

# **Comparing FrameMaker and Quicksilver as Tools for Producing Single Sourced Content from XML**

University of Tampere  
School of Modern Languages  
and Translation Studies  
Henri Huhtamäki  
Pro Gradu Thesis  
November 2005

## **Acknowledgements**

Timo Wallin, Metso Automation Inc.

Kari Kovanen, DokuMentori Oy

Kalle-Pekka Hietalahti, DokuMentori Oy

TAMPEREEN YLIOPISTO  
Kieli- ja käännöstieteiden laitos  
Englantilainen filologia

HUHTAMÄKI, HENRI:

Comparing FrameMaker and Quicksilver as Tools for Producing Single Sourced Content from XML

Pro gradu -tutkielma, 82 sivua, suomenkielinen lyhennelmä 1 sivu.  
Marraskuu 2005

---

Tutkimuksen tarkoituksena on vertailla kahta yleisesti teknisen dokumentaation tuottamiseen tarkoitettua ohjelmaa yksilähteistämisen näkökulmasta, kun lähdemateriaali on XML-muodossa: Adobe FrameMakeria ja Broadvision Quicksilveriä. Tarkoituksena on antaa teknisille kirjoittajille ja tekniseen viestintään erikoistuneille yrityksille tarpeeksi tietoa, jotta he osaisivat valita oikean työkalun omiin tarkoituksiinsa. Työkalut testataan tutkimuskentän rajoittamiseksi sellaisina kokonaisuuksina kuin ne tuotepakkauksessa ovat.

Kiinnostus yksilähteistämisen toteuttamiseen on noussut yritysten ja teknisten kirjoittajien yhteydessä monista syistä, lähinnä se on rahan- ja ajansäästökysymys. Yksilähteistäminen ei ole täysin ongelmaton ratkaisu, mutta tietyissä oloissa se voi säästää selvästi resursseja.

Yksilähteistämistä tarkasteltiin työkalujen yhteydessä tutkimalla niiden rakenteisia ympäristöjä, ehdollista tekstiä ja HTML-julkaisua. Tarkoituksena oli saada tuotua Metso Automationilta saatuja XML-dokumentteja työkaluun, antaa tiedolle uusia ulkoasumäärittelyjä tuonnin yhteydessä tai sen jälkeen ja julkaista ehdollista tekstiä ja HTML:ää tästä testidatasta. XML-ominaisuuksien tutkiminen jakautui selkeästi rakenteisen ympäristön käyttöönottoon ja sen testaamiseen. Päätösentöön helpottamiseksi XML-ominaisuudet pyrittiin myös kuvaamaan mahdollisimman tarkasti. Testauksen yhteydessä kiinnitettiin myös huomiota työkalun rakenteisten ominaisuuksien dokumentaatioon.

Testitulokset voidaan kiteyttää siten, että FrameMaker on helpompi käyttöönottaa kuin Quicksilver, sen HTML-julkaisu on ammattimaisempi koska sen mukana tulee erillinen työkalu (Webworks Publisher) tähän tarkoitukseen. FrameMaker antaa käyttäjälle mahdollisuuden antaa eri muotoiluja erilaisille ehdollisille teksteille, mutta sen rakenteinen työskentelyympäristö on hieman epäjärjestelmällinen verrattuna Quicksilveriin. Quicksilverissä puolestaan on kompakti käyttöliittymä rakenteisten dokumenttien kanssa työskentelyyn, sisään tulevan XML-tiedon prosessointiohjeet ovat monipuolisesti räätälöitävissä, ja se tarjoaa tehokkaat puitteet ehdollisen tekstin tuottamiseen. Quicksilverin prosessointiohjeiden tekeminen vaatii kuitenkin Interleaf Lisp -ohjelmointikielen hallintaa ja HTML-julkaisu ei ole yhtä kehittynyt kuin FrameMakerin mukana tulevassa Webworks Publisherissa.

Jatkotutkimuksia voi toteuttaa tutkittujen ohjelmien uudemmista versioista, tai testamalla yksilähteistämistä XML-tyylikielillä ja vertailemalla tätä tapaa tutkielmassa esitettyyn yksilähteistämisympäristöön.

## TABLE OF CONTENTS

1. Introduction .....	1
1.1. Focusing on the research field .....	2
1.2. Research material and methods .....	8
1.3. Defining concepts used in this thesis .....	9
2. Single sourcing, XML and relevant conceptual issues .....	12
2.1. A look at single sourcing .....	12
2.2. A look at XML .....	20
2.3. The connections of single sourcing, XML and the researched tools .....	30
2.4. Some other relevant conceptual issues .....	31
3. An examination of Adobe Framemaker 7.0 .....	35
3.1. FrameMaker structured environment .....	36
3.2. Conditional text in FrameMaker .....	48
3.3. HTML publishing in FrameMaker .....	49
3.4. FrameMaker 7.0 documentation .....	51
4. An examination of Broadvision Quicksilver 1.6.1 .....	54
4.1. Quicksilver structured environment .....	54
4.2. Conditional text in Quicksilver .....	67
4.3. HTML publishing in Quicksilver .....	69
4.4. Quicksilver 1.6.1 documentation .....	71
5. Comparison and conclusions .....	74
5.1. Structured environments .....	74
5.2. Conditional text and HTML publishing .....	76
5.3. Software documentation .....	77
5.4. Recommendations .....	77
5.5. Further research opportunities .....	79
6. References .....	80

## LIST OF FIGURES

1. Providing layout for XML using the tools .....	5
2. Dokumentori's view of XML benefits .....	29
3. Framemaker structured environment .....	36
4. An example of a FrameMaker structure application .....	37
5. An example of an Element Document Definition (EDD) in FrameMaker .....	38
6. An example of a read/write rules document .....	40
7. FrameMaker Structure View .....	42
8. FrameMaker Element catalog .....	43
9. FrameMaker Attribute selection .....	44
10. Conditional text dialog in FrameMaker .....	48
11. XML to Interleaf processing .....	55
12. Quicksilver Implementor's Toolkit .....	56
13. An example of a Quicksilver processing definition .....	57
14. XML processing window in Quicksilver .....	59
15. Document structure view in Quicksilver .....	60
16. Inserting elements in Quicksilver .....	61
17. Setting attributes in Quicksilver .....	62
18. Search XML dialog in Quicksilver .....	63
19. Error dialog when adding a DTD in Quicksilver .....	64

# 1. Introduction

In this thesis I will focus on comparing two software used in technical documentation: Adobe FrameMaker 7.0 and Broadvision Quicksilver 1.6.1. I will examine these tools from the aspect of using their structured environments to import XML, add layout to the imported content and produce single sourced documents (using conditional text and HTML publishing), and then compare the two tools' features and documentation. My goal is to provide a technical writer, or a business specializing in technical communication aspiring to implement single sourcing in a scenario, where the source material is in XML format, with the necessary information to be able to choose which software to purchase and implement. The results do not necessarily show which tool is better for producing single sourced content from XML since the tools, as we shall later see, are quite different on their approach to providing an environment for working with structured documents. The results are, however, sufficient for providing the necessary information for making decisions based on individual needs.

One reason why I chose this subject is that businesses specializing in technical writing are interested in documentation tool research<sup>1</sup>, and this implies that there is a gap to be filled in this particular field of research. This interest also makes the practical implementation of a study like this feasible as cooperation is possible with the business' documentation experts, who may be doing their own explorations in the tool research field to update their documentation systems and processes. These experts have valuable practical information about the different tools, information such as why a particular tool was chosen over another and what the feature requirements of the particular business are.

---

1. According to a survey given out to businesses by Lassi Kaleva (Kaleva 2000).

In this thesis I will use the word “he” as a neuter pronoun, following Sidney I. Landau’s suggestion:

My own solution is to recommend that men use masculine pronouns for neutral use, because they naturally identify with the masculine gender, and that women use feminine pronouns for the analogous reason. (Landau 1989: 3)

## 1.1 Focusing on the research field

Civilization is a cumulative enterprise, and communication has always been a vital component of that cumulation process. From the fourteenth century on, the social system of science has depended on technical communication to describe, disseminate, criticize, use, and improve innovations and advances in science, medicine, and technology. (O’Hara 2001: 1)

Technical communication has existed for hundreds of years, but the main reason for its development into its modern form is largely due to the work of William Shockley, John Bardeen, and Walter Brattain, who invented the transfer-resistance device in 1947. This device has come to be called the transistor and combined with another advance, the printed circuit, has been used to create computers. Computers had a two-sided effect on technical communication: on one hand, the computers created a need for new documentation, for example software user manuals and hardware installation manuals. On the other hand, computers had a dramatic effect on the documentation work practices, leading to, for example, advanced text and graphics manipulation and printing. (O’Hara 2001: 3-4)

These changes have affected the skill levels required from technical writers. Although the work of writers, illustrators and editors still requires the same core competencies as before the electronic revolution, new knowledge and capabilities are essential as new, specialized fields in technical communication appear. (O’Hara 2001: 4) Changes in the work field have also been acknowledged by Nancy Hoft, who discusses the changes involved with interna-

tional technical communication, that is, communication that is understandable to an international audience. (Hoft 1995: 7-9) While O'Hara discusses the specialization of technical writers into different areas (O'Hara 2001: 3-4), Karen Schriever discusses the eroding of the traditional separate classification of writers and designers, and argues that today's professional must be ready to cross the boundary between writing and designing. All in all, the profession seems dynamic, increasingly challenging and growing, where the writers have to adapt new skills relatively quickly and be ready to cross boundaries between different areas of technical communication.

One of the challenges presented to technical writers is single sourcing. Single sourcing is a relatively new innovation in the field of technical communication, stemming from the need for faster and easier designing, creating, and maintaining the large amount of information needed for describing a product, for example. (Simply Written Inc. 2002: 2) This is achieved by creating the multiple media output from a single source file, which, in some cases, can save a company or a freelance technical writer money and time. Budgets and timelines are often tight, which makes single sourcing an appealing option for producing documents for multiple media. (Rockley and Hackos. 1999: 2)

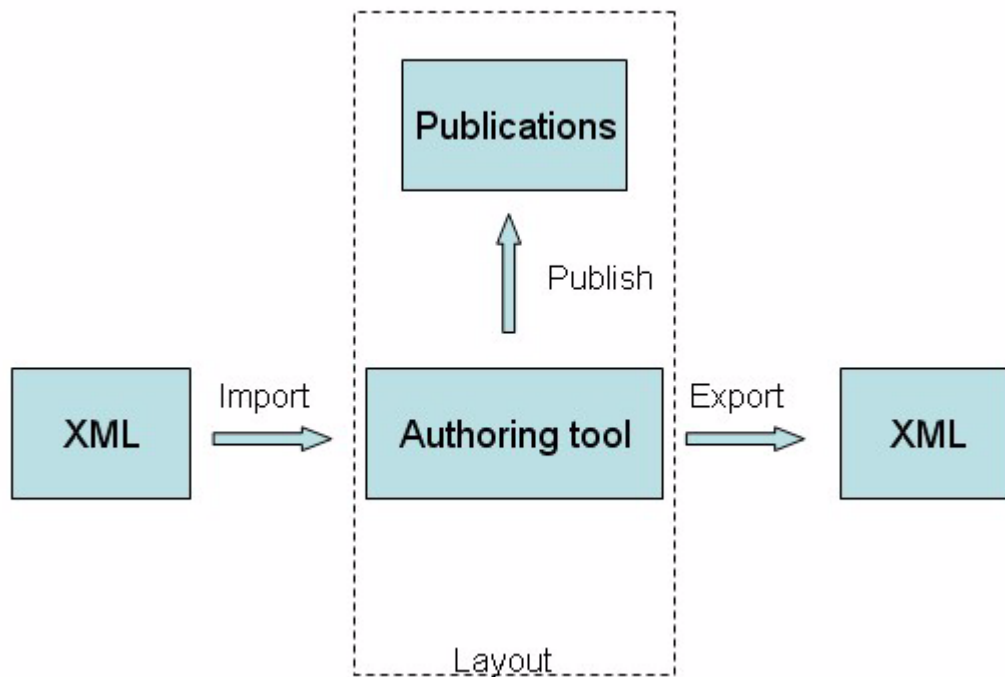
Single sourcing does have its problems, as I shall discuss later in this thesis. But I will not discuss the question whether single sourcing should be implemented or not. I will assume that people interested in integrating single sourcing into their documentation producing environment already have a need to make this change, and are looking into what tool would suit them best. Also, the implications of implementing a single source publishing system on the work of technical writers -- albeit they are very important matters to discuss before implementation -- are not an important part of this thesis, thus they will be touched on only briefly.



XML, or eXtensible Mark-up Language, provides a platform-independent way of representing information in a structured format. Structured representation is not a new concept in itself, since SGML, or Standard General Markup Language, which XML is based on, was published already in 1980, and its ancestor, the GML saw daylight in the late 1960s. (SGML Users' Group 1990) XML, however, is designed to be easier to use than SGML, which has affected the rapid growth of interest in it. One thing that makes XML so interesting in conjunction with the concept of single sourcing is that it separates the way the information is presented from the actual information content, making it possible to use different presentation styles for the same data, thus creating single sourced outputs.

It is important to note that these tools will not be handled in this thesis as tools for producing XML content. XML will be discussed since it is a central theme, and the terms and concepts of XML will be used when examining the tools. The focus is using the tools in conjunction with XML to produce output in different formats with different appearance, or layout.

The following figure illustrates the selected tools' function in the process of using XML in single sourcing:



**Figure 1: Providing layout for XML using the tools**

Technical communication employs various tools for different purposes: for producing and managing text, creating layouts, designing graphics, managing documents and creating XML, HTML or SGML content, to name a few. This thesis will focus on the field of text authoring and publishing tools that have XML support, specifically on FrameMaker and Quicksilver. The reason for choosing these tools is partly affected by the request of my current employer DokuMentori, a medium-sized business specializing in technical documentation, partly because one or both are used in so many companies' documentation departments (DokuMentori uses both and Metso Automation, which I will discuss later, uses the Unix version of Quicksilver, Interleaf) and partly because of my own interest in these tools. It is impor-

tant to note that I will refer to these tools as FrameMaker and Quicksilver, leaving out the version number to avoid unnecessary repetition. Different versions of the tools can be very unlike each other, thus to avoid any misunderstandings, it is important to keep in mind that the versions discussed in this thesis are specifically FrameMaker 7.0 and Quicksilver 1.6.1.

The intended audience for this thesis are freelance technical writers, companies or company documentation departments specializing in producing technical documentation who are interested in using FrameMaker or Quicksilver to produce single sourced documents from XML data. Although formatting can be added to XML data by using stylesheets (which is discussed more in chapter 2.2.1, under “Extensible stylesheet language” on pages 25-26), considering the field of work of the audience, it is very likely that tools for producing information products (information products are discussed in chapter 1.3, under “Technical communication” on pages 9-10) are already in use. Hence there is a possibility that existing tools can be used for this purpose, avoiding a few of the problems of single sourcing, including the costs of new tools.

I have decided not to handle matters such as the price of the tools, implementation expenses or possible implementation problems in this thesis. Costs are always interesting to know, especially from the viewpoint of the person who has to pay for the new tools, but I consider such matters trivia when the focus is on the features of the tools.

I wanted each tool to stand on their own, that is, I wanted to test only the software that comes in the product package by default. By this I mean that I left out any possible free add-on software that would enhance or add tool features. This is basically due to the fact that the research field would expand too wide if I chose to include every software that enhances the researched tools’ capabilities. Another reason for examining the tools individually is that

someone wanting to choose the right tool based on a comparison very likely does not like a myriad of tool sets where the changing of one component may have a negative effect on the whole.

It is relevant to discuss my starting experience in this field of research, since this makes all the difference when reading the test results. For example, freelance technical writers undoubtedly have few choices as to how their XML based single sourcing system is built, most likely they have to do it themselves. Even larger companies do not always have the chance of relying on XML and documentation tool experts to set the system up for them. Hence the need to discuss my own starting points as a technical writer setting up a system using the tools to produce single sourced content from XML data, for the readers to be able to relate their own experience in this field to mine, and thus benefit more from the results.

I have worked for several years as a technical writer, producing user manuals for software using mainly Microsoft Word and Macromedia RoboHelp. Before I started this research I had used both Quicksilver and FrameMaker only briefly, and the Unix version of Quicksilver in a few projects. This is relevant to this thesis, since I learned to use both the tools' structured environments at the same time while I conducted this research, making it harder for me to (subconsciously) favor one tool over the other based on my previous experiences. Also, before starting work on this thesis I had a basic understanding of HTML, but I had never studied XML. I also had never participated in any single sourcing projects, nor used a single sourcing system. During the writing of the thesis I have participated in a project aimed to create an XML based single sourcing system and I have used a functional single sourcing system built on Quicksilver for production of spare part books. As a summary, I was a novice in all of the practical sides of this research, meaning the tools, XML and the practical implementation of

single sourcing when I started working on this thesis. Basically my starting point was similar to that of a technical writer who has to set up an XML based single sourcing environment with little previous experience.

## 1.2 Research material and methods

For examining the XML import features of the tools I needed sample XML files. For research purposes, XML data that is actually used in a company's XML publishing system would be optimal, since then the testing reflects actual conditions. Metso Automation's documentation department was kind enough to provide me with the needed files. Thus, in this thesis I will use XML files and a DTD obtained from Metso Automation's documentation department as test material to evaluate the XML capabilities of the tools. Using third-party research material is also fair for both of the tools, as it provides more objective results as, for example, using the tools' own example files. I will discuss Metso Automation's use of single sourcing and the use of XML in technical writing more later.

The material obtained from Metso Automation will be used in conjunction with setting up the XML import environments of the researched software. The DTD will be used to provide the structural definitions to the software, and the XML will be used as test material when testing the XML import and export features. The rules provided by the DTD are also going to be used for purposes of validity checking within the software, and it will also provide the elements that are going to be used when working with structured documents. Common structured document features, such as adding attributes will also be tested, as well as adding layout to the imported XML data. Any additional features relevant to working with structured documents

will also be examined, as well as the documentation supporting the whole process of setting up the XML import and export environment.

The resulting documents from the XML import are going to be used for evaluating the single sourcing features of the tools, that is, the conditional content features, as well as HTML publishing.

Considering the goal of this thesis, which is to provide enough information for the audience to be able to make a decision between two tools, I will also provide pictures and descriptions of the examined features.

### **1.3 Defining concepts used in this thesis**

I have devoted the next chapter to single sourcing, XML and other important themes, as they are such large concepts, but I will be discussing some of the more general central concepts used in this thesis here to avoid lengthy sidenotes in the text. As we shall see with the first term, the definitions are also not clear cut, and hence also deserve some attention.

#### **Technical communication**

In this thesis I will follow the lead of Nancy Hoft and use “technical communication” to refer to the profession of designing information products. Information products is a very broad term that encompasses, for example, multimedia presentations, printed user guides, online tutorials, marketing brochures, technical specifications, etc. (Hoft 1995: X). It is relevant to note, however, that the term “technical communication” is held controversial by some researchers in this field. For example, Schriever argues that although the term is familiar to the insiders of this field, it may not represent the diversity of the field adequately to the outsiders.

Schiever suggests the term “document design” instead. (Schriever 1997: 9) However, I consider the audience of this thesis as “insiders” of the field, and I see no need to replace “technical communication” with “document design”.

### **Technical documentation**

In short, I view technical documentation as the information products - such as manuals and spare part books - that make the product, be it software or heavy machinery, easier to use for end users and maintenance personnel. Technical documentation can mean both the process of producing documents and the end products.

### **Technical writer**

Technical writer is someone who actually produces text for instructions, as opposed to people who design non-verbal instructions, create illustrations or produce, for example, spare part books (that often does not involve writing). Technical writers have to understand how products work and be able to communicate that information. Without proper education these skills are often mutually exclusive. Also, the information is often published through multiple media so the writers have knowledge about publishing in a variety of environments.

### **Structured vs. unstructured documentation**

Structured documents have their text wrapped into elements that are in turn contained within other elements, thus the elements form a structure within the document. This assembly enables a construction that has a hierarchical content description. Structured elements may

have attributes, also known as meta-data (information about information). (Adobe Systems Incorporated 2002 c: 5)



## 2. Single sourcing, XML and relevant conceptual issues

In this chapter I will introduce single sourcing, discuss the different levels of it, and introduce the basic concepts of XML. I will also discuss the connection of XML and single sourcing and other conceptual issues relevant to this thesis. The central themes discussed in this chapter must be examined thoroughly to get a good grasp of the research area, and provide explanations for the terms that are going to be used later when examining the tools. This chapter also explains the themes' connection to each other and further clarifies the scenario of producing single sourced content from XML using FrameMaker or Quicksilver.

### 2.1 A look at single sourcing

Single sourcing involves identifying all the needed information requirements and then creating these from a single source. As opposed to traditional publishing, where documents are assembled from chapters and sections, single sourced files are created from a single source file. The desired different information types, for example training material, marketing material and usage instructions, are all created from the same source. (Rockley and Hackos 1999: 3)

Rockley and Hackos summarize where the power of single sourcing lies:

The power of single sourcing lies in effectively reusing information -- whether it is a paragraph, procedure, or even sentence--over and over again, while changing the “glue” that holds the information together for whichever medium, audience or output. (Rockley and Hackos 1999: 3)

There are other benefits in single sourcing too, but the effective reuse of information is undeniably important. Work in single sourcing is centered around writing the information so that it

can be effectively reused in different contexts, breaking the information into smaller “modules”, again affecting effective reuse, and identifying the information according to its usage. (Rockley and Hackos 1999: 3)

Information reuse requires the documentation to be modular. Modular means that the information is cut into modules the size of which depend on the need, they may be as short as sentences, or as long as whole chapters. The needed documentation is constructed from these modules and, obviously, the same modules are reused as much as possible in different contexts.

### **2.1.1 Levels of single sourcing**

Ann Rockley identifies four levels of single sourcing (Rockley 2001 b: 2-3). The different levels and their practical uses are important for understanding the effective usage of single sourcing.

#### **Level 1: Identical content, multiple media**

This is the most basic level of single sourcing, where efforts are not made to differentiate the representation or content of the information considering the differences in the multiple media formats. An example of single sourcing on this level is converting a paper-based document into PDF format. Considering that the different formats share identical content and representation, valid concerns have been expressed as to whether this is an effective way of single sourcing documents. (Rockley 2001 b: 2) For example, a paper-based document containing large picture files for printing is not suitable to be converted to HTML as such. The large files

can make the HTML heavy to use over the Internet, and the pictures are also often too large for the screen.

Hackos and Rockley also discuss this level of single sourcing, arguing that it is not an effective use of it, since identical content does not suit multiple media. But Hackos and Rockley also go as far as to argue that "this type of 'single sourcing' is actually conversion, not single sourcing." (Rockley and Hackos 1999: 3)

## **Level 2: Static customized content**

The second level of single sourcing utilizes more effective ways of designing information for different outputs. This level output has two types of information: the core information is still shared by the different formats, but they also have format-specific information designed by the technical writer. The "static" part means that the user cannot change the information according to his needs, changes to the content are made by the technical writer.

An example of single sourcing on this level is creating information for multiple media, taking the specific media needs and capabilities into consideration. Hence, for example, a document published both for paper and web would have different sets of pictures for the two media. Another example of single sourcing on this level would be multiple audiences. Different audiences have different needs and knowledge levels, thus customized documentation to meet these requirements is appropriate. For example, an engineer who is responsible for installing and maintaining a product needs much more detailed information than a typical end user, thus two different manuals are required, since the extra information needed by the engineer can actually make the product harder to understand for the typical user.

Another important aspect of this level is creating different information products, as Rockley calls them (Rockley 2001 b: 2), from a single source. These information products include user guides, training material and marketing material. This type of cross-departmental material single sourcing is a good example of information reuse within a business. Instead of every department creating its own needed information, it is single sourced from the same core information. The materials deal with the same product(s), only the view and needs change.

### **Level 3: Dynamic customized content**

While level 2 produces content that cannot be changed by the user, level 3 provides content that is customized according to the user through a user profile or his selections. The information is retrieved from a database according to pre-designed user profiles from which the user can choose, or according to the user selecting what he wishes to view. The information can also be offered according to the user's answers to questions presented to him. (Rockley 2001 b: 3)

### **Level 4: Electronic Performance Support System**

This level of single sourcing, Electronic Performance Support System, or EPSS, is based on level 3, Dynamic customized content, but here the information is offered to the users when they need it. This means that the EPSS tracks the user's progress within the product (e.g. software) and determines what information he might need at any point. This level is appropriate when the users can be defined quite accurately beforehand, and is not suitable for a product that has a wide audience, because of the scope of information needed by the EPSS. (Rockley 2001 b: 3-4)

The features of the documentation tools tested in this thesis enable level 2 single sourcing in Rockley's categorization. Levels 3 and 4 are more advanced levels of single sourcing, requiring a real-time access to a database holding all the necessary information, and level 4 also requiring a special software to monitor user movements in the product.

### **2.1.2 The benefits of single sourcing**

The benefits of single sourcing concretely show why this is such an important topic in the field of technical communication, and at the same time why single sourcing was an interesting viewpoint from which to approach the tools. Hackos and Rockley identify the following cost-saving benefits of single sourcing:

- It eliminates redundant or repetitive information
- It improves consistency across a documentation set or library
- It reduces errors when information is updated; instead of having to update several separate documents, writers only update the single source file from which the documents are created
- It improves staff productivity by eliminating both rewriting and clerical tasks
- It frees writers to focus on content instead of format
- It allows users to customize their own information set
- It reduces the time it takes technical reviewers to review content; instead of reviewing several separate documents, they only need to review the single source document (Rockley and Hackos 1999: 5)

These alone are enough to explain the interest in single sourcing. Large documentation companies eventually run into problems with the consistency of the documentation, for example the online and paper versions may have differences due to the fact that they are not generated from the same source. In other words, when the documents are updated, the same changes have to be made to all the different versions, thus the possibility of errors increases and even-

tually the documents are bound to have differences, not to mention that it is much more time-consuming.

Single sourcing can also be used in a cross-departmental way: for example the marketing, training and documentation departments can get their material from the same source. The product information is stored in a database, from which the workers retrieve data that is relevant for their purposes or, for example, the marketing department can programmatically retrieve any block of information that is marked as marketing information. The aforementioned block can be designed so that it can also be used in product documentation, hence it is also marked as a documentation block. This is a good example of the efficiency of single sourcing; all the redundant information in the different departments' output is identified and reused instead of rewritten every time.

### **2.1.3 The problems of implementing single sourcing**

Hackos and Rockley identify these as the initial costs of implementing single sourcing:

- Reorganizing, restructuring, and redesigning materials
- Retraining staff in single sourcing principles and techniques
- Acquiring and learning how to use new tools (Rockley and Hackos 1999: 5)

Processing the existing material to suit single sourcing can be an arduous task, especially if there is a great deal of it. Every document and manual that is included in the single sourcing system must be cut into small modules, while concentrating on creating modules that can be applied in many instances, thus reusing the material. Anyone doing this has to have a good understanding of the whole material, otherwise deciding which parts can be reused and how they have to be written in order to be able to use them is quite hard. Hence, even if a new

employee is hired for this purpose, this takes much of the company's existing technical writers' time since newly hired employees do not have the same knowledge.

After single sourcing has been implemented, new information cannot be freely written any more, it has to be planned and designed beforehand so that it can be reused. Besides this, the staff has to be trained to understand the principles of single sourcing and to be able to use any new tools that single sourcing brings with it. Single sourcing is a great change in the ways of working, and this change can even be resisted by the employees, which brings new problems to the process.

#### **2.1.4 The effects of single sourcing on the role of a technical writer**

The role of a technical writer is a wide concept, since the tasks of a technical writer vary from company to another. The changes that single sourcing brings mostly affect technical writers' specialization into different areas, supporting O'Hara's view of the recent developments in the role of a technical writer, which was discussed in chapter 1.1, on pages 2-3. An example of the impending changes is related to the fact that single sourcing separates the content from the representation. Before single sourcing a technical writer had to both write and publish the information in various formats. If the content and representation are separated, an effective way of producing the documentation is to separate the writing and publishing processes to separate people, making one an expert in producing the content while the other focuses on publishing. Also, the separation of core content and the content that is customized provides another chance for specialization: others focus on core content while others write the customized material. (Rockley 2001 b: 4)

### 2.1.5 An example of the need for a single sourcing system

Here I will discuss the documentation needs of the corporation that provided the example XML files and the DTD for this thesis. The manager of the documentation department of Metso Automation, Timo Wallin, has over the past few years realized that many of their problems could be solved by implementing a single sourcing system based on XML, progress towards which has gradually begun. When finished, it will be a level 2 single sourcing system according to Rockley's categorization, utilizing cross-departmental information reuse.

According to Wallin, the main reasons for the need of this system are:

- the need for easier publishing to multiple media
- the need for easier information reuse between different writers and departments
- the need for enhanced document coherence
- the need to be able to integrate documentation to Metso's documentation structure

The documentation department produces documentation in, for example, both paper and HTML formats. A single sourcing system as described above would enable publishing both formats easily, using different sets of pictures for paper and HTML versions.

The documents are written both by software developers and technical writers, some documents having more than one master version in use. The single sourcing system will have centralized core data to prevent multiple versions of one document, it will make document updating easier for different writer groups, and make information reuse easier for different departments.

When finished, the single sourcing system of Metso Automation is going to be a high-end, effective and cost-saving system for multiple media publishing and information reuse. The effectiveness and cost-savings come from the fact that it will use customized content for



different publications, there will be a cross-departmental information reuse system and the manual work needed when publishing in different formats is reduced considerably.

## 2.2 A look at XML

XML (eXtensible Markup Language) has a feature that makes it particularly interesting in conjunction with single sourcing: XML separates content from representation, as was mentioned in chapter 1.1, on page 4. This enables the writer to retain the core information in XML documents, while using stylesheets, or word processing tools equipped with XML support, to produce the single sourced content for multiple media. XML is also platform-independent and application-independent, which makes it a versatile and, considering possible future needs, a safe format for information. Even though I will not discuss the usage of XML in single sourcing in this paper, I will discuss the evolution and main concepts of it to give a more broad view of the concept, to clarify some terms used later in tool examination, and to explain the interest in using XML as a file format.

XML is based on SGML<sup>1</sup> (Standard Generalized Markup Language) as is HTML (Hypertext Markup Language). SGML is an ISO (International Standards Organization) standard for text markup, which was approved in 1986<sup>2</sup>. (SGML Users' Group. 1990) SGML provided the authors with great power over their documents, but it was very complicated to use. HTML is not as complicated as SGML, but it offers little of the features that SGML does.

---

1. The goal of SGML was to create a “standardized method of describing and creating structured documents independent of any hardware, software, formats, or operating system.” (Adobe Systems Incorporated 2002 c: 5)  
2. SGML's roots, meanwhile, are in the work of Charles Goldfarb, Edward Mosher and Raymond Lorie, who created GML (General Markup Language) in 1969. (SGML User's Group 1990)

Hence the XML standard was developed, where the powerful SGML features and HTML easy usability meet, to bridge the gap between HTML and SGML.

Discussion of combining the simplicity of HTML and extensibility of SGML begun in 1996, and in 1998, the first edition of XML, version 1.0, was published by the World Wide Web Consortium. (Sankaranarayana 2001) Whereas HTML provides a set of specific tags, between which the information is inserted, XML allows the user to create his own tags. This means that the user can tag information according to content rather than format structure. This enables, for example, easy conditional publishing. (Manning 2002) Conditional publishing will be discussed further below, in chapter 2.4.1 on page 31. For example, an author may want to add a tag named “internal” to XML data according to the type of information, all data that is meant only for internal versions of a document is inside the tag “internal” and is ignored when publishing an external version. I will discuss this more below in connection with metadata in chapter 2.2.1 under “Metadata in content management” on page 24.

Next I will discuss the main concepts of XML that I find are the most relevant to my focus.

## 2.2.1 The main concepts of XML relevant to this thesis

The concepts discussed in this chapter are good to know and understand since most of the concepts defined here will be used later on in this thesis.

### Document Type Definitions and schemas

Document Type Definitions (DTD) and schemas define the structure of XML documents. However, they are a little bit different in that schemas enable very strict rules. I will discuss DTDs first and then briefly explain the differences of a schema compared to a DTD.

DTDs consist of entities, elements and attributes. Here are a few sample lines of what a DTD may look like:

#### Example 1: DTD sample lines

```
<!ENTITY % info.elem 'create_data, modify_data, check_up_data, approve_data, translation_data?, version_data?>  
<!ELEMENT book (book_info?, title, part)>  
<!ELEMENT book_info (%info.elem;)>  
<!ATTLIST book %block.att; lang CDATA #REQUIRED>
```

These sample lines are from a Metso Automation DTD defining the structure of a book. The first line defines an entity, which is a static variable that can be used through the rest of the document without repeating its contents. The next two lines define two elements, “book” and “book\_info”. These element names are the tags that are used in XML to enclose information in. The DTD also defines what elements can or must be inside the defined elements, for example the element “book” can have a “book\_info” element and must have one “title” element and one “part” element. The last line defines attributes for the “book” element, the “%block.att” is

an entity reference, while the “lang” is a required attribute that will contain language information.

This DTD would be then used to control the creation of XML data so that, for example, “book” element would not contain any other elements that are defined, and that every “book” element has the required “lang” attribute. The checking of XML data against a DTD is called validation, which I shall discuss more below in section “Valid vs. well-formed XML” on pages 26-27.

A schema is basically the same as a DTD, as in that it defines the structure for XML documents, but schemas are written in XML and they allow more strict rules. For example, a DTD can define that an element is optional, it has to appear once, or once or more. A schema, on the other hand, can define that an element may appear, for example, five times. Since schemas are written in XML, XML editors can check their correctness and they can be transformed with XSLT (see section “Extensible Stylesheet Language” below on pages 25-26).

Since DTDs and schemas keep the information in control, every document created using them has the same structure. They assist the user in writing, since the structure is planned beforehand, thus the writer can focus solely on writing the content. When a writer opens a structured document in, for example FrameMaker, the provided structure definitions show what elements can be inserted where, and the program highlights elements that are in the wrong place. Thus, the writer does not have to think about the structure any more, and can focus solely on creating the content.

## Metadata in content management

When discussing single sourcing, I mentioned that an effective single sourcing system requires the information to be cut into small modules that are reused in different contexts. The problem with maintaining these modules is that a large single sourcing system may eventually have tens of thousands of these modules, considering that they can be as small as a sentence. The answer to this is metadata, or data about data. Metadata can be added to XML in the form of tags or attributes. This metadata can be added to modules to be able to keep track of, for example, what product the module belongs to and what version of the product, what language it is in, where it can be used, etc. (Rockley 2001 a: 2) Metadata is not solely for module identification purposes, it can be used to differentiate, for example, the internal and external information in a document so that publishing different versions is possible.

## Scalable Vector Graphics

Scalable Vector Graphics (SVG) is a language for describing two-dimensional graphics and graphical applications in XML. What makes SVG so interesting is that the data is saved in text format (XML), thus one can, for example, edit an SVG picture in a text editor, and it makes version control easier. SVG enables data-driven (an SVG picture can dynamically change according to its source data), interactive and - again relating to single sourcing - personalized graphics. (Fibinger 2002: 2) Also, SVG is the first vector graphics<sup>1</sup> format that web browsers understand, thus one can zoom in and out of an SVG graphic and the quality of the

---

1. Vector graphics consist of mathematical functions, instead of just dots (bitmap graphics). In other words, a line in vector graphics is drawn from point X to Y, thus only the start and end points need to be stored.

picture never degrades. Practically, SVG can be used to programmatically create pictures from text, for example create pictures from a database.

### **Extensible Stylesheet Language**

The two tools examined in this thesis are by no means the only way to produce single sourced content from XML. An alternative way is to use Extensible Stylesheet Language. I will discuss the relation of these different approaches to single sourcing more in chapter 2.3 on page 30.

Extensible Stylesheet Language (XSL) consists of three parts: the XSLT, the XPath and the XSL-FO. (W3C) I will discuss these, and also Cascading StyleSheets (CSS).

XSLT stands for XSL Transformations. XSLT is not used to provide style to an XML document, but it is used to edit a document programmatically; for example change document structure, delete elements in a document, add elements to the document, or remove all the tags in a document making it a normal text document. (Ruini 2001)

XSLT is important when the focus is on single sourcing and XML, since XSLT can be used to publish documents from a single source. For example, since XSLT makes it possible to affect the XML documents' elements, one XSLT code can be created to produce a document from the source document that includes only the elements that have an "internal" attribute, and one XSLT code can change the element names to HTML tagging, producing an HTML document.

The XML Path Language, or XPath, is used by the XSLT to refer to the elements in an XML document. Also, to support this it has basic features for manipulation of strings, numbers and booleans (logical operators). (W3C 1999)

XSL-FO stands for XSL Formatting Objects, and it can be used to provide an XML document its style properties. XSL-FO actually consists of the formatting objects and formatting properties. Formatting objects provide such typographical concepts as page and paragraph. Formatting properties, on the other hand, provide the finer points of style, such as indents and word spacing. The XSL-FO style definitions are embedded in the XML document itself, so no external stylesheets are used when creating style with XSL-FO. (W3C 2001)

While the XSL-FO style definitions are embedded in XML, an external stylesheet, the Cascading StyleSheet (CSS) is used as a separate stylesheet to XML. CSS offers less of the features that XSL-FO does, but it can also be used with HTML. CSS can be used to define the font, color and layout.

### **Valid vs. well-formed XML**

As mentioned earlier, XML validation is the process of comparing the XML structure to the DTD's definitions to see if the XML adheres to the DTD, i.e. whether it is valid or not. Well-formed XML, on the other hand, is XML that adheres to the rules defined in the XML 1.0 specification document. These rules define, for example, that nested elements may not overlap, which means that an element's starting and ending tags must be on the same level of document hierarchy. (Johnson 2000)

For example, the following XML lines are not well-formed:

**Example 2: Ill-formed XML**

```
<element A>  
  <element B>  
</element A>  
  </element B>
```

These lines are not well-formed, since the ending tag of element B is not inside the element A's tags, supposing that element A is the parent element of element B.

### **2.2.2 The key benefits of XML**

I touched this subject briefly in the introduction to XML, saying that XML suits single sourcing so well because it separates the content from the representation. But to give a broad enough view of XML, I will briefly also discuss other benefits, which may themselves explain the raising interest in using it, and explains why the source material for single sourcing might be in XML format. The main benefits are:

- Easy exchange of information
- Ability to tailor markup language
- Self-descriptive data
- Structured and integrated document (Holzner 2001: 27-29)

Easy exchange of information relates to the fact that the data is stored in text format. This enables platform independent information exchange. XML files are also small in their size when compared, for example, with a Word 97 document. Thus the text format is actually more economical disc-space-wise than the Word binary format. (Holzner 2001: 27)

Markup languages can be tailored to suit different needs. Hundreds of tailored markup languages have already been standardized, for example the Interactive Financial Exchange



(IFX) and Bank Internet Payment System (BIPS). XML can also be easily extended with XML extensions. (Holzner 2001: 27-28)

XML information is self-descriptive, which means that the data itself holds metadata about the information content of the elements. This causes the data to mostly be self-documenting. For example, the function of an element named “Customer” needs no external documentation to be understandable. (Holzner 2001: 28)

XML can define the structure of a document and rules how its elements are integrated to another document. This is important when handling important and complex information. For example, the DTD can define that only a certain type element may exist inside a certain element. This is much different from HTML, since the HTML browsers repair many mistakes in the language. XML browsers, on the other hand, check both the validity and well-formedness. (Holzner 2001: 28-29)

### **2.2.3 An example of XML usage from DokuMentori**

Here I will show a practical example of XML usage with a customer to assist in understanding the actual use of XML.

DokuMentori created a documentation system based on XML that enabled single sourcing for one of its customers. The idea was for the customer to be able to create documentation from existing information and from a single source.

The figure below illustrates DokuMentori's view of the process benefits for the customer:



**Figure 2: Dokumentori's view of XML benefits**

When in use, the XML based documentation system would affect the quality of end products and the production time, leading to a more satisfied customer, cost savings and a sound company image. These were the basic premises for building the system.

The initial step was to create models of the customer's documentation and to create a new DTD model to which all of the documents would from there on adhere. The DTDs improve consistency between the manuals, affecting quality and coherence from the beginning. The second step was to break down the information into modules, in other words, into pieces of information that (when combined) would form the documents. These modules were stored into a database, from which the author can call for these modules when needed. The third step was to create metadata that, for example, describes what a module contains, what is the module's version, what language it is in, etc. to help in identifying the modules. After these

steps a theoretical model for an XML based single sourcing enabling documentation environment had been created. By theoretical I mean that it was not yet functional, but in theory, the functionality was already there; an author could create documentation by selecting modules he wanted to include in the documentation, and using XML filters based on the metadata included in the XML data, filter out the needed modules from the database.

### **2.3 The connections of single sourcing, XML and the researched tools**

As I discussed earlier in chapter 2.2.1 under “Extensible Stylesheet Language” on pages 25-26, XSLT can be used to affect the content of documents and XSL-FO or a CSS can be used to format that content, thus a combination of XSLT and XSL-FO or CSS is entirely sufficient to produce single sourced documents from an XML source document, where both content and layout is taken into consideration when designing the different outputs. However, I argue that it is more effective to use the existing authoring tools, such as FrameMaker or Quicksilver, (if the existing tools are capable of this, obviously) to provide the layout and publishing formats. This is related to some of the problems of single sourcing discussed earlier in chapter 2.1.3 on pages 17-18. Since the technical writers of a company are already used to working with the existing tools, it reduces the amount of training needed, saving money and time. And, since the tools that the technical writers are familiar with are used, this is a smaller change in the working ways for them to adapt to, reducing problems that come from people resisting change.

## **2.4 Some other relevant conceptual issues**

The following three areas will be examined separately from the structured documentation features of the two tools, hence, they are discussed here separately to clarify the structure of this thesis.

### **2.4.1 Conditional publishing**

When discussing the different features of XML, I briefly touched the subject of metadata in XML. Metadata can be added to XML in the form of tags or attributes, that is, element attributes or tags (i.e. element names). Conditional publishing depends on metadata, and it means publishing with a certain condition. Everything that meets this condition will be published and everything else will be left out. For example, certain elements in a document may have the attribute “internal”. When the writer decides to publish an internal version, nothing is removed from the document. But when he publishes an external version, all the elements with the “internal” attribute are ignored. This one kind of simple single sourcing, publishing different versions from the same core information.

### **2.4.2 HTML publishing**

Since this thesis examines the tools from a single sourcing point of view, the publishing formats are obviously quite important. In addition to printed documentation, PDF and HTML are also quite important formats. However, since PDF creation is handled by another program (Adobe Distiller), it will not be examined in this thesis. Since the Internet revolutionized electronic communication, HTML has become one of the most important formats for documents, for it is an effective way to offer customers easy access to documentation over the Internet.

I will examine the HTML publishing features of the tools by publishing the Metso sample project to HTML. I will focus on the following features:

- HTML + CSS publishing
- Structured element mapping to HTML styles
- Automatic splitting of structured documents into smaller HTML documents
- Picture conversion

By HTML + CSS publishing I mean the ability to publish the HTML with its styles in a separate Cascading Style Sheet. The separation of the style from content offers more control over the HTML, enabling, for example, easy style changes for the whole HTML with just one change in the stylesheet. Structured elements have to be mapped to HTML styles, and this basically means that for every structured element there is an HTML style match defined. This allows, for example, the mapping of heading level 1 to a “H1” HTML style. The automatic splitting relates to the usability of HTML. The documents in FrameMaker and Quicksilver can be hundreds of pages long, but HTML has to be split into small sections so that it can be loaded quickly and thus viewed effortlessly over the Internet. Also, pictures have to be converted to common formats that most Internet browsers understand<sup>1</sup>, and I will examine whether it is possible to affect the picture conversion, again relating to quicker downloading through changing picture size or quality. In addition to these, any other relevant features that the tools may have will also be discussed.

---

1. The most common picture formats on the Internet are GIF and JPEG (Zhang 1995).

### 2.4.3 Software documentation

Considering the audience of this thesis, which includes technical writers wanting to implement XML authoring, documentation quality may prove very important, since not all technical writers are XML or XML authoring tool experts. This does not mean, however, that a technical writer may not have to learn to work with XML documents. As discussed before in chapter 1.1, on pages 2-3, technical communication is a dynamic and increasingly challenging profession that may require new knowledge and capabilities of the writers as new challenges are presented. As outside help for implementing new documentation systems can be expensive, due to company documentation budget constraints and the fact that existing employees already have a good knowledge of the information products, a technical writer may very well be appointed to implement a documentation system that can also handle XML data.

Considering the focus of this thesis, I will narrow the examined documentation to the parts handling the structured environments of the programs. I will not discuss such issues as documentation legibility or language quality, for example, since they have less effect on how easy the setup of the aforementioned environment is.

When examining the software documentation, I will be mainly focusing on the following aspects:

- use of illustrations
- are the used terms explained
- do the instructions identify a user level
- are workflows provided
- is the information provided sufficient

When discussing the use of illustrations, I will focus on whether there are enough illustrations to assist the user and whether the instructions illustrate both the user interface and the

processes and general environment through pictures and diagrams. The used terms and concepts have to be explained in the manual, since working with structured documents involves a lot of new terms, concepts and acronyms for beginners. Since users have different needs from the documentation depending on their experience, the documentation must have some method of identifying the information meant for the different users. For example, the documentation can be divided into parts aimed at different users. Workflows are equally important, since the setup of the structured environments can quite complicated, especially for a beginning user. Simple workflows guarantee a working outcome, as opposed to just explaining the different features of the tool and then letting the user figure the actual use out by himself. The ISO/IEC Guide 37, Instructions for use of products of consumer interest, provides a general principle of good documentation that also summarizes the last point in my focus on the documentation: “Instructions for use should...contain all information required for correct and safe use of the product and/or for service and maintenance.” (ISO/IEC 1995: 1)

### 3. An examination of Adobe Framemaker 7.0

FrameMaker has a long history, with the first version seeing daylight already in 1986. Created by Charles Corfield, a mathematician at Cambridge University, the first version was a WYSIWYG<sup>1</sup> text editor for a Sun 2 workstation. By 1995 FrameMaker had reached “its zenith ... at that point, Frame's client base consisted mainly of large companies, many of which owned thousands of licenses.” (Dan Emory & Associates). Adobe FrameMaker 7.0 was released in 2002 and is the first FrameMaker version which incorporates both the structured (previously +SGML) and non-structured (previously FrameMaker Standard) versions of FrameMaker. Before the structured and non-structured versions were incorporated into the same product, an author aspiring for a structured authoring and publishing environment, in addition to the standard FrameMaker, had to buy two different products. This version also brought the ability to work with XML documents, based on the SGML environment. Also, this version could now import SVG graphics into documents. (Docu+Design Daube 2004, Dan Emory & Associates 2005)

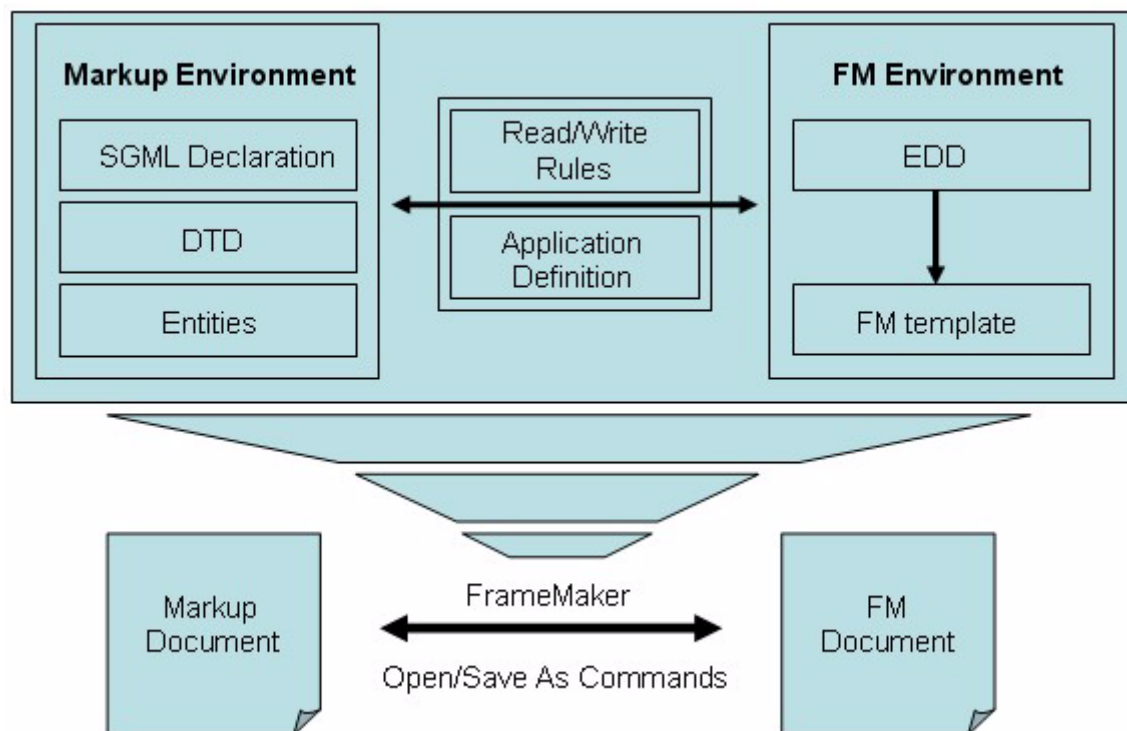
---

1. What You See Is What You Get; i.e. what the user sees on the screen is what he will get when, for example, printing.



### 3.1 FrameMaker structured environment

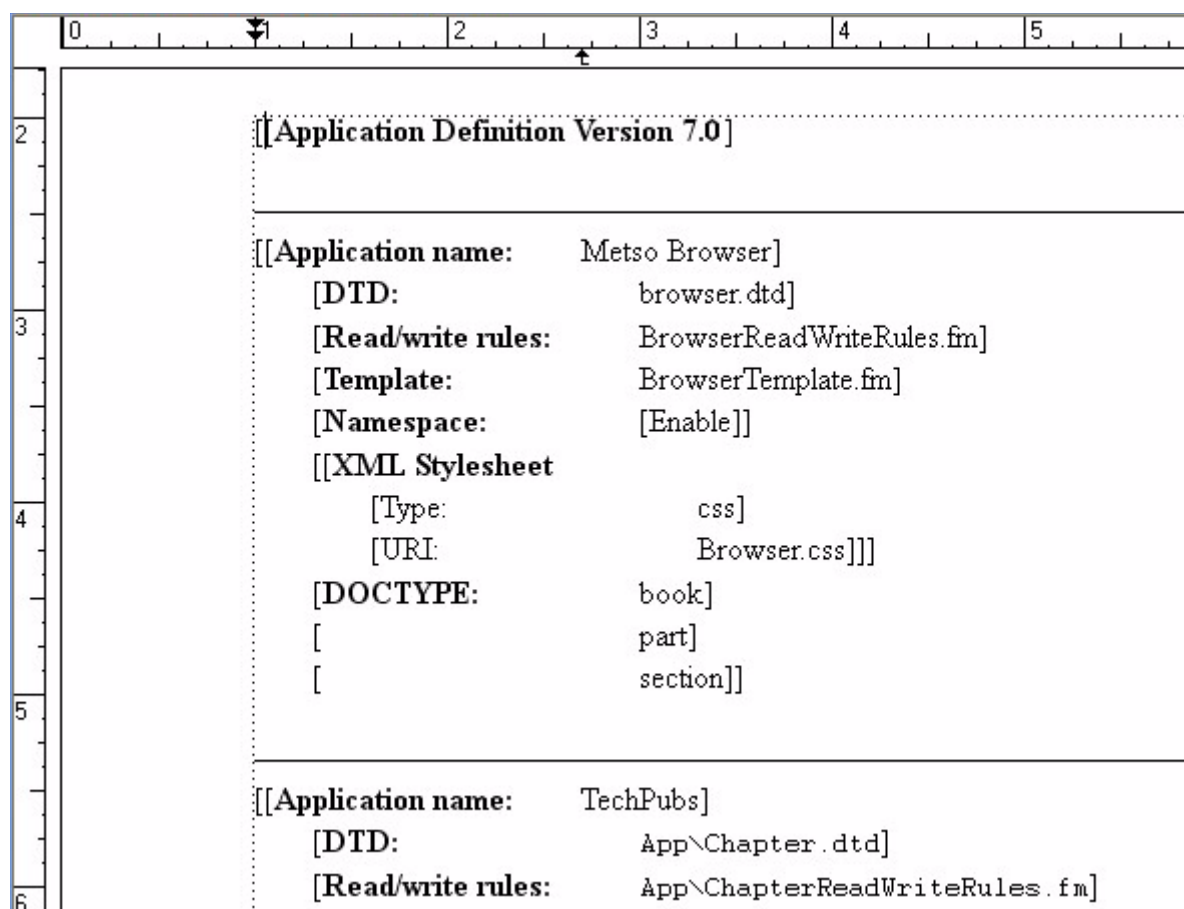
Figure 3 below illustrates the environment of FrameMaker used in importing or exporting structured documents.



**Figure 3: Framemaker structured environment (Adobe Systems Incorporated 2002 b: 29)**

The “Markup Environment” represents either SGML or XML (SGML is not a part of this thesis, but the illustration would be inaccurate if it was omitted). The “Read/Write Rules” define how the incoming structured data is processed in FrameMaker and the “Application Definition” defines which structured application is used when importing the data. The “FM Environment” represents FrameMaker environment, where the “EDD”, or Element Definition Document, defines the structure and the “Template” the representation of the FrameMaker document.

FrameMaker uses structure applications (illustrated in figure 4 below) to deal with XML documents. The applications define how the XML data is treated when importing or exporting from FrameMaker. There are ready-made applications for, for example, DocBook (an XML document model). For research purposes I created my own XML application for importing and exporting the Metso sample XML documents.



```
[[Application Definition Version 7.0]]
-----
[[Application name:      Metso Browser]
 [DTD:                  browser.dtd]
 [Read/write rules:     BrowserReadWriteRules.fm]
 [Template:             BrowserTemplate.fm]
 [Namespace:           [Enable]]
 [[XML Stylesheet
   [Type:                css]
   [URI:                 Browser.css]]]
 [DOCTYPE:              book]
 [                       part]
 [                       section]]
-----
[[Application name:      TechPubs]
 [DTD:                  App\Chapter.dtd]
 [Read/write rules:     App\ChapterReadWriteRules.fm]
```

Figure 4: An example of a FrameMaker structure application

A FrameMaker structure application consists of the following parts:

## Element Definition Document

The Element Definition Document (EDD, illustrated in figure 5 below) specifies the structure of the FrameMaker documents. It also specifies the formatting information of the elements. Considering structure, the XML equivalent of an EDD is the DTD.

```
EDD Version is 7.0
Structured Application: Metso Browser

Element (Container): Book
Valid as the highest-level element.
General rule: Book_Info?, Title, Part

Attribute list
1. Name: id                Unique ID        Optional
2. Name: secure            Choice           Optional
   Choices: internal | external
   Default: external
3. Name: lang              String           Required

Element (Container): Book_Info
General rule: Create_Data, Modify_Data, Check_Up_Data, Approve_Data,
translation_data?, Version_Data
```

Figure 5: An example of an Element Document Definition (EDD) in FrameMaker

Similarly to the DTD, the EDD contains element definitions, attribute definitions and comments. In the figure above, the element “Book” is shown as the highest-level element, meaning that it is the root element of a document. “General rule” shows the elements that the Book element can contain, and “Attribute list” shows the element’s attribute data.

All this information is received from the DTD, where the similar information would look like the following:

**Example 3: Equivalent DTD lines to the EDD in figure 5**

```
<!ELEMENT book (book_info?, title, part)>
<!ATTLIST book
ID #IMPLIED
secure (internal | external) "external"
lang CDATA #REQUIRED >
```

In addition to the components found in the DTD, the EDD also contains the following components:

- Formatting information - the EDD can contain formatting for elements, in other words, when the XML file is imported, the formatting is applied for the element. For example, a “title” element can acquire a larger font.
- Element type information - this maps elements to FrameMaker objects, for example markers and equations.
- Miscellaneous settings - these provide additional information, such as the name of the structured application. (Scriptorium Publishing Services, Inc. 2004)

FrameMaker allows creating an EDD from an existing DTD, and vice versa. When an EDD is created from a DTD, FrameMaker reads through the DTD, processing elements and their attributes one at a time, and makes assumptions on how the constructs from the DTD should be represented. (Adobe Systems Incorporated 2002 b: 78)

## Template file

A template is a document that stores properties that are used in more than one place. When an XML document is imported into FrameMaker, the information flows through the template file. The template file contains structure definitions and formatting information, which are derived from the EDD. The template can also store page layouts that determine, for example, the position and number of columns on pages. (Adobe Systems Incorporated 2002 a)

## Read/write rules

When XML is imported into FrameMaker, the read/write rules define how elements such as tables or graphics are converted. Using these rules the author can, for example, rename elements, delete elements or map entities to FrameMaker variables. The read/write rules are not necessary for the Structured Application's functioning, if the rules are not available, FrameMaker will not alter the elements when importing or exporting.

```
/*
 * Include all ISO entity mapping rules.
 */
#include "isoall.rw"

element "book" is fm element "Book";
element "book_info" is fm element "Book_Info";
element "part" is fm element "Part";
element "part_info" is fm element "Part_Info";
element "create_data" is fm element "Create_Data";
element "modify_data" is fm element "Modify_Data";
element "version_data" is fm element "Version_Data";
element "check_up_data" is fm element "Check_Up_Data";
element "approve_data" is fm element "Approve_Data";
element "translation_data " is fm element "Translation_Data";
element "handling_data " is fm element "Handling_Data";
element "by_who " is fm element "By_Who";
element "date" is fm element "Date";
element "version" is fm element "Version";
```

Figure 6: An example of a read/write rules document

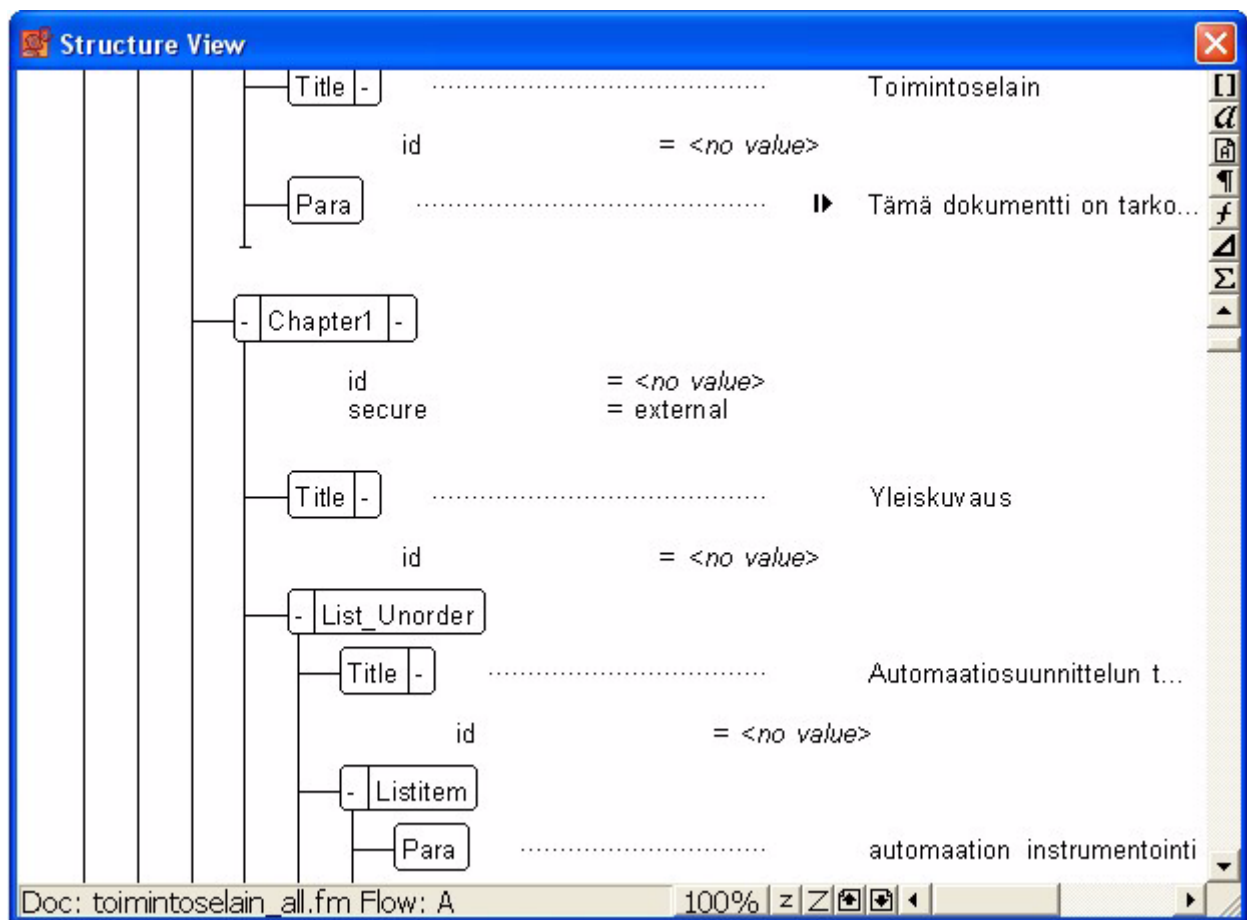
The above figure shows a very simple read/write rules document that capitalizes element names when XML is imported into FrameMaker.

### **3.1.1 Working with structured documents in FrameMaker**

Now that I have introduced the components of a structured application in FrameMaker, I will examine the features provided for working with structured documents. I will discuss their practicality and usability more below, in chapter 3.1.2 on pages 45-48.

#### **Structure View**

Structure View (illustrated in figure 7 below) shows the structure of an open document. Structure View also provides visual clues of whether the structure of the document adheres to the structure defined in the EDD. Elements that are in the wrong place are marked with red, also their connecting lines to the structure are marked red to make it easier to see where the element is placed. Also, missing elements are marked with a red rectangle.



**Figure 7: FrameMaker Structure View**

Structure View also provides the user with information of the element attributes. The attributes can also be hid from view if there is a need for it. These attributes are managed in the Attributes selection window (see more below, under “Managing attributes” on page 44). The Structure View also marks missing attribute values with red.

The Structure View can be kept open on the side of the normal view, enabling the user to see them both at the same time, which is useful when inserting new elements. The views follow each other’s position, clicking in either view will focus the cursor on the selected place in both views.

## Element catalog

Element catalog (illustrated in figure 8 below) is a context-sensitive element selection tool. Element catalog shows which elements are required and which elements are possible according to the FrameMaker element definition document.



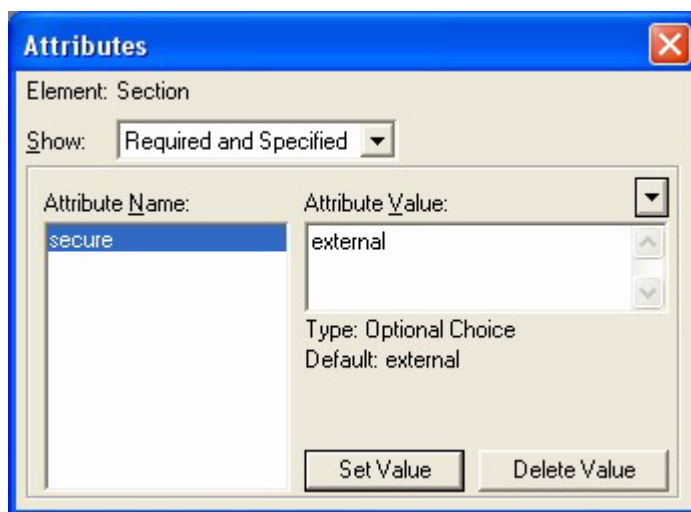
**Figure 8: FrameMaker Element catalog**

Valid elements are marked with a heavy check mark (as in “Title” in figure 8), and elements that are valid later in the element structure are marked with a light check mark. FrameMaker checks what elements can be inserted according to cursor placement in the Structure view.



## Managing attributes

Attribute selection window (illustrated in figure 9 below) is a graphical interface for managing element attributes. As with the Structure view and Element catalog, the valid attribute information comes from the definitions in the EDD. The Attribute Value pop-up menu allows the user to select a predefined attribute value for the attribute.



**Figure 9: FrameMaker Attribute selection**

### 3.1.2 Setting up and testing the XML features

In this chapter I will handle the set up of the FrameMaker XML import and export environment and test the features I defined in chapter 1.2 on pages 8-9.

As the previous chapter shows, there are several elements to a Structured Application. This can get confusing, especially for someone who is setting up Structured FrameMaker for the first time. The number of components and the amount of work needed to set up the application means that there is a lot of navigating from component to component making small changes, and it is easy to get “lost” while working, forcing you to backtrack your steps, close application components and start again. Also, when using custom structure applications, the application definition document has to be “read” into the program’s memory first every time, which is a slight inconvenience, as the structure application definition has to be opened every time the user wants to use them.

Also, the need for a separate structure rule document (EDD) and a template file is interesting. The EDD is not needed in import per say, the template file receives the element definitions from the EDD, and thus only the template file is referenced in the structure application. This is because it enables the author to create several layouts (templates) for a single import/export application. When the template and EDD are separate, different layouts are achieved by only defining a different template in the structured application, thus effectively removing the extra work caused by the need to create separate structured applications for different layouts.

Setting up the EDD was the hardest part in setting up the structure application. The first version of the EDD I created seemed to function properly (I could create structured documents using the EDD) until I tried to actually import XML data into FrameMaker. The import revealed that the EDD was nothing like the original DTD’s structure. The elements were in the

wrong places and the result was actually a chaotic representation of the original XML document, filled with FrameMaker's red marking, meaning that the data was not valid. Another problem I encountered when creating the EDD was the external DTD used in Metso Browser's DTD. For some reason FrameMaker treated the root element of the external DTD as a root element in the EDD. In other words, the actual DTD structure changed when I tried to create an EDD based on an existing DTD. Manually moving the external DTD elements created in the EDD solved the problem, but did not seem a very professional solution. Despite these problems it must be noted, however, that creating the processing instructions in the EDD is very simple, as they are created from components in a list that only shows the valid components depending on cursor placement in the EDD.

The Structure View in conjunction with the Element catalog and Attribute selection window provide a good working environment for producing structured documentation, or just editing existing documents. The Structure View provides good, clear visual clues of, for example, missing elements or attributes. This visually guided, EDD structure following working environment enables the user to quite easily produce valid structured documents. However, the Structure View is not perfect in its operation. Since some of the actual content of the elements is also shown in the Structure View (this can be seen on the right side of figure 7) it is rather large. Scaling the view smaller does not help, since when text is clicked in normal view, the structure view focuses on the element contents, rather than on the element, and thus the actual structure in the Structure View window does not show completely.

Validity checking works real-time in the Structure View. By this I mean that any ill-formed structure created is marked with red color instantly, without the user having to validate the document by selecting a validation command. The manually selectable validity check

command inserts the cursor to the problem spot it found. This is a faster problem identification process than, for example, with some DTD validity checking tools I have used, which usually report only the faulty line's number in the DTD. Also, considering document validity, FrameMaker does not allow invalid attribute values.

Appearance to structured elements can be added already in the import phase, as was mentioned when introducing the function of the EDD. When the incoming XML is processed, heading levels, for example, can be set to be in bold and in a larger font. This formatting information is stored into a template through which the incoming information flows. The formatting information can also be defined only in the template. Thus, quick changes to text appearance are possible. The template can also hold layout definitions. Adding layout is as simple as defining, for example, the wanted number of columns in the template file. The changes will take effect the next time the template in question is used in when importing XML.

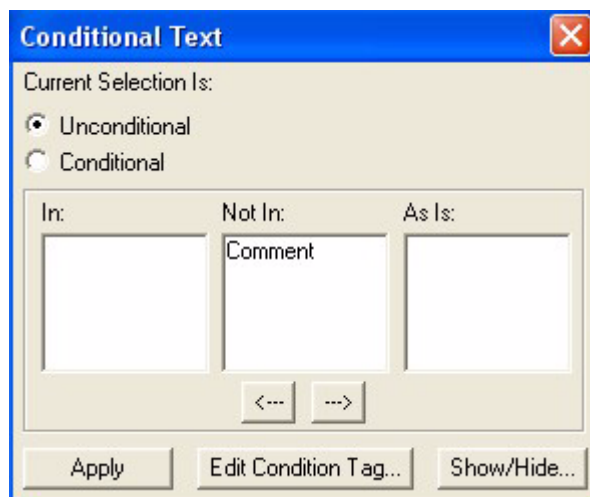
Exporting the imported XML data functions after the structure application is set up, and no extra work is needed. The exported XML data is exactly like when it was imported, so FrameMaker made no changes to the original data. Obviously, when XML data is exported from FrameMaker, all the style definitions defined in the EDD are removed from the information. FrameMaker also includes the option to add a CSS for the exported structured information. Adding the CSS to the structure application was quite easy to perform, and after the CSS was added all export functions now included the CSS into the output directory, making the XML result instantly ready for viewing.

Despite the slight problem with the structure view's usability, after the EDD is set up properly, FrameMaker provides a good environment for working with structured documents.

The real-time validity checking provides good visual clues of possible validity problems and the element catalog and attribute selection windows are simple and easy to use.

## 3.2 Conditional text in FrameMaker

Conditional text in FrameMaker is handled through the Conditional Text dialog (illustrated in figure 10 below).



**Figure 10: Conditional text dialog in FrameMaker**

Working with conditional text in FrameMaker is quite straightforward. The Conditional Text dialog shows all the available condition tags, and text marked with these tags can be hidden from view or shown when working with the document. When the document is published, only the text that is showing will appear in the output. Condition tags can be created and edited through the “Edit Condition Tag...” button and the tags can have different styles, such as underlines, strikethroughs and different colors. This way different conditional contents are easily recognized.

The interface is simple to work with, and the ability to give different style formatting to different tags is convenient. Condition tags can only be used within a document, hence document container level conditional publishing, where entire documents could be included or excluded, is not enabled.

### 3.3 HTML publishing in FrameMaker

FrameMaker includes a basic “save as HTML” function, that automatically processes the saved content into HTML. However, for more advanced HTML publishing, FrameMaker also includes a separate product named Quadralay Webworks Publisher Standard Edition 7.0. This tool will be used in evaluating the HTML publishing of FrameMaker, since it is included by default in FrameMaker product package.

Webworks Publisher is a tool that can be used to, among other things,

- Map element tags in structured FrameMaker documents to online styles
- Determine how a FrameMaker document is divided into one or more HTML files
- Convert any images into online formats
- Include alternative text descriptions for page elements
- Specify how navigation bars appear at the top and bottom of a page (Adobe Systems Incorporated 2002 a)

The feature list sounds very promising, as it suggests that the tool grants much control over the HTML conversion. The ability to create navigation bars in the published HTML is a very interesting feature.

Aside from producing formats for handheld devices, and XML + CSS and XML + XSL (XML with a stylesheet) Webworks publisher offers two HTML formats to be generated, Portable HTML and Dynamic HTML. The main difference between the two is that the Dynamic

HTML uses a CSS file to provide outlook for the text. I will use the Dynamic HTML template in my test, since the separate stylesheet provides more control over the produced output, as I discussed earlier in chapter 2.4.2 on pages 31-32.

### **3.3.1 Testing Webworks Publisher**

When the project is started the user gets to define that the content is structured and that elements will be mapped instead of styles. Elements are chosen HTML equivalents from a drop-down list and all the elements that are not selected an equivalent will be automapped by Webworks Publisher using default settings. Elements with the default mapping are analyzed by Publisher that tries to create equivalent HTML styles from them.

Mapping is restricted to the predefined styles in Webworks Publisher, which can cause some troubles since not all HTML styles are included. For example, Webworks Publisher provides only 4 heading levels, although there are six heading levels available in HTML 4.0 (Web Design Group 1998).

The splitting of a FrameMaker document into one or more individual HTML documents is done by mapping a FrameMaker element into a Publisher paragraph style that starts a new page. Publisher generates new output files every time it comes across the defined element in the FrameMaker source files. (Quadralay Corporation 2002)

The styles for the HTML output can be modified by editing the default stylesheet Webworks Publisher adds to the output directory. The user can also create his own stylesheet and set the Publisher templates to use that stylesheet.

The JPEG graphics quality can be affected by using a macro for that purpose. Users can define the output quality of JPEG graphics on a scale of 1 - 100. Higher values produce better

quality, but with lesser compression. Also, alternative text descriptions can be added to graphics, which is a good feature if the HTML user does not see the pictures for some reason or is using a nongraphic browser. (Quadralay Corporation 2002)

The navigation bars are added to the HTML pages' lower and upper parts. The navigation bar includes four buttons, TOC, Prev, Next and Index. The table of contents and index are converted from FrameMaker's equivalents by mapping them as the HTML TOC and Index pages when working with the Publisher project. The navigation bars in the produced HTML output provide an interface for accessing the HTML information effortlessly and they also give a professional look for the outcome. Without navigation, HTML can be a quite confusing web of text.

All in all, the Webworks Publisher publishing is quite professional, and provides the HTML + CSS option (and many more). Automatic splitting of documents, the ability to affect JPEG quality and the navigation buttons on the produced HTML make Webworks Publisher a good tool for HTML publishing.

### **3.4 FrameMaker 7.0 documentation**

FrameMaker documentation relevant to my thesis consists of PDF files for structured environments. There are many documents covering different areas of structured applications, but the most relevant are The Adobe FrameMaker 7.0 XML Cookbook and the Structure Application Developer's Guide Online Manual, which are provided in PDF format with FrameMaker.

Different user needs are considered in the division of the information between the XML Cookbook and the Developer's Guide. The XML Cookbook is a basic introduction to creating



structured applications in FrameMaker. The components of a structured application are explained clearly and it offers step-by-step instructions for creating a simple, yet fully functional, application. The example files needed to create the first application are provided with FrameMaker. The instructions are very simple and easy to follow, and the step-by-step progress ensures that all the necessary actions needed to create a working application are taken by the user.

The Structure Application Developer's Guide is a deeper look into the functioning of a structured application. Any questions or needs that were not answered by the XML Cookbook can be found here. Being nearly 600 pages long, it covers everything about creating XML or SGML based structured applications in FrameMaker.

All the necessary terms, concepts and acronyms are explained in the documentation. In, for example, the Cookbook there is no special section for explaining the terms, but all the definitions can be found in places considered most appropriate by the author(s). The instructions also provide plenty of illustrations. Using the Cookbook once again as an example, the whole process of creating a structured application is supported by pictures of the user interface. The Developer's Guide has diagrams supporting the explanations of the components and processes of the structured environment.

It must be noted, however, that not all of the documentation included in FrameMaker is discussed here. The documentation is very thorough and separate PDF files are included on, for example, using different character sets and filters.

All in all the FrameMaker documentation is more than sufficient. The division of the documentation on working with structured documents in FrameMaker into basic level (Cookbook) and advanced level (Developer's Guide) simplifies the process of learning to use

FrameMaker's structured features. The used terms are explained and illustrations are used well to support the text.

## 4. An examination of Broadvision Quicksilver 1.6.1

Quicksilver's history is long and varied. The original name of the product is Interleaf, which changed when the software was bought by Broadvision. The first version was published over 20 years ago and since then it has run on a large variety of computers, including IBM mainframes and Apollo. (McGahey 2005)

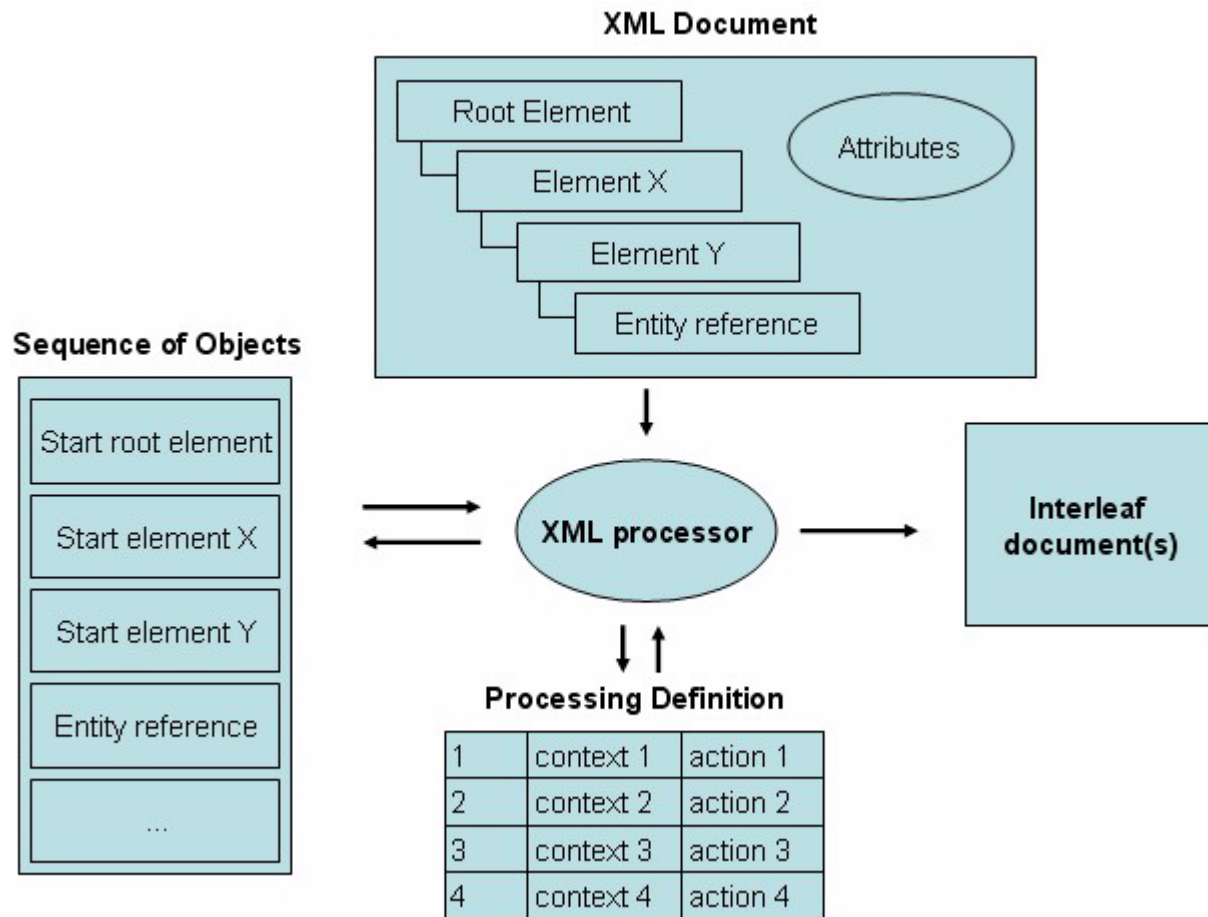
### 4.1 Quicksilver structured environment

It is clear from the start that Quicksilver's structured environment tools are not meant for beginners to use. The Quicksilver XML Publisher Implementor's Guide lists the following prerequisite skills and experience that are needed in order to be able to successfully create XML applications in Quicksilver:

- XML experience
- Interleaf publishing skills
- programming skills
- experience with Quicksilver XML publisher processing primitives
- familiarity with Interleaf Lisp (Interleaf Inc. 1999: viii)

The prerequisites sound very demanding, and might raise the threshold for a beginning user to start learning the XML features in Quicksilver. All the skills except one, however, do have a sense of acquirability, that is, they do not seem impossible to get a hold of in a reasonable time. The exception is programming skills. The more unfamiliar prerequisites, i.e. processing primitives and Interleaf Lisp will be discussed more below, when the Implementor's Toolkit is examined.

The following figure illustrates the environment of XML to Interleaf processing:

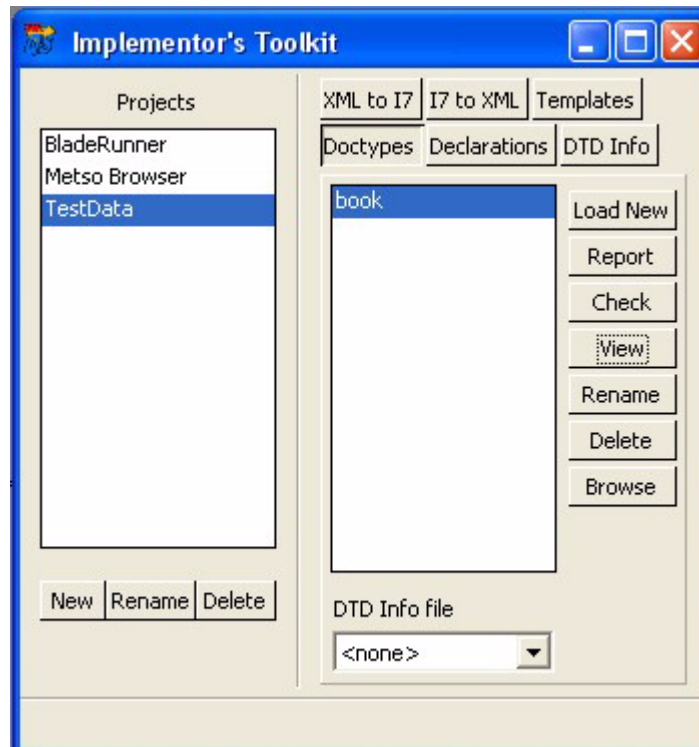


**Figure 11: XML to Interleaf processing (Interleaf Inc. 1999: 4-9)**

An XML processor processes the incoming XML data to Interleaf format. The processing definitions define how the incoming XML data is treated. Object sequence corresponds to the XML data sequence and the outcome of the process is one or more Interleaf documents. (Interleaf Inc. 1999: 4-8) The actual components needed by this process are handled with the Quicksilver Implementor's Toolkit, which is discussed next.

#### 4.1.1 The Quicksilver Implementor's Toolkit

With the Implementor's Toolkit (ITK, illustrated in figure 11 below) the author can design and create applications for processing incoming or outgoing information.



**Figure 12: Quicksilver Implementor's Toolkit**

The applications are shown under the “Projects” title on the left side of the ITK dialog, whereas the components, which will be discussed next, that make up the application are shown on the top of the right pane.

## XML to Interleaf 7 to XML

XML-to-Interleaf 7 processing definitions define rules for processing incoming XML data, whereas Interleaf 7-to-XML defines how the data is processed when publishing to XML. The processing definitions define how XML element names are mapped to Quicksilver (and vice versa) and how, for example, graphics and tables are handled.

### Processing Definition

Context	Action	Comment
<code>(context :type :element :name "TABLE")</code>	<code>(let ( (caption (get-info :descendant :level 1 :posi- tion 0)) ) ) (when (and caption (string-equal "caption" (mid:get-name cap- tion)) ) (create-cmpn :name "table-title" :element cap- tion) ) (mid:process-object (get-current-processor) cap- tion) ) ) (continue-processing)</code>	preprocess table get caption of table and display it process also the caption to make sure the text is han- dled
<code>(apply-table-model "HTML4" :table-name "html4" :row- names '(("header" . "hea- drow") ("body" . "row") ("footer" . "footrow"))</code>		Apply HTML table model
<code>(context :type :element :name "Vendorinfo")</code>	<code>(create-cmpn :name "ruleven- dor")</code>	divide sections with lines
<code>(context :type :element :name "Partinfo")</code>	<code>(create-cmpn :name "rule- part")</code>	divide sections with lines
<code>(context :type :element :name "Description")</code>	<code>(create-cmpn :name "rule- desc")</code>	divide sections with lines

**Figure 13: An example of a Quicksilver processing definition**

The above figure shows an example of a processing definition document in Quicksilver. The processing document consists of individual processing rules. Processing rules themselves consist of a context statement and an action statement, which are Interleaf Lisp statements. (Interleaf Inc. 1999: 4-3) Interleaf Lisp is based on Common Lisp, which is a popular programming

language dating back to the 1960s. Interleaf Lisp, however, contains also classes, methods, and functions for processing Interleaf objects. (Morris 1998)

## **Templates**

Templates are used in XML to Interleaf 7 processing to add formatting and other information to structural elements, such as paragraphs, headings and graphics. Templates ensure consistent layout throughout the document. XML processor templates are regular Interleaf documents. (Interleaf Inc.1999: 3-2 - 3-3)

## **Doctypes, declarations and DTD info**

The separation between doctype and declaration was actually quite confusing to me. A doctype document added into Quicksilver holds only a few lines of text, whereas the actual declarations that define document structure are in the declaration document. After some research I managed to get an explanation for this: the doctype statement defines the module (XML document) name, its ID and the DTD name. Doctype statement defines, as the name suggests, the document type. When the similar doctype statement is found in an XML document, Quicksilver knows what tailored application it will use to process the incoming XML document. Different XML documents can have different DTDs defined, since their structural needs differ. Hence, for example, a chapter in a book can have a different processing application than the book itself. DTD info holds user-defined information about the used DTDs.

## XML processing window

After all the components have been added to the Implementor's Toolkit, a user can import XML through an XML processing window, which is illustrated below.

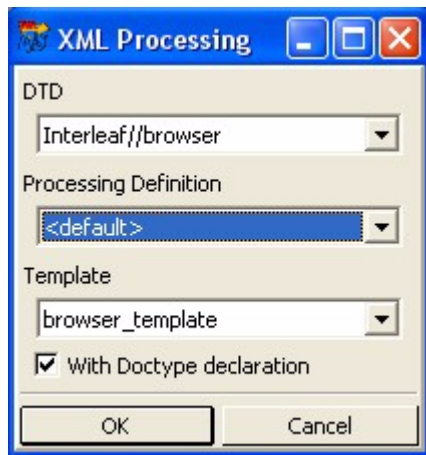


Figure 14: XML processing window in Quicksilver

The DTD defines the DTD that the incoming XML file is supposed to follow. The processing definition defines the used processing rules, i.e. Lisp statements that affect the elements of the incoming XML file. The Template is the template created for adding styles for the incoming elements and layout for the end product, that is, the Interleaf document.

### 4.1.2 Working with structured documents in Quicksilver

Now that I have discussed the elements of an XML processor in Quicksilver, I will proceed to examining what features are provided for working with the structured documents. Their practicability and usability will be discussed later in chapter 4.1.3 on pages 64-67.



## Structure and normal view

Quicksilver shows the structure of the open document in the left-hand pane, and the normal view on the right as illustrated in the figure below. (No styles have been added to the text.)

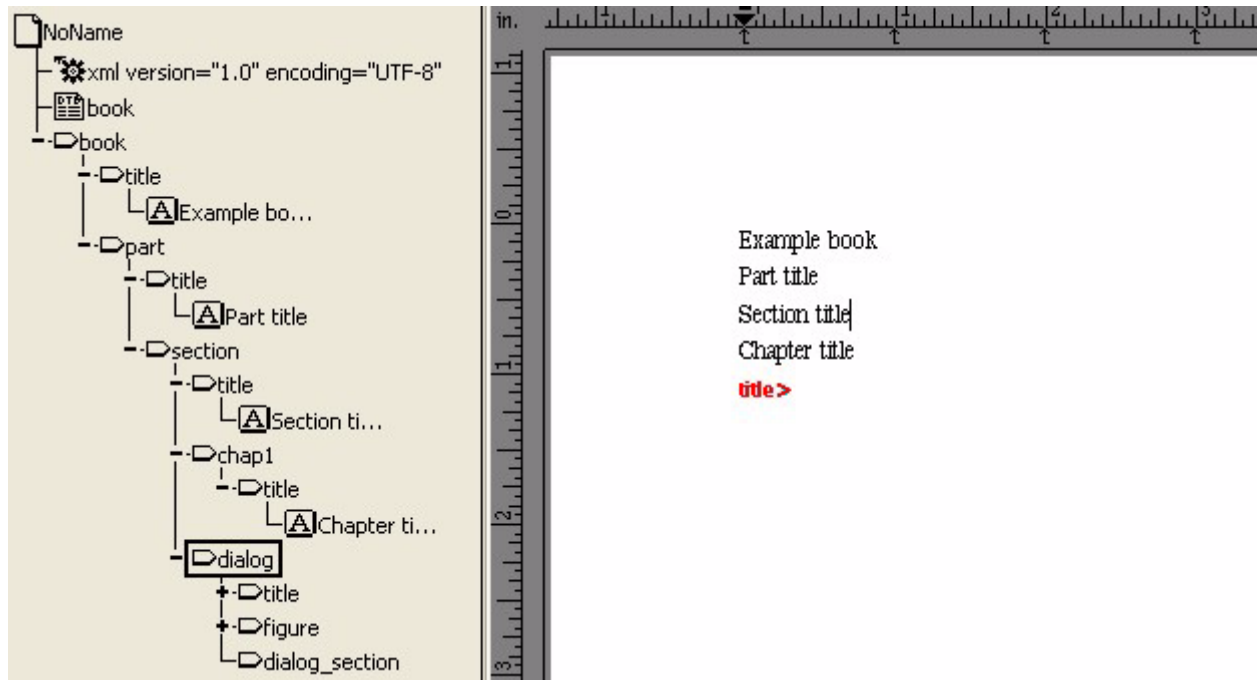
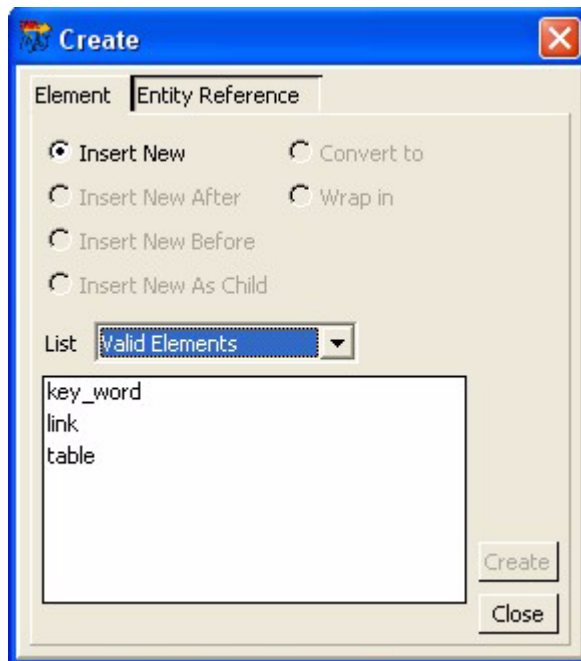


Figure 15: Document structure view in Quicksilver

While FrameMaker highlights the missing elements or elements that are in the wrong place in the structure view, Quicksilver highlights the missing element containments in red in the text view on the right, as shown in the figure. It is important to note that Quicksilver does not allow invalid structure to a document following a DTD, hence missing elements or elements in the wrong place are not required to be highlighted for the user.

## Inserting new elements and viewing attributes

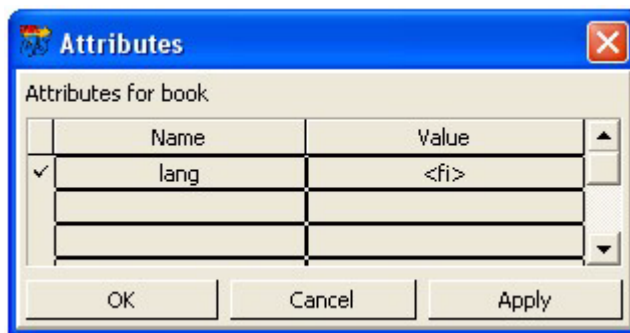
Elements are inserted in Quicksilver through the selection window illustrated below.



**Figure 16: Inserting elements in Quicksilver**

Quicksilver uses the DTD reference to discern what elements are available for the location selected in the structure view of a document. By default the window only lists valid elements for the current location, but the user can also change that if he wishes to see the rest of the elements available. Elements can also be inserted through a shortcut menu that opens when right-clicking an element in the structure view. This is a quick way to add elements, and it is not as versatile in its selections as the element selection window. The Entity Reference pane on the right is for creating references to entities defined in the used DTD.

The attribute selection window is illustrated below.

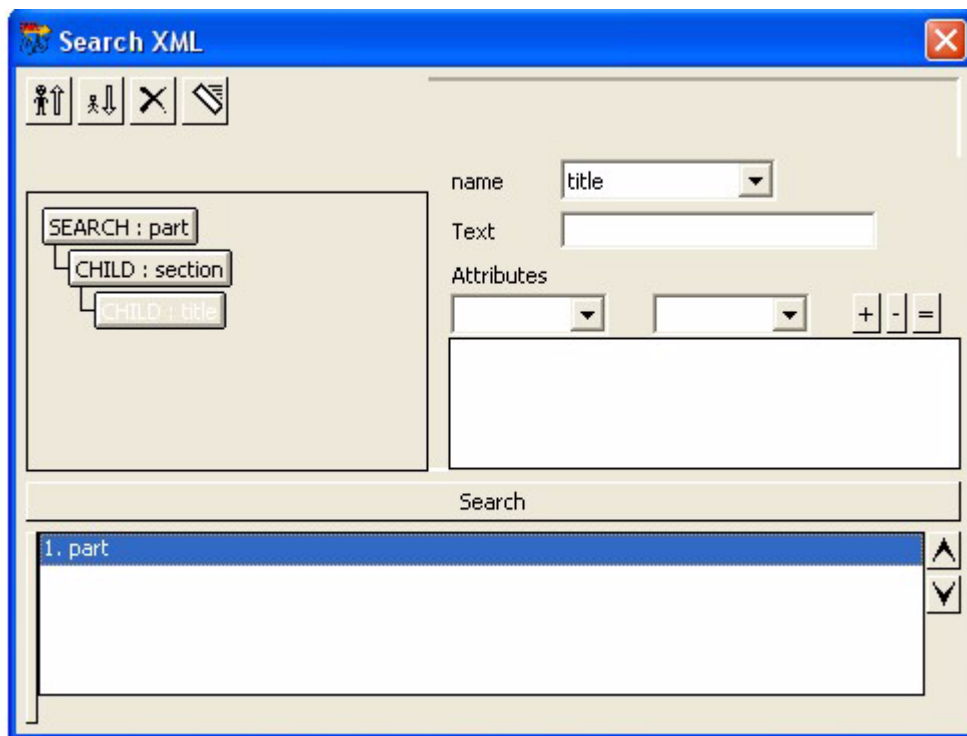


**Figure 17: Setting attributes in Quicksilver**

The attribute selection window retrieves possible attributes for an element from the DTD. Values can be set by clicking on the value and selecting a valid value from a list or typing the value if the values can be defined by the user (i.e. the structure definitions do not restrict attribute values).

### **Search XML function**

Quicksilver has also a tool for searching elements from the XML structure. The dialog is illustrated in figure 18.



**Figure 18: Search XML dialog in Quicksilver**

The search function is a versatile tool, allowing for searches involving the

- structure (parent or child element)
- name of element
- text that the element contains
- attributes and values associated with an element (Interleaf Inc. 1996-1999)

By using these search criteria, or combinations of these, the user can search for structural elements within the document. The lower part of the window shows all the elements meeting the search conditions, and can be used to navigate to the elements directly.

### 4.1.3 Setting up and testing the XML features

Although the documentation is discussed further below, a mention of the partial inadequacy of Quicksilver's documentation is relevant in here, since it affected especially the setup of the software. A substantial part of my time devoted to Quicksilver research was used in trying to create a working XML application, and the documentation did not support this adequately. This obviously also affected my research results, which is important to keep in mind when reading this chapter. Closely related to the negative effects of the lacking documentation is the cryptical error feedback in Quicksilver. The following figure shows the error dialog that opened when I tried to add and view the Metso DTD to the XML processing application I was creating for test purposes:



**Figure 19: Error dialog when adding a DTD in Quicksilver**

When adding the DTD to the XML application, Quicksilver gave the above error dialog, with the options of canceling the operation or exiting Quicksilver and writing an error log to the desktop. When the error log itself looked like the one in the example below, it was a matter of guessing what caused the error:

#### Example 4: Quicksilver error log sample

```
Reported by Admin@wings on Sunday, May 8, 2005 4:16 pm.
Broadvision Quicksilver 1.6.1 / i386
Memory allocated/freed: 19115/13290, page faults: 0
*document* is "d:\\desktop.ileaf\\chap-01_00-yleiskuvaus.ilxad".
Error: Wrong argument type: eof in: ml-analyzer-read-name
Backtrace:
(ml-analyzer-read-name #<obj-user 1fcc4d8>)
(mid:read-name #<obj-user 1fcc4d8>)
(ml-analyzer-read-generic-identifier #<obj-user 1fcc4d8>)
(mid:read-generic-identifier #<obj-user 1fcc4d8>)
(ml-analyzer-read-content-token #<obj-user 1fcc4d8>)
```

It was, however, clear that the Metso's DTD required some modifications before it could be used in an XML processing application in Quicksilver. After successfully using a simple test DTD, I started examining what needed to be changed in the Metso DTD.

After removing a reference to an external DTD (referencing a DTD subset in the ITK did not help), which is used with tables in XML, and removing entity references from attribute lists, I managed to add the DTD to the XML application without errors. This solution did not seem very professional, but in order to get on with the research, I had to settle for this arrangement.

The separation of the Doctype and the Declaration in the Implementor's Toolkit required much time and effort to understand, however, after trial and error, I eventually had a functioning XML application in Quicksilver.

Unlike the creation of the EDD for incoming element processing in FrameMaker, creating similar processing instructions require programming experience in Quicksilver, as they are created with Interleaf Lisp. As I have no programming experience, I decided not to create my own processing instructions in Quicksilver. The time and effort required of learning Interleaf Lisp does not match the possible gains to this thesis. If a user does, however, have program-

ming experience with Interleaf Lisp, the processing instructions can be tailored with flexibility, as the user is not restricted to selecting predefined processing components from a list as in FrameMaker.

Everything related to XML application creation is done through the Implementor's Toolkit, which makes the work quite concise and concentrated. FrameMaker does not have a similar compact interface, thus, after using the ITK, working with structured documents in Quicksilver gives a more organized impression.

It is useful that the technical writer sees both of the representations at the same time, so he can add content on the text view and take advantage of the structure view when inserting new elements. While FrameMaker has separate structure and normal views of the open document, Quicksilver combines them together, with the structure view on the left and the actual text content on the right. This is quite logical and convenient. The combined view works better than the FrameMaker separated solution, where the structure in the view can be lost when it focuses on the element containments, while the structure is always visible in Quicksilver.

Quicksilver has a real-time validity check of the documents, as does FrameMaker, but Quicksilver is more strict about the validity of the documents than FrameMaker. Quicksilver does not let the user to add or remove elements from the structure if the action would cause invalid markup. Also, invalid attribute values cannot be set, and the validity check function that can be selected manually reports of missing attribute values.

Layout to the structured data is easily added with a template. After the necessary formatting information is defined to the template, it is added in the ITK templates for the desired XML application. After this the application uses the template to produce structured documents with the formatting defined.

Exporting the imported structured material back to XML works properly and as the rules for a document following a DTD are pretty strict, no unwanted structural changes appear.

All in all, the Quicksilver structured environment setup requires more from the user, as the documentation or error feedback are not very good, and the processing instructions require programming experience. Inserting new elements and working with attributes is as straightforward as in FrameMaker. Quicksilver provides a separate interface for working with the XML applications, which is a more concentrated solution than FrameMaker's. Also the combination of the structure and normal views is a better solution than having separate ones as in FrameMaker.

## 4.2 Conditional text in Quicksilver

Conditional text in Quicksilver can be created using attributes. First the required conditional elements are assigned attributes and attribute values according to what their usage is. (One must notice, however, that when speaking of a structured document that follows a DTD, attributes cannot be added without defining them in the DTD first). After all the necessary elements have been tagged, control expressions are used to define how the tagged or untagged content is processed. Here is an example of what a control expression may look like:

**Example 5: Control expressions in Quicksilver**  
Internal = 1  
Internal # \$empty

The first control expression would show all the untagged elements and the elements that have the attribute "Internal" with value 1. If needed, also the untagged elements can be removed



from sight with the latter control expression. These two examples are, however, quite simple.

The control expressions can have a more complicated function as in the following example:

**Example 6: A more complex control expression in Quicksilver**

```
color = red | color = blue & model = 20
```

The control expression in the above example would show everything that has either “red” as their attribute “color” value or that has “blue” as “color” value and has “20” as “model” value.

Since control expressions can be conjoined this way Quicksilver offers quite flexible control over conditional content.

The conditional content is not restricted to document level, also document containers can have control expressions. This way also books can have several different editions in one. On a container level attributes are assigned to documents instead of elements, and a control expression is used to define how the documents inside a book are treated. When all the necessary attributes have been assigned, the control expressions can be used to change the outcome of the whole documentation quite simply. While the FrameMaker conditional text features can be used to define whether text meeting certain conditions is visible or not within a document, in Quicksilver a whole book can be changed by adding a control expression, or several, for the highest level container, without even having to open the documents as in FrameMaker. This making the conditional content features of Quicksilver quicker to use.

### 4.3 HTML publishing in Quicksilver

While FrameMaker has a separate tool for creating HTML, the Webworks Publisher, Quicksilver has a simpler “publish as HTML” function. When the documents are published as HTML each component, frame and table are mapped to a predefined style. There are three ways to affect the style mappings in Quicksilver:

- by using attributes and attribute values
- by using a template provided for this purpose
- by editing the definitions directly

The first two ways are actually quite close to each other, since in both attribute values are used. The difference is that the template has the attribute values predefined. Mapping using attribute values works in the following way: first, the attribute `HTML_STYLE_MAP` is defined for the document, or the template is used to provide the attribute. Secondly, the attribute is assigned values either manually or using the template. Then, elements that need to be mapped differently from default mapping are assigned the attribute and one of the attribute values. When the document is then published to HTML, Quicksilver reads through the `HTML_STYLE_MAP` attribute values and maps the elements to corresponding HTML elements.

Changing the style mappings manually requires a little more knowledge of HTML. All of the mappings can be found in the Quicksilver installation directory, where they can be edited manually. The definitions can be edited, but the user cannot create new definitions, thus these can only be used to affect the HTML element style, not create mappings. A mapping file consists of the Quicksilver element name and the resulting HTML code.

Quicksilver converts graphics automatically to GIF format, and this cannot be affected by the user.

#### **4.3.1 Testing the HTML publishing**

Using the HTML template with a book containing Quicksilver documents is easy. After the template is added to a book all the documents will have the predefined attribute `HTML_STYLE_MAP` which can then be used to give mapping instructions to elements. However, a structured document following a DTD is not as simple. In order to ensure the document's validity, attributes cannot be added when necessary.

A way around this problem could be to define an attribute `HTML_STYLE_MAP` in the used DTD for all the necessary elements. The attribute would have all the same valid values as the same attribute in Quicksilver, and thus it could be used to map elements. However, this method is not documented anywhere in the Quicksilver documentation, and testing this method would require making changes in the sample documents, hence it will not be tested. Thus, from the viewpoint of this thesis, structured documents will have to be published to HTML using the default mapping, and it has to be concluded that structured elements cannot be mapped to HTML elements without making either invalid structured documents or making substantial changes (since all the used HTML element styles will have to be defined) to the used structure definitions first.

New attribute values cannot be added for the `HTML_STYLE_MAP` attribute. When examining FrameMaker, there was an insufficient number of heading levels, which created the need to make new mappings. The `HTML_STYLE_MAP` attribute, on the other hand, has all HTML heading levels 1 through 6 predefined.

Quicksilver uses HTML + CSS publishing by default, and adds an empty stylesheet into the output directory. The HTML uses default style formats, and the function of the empty stylesheet is serve as a marker of what stylesheet name has been defined in the HTML code. Thus the user can add his own stylesheet by copying over the default stylesheet and no changes are needed in the HTML code.

These are the features of Quicksilver's HTML publishing, and the differences to Webworks Publisher are dramatic. Element mapping is not as straightforward as in the Publisher, pictures are converted to GIF but the conversion cannot be affected and documents cannot be split into smaller HTML modules when publishing. As no navigation for the resulting HTML is provided with Quicksilver, the HTML is not ready for convenient use.

#### **4.4 Quicksilver 1.6.1 documentation**

Quicksilver documentation relevant to my thesis includes an online help for the Implementor's Toolkit and a printed book named Quicksilver XML Publisher Implementor's Guide.

The online documentation covers the basics of structured documents and working with structured documents in Quicksilver sufficiently, but when the user reaches the Implementor's Toolkit part, instead of instructions on how to use the feature, only a brief explanation of what ITK is and this warning are given:

Caution. The Implementor's Toolkit is intended for use only by expert XML Publisher implementors, such as the specially-trained consultants in the Interleaf Professional Services Group (IPSG), or other developers knowledgeable in XML. Using the ITK without such expertise can cause unintended, potentially irreversible errors with your XML Publisher applications. (Interleaf Inc. 1996-1999)

The same warning style continues in the printed book provided with Quicksilver. The book introduces the basic functions of the Implementor's Toolkit, but seems to fall short in the usage instructions, since most space in the book is reserved for explaining the processing instructions. Thus, the processing definitions are thoroughly documented, but that serves little purpose if the documentation leading to a basic functional application falls short. The documentation is good on the basic and advanced levels, but is inadequate in between. What I mean by this is that the basic functions are explained adequately and a lot of the documentation concentrates on the advanced features of an XML application, but the actual creation of an XML application is inadequately documented. A step-by-step instruction for creating a basic XML application, as in the FrameMaker documentation, would be one solution for this problem. The book also provides general guidelines for planning and creating XML applications. These instructions seem to serve as checklists for people who already know how to work with these projects in Quicksilver.

The Quicksilver XML Publisher Implementor's Guide provides good illustrations on both the parts of the user interface being explained and diagrams explaining the general structured environment. All the used terms are not explained in the book, which comes down to the presupposed experience of the user. The online help, however, has explanations for all the terms used.

As a summary, considering the intended user the documentation is quite good with adequate illustrations, but I would have needed a workflow type instruction for creating an XML application. Relating to the intended user, one must note that some of what I found falling short in the Quicksilver documentation can partly be explained by the fact that I myself do not fill all the requirements defined in the beginning of the printed book (these were discussed in

the beginning of chapter 4.1). However, I do not think that the need for good documentation supporting the product can be circumvented with mere warnings.

## 5. Comparison and conclusions

FrameMaker and Quicksilver were examined in this thesis from a viewpoint of how they can be used to create single sourced documents from XML data. A test project was created in both tools and both were used to import and export XML files received from Metso Automation. This test data was used to examine the structured environments, conditional text features and HTML publishing, and the parts of the documentation that were related to the structured environments were also examined. Now I will discuss the tool differences, and the strong and weak points discovered.

### 5.1 Structured environments

The first difference that I noticed when creating my own structure application (FrameMaker) and XML processor (Quicksilver) is that Quicksilver has a separate interface for this purpose, the Implementor's Toolkit. This separation makes the work much more concise as it focuses the features related to XML applications in one small window, while in FrameMaker the equivalent features share the same interface. This effectively means that most of what a FrameMaker user sees while working with structured documents are features that he does not need for the purpose at hand. While the structure applications in FrameMaker have several parts, an own interface for these would enhance usability.

Quicksilver is more compact also in the mapping and processing instructions, in other words, the definitions of how XML elements are mapped to the tool's elements and what is done to the elements when importing the XML data. Both of these are defined in the process-

ing definitions in Quicksilver, whereas in FrameMaker two separate documents are needed: one for the read/write rules and one for the Element Definition Document.

FrameMaker's processing instructions are more simple than Quicksilver's. While the processing instructions are added to the EDD from a list showing all the possible components, creating processing instructions requires Interleaf Lisp programming experience. On the other hand, using a programming language to create processing instructions grants more flexible control over content processing.

In FrameMaker the used DTD has to be converted to a native FrameMaker document, the EDD, to provide structured instructions for documents. Quicksilver, on the other hand, can use a DTD as it is, and no conversion is needed. Thus, especially in a situation where several DTDs would be used, Quicksilver would be faster. Structural changes are more easily implemented in Quicksilver as only the DTD needs editing.

When the XML application is finished in Quicksilver, it can be used to process XML data when needed without any additional procedures. In FrameMaker, on the other hand, the user needs to "read" the structure application definition, in other words the document containing all the structure applications, into the programs memory before using one of his own applications. This is only a slight inconvenience, but seems like something that could be avoided by reading the application definitions automatically at startup.

The integration of the structure view and normal view in Quicksilver is more convenient than in FrameMaker, where the structure view and normal view are separate windows. As the structure view also shows some of the element contents, and as it focuses on the contents when the user clicks somewhere in the normal view, the structure can be lost from view if the window is scaled smaller.



To sum up, my general view of FrameMaker is that it is slightly less organized than Quicksilver, but simpler to use and to set up. Quicksilver, on the other hand, requires more from the user but offers more control over the documents. The differences in conditional text follow along the same footsteps.

## 5.2 Conditional text and HTML publishing

While in FrameMaker conditional text can be hid or brought into view within a document, in Quicksilver also the documents may have conditions, enabling the user to hide or show whole documents using book-level control expressions when, for example, printing a book. The control expressions for affecting conditional content in Quicksilver require slightly more from the user, but again offer more control over the documents. FrameMaker does, however, offer more control over the formatting of conditional content, enabling the user to, for example, use a different color or style for different conditional content.

The effects of FrameMaker having a separate tool, the Webworks Publisher, for HTML publishing are clearly visible in the differences of the tool's HTML publishing. The main differences are the Webworks Publisher's feature for automatical document splitting into smaller HTML sections and its ability to add navigation buttons, a table of contents and an index to the produced HTML. Hence Webworks Publisher clearly produces more professional HTML that is ready for distribution. The Publisher has an option for producing HTML with a cascading stylesheet and Quicksilver produces this format by default. As the element mapping in Quicksilver is handled through defining attribute values for the structured document in question, a problem arises when using structured document that uses a DTD. Thus, element mapping is not quite as straightforward as in the Webworks Publisher. On the other hand,

Webworks Publisher does not have all the available HTML elements predefined, and since new elements mappings cannot be created, this can cause some problems.

### **5.3 Software documentation**

FrameMaker's documentation provides more information for beginning users than that of Quicksilver's. The step by step workflow instructions in FrameMaker documentation are sorely missed when trying to set up the Quicksilver environment using the provided instructions. Both have plenty of illustrations supporting the text, which helps the user link what he is reading to the actual tool components and assists in understanding the general environment of the tool. The FrameMaker documentation has more explanations for the terms used than the printed book that comes with Quicksilver, which can be attributed to the presupposed user of the Quicksilver structured environment, that is, a quite advanced user.

### **5.4 Recommendations**

Both tools can suit different users depending on their needs. If a tool is needed which setup is relatively simple and does not have demanding skill or experience prerequisites, has simple conditional content features, professional HTML publishing features and good documentation, FrameMaker is the right option. If a tool is required that can handle DTDs as such, has versatile and flexibly tailored processing instructions (provided Interleaf Lisp is not a problem), has very powerful conditional content features, Quicksilver is the right option.

**Table 1: FrameMaker's strengths and weaknesses**

+ Easier to set up than Quicksilver
+ Professional HTML publishing with Webworks Publisher
+ Enables the user to give different styles to different conditional content
+ Thorough documentation for both beginners and more advanced users
- The structured working environment is slightly less organized than Quicksilver's
- The conditional text features are not as versatile as Quicksilver's

**Table 2: Quicksilver's strengths and weaknesses**

+ Compact interface for working with structured documents
+ Processing instructions can be flexibly tailored
+ Powerful conditional content features
- Processing instructions require Interleaf Lisp programming experience
- HTML publishing is very basic compared to Webworks Publisher
- Documentation presupposes an advanced user

## 5.5 Further research opportunities

As I have worked on this thesis, newer versions of the two researched programs have been released: FrameMaker version 7.1 and Quicksilver version 2.0. A logical way to proceed from this thesis would be to examine how the newer versions have been developed; what new features have been included, have the problems discussed in this thesis been fixed and whether these affect the conclusions made in this thesis.

Also, as I have argued that using FrameMaker or Quicksilver may prove to be a better solution than using XML stylesheets for single sourcing, it would be interesting to conduct a research that would examine this from the viewpoint of creating single sourced content solely using XML and see how it reflects to my solution in this paper. This would give the technical communication community another tool-related research, which are in demand, as was mentioned in chapter 1 on page 1.

## References

- Adobe Systems Incorporated. 2002 a. "Adobe FrameMaker 7.0 User Guide for Windows, Macintosh, and UNIX."
- Adobe Systems Incorporated. 2002 b. "Structure Application Developer's Guide Online Manual."
- Adobe Systems Incorporated. 2002 c. "The Adobe FrameMaker XML Cookbook."
- Costur, Pamela and Ann Rockley. 2001. "Information Modeling for Single Sourcing." (PVM) In the proceedings of the STC's 48th annual conference.  
URL: <http://www.sct.org/proceedings/ConfProceed/2001/PDFs/STC48-000158.pdf>
- Dan Emory & Associates. 2005. "The decline of FrameMaker in the arms of Adobe." (29.01.2005)  
URL: <http://www.daube.ch/docu/fmhist06.html>
- Docu+Design Daube. 2004. "FrameMaker history." (29.01.2005)  
URL: <http://www.daube.ch/docu/fmhist00.html>
- Fibinger, Iris. 2002. "Single Sourcing in Technical Communication." (09.02.2004)  
URL: [http://www.svgopen.org/2002/papers/fibinger\\_\\_singlesourcing\\_in\\_technical\\_documentation/index.html](http://www.svgopen.org/2002/papers/fibinger__singlesourcing_in_technical_documentation/index.html)
- Hoft, Nancy L. 1995. International Technical Communication: How to Export Information about High Technology. New York: John Wiley & Sons, Inc.
- Holzner, Steven. 2001. Inside XML. Indianapolis: New Riders Publishing.
- Interleaf Inc. 1996-1999. "Quicksilver Documentation."
- Interleaf Inc. 1999. The Quicksilver XML Publisher Implementor's Guide. Massachusetts: Interleaf Inc.
- ISO/IEC. 1995. Guide 37: Instructions for use of products of consumer interest. Switzerland: ISO/IEC.
- Johnson, Mark. 2000. "Well-formed vs. Valid XML Documents." (13.03.2005)  
URL: [http://www.itworld.com/nl/xml\\_prac/11022000/](http://www.itworld.com/nl/xml_prac/11022000/)
- Kaleva, Lassi. 2000. "Selvitys yritysten teknisen viestinnän tutkimustarpeista." (05.5.2005)  
URL: <http://www.uta.fi/~trtysu/mmtc/suomi/selvitystyo.html>
- Koikkalainen, Tanja. 2002. "Single sourcing: a system for reusing information in documentation." Pro gradu thesis, University of Tampere

Landau, Sidney I. 1989. Dictionaries: The Art and Craft of Lexicography. Cambridge: Cambridge University Press

Manning, Steve. 2002. "Introduction to XML for Technical Writers." (09.02.2004)  
URL: <http://www.stc.org/confproceed/2002/PDFs/STC49-00037.pdf>

McGahey, Kitty. 2005. In an e-mail conversation on 11 April 2005.

Morris, Bob. 1998. "Interleaf FAQ." (21.02.2005)  
URL: <http://www.faqs.org/faqs/interleaf-faq/>

O'Hara, Frederick M., Jr. 2001. "A Brief History of Technical Communication." (3.3.2005)  
URL: <http://www.stc.org/confproceed/2001/PDFs/STC48-000052.pdf>

Quadralay Corporation. 2002. "WebWorks Publisher Standard Edition 7.0 Help."

Rockley, Ann. 2001 a. "Content Management for Single Sourcing." (13.03.2005)  
URL: [www.stc.org/confproceed/2001/PDFs/STC48-000171.PDF](http://www.stc.org/confproceed/2001/PDFs/STC48-000171.PDF)

Rockley, Ann. 2001 b. "The Impact of Single Sourcing and Technology." (27.03.2005)  
URL: [http://www.rockley.com/articles/Single\\_Sourcing\\_and\\_Technology.pdf](http://www.rockley.com/articles/Single_Sourcing_and_Technology.pdf)

Rockley, Ann and JoAnn Hackos. 1999. "Single Sourcing White Paper." (05.03.2005)  
URL: [http://www.infomanagementcenter.com/pdfs/white\\_paper\\_001.pdf](http://www.infomanagementcenter.com/pdfs/white_paper_001.pdf)

Ruini, Henri. 2001. "Tyyli- ja muunnoskielet." (13.03.2005)  
URL: <http://www.cs.helsinki.fi/u/ruini/structure/xml/intro/maarittelyt/tyyli.html>

Sankaranarayana, Ramesh. 2001. "History of XML." (12.03.2005)  
URL: <http://mobile.act.cmis.csiro.au/comp3410/History.asp>

Schriever, Karen A. 1997. Dynamics in Document Design: Creating Text for Readers. New York: John Wiley & Sons, Inc.

Scriptorium Publishing Services, Inc. 2004. "Integrating XML and FrameMaker." (30.01.2005).  
URL: <http://www.scriptorium.com/>

SGML Users' Group. 1990. "A Brief History of the Development of SGML." (05.03.2005)  
URL: <http://www.sgmlsource.com/history/sgmlhist.htm>

Simply Written Inc. 2002. "An Introduction to Single Sourcing." (12.04.2004)  
URL: <http://www.simplywritten.com/library/SSIntro.pdf>

W3C. 2001. "Extensible Stylesheet Language (XSL) Version 1.0." (13.03.2005)  
URL: <http://www.w3.org/TR/xsl/>

W3C. "The Extensible Stylesheet Language Family (XSL)." (27.03.2005)  
URL: <http://www.w3.org/Style/XSL/>

W3C. 1999. "XML Path Language (XPath)." (20.03.2005)  
URL: <http://www.w3.org/TR/xpath#section-Introduction>

Walsh, Norman 1998. "What is XML?" (09.02.2005)  
URL: <http://www.xml.com/pub/a/98/10/guide0.htm>

Web Design Group. 1998. "HTML 4.0 Elements." (17.08.2005)  
URL: <http://www.htmlhelp.com/reference/html40/olist.html>

Zhang, Allison. 1995. "Multimedia File Formats on the Internet: Chapter 8, Pictures." (25.8.2005)  
URL: <http://www.is-edu.hcmuns.edu.vn/WebLib/Books/Internet/multimed/image.htm>