting and ending character position. The text range of Range object can be very different, a whole document, several paragraphs, a paragraph, a list, or a word. A range can also be several paragraphs plus several tables, as long as they are continuous areas. Just because Range is such a widely used object type, most attributes like formats (Font, Style) are obtained from Range object as properties.

7.5.4 Find Object

Find object is only used for searching. After searching, its *execute* method can return the result *Range* instance of document.

6.5.5 Selection Object

Only one *selection* object is available in a document, while multiple *range* objects can exist.

7.6 Different Mechanisms Parsing Word Documents

In this paper, I propose to parse document and map different structured content elements with relative elements in FDM model, and then this model will map all elements with XML tags. I have talked about *Style* and *Font* of document. Here I will explain more detail.

7.6.1 Style

Normally we change text by using one of two methods: by applying a style or by apply formatting. Style is not a full solution to all Word document as what I have said. But it is a good solution to a large deposit of document created by a specific Word Template. Documents have different structural elements like headline, bylines and hyperlinks. When applying a style, you just need to select the text range and apply something: a heading, a subheading, a code block, a quotation, or others. Maybe later most headlines will not be the same size, weight, or even font. But if you don't care these, you only want to know all headlines and sub-headlines in the Digital Library Systems, that using *Style* as rule for parsing will be very easy to define rule and be very efficient.

Style Object
Document instance can use *get_Styles()* method to return a collection of style objects. The Style object in Word object model represents a *Style Definition*, no matter built-in or customized styles. This is not an instance of the style in the document. This will result in a problem when parsing the document. All content in document have styles, but we cannot get them directly.

The parsing algorithm for this is by using *search* mechanism. You can search for all

instances of a specific *Style*, but cannot search for the next instance of any *Style*. So, we can create a loop for the *Style* object collection. In each loop, we use a *Find* object to search for all instances of text ranges for specific *Style* and store then in a list collection, and then we can do the same thing to another specific Style. Problem still comes out. Text ranges in result list collection are out of order. So, to solve these, we need to record the start position of range and end position of range when doing searching in the loop and finally resort all the text ranges.

To sum up, using *Style* mechanism, we need to get *Styles* object collection from document and use *Find* object and a list collection to store all instance of text Ranges (with start & end positions) and sort them in the end.

7.6.2 Structured-Based Elements

Style is a higher-level format. Inside style, we can define different formats (mainly Font format) that maybe too fragmentary to use the same search mechanism like Style. So, the parsing should be based on structural element of document. The parsing mechanism is like what I say in **section 6.6 - Parsing Algorithm**. In my implementation, root document element can obtain sub-element list by calling *processParagraphs()* method. This will be a loop which checks each paragraph type (list, table, text, hyperlink, graphic or others) by calling *processParagraph()* method. Now problems come to how to recognize and handle these structural elements.

List

In Word document, each *listitem* in a *list* is a *Paragraph* object. We can judge by using *get_ListParagraphs()* method of the *Range* object (*Range* object of *Paragraph* object). Another thing, if list exists, the number of *ListParagraphs* we get is always one. That means, one paragraph can only hold one *list/listitem*.

About the format of list item, we can obtain a ListFormat object for each listitem (Do not get it for all listitems of list, because each listitem ListFormat maybe not the same.) From ListFormat object, we can get ListValue, ListString, ListLevelNumber, ListType properties. As we know, list in Word has at least two types, with number or without number, which is decided by ListType property. So when we further use list content for presentation, first thing is to check the ListType. If it is list without number, we need not to see ListValue and ListString property, because they are not normal ASCII characters.

ListTemple is a feature only MS Word has. It is some pre-defined List format templates which can be chosen by user. Whether to include in FDM model depends on requirement.

Like I have talked above, each list item is in a paragraph. I will add all consecutive list items as sub-elements of same FDMList object. When parsing the document, we will have a loop which checks paragraphs one by one. The first time I meet a list item object, I will create a new instance of FDMList, then continue parsing next paragraph. If next paragraph is also a list item (FDMList instance is available), I will not create new instance of FDMList, instead adding FDMListItem as sub-element of FDMList. If next paragraph is not a list item, I will release the former FDMList instance. I use an State flag to control which action should be done. This process seems to be simple. But when taking account of nested-content, it will be more complex. I will discuss this later.

There is another thing unsolved. I assume all consecutive ListItems are sub-elements of a list instance. It is not good. Actually, in DocBook definition, ListItem in higher level should be sub-elements of lower level ListItem (so called nested lists). So, if need to realize that, this needs further processing to list instance to rearrange relationship between list and listitems.

Table

Tables in Word documents present a similar challenge as lists, but even worse. Because of intricate and limitation of Word Object model, until now, I still cannot find a full solution to parse tables. We have to make some trade-off to get reasonable result.

Like list, in Word document, each cell of table is a Paragraph object (To simplify explanation, assume that only one paragraph in cell. If several paragraphs, that will be more complexed.). We can judge to know existence by using get_Tables() method of the Range object (Range object of Paragraph object) when checking paragraph one by one. Same like list, the number of Tables we get is always one.

I also use a status flag to symbolize that I meet a table when parsing

There are 3 mechanisms to parse table

- When table is found, use get_Rows() to return Row objects collection. Create a loop for rows. In each row, use get_Columns() to return Column objects collection. Each column is actually a cell in table. This seems a perfect result and solution to most tables. But if table has combining cells, when accessing such cell, exception will be thrown. So, this mechanism does not support tables containing combination cells.

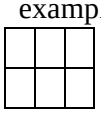
  Another important things need to note, as we known, each cell is a paragraph (Assume only one paragraph in cell now). That means if 8 cells exist, in paragraphs collection, consecutive 8 paragraphs is for this table. (In fact, that is not right number.) After we use this mechanism to parse table content by calling a processTable() method, we must omit parsing the following several paragraphs, because they are already parsed.

- Another mechanism is here. When table is found (that is, get_Tables() returns 1), we set the status flag to "processing table" and parse Range object of this paragraph, then come to next paragraph. If still found that get_Tables() returns 1, that means we are still parsing cells of table. So, parse until we find that get_Tables() return 0, change the status flag back to "processing Text Paragraph", then continue parsing.

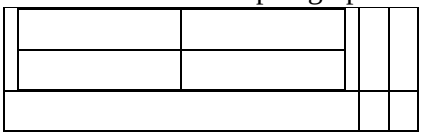
  This mechanism is that parse paragraph one by one and use status flag control when in table.

  This still seems work. But in fact, case is not simple like that.

  It is right that each cell is a paragraph. But I found that when doing debugging, after last cell of row there exists a one-character paragraph which is ASCII 7 (Control code in ASCII – "BEL", caused teletype machines to ring a bell. Causes a beep in many common terminals and terminal emulation programs.) For example, a 2*3 table (without nested table) will occupy 2*3+2=8 paragraphs.



  Well, we can assume that symbolizes change row and that seems not big problem. But if nested tables exist, if we only parse paragraph one by one, when we meet "BEL", how we know this symbolizes change row of outer table or inner table. For example, in a 2*3 table, the first cell of table contains a 2*2 nested table. Then the number of paragraphs this table uses is 2*3+2+2*2+2=14.



  If we consider that in the cells of table, some cells have text content (some are single paragraph, some are multiple paragraph), and some cell is empty, with no text content. Then the algorithm to count the number of paragraph and from this limited information to estimate the position of cell is very tricky and not applicable.

Test test	test		Test test
	test		
test		Test test	

- The third mechanism is present here. Although it is not perfect, it can solve most cases above. I define 2 statuses for parsing table, one for parsing outer table, one for parsing inner tables. When first time use get_Tables() method of Range object return 1, I know that I meet a table and change the status to parsing outer table and call processTable() method to parse table. When finish parsing this table, I continue to check next paragraph. If in next paragraph I still find that get_Tables() returns 1, that means this is still paragraph for table which has been parsed in processTable() method. I will omit this paragraph and return null as return value

and check next paragraph until I meet with paragraph whose get_Tables() returns 0.

In processTable() method, I use get_Cells() to obtain all cell objects collection from table object and create a loop to parse each cell by calling processCell() method.

MS Word is very flexible. In the cell of table, we can have any structure elements inside, including text paragraphs, images, lists, tables and etc. Here is another paragraphs objects collection like document root element's sub-elements. So, it seems obviously that we can call processParagraphs() like root element to process the nested content in table cell.

But One trick way to this is that due to limitation of MS Word Object Model, we can not use some command like ***cell(x,y).get_Range.get_Tables().get_Count()*** to know if nested-tables exist. Because in any cell of table, no matter nested-tables exist or not, as long as it is inside a table, it must return 1 (It return the outer table instance, not concerning nested tables). The only way is using command like ***cell(x,y).get_Tables().get_Count().*** So, in processCell() method, if we call processParagraphs(), we will result in endless loop parsing.

My solution is that do not directly use processParagraphs() method. In processCell(), first thing is to judge whether nested-tables exist by using ***cell(x,y).get_Tables().get_Count().*** If yes, create a loop for these nested-tables collection and calling processTable() method for each of them. NOTE: before calling method, be sure push current status into stack and after the method return pop up and restore process status. (Because the level of nesting is unsure, we need to store low-level process statuses.) If there are no nested-tables existing, we can call processParagraphs() to parse that paragraph objects collection. (If this is empty cell, this paragraph object collection is also empty.) NOTE: before calling this method, change the process status to "Parsing paragraphs in table cell" and change back after method returns. Why? Remember the condition of "***cell(x,y).get_Range.get_Tables().get_Count()"*** not work here. After change status, processParagraphs() method will omit this judgment when parsing. It works.

Until, a long explanation for parsing table is finished temporarily. This solution is perfect? No, anybody can see that if, in the same table cell, both nested tables and list and paragraphs exist. It will still be a problem. My implementation still has to assume this will not happen. (Still, there are tricky ways to solve this. But I am still not satisfactory with that.)

Text

There is no Text object in Word object model, but Range object. Because Range object can be too big or be mixed content, it does not fit in with FDM model. I use Text object as basic unit for text content in list items, paragraphs and so on. I define it as element of text content with same font format. If all contents of paragraph use same

font, paragraph object contain only one text object.

For Text object, how to parse for font is most important consideration. Font has different levels in document. “Character”, “Word”, “Sentence”, “Paragraph” or even larger block. Fix the font level impossible because different documents’ fonts vary dramatically. Fix with too low-level font like character level too low efficiency for parsing. Fix font parsing will bother client to configure too much.

In implementation, use paragraph as basic block of text for font. If inside different fonts exist, that paragraph (not all) will change to sentence level parsing; if inside some sentence contains different fonts, that sentence will change level to parsing… until character level to fix font. Same font continuous text content will be put in same **FDMText** instance object. This **self-adaptive parsing** can improve efficiency and need not configurate. If large document with same font, very fast to parse.

To accelerate parsing, normally mostly texts are same font with former text block. So, use a **cacheTextFont** variable to improve searching Font algorithm efficiency.

To demo, only consider Font name, size, bold, Italic, Underline, Forecolor, Backcolor most common properties. Easy to add more in future.

All elements in **FDMFontMap** are unique fonts. Can export the map to MS-Word document format (named Formatting table file). Client can modify and add TAG information and re-import into FDMFontMap. Can verify the client’s change to Formatting table file to ensure consistent of modification. These processes can be independent of parsing source document.

Hyperlink

This implementation does not handle Hyperlink in current version. It is not very complicated to support it. Just need to add code when calling processText() method to check the style of Range object. But for convenience, it is easier to handle a hyperlink as on consecutive text stream. Known from above, different Font formats text will be broken into various text pieces. So, we need to make some trade off that whether to break text.

Graphic

This implementation does not handle Graphic in current version. Graphics in Word document are in the same situation as styles. There are two types of graphics. One is Shape, and another is InlineShape.

InlineShape represents an object in the text layer of a document. An inline shape can only be a picture, an OLE object, or an ActiveX control. InlineShape objects are treated like characters and are positioned as characters within a line of text.

Shape represents an object in the drawing layer, such as an AutoShape, freeform, OLE object, ActiveX control, or picture.

Like Style, we can use get_Shapes() method of document instance to get all the Shape objects, and use get_InlineShapes() method of document instance to get all the InlineShape objects. And still like Style, we need a list collection to store all Shapes/InlineShape instances and sort them. Note: The mechanism to get position of them is different because Shape is in drawing layer, not like InlineShape and Style in text layer. Start/End position of InlineShape (and Style) object is retrieved from Start/End property of Range object, while Start/End position of Shape object is retrieved from Start/End property of Anchor object.

Chapter 8

Discussion and Open Issues

7.1 Current Implementation Limitation

- Not all spaces of Text Contents are preserved.
- Can not create markup like <A> aaa <B> bb </B> aaa </A>
- Styles or more formatting information support

Appendix A

API Reference

Packages	
com.model	Package to store client-side classes
com.model.FDM	Package to store all classes of FDM
com.model.util	Package to store server-side classes and utility classes

Package com.model

Class Summary	
Client_Run	Client_Run is a client class, which demo how to call classes in server side for all functions.
DefaultConfiguration	DefaultConfiguration class sets some default information here for easier extension In current version program, not use it
Version	Version class stores the version information for the project

Package com.model.util

Interface Summary	
PowerPointParserInt	PowerPointParserIntinterface define the basic functions for MS PowerPoint Parser
WordParserInt	WordParserIntinterface define the basic functions for MS Word Parser

Class Summary	
AbstractParser	AbstractParseabstract class implement all different function interfaces.
Configuration	Configurationclass • nitializati all methods for handling content in configuration file, including • nitialization, reading, saving and so

	on.
Debug	Debugclass contains utilities for debugging, error handling and other functions Feature:
DupPrintStream	DupPrintStream class tees all PrintStream operations into a file, rather like the UNIX tee(1) command.
Office2kParser	Office2kParserclass extends AbstractParser.
Office97Parser	Office97Parserclass is the implementation of parser to Office 97 documents.
Parser	Parserclass is the information class which client program can only access.

Package com.model.FDM

Class Summary	
FDMBlockElement	FDMBlockElementabstract class, for elements that can contain all other FDMBlockElement and FDMUnblockElement
FDMCell	FDMCellclass, for element that can use as table cell
FDMContentElement	FDMContentElementclass, for element that can use for content navigating, like TOC, indexing
FDMCustomInfos	FDMCustomInfosclass, for element that used to store client customized information
FDMDocument	FDMDocumentclass, root element of FDM Document model
FDMDocumentInfo	FDMDocumentInfoclass, for storing document information
FDMFont	Fontclass, for storing Font formatting information Font is most important format of document.
FDMFontMap	FDMFontMapclass, for storeing formatting information of model
FDMFormat	FDMFormat Generic class for formatting in FDM Actually formats can means a lot in document.
FDMImageObject	FDMImageObjectclass, as element for image file
FDMInlineMediaObject	FDMInlineMediaOjbectabstract class, for elements of media file, can be inherited by InlineGraphiObject, AudioObject, VideoObject and etc.

FDMLink	FDMLinkclass, for elements that store hyperlink
FDMList	FDMListclass, as element for list in model
FDMListItem	FDMListItemclass, as element for items in the lists inside document
FDMMediaObject	FDMMediaObjectclass, as element for media content.
FDMNavigationElement	FDMNavigationElementclass, as element for navigating through the document model
FDMNonBlockElement	FDMNonBlockElementclass, as element which be inherited by text, list item and so on.
FDMParagraph	FDMParagraphclass, as element for paragraph content
FDMParagraphFormat	FDMParagraphFormatclass, as element for storing paragraph format
FDMReference	FDMReferentclass, as element for something like hyperlink
FDMRevisions	FDMRevisionsclass, as element for storing document revisions information
FDMStructureElement	FDMStructureElementclass is Base element class in FDM model, which will be inherited by others element types.
FDMStructureElementList	FDMStructureElementListclass, as subelement list
FDMStyle	FDMStyle is a predefined collection of different formats.
FDMStyleMap	FDMStyleMapcollection class, extends from HashMap, used to stored formatting information
FDMTable	FDMTableclass, table element of FDM Document model
FDMText	FDMTextclass, for same text clip inside the paragraph

Bibliography

[Matt01]	Florian Matthes, Kolibri Phase III: Antrag auf Foerderung eines Forschungsvorhabens durch die DFG, Technical Report, Arbeitsbereich Softwaresysteme, TUHH, Germany, 2001
[McDo01]	Kevin McDowell, Export a Word Document to XML, MSDN Library, Microsoft co., May 2001
[WaDa01]	Helen Watchorn, Paul Daly, Word And XML: Making The 'Twain Meet, XML Europe 2001 Conference, 21-25, May 2001, Berlin, Germany
[Majix00]	Majix, an open source Word-XML Converter, 2000, http://www.tetrasix.com/
[WaMu99]	N. Walsh, L. Muellner, DocBook: The Definitive Guide, O'Reilly, 1999, http://docbook.org
[Phil02]	Bill Philips, Bridge2Java Toolkit, IBM Alphaworks, 2002, http://www.alphaworks.ibm.com/tech/bridge2java/
[HunMc02]	Jason Hunter, Brett McLaughlin, JDOM Library, Apache Foundation, 2002, http://jdom.org
[Apa02]	The Apache Software Foundation, http://www.apache.org
[Sun02]	Java Technology in Sun, http://java.sun.com
[Har01]	Elliotte Rusty Harold, XML Bibile 2nd Edition, 2001
[Kay01]	Michael Kay, XSLT Programmer's Reference 2nd Edition, Wrox Press Ltd, 2001
[HarMea01]	Elliotte Rusty Harold, W. Scott Means, XML in a Nutshell Chinese Edition, 2001
[Fung01]	Khun Yee Fung, XSLT: Working with XML and HTML, Addison-Wesley, 2001
[Mclau00]	Brett Mclaughlin, Java and XML, O'Reilly, 2000
[Dar01]	Ian F. Darwin, Java Cookbook, O'Reilly, 2001
[TVDB01]	Ed Tittel, Chelsea Valentine, Lucinda Dykes, Mary Burmeister, Mastering XHTML, SYBEX Inc, 2001
[Burke01]	Eric M. Burke, Java and XSLT, O'Reilly, 2001
[Tess00]	XML, Issue: VI.R4.M0, Tessella Support Services PLC, 2000
[W3C02]	World Wide Web Consoritium, http://www.w3c.org, 2002
[[Micr99]]	Microsoft corporation: Rich Text Format (RTF) Specification 1.6. 1999.

[Word00]	Microsoft Word Visual Basic Reference
[Rain01]	Rain1977, Cascading Style Sheet 2.0 Handbook