Software Documentation

A Thesis

Submitted to the Department of Computer Science and Engineering

of

BRAC University

by

Tahmina Zaman Khan

ID: 02101083

CS

In Partial Fulfillment of the

Requirements for the Degree

of

Bachelor of Science in Computer Science

August 2006

Declaration

I hereby declare that this thesis is based on the results found by myself. Materials of work found by other researcher are mentioned by reference.

Signature of Supervisor

Signature of Author

Acknowledgement

It was my great pleasure to be able to conduct a research work and complete the thesis paper on “Software Documentation”.

I would like to extend my sincere thank to my supervisor Dr. Mumit Khan, Associate Professor, CSE Dept. BRAC University, Dhaka for his precious time and perpetual support extended relentlessly to fulfill my academic assignment.

Also I honor his immense contribution throughout the process of study. His contribution in the preparation of the concept paper, literature, methodology and writing this report is highly acknowledged. I am immensely grateful to him for his extensive assistance.

I am also thankful to my co-supervisor Mr. Matin Saad Abdullah for his helpful and appreciable assistance at the beginning of the study and clearing the subject matters.

Tahmina Zaman Khan
02101083
CSE Dept.
BRAC University.

Abstract

The main objective of my thesis is to generate a user manual that would be very much comprehensible at the same time well structured and would act as an effective navigator. Documentation is mainly requisite for better communication among the different members of a software development team, such as designers of finer grained components, builders of interfacing system, implementers, testers, performance engineers, technical managers, analysts, quality specialists. In order to develop a very comprehensive documentation there are certain conventions that are requisite to be taken care of. Those conventions and rules have been high lighted extensively.

There are different types of documentation based on the requirements of each individual associated with the software development life cycle and they are design, code, user, architectural, trade study and marketing are few to mention.

However, my focus area is user documentation. Unlike code documents, user documents are usually far divorced from the source code of the program, and instead simply describe how it is used. The use of XML and Docbook is there. DocBook simply provides a framework. All the presentation issues are devolved to style sheets.

Contents

List of Figures

CHAPTER 1: INTRODUCTION

Software Documentation

It is stated that any document (hardcopy or softcopy) which defines the functional requirements, design, implementation, operation or support arrangements that pertain to a software item or any data used by a software item is known as software documentation. Essentially it is written text that accompanies computer software. It either explains how it operates or how to use it. In fact, the term software documentation means different things to different people. It is mainly requisite for better communication among the different members of a software development team, such as designers of finer grained components, builders of interfacing system, implementers, testers, performance engineers, technical managers, analysts, quality specialists. The goal of software documentation is to provide a framework and model for recording the essential information needed throughout the development life cycle and maintenance of a software system. However, all documentation must be done abiding by certain basic rules for the convenience of the readers and this will be described shortly. There are different types of documentation meant to cater different categories of users. Such as:

Architectural: merely lays out the general requirements that would motivate the existence of a routine. A good architecture document is short on details but thick on explanation.

User: describes how the software is used.

Code: text attached intended to explain various operations. This writing can be highly technical and is mainly used to define and explain the APIs, data structures and algorithm.

Design: takes a much broader view and emphasize on WHY the concerned software is designed in that manner.

Trade study: It focuses on one specific aspect of the system and suggests alternate
approaches. It could be at the user interface, code, design, or even architectural level.

Marketing: used to allure the potential user, explain the product's functionality, and focus on the position of the product in comparison to other alternatives.

Nevertheless, different methodologies are there for each of the aforementioned approaches to documentations, where few of them have been covered in this paper. EEE/ANSI 1471, View and Beyond (V & B) and 4 + 1 Model are few to mention. These mainly focus the architectural view of the software.

This paper mainly is designed to encompass every nook and corner of the user documentation and tool and techniques associated with in creating an effective and efficient user manual. Which would enable any and every user to get benefited; in short it is aimed to create a very helpful navigator. In order to deal with the formatting and structuring of the document consistently, maintaining cohesiveness all along the way, DocBook has been incorporated. DocBook simply provides a framework and all the presentation issues are devolved to style sheets. DocBook has its own set of tags, processing tools and transformation tools.

CHAPTER 2: BASIC RULES FOR SOFTWARE DOCUMENTATION

2.1 The importance of audience:

The audience is a key element that many people fail to consider. When the job is to convey knowledge to someone, we need to understand what that person already knows and develop a sense of the teaching methods likely to work for that person. If it is a writing instruction for a customer service representative it is better not to assume they understand database theory or any of the associated terminology. It sounds obvious, but the failure to properly analyze intended audience is the single largest pitfall facing technical writers.

2.1.1 Examining the software to determine the user base

If we don't understand our audience then chances are we won't be able to write documentation they'll understand. Evaluating the audience can be a difficult task but it is absolutely essential to do so. The best way to start analyzing the audience is by evaluating the product we need to document. Quite a lot about our intended audience can be told from the software since they are also the intended users of the software.

If we're tasked with teaching developers how to write front ends for the product using Visual Basic, then we already know quite a lot about our audience. We know that:

- They are developers
- They know Visual Basic
- They are familiar with user interface design

In actuality, perhaps some of the readers will not fit this description, but it is our task to define the supported skill set, make it clear to the readers what we expect them to already know, and then write the book as if they know precisely that - no less and no more. If some of the readers do not meet all of the prerequisites we've set out for the book then it is their responsibility to gain those skills before attempting to use our book.

In some cases, our intended audience will not be so clearly defined. Parts of the book may include overview information while other parts delve deep into the inner workings of the product. In these cases we'll need to work particularly hard to develop a list of things and terminology all of the readers will understand, and perhaps divide the book into several sections for analysis. Just it should be made sure to clearly state whom each section is intended for within the introductory material of the documentation.

2.2 Importance of Writing Procedures

Most technical writers spend at least part of their time writing procedures. Procedures are essentially detailed instructions for performing one or more tasks. Unless the documentation is strictly conceptual (explaining a topic rather than telling the readers how to use the software), it is required to write procedures. Since readers will be following instructions on real applications with real data, it is particularly important that our directions be clear, concise, accurate, and easy to follow.

2.2.1 The importance of word choice and consistent writing

A good technical writer not only understands his/her intended audience, s/he understands the importance of clear and concise writing. Although most writers dream of beautiful prose that can move readers to another plane of existence, technical writing is all about removing any possible ambiguities so the reader

understands precisely what to do. Short descriptive active sentences are generally better than long flowery lines filled with metaphor. Word choice is paramount to a technical writer. If there is any possible way an instruction can be misread or misconstrued someone will do so. When possible consequences of wrong actions are data corruption and data loss it becomes very important that users understand what we are telling them to do.

Consistency is also important. If we lead our users to expect all functions to appear in boldface followed by a set of parentheses, then it must be made sure all functions appear that way. Otherwise, the readers may not realize that they're all functions and could get confused. Similarly, we should keep a similar style and vocabulary throughout each document. If we're writing an introduction filled with very general and basic conceptual material we must make sure to avoid as much technical terminology as possible and that we always define all terminology used. If several writers are collaborating on the same document it must be made sure that individual voice doesn't shine through. When the document is finished we should not be able to tell where one person stopped and another picked up. We shouldn't even be able to tell that more than one person worked on the documentation.

2.2.2: Use active language

A procedure tells a reader to perform a specific action or set of actions. Good instructions use active language, commanding the reader to act rather than discussing actions that have been taken or could be taken by others. There's a big difference between saying "the dropdown box in the corner of the page can be used to select the date" and "select a date using the dropdown box in the corner". The first informs the reader what that particular dropdown box does and the second tells them to do it. The text introducing a procedure can discuss when to perform the procedure; the procedure itself needs to tell the user how to do the needed actions and assume that they are, in fact, doing them.

2.2.3 Use short phrases and sentences

It's important to give each instruction as clearly as possible. The fewer words used to convey a particular task, the less places there are for a user to get confused. Anyplace if it is possible to use a single word to convey an action, it is best to do so.

Avoid any elaboration or explanation within the procedure itself. If we need to provide options or explain why a particular task is important, we should do it in paragraph form before beginning the procedure. If the user can choose more than one way to perform a specific task, it is better to present each procedure separately after explanatory text discussing when that particular option is a good choice.

2.2.4 Use ordered lists

By definition a procedure consists of a set of tasks that need to be performed in a specific order. By presenting these tasks in an ordered list, with steps 1, 2, 3, etc., it reinforces the need to follow a set order. Numbered lists have an ingrained implied order that's almost subconscious and thus provide much stronger impetus that simply supplying paragraphs filled with "Do this then do that" terminology.

Ordered lists also help bone down the language to its simplest, and thus clearest, form. By removing all of the extraneous connective words needed to indicate order in paragraph form it is best to remove verbiage that could confuse the readers.

2.2.5 Provide one instruction per step

One of the common mistakes people make when writing procedures is trying to stuff several steps into one line. They think that each instruction is simple so they can safely combine a few easy directions into one combined step. Unfortunately no matter how simple those individual instructions are any step that tells us how to do more than one thing is potentially confusing. Avoid that confusion by limiting each step to one instruction no matter how simple or mundane. For instance, when telling users how to create a new account you could say:

Enter the desired username and password into the appropriate fields
Select the desired permissions for the new user
Hit enter to add the user

But this is clearer:

1. Enter the desired username in the username field
2. Enter the desired password in the password field
3. Select the Read checkbox if this user has read privileges
4. Select the Write checkbox if this user has write privileges
5. Select the Exec checkbox if this user has execute privileges
6. Hit enter to add the user

2.2.6 Be consistent

Consistency is important in all documentation, but especially so when writing procedures. In particular, it is paramount to use consistent names for screen elements like buttons, forms, tabs, and windows. Many of these elements won't be named within the software so we'll have to name them. Choose a logical name that makes it obvious what particular element we're referring to and then propagate that name throughout the entire documentation set.

Collect all of the names created and is required to make sure they're added to a common database of product terminology. In an ideal world, these terms would be added to a group style guide or glossary, but even if we don't do that, is it needed to make sure that they are preserved and available to everyone working on that product.

Don't feel like we need to come up with ten creative ways to tell people how to press a button. Tell them to press the button (or tap it, or choose it, or whatever terminology we've decided on) each and every time. We shouldn't break out the thesaurus, but rather use repetitiveness to really pound home clarity.

In addition to consistent use of names and terminology in general, it's also important to use parallel constructs within the procedures. The human brain looks for parallelism and is bothered when it's not there (even if it doesn't completely understand what's wrong with a particular sentence). By beginning each step with a similar phrase and tone, we're allowing the user to ignore the writing and concentrate on the content.

2.3 The importance of organization

Technical writers tend to pay more attention to organization and the order material gets presented than most other writers. In many cases, other writers have some leeway in presentation. More often than not the material can be organized in several different ways and still retain its effectiveness. This is not often true for technical writing. Since much of the time we're either providing instructions - by their very nature a series of ordered steps - or providing background information that builds on prior material and tasks already completed organization is extremely important. If we fail to give users an essential step of any procedure they will not successfully complete that procedure. If we fail to discuss a basic topic before moving on to a related advanced topic they most likely will get confused.

It's our job to analyze everything the user needs to do to successfully use a product and determine the order they need to do those items. Then we need to determine what the user is likely to already know and make sure that we provide every other scrap of needed information. In addition we must analyze how someone who doesn't understand the product as well as we does would think about the product and provide logical entry points to the necessary material given those expectations. It doesn't help to provide every scrap of information our users need if they can't find that information within our document

2.4 The importance of indexing

Indexing is an important but often overlooked step in this process. Most users are going to either turn directly to the table of contents or turn directly to the index and start looking for the material they think they need. We need to make sure that information can be found from either method even if the user doesn't yet know precise terminology or an exact word to look up. Cross-indexing is an art, one that eludes many otherwise very competent technical writers. Indexing is a skill that needs to be developed and practiced regularly.

2.5 The importance of editing and review

We might think the process of technical writing is complete once the last word is put on the page, but it isn't. Every document needs to be reviewed and edited. In some cases we'll be fortunate enough to have an actual editor in place to edit our document, but most technical writers must rely on another technical writer or even do it themselves. In addition to the normal editing tasks of checking grammar, general word usage, and spelling, technical editing includes several other elements including checks for consistent word usage, checks that a consistent audience is maintained, and checks that the organizational choices make sense.

Reviewing often occurs at the same time as the editing process but it serves a very different purpose. While editing ensures a clean grammatically correct document with consistent style, it doesn't test the accuracy of the content in any way. Reviewing does. If possible, we should have at least one of the developers who wrote the product participate in a review as well as at least one person who fits the target audience profile. If the person within the target audience doesn't understand everything we wrote then even if it's accurate we haven't successfully met your mandate and need to re-write the document accordingly. If the developer finds inaccuracies or points out areas where the product was changed then similarly we need to fix those before declaring the document finished.

CHAPTER 3: ARCHITECTURAL DOCUMENTATION

Software's architecture for a program or computing system consists of the structure or structures of that system, which comprise elements, the externally visible properties of those elements, and the relationships among them the quality attributes of a software system, such as performance, modifiability, and security, are bound up in its software architecture. A system with the wrong architecture will be a failure.

Software architecture also determines the blueprint for the project developing the software. Teams are formed around architectural elements, which are the units of implementation, unit testing, integration, configuration management, documentation, and a host of other activities.

Unlike code, architecture is a design artifact largely intended for use and analysis by humans. Hence, representing it in a readable, accessible fashion for its stakeholders becomes an issue of importance. Architecture gives the marching orders to implementers, telling them what pieces to build and how those pieces should behave and interact with each other. It also determines the project structure for managers, who use it to plan, schedule, and budget. It gives the first glimpse of the system to maintainers who must change the architecture and new team members who must become familiar with it.

Therefore, architecture documentation has emerged as an important architecture-related practice. In 2002, researchers at the Carnegie Mellon Software Engineering Institute (SEI) completed Documenting Software Architectures: Views and Beyond, which puts forth a documentation philosophy as well as a detailed approach. The philosophy is embodied in the title: "views and beyond." The V&B approach, as it is known, holds that documenting

software architecture is a matter of choosing a set of relevant views of the architecture, documenting each of those views, and then documenting information that applies to more than one view or to the set of views as a whole. The last step ties the views together and makes them become a holistic and integrated representation of the architecture, as opposed to disjoint snapshots taken from different angles. The detailed approach includes a method to choose the most relevant views, standard templates for documenting a view and documenting the information beyond views, and definitions of the templates' content.

While the V&B approach was being solidified, a new recommended best practice was being formed by the Institute of Electrical and Electronics Engineers (IEEE). The IEEE Architecture Planning Group (APG) was formed in August 1995 and chartered by the IEEE Software Engineering Standards Committee (SESC) to set a direction for incorporating architectural thinking into IEEE standards. The result of the APG's deliberations was to recommend an IEEE activity with goals as to define useful terms, principles, and guidelines for the consistent application of architectural precepts to systems throughout their life cycle, to elaborate architectural precepts and their anticipated benefits for software products, systems, and aggregated systems ("systems of systems"), to provide a framework for the collection and consideration of architectural attributes and related information for use in IEEE standards, to provide a useful roadmap for the incorporation of architectural precepts in the generation, revision, and application of IEEE standards

In April 1996, the SESC created the Architecture Working Group (AWG) to implement those recommendations eventually became ANSI/IEEE Std. 1471-2000]. That standard, henceforth referred to as 1471, is a recommended practice that addresses the activities of the creation, analysis, and sustainment of architectures of software-intensive systems and the recording of such architectures in terms of architectural descriptions. The standard establishes a

conceptual framework for architectural description and defines the content of an architectural description.

Interest in 1471 is growing and will likely continue to grow. Although it is impossible to tell how many projects invoke it, it is mandated for use in the Future Combat Systems (FCS) project, a U.S. Army command-and-control system that is expected to comprise over 30 million lines of code and (therefore) whose software architecture is of supreme importance.

3.1 The Views and Beyond Approach to Software Architecture Documentation

3.1.2 A Multi-View Approach

Modern software architecture practice embraces the concept of architectural views. A view is a representation of a set of system elements and the relations associated with them. Views are representations of the many system structures present simultaneously in software systems. Modern systems are too complex to be grasped all at once. Instead, we restrict our attention at any one moment to one (or a small number) of the software system's structures, which we represent as views. Some authors prescribe a fixed set of views with which to engineer and communicate architecture; for example, the Rational Unified Process (RUP), which is based on Kruchten's 4+1 view approach to software and the Siemens Four Views model. A recent trend, however, is to recognize that architects should produce whatever views are useful for the system at hand, and the V&B approach adopts that policy. This trend leads to the fundamental philosophy of the V&B approach stated earlier: Documenting architecture is a matter of documenting the relevant views and then adding documentation that applies to more than one view.

3.1.3 Different Kinds of Views

There is an almost unlimited supply of views to choose from. To lend some order to an otherwise chaotic collection of possible views, it's helpful to think about views in groups, according to the kind of information they convey:

Module views describe how the system is to be structured as a set of code units. Component-and-connector (C&C) views describe how the system is to be structured as a set of interacting runtime elements. Allocation views describe how the system relates to non-software structures in its environment. A particular view of a system may fall squarely into one of these categories or combine information from more than one category.

3.1.4 Styles

A view is a representation of a structure that is present in a software system. One might show the hierarchical decomposition of the system's functionality into modules or how the system is arranged into layers; another might show how the system accomplishes work through communicating processes or the interaction of clients and servers. Still another might show how software elements are deployed onto hardware processing and communication nodes.

An architect chooses the structures to work with and designs them to achieve particular quality attributes using architectural styles.[1] A style is a specialization of element types (e.g., "client," "layer") and relationship types (e.g., "is part of," "request-reply connection," "is allowed to use"), along with any restrictions (e.g., "clients interact with servers but not each other" or "all the software comprises layers arranged in a stack such that each layer can only use software in the next lower layer").

Styles are documented in a style guide that defines each style by defining the element types and relationship types indigenous to the style, along with any semantic restrictions on their use. It lists what design problems the style is and is not good at addressing. The guide also discusses any notations or analytical approaches available to the architect using that style and refers to any related styles.

3.1.5 Choosing the Views

The V&B approach to choosing the views to document is a simple three-step procedure based on the structures that are inherently present in the software and on the stakeholders and the concerns they have that would motivate documenting the corresponding view. The steps are described below.

Step 1: Produce a Candidate View List

Begin by building a stakeholder/view table for your project. Enumerate the stakeholders for your project's software architecture documentation down the rows. Be as comprehensive as you can. For the columns, enumerate the views that apply to your system. Some views (e.g., decomposition, uses, and work assignment) apply to every system, while others (e.g., pipe-and-filter, layered) only apply to systems designed according to the corresponding styles.

Once you have the rows and columns defined, fill in each cell to describe how much information the stakeholder requires from the view: none, overview only, or detailed. We encourage architects to hold a workshop with stakeholders or their representatives to begin a dialogue about what information they will need from the documentation. The candidate view list consists of those views in which some stakeholder has a vested interest.

Step 2: Combine Views

The candidate view list from Step 1 is likely to yield an impractical number of views. Step 2 winnows the list to a manageable size.

First, look for views in the table that require only overview depth or that serve very few stakeholders. See if the stakeholders could be equally well served by another view that has a stronger constituency.

Next, look for views that are good candidates to become combined views. A combined view shows information native to two or more separate views. A rule of thumb is that if there is a strong correspondence between the elements in two views, they are good candidates to be combined.

Step 3: Prioritize

After Step 2; you should have the minimum set of views needed to serve your stakeholder community. At this point, you need to decide what to do first. For example, some stakeholders' interests supersede others. The project manager of a company you are partnering with often demands attention and information early and often, and you may want to cater to his/her needs first.

3.1.6 A Template for Views and Information beyond Views

No matter the view, the documentation for it is placed into a standard organization or template comprising seven parts: A primary presentation shows the elements and relationships among them that populate the portion of the view shown in this view packet. The primary presentation should contain the information you wish to convey about the system (in the vocabulary of that view) first. The primary presentation is usually graphical. If so, it must be accompanied by a key that explains or points to an explanation of the notation.

An element catalog details the elements (and their properties, including interfaces) depicted in the primary presentation. In addition, if elements or relations relevant to this view packet were omitted from the primary presentation, the catalog is where they are introduced and explained.

A context diagram shows how the system (or portion of the system) depicted in the primary presentation relates to its environment.

A variability guide shows how to exercise any variation points that are part of the architecture shown in this view packet.

An architecture background or rationale explains why the design reflected in the view packet came to be.

An "other information" section contains items that vary according to the standard practices of each organization or the needs of the particular project.

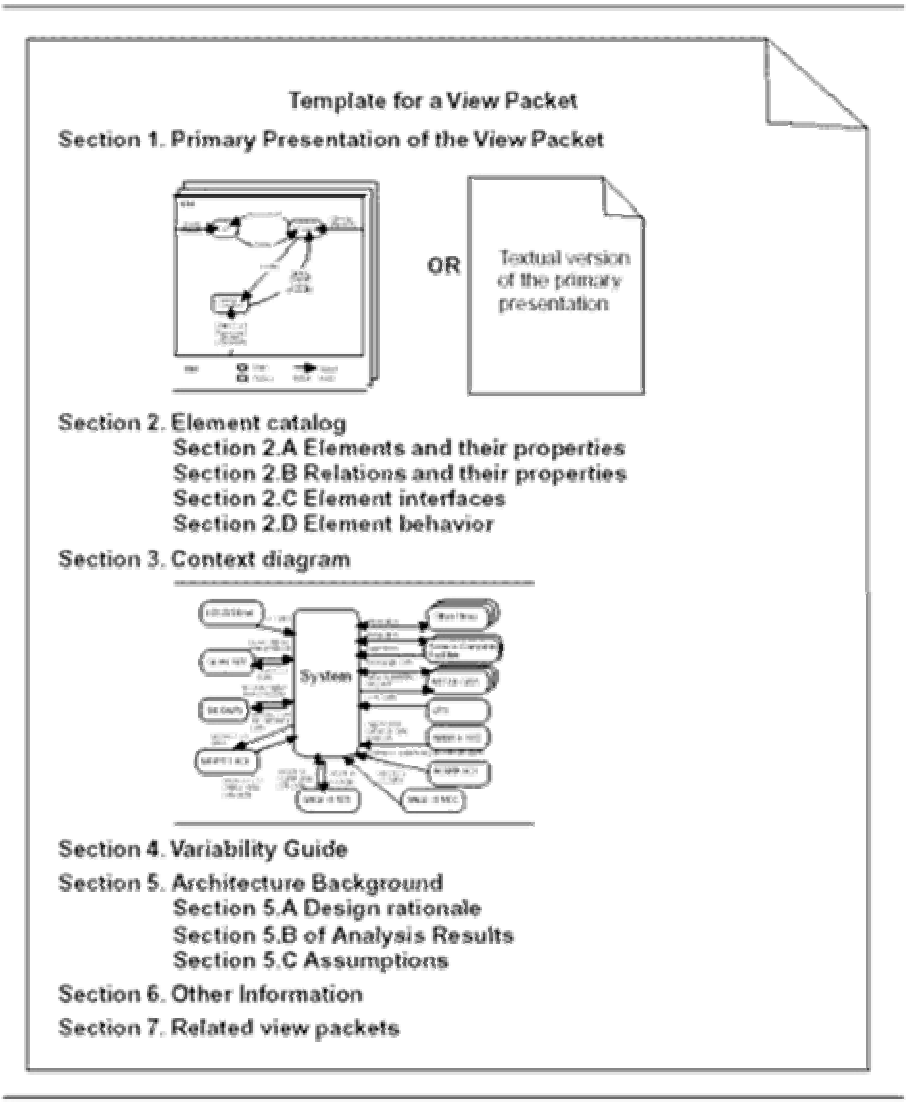

Figure 3.1 The Template for a View

The final piece of architecture documentation is the information that applies to more than one view and to the entire package. It ties together the views and provides a holistic picture of the total design. Cross-view or "beyond views" documentation consists of the following sections:

Documentation roadmap. The documentation roadmap is the reader's introduction to the information that the architect has chosen to include in the suite of documentation. A roadmap begins with a brief description of each part of the documentation package. For each view in the package, the roadmap gives a

description of the view's element types, relation types, and property types. The roadmap also gives a description of the view's purpose. The information can be presented by listing the stakeholders who are likely to find the view of interest and by listing a series of questions that can be answered by examining the view. The roadmap follows with a section describing how various stakeholders might access the package to help address their concerns. This section might include short scenarios such as "a maintainer wishes to know the units of software that are likely to be changed by a proposed modification."

View template. A view template is the standard organization for a view. Its purpose is to help a reader navigate quickly to a section of interest. It helps a writer organize the information and establish criteria for knowing how much work is left to do.

System overview. A system overview is a short prose description of what the system's function is, who its users are, and any important background or constraints. The purpose is to provide readers with a consistent mental model of the system and its purpose.

Directory. The directory is simply an index of all the elements, relations, and properties that appear in any of the views, along with a pointer to where each one is defined and used.

Mapping between views. This shows the correspondence between individual elements in different views. Helping a reader or other consumer of the documentation understand the relationship between views will help that reader gain a powerful insight into how the architecture works as a unified conceptual whole.

Project glossary and acronym list. The glossary and acronym list define terms unique to the system that have special meaning. These lists, if they exist as part

of the overall system or project documentation, might be given as pointers in the architecture package.

Cross-view rationale. This section documents the reasoning behind decisions that apply to more than one view. Prime candidates for cross-view rationale include documentation of background or organizational constraints that led to decisions of system-wide import.

The following figure illustrates the seven pieces of cross-view or "beyond view" documentation.

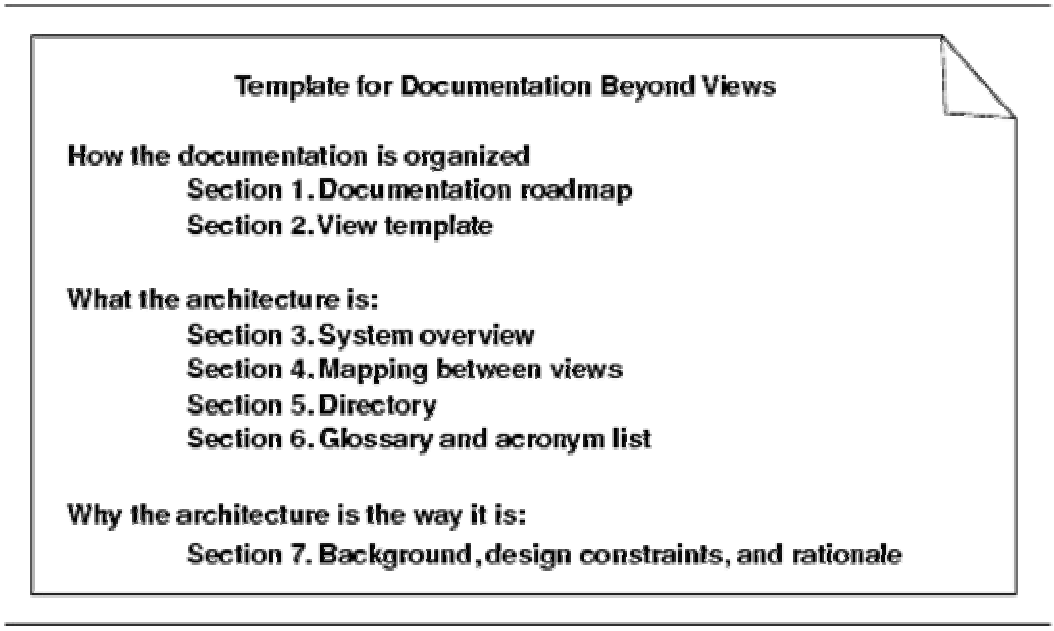

Figure 3. 2: The Template for Documentation beyond Views

3.2 1471 (IEEE Recommended Practice for Architectural Description of Software-Intensive Systems)

1471 draws on experience from industry, academia, and other standards bodies. The recommendations of 1471 center on two key ideas: (1) a conceptual framework for architectural description and (2) a statement of what information must be found in any 1471-compliant architectural description. The conceptual

framework described in the standard ties together such concepts as system, architectural description, and view.

The following figure summarizes a portion of this framework in UML. In 1471, views have a central role in documenting software architecture. In the standard, each view is "a representation of a whole system from the perspective of a related set of concerns." The architectural description of a system includes one or more views. In this framework, a view conforms to a viewpoint. A viewpoint is "a pattern or template from which to develop individual views by establishing the purposes and audience for a view and the techniques for its creation and analysis. In 1471, the emphasis is on what drives the perspective of a view or a viewpoint. Viewpoints are defined with specific stakeholder concerns in mind, and the definition of a viewpoint includes a description of any associated analysis techniques.

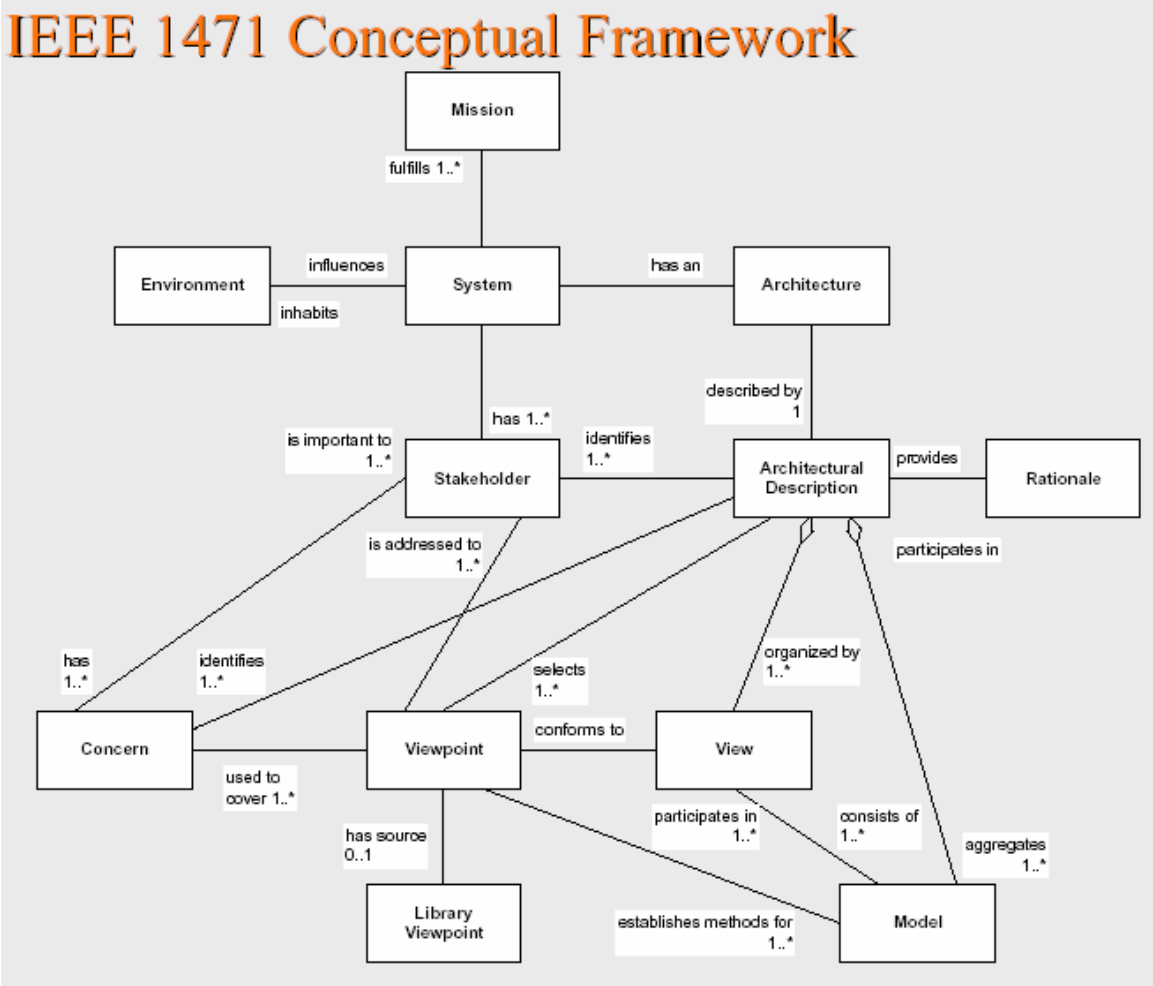

Figure 3.3: Excerpt from Conceptual Framework of 1471[2]

In addition to the conceptual framework, 1471 includes a statement of what information must be in any compliant architectural description. These "shalls" include the following:

Identification and overview information: This information includes the date of issue and status, identification of the issuing organization, a revision history, a summary and scope statement, the context of the system, a glossary, and a set of references.

Stakeholders and their concerns: The architecture description is required to include the stakeholders for whom the description is produced and who the

architecture is intended to satisfy. It is also required to state "...the concerns considered by the architect in formulating the architectural concept for the system." At a minimum, the description is required to address users, acquirers, developers, and maintainers.

Viewpoints: An architecture description is required to identify and define the viewpoints that form the views contained therein. Each viewpoint is described by its name, the stakeholders and concerns it addresses, any language and modeling techniques to be used in constructing a view based on it, any analytical methods to be used in reasoning about the quality attributes of the system described in a view, and a rationale for selecting it.

Views: Each view must contain an identifier or other introductory information, a representation of the system (conforming to the viewpoint), and configuration information.

Consistency among views: Although the standard is somewhat vague on this point, the architecture description needs indicate that the views are consistent with each other. In addition, the description is required to include a record of any known inconsistencies among the system's views.

Rationale: The description must include the rationale for the architectural concepts selected, preferably accompanied by evidence of the alternatives considered and the rationale for the choices made.

3.3 The “4+1” View

This approach organizes the description of a software architecture using several concurrent views, each one addressing one specific set of concerns.

An Architectural Model

The 4+1 view model is a useful, standardized method for studying and documenting a software system from an architectural perspective. Software architecture deals with the design and implementation of the high-level structure of the software. It is the result of assembling a certain number of architectural elements in some well-chosen forms to satisfy the major functionality and performance requirements of the system, as well as some other, non-functional requirements such as reliability, scalability, portability, and availability. Software architecture deals with abstraction, with decomposition and composition, with style and esthetics. In order to eventually address large and challenging architectures, the model is made up of five main views

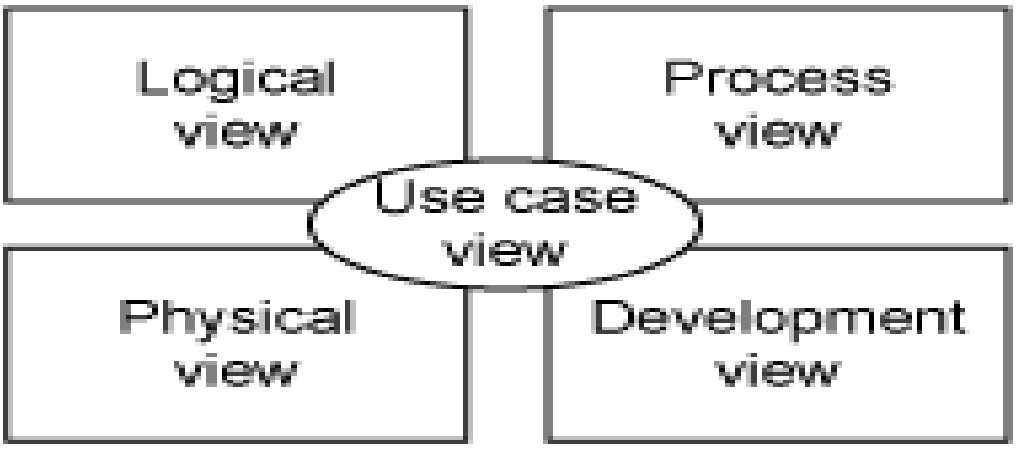

Figure 3.4: shows the five views of the 4+1 view model.

Each of the five views in the 4+1 view model highlights some elements of the system, while intentionally suppressing others. The 4+1 view model is an excellent way for both architects and other team members to learn about a system's architecture. Architects use it to understand and document the many layers of an application in a systematic, standardized way. Documents created using the 4+1 view process are easily used by all members of the development team.

The first four views represent the logical, processing, physical, and developmental aspects of the architecture. The fifth view consists of use cases and scenarios that might further describe or consolidate the other views.

- The **logical view:** describes the (object-oriented system) system in terms of abstractions, such as classes and objects. The logical view typically contains class diagrams and collaboration diagrams. Other types of diagrams can be used where applicable.
- The **development view:** describe the structure of modules, files, and/or packages in the system. The package diagram can be used to describe this view.
- The **process view:** describes the processes of the system and how they communicate with each other.
- The **physical view**: describes how the system is installed and how it executes in a network of computers. Deployment diagrams are often used to describe this view.
- The **use case view:** describes the functionality of the system. This view can be described using case diagrams and use case specifications.

The logical view

The 4+1's logical view supports behavioral requirements and shows how the system is decomposed into a set of abstractions. Classes and objects are the main elements studied in this view. You can use class diagrams, collaboration diagrams, and sequence diagrams, among others, to show the relationship of these elements from a logical view.

Class diagrams show classes and their attributes, methods, and associations to other classes in the system. The class diagram in Figure 5 shows a simple use case of an ordering system (or part of one). The customer can have from zero to several orders, and an order can have from one to several items.

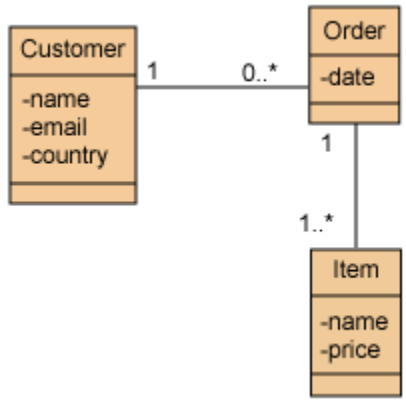

Figure 3.5: A class diagram of an ordering system

While useful, the class diagram hardly gives you a complete picture of the system. For one thing, class diagrams are static, so they tell you nothing about how the system will react to user input. For another, class diagrams are often too detailed to offer a useful overview of the system. You can only learn so much from studying a class diagram for a system comprised of thousands of classes.

You can use collaboration diagrams (or communication diagrams) and sequence diagrams to see how objects in the system interact. A collaboration diagram is a simple way to show system objects and the messages and calls that pass between them. Figure 6 is a simple collaboration diagram. Note that each message is assigned a number that indicates its order in the sequence.

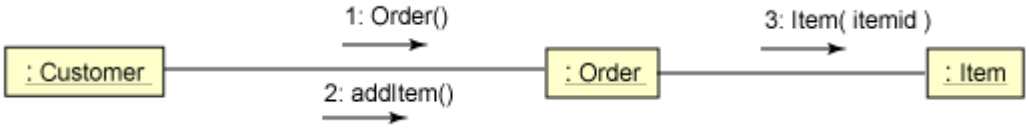

Figure 3.6: A collaboration diagram of an ordering system

Collaboration diagrams are very practical for showing a birds-eye view of collaborating objects in the system. If you want a more detailed window into the system's logic you might want to try drawing a sequence diagram. Sequence diagrams provide more detail than collaboration diagrams, but still let you study the system from a distance. Architects and designers often use sequence

diagrams to fine-tune system design. For example, looking at the sequence diagram in Figure 4 might lead you to change a number of the system's method calls to reduce their number. Alternately, you might change the design by creating a vector (or similar collection) of all the Items. You could then pass the vector and a Customer id to the Order constructor. (Note that doing this would change the roles of the Customer and Order classes completely.)

The development view

The development view is used to describe the modules of the system. Modules are bigger building blocks than classes and objects and vary according to the development environment. Packages, subsystems, and class libraries are all considered modules. Figure 7 is a package diagram showing how packages are nested in the system.

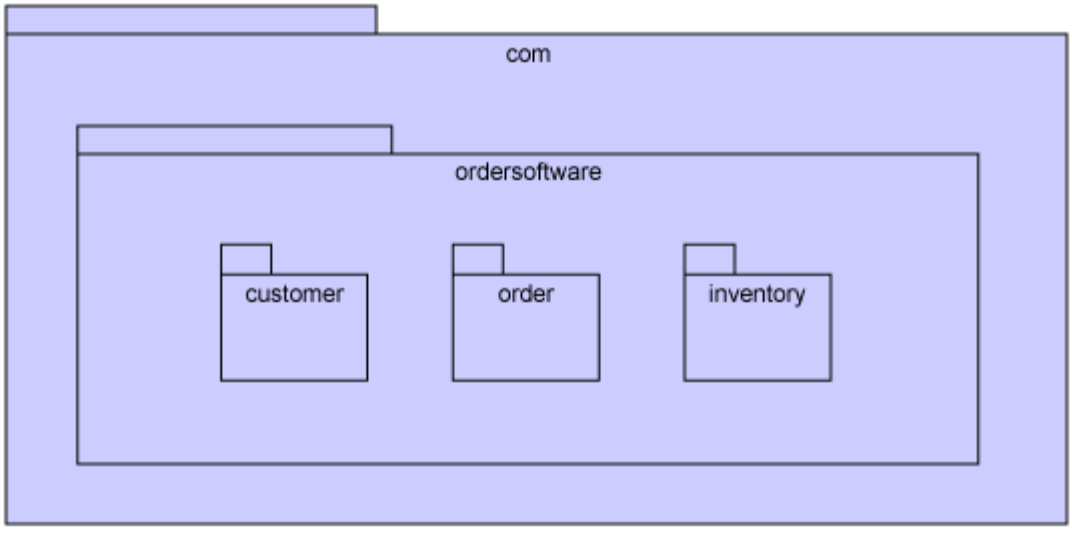

Figure 3.7: A package diagram shows how packages are nested in the system

You can also use the development view to study the placement of actual files in the system and development environment. Alternately, it is a good way to view the layers of a system in a layered architecture. A typical layered architecture might contain a UI layer, a Presentation layer, an Application Logic layer, a Business Logic layer, and a Persistence layer.

The process view

The process view lets you describe and study the system's processes and how they communicate, if they communicate with each other at all. An overview of the processes and their communication can help you avert unintentional errors. This view is helpful when you have multiple, simultaneous processes or threads in your software.

For example, Java servlets usually create threads of one servlet instance to serve requests. Without access to a process view, a developer might unintentionally store something in the servlet class's attributes, which could lead to complex errors if other threads did the same. The process view could reduce this type of problem by describing clearly how to communicate.

The process view can be described from several levels of abstraction, starting from independently executing logical networks of communicating programs. The process view takes into account many of the nonfunctional requirements or quality requirements (which last month's column talked about) like performance, availability, etc. Activity diagrams are quite often used to describe this view.

The physical view

The physical view describes how the application is installed and how it executes in a network of computers. This view takes into account nonfunctional requirements like availability, reliability, performance, and scalability.

Figure 8 is a deployment diagram of the example ordering system. It has one node for users who run the Web browser on their own computers. The ordering system and database are on their own nodes. The nodes contain one or more components, which can be either larger entities or smaller actual components.

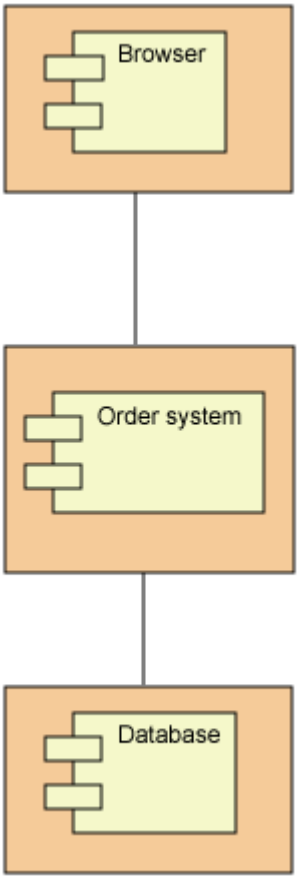

Figure 3.8: A deployment diagram of an ordering system

The 'plus-one' view

The "plus-one" view of the 4+1 view model consists of use cases and scenarios that further describe or consolidate the other views. Use cases represent the functional side of the system. In the case of the 4+1 model they are used to explain the functionality and structures described by the other views. The use case view consists of use case diagrams and specifications detailing the actions and conditions inside each use case.

CHAPTER 4: USER DOCUMENTATION/ MANUAL

Unlike code documents, user documents are usually far divorced from the source code of the program, and instead simply describe how it is used.

In the case of a software library the code documents and user documents could be effectively equivalent and are worth conjoining, but for a general application this is not often true. On the other hand, the lisp machine grew out of a tradition in which every piece of code had an attached documentation string. In combination with strong search capabilities (based on a Unix-like apropos command), and online sources, Lisp machine users could look up documentation and paste the associated function directly into their own code. This level of ease of use is unheard of in putatively more modern systems.

Typically, the user documentation describes each feature of the program, and the various steps required to invoke it. A good user document can also go so far as to provide thorough troubleshooting assistance. It is very important for user documents to not be confusing, and for them to be up to date. User documents need not be organized in any particular way, but it is very important for them to have a thorough index. Consistency and simplicity are also very valuable. User documentation is considered to constitute a contract specifying what the software will do and should be free from undocumented features.

There are three broad ways in which user documentation can be organized. A **tutorial approach** is considered the most useful for a new user, in which they are guided through each step of accomplishing particular tasks. A **thematic approach**, where chapters or sections concentrate on one particular area of interest, is of more general use to an intermediate user. The final type of

organizing principle is one in which commands or tasks are simply listed alphabetically, often via cross-referenced indices. This latter approach is of the greatest use to advanced users who know exactly what sort of information they are looking for. A common complaint among users regarding software documentation is that only one of these three approaches was taken to the near-exclusion of the other two

It is common to limit provided software documentation for personal computers to online help that give only reference information on commands or menu items. The job of tutoring new users or helping more experienced users get the most out of a program is left to private publishers, who are often given significant assistance by the software developer.

4.1 The following is a generic user manual structure:

1. Introduction
   - The product - introduce the product to the user.
   - The user manual
     - Scope/Purpose
     - Flow
     - Conventions
     - Glossary
2. Installing the software (assuming it's not a separate guide)
   - System requirements
     - Platform Support
   - Information/resources required in the process of installation
   - Installation steps

At this point there is usually a decision to be made about how to depict installation procedures for different platforms. The main criteriion here is how different the procedures are - if the steps are drastically different, you will have to

explain the procedure separately for each platform. But if the steps are not very different, you could choose the most common platform as your base, and wherever the steps are different, indicate the steps for the different platforms as indented text.

3. Using the software
    - Introduction
        - Purpose of the software
        - What it does and does not do (list the exact tasks)
        - User levels and the implications (segregate the user and admin level tasks)
    - Best configuration (for example, best viewed in 640x480 resolution)
    - Invoking the software
    - Interface elements
    - Steps to perform the required tasks
4. Administration
    - Reiterate the administration level tasks
    - Segregate (if possible) into administration, maintenance and troubleshooting functions, and then get into explanations
    - Always lead in to a task with scenarios (for example, you need to shut down the server over the weekends and at the end of the day - here's how...)
    - Also, try and bring out exceptional scenarios at the same time (to continue the above example, the administrator would not shut down the server over the weekend if there has been a request for remote access by one of the users)
5. Troubleshooting
    - For each error condition describe:
        - The error message displayed
        - What it means and what is the implication with respect to the attempted action (for example, the user will have to re-enter information)

- Steps to take to rectify the error

6. Appendix

An appendix allows you to expound on peripheral information that would be detracting when given in the main body. Detailed diagrams, flow charts, or references to books/tutorials on related software could be included here.

Here are some quick tips to assist you in developing this structure:

- Be visual: The most comforting thing for the user will be to see on screen what they've seen on the manual's pages, or vice-versa. Try and use screen grabs and small schematic diagrams wherever appropriate.
- Importance of relevant analogies: Essential if your software introduces concepts new to the user.
  - Use of transition words: "because", "therefore" and "consequently" are powerful words when talking about cause-effect relationships that the user isn't aware of.

4.2 Need to review

User reviews are a tad trickier than the others are because of the lack of resources. First, you may not have access to the actual users to review your document. And second, they may not really be motivated at that point to take the time to review your document. The workaround is to use your marketing and QA departments, and perhaps the people from the customer's end who are involved in the project.

Once the reviews are in, you need to get down to implementing the changes suggested. One tip would be to start revision on a document only after all the review comments are in. Also, while we won't get into the art of accepting feedback, you have to be in control of the changes that you agree to make. While you can change the information quite a bit at the time of the first review, you

should try and restrict your changes to corrections only after the second review; structural changes this late in the process will throw you off.

A characteristic of documentation is that if you notice even one inaccuracy in a document, it will put you off going through the rest of it. The gravity of this increases manifold when you're talking about a user who's looking to this document to understand your software. Ensuring that your manual reflects the latest version of the software is crucial, and this is where tying the document version number with that of the software comes in.

Another consideration here is version nomenclature. You could tie this in with the software, using x.y nomenclature that has x changing with every baseline change and y changing for every intermediate release of the document. Also, when you revise the document, you should record the reason/description of the change in the document's revision log.

With all of this behind you, you will finally be ready to release the manual. The following frills will complete the package:

1. Cover page
   - The name of the software should be written in accordance with the brand decided.
   - The version number of the software should be clearly stated.
   - The name of the developer with address and contact numbers.
2. Table of contents
   - The topics should be linked to the matter inside.
3. Notifications for proprietorship and confidentiality
4. Headers and footers
   - Headers could include the project name and version number of the document.
   - Footers can have the page numbers and a short confidentiality notice.

It might also be a good idea to include a feedback form as the last page, as your users will probably get back to you with suggestions. This will be especially useful if there is a second phase of development for the software.

CHAPTER 5: BEST PRACTICE METHOD FOR GENERATING EFFECTIVE USER DOCUMENTATION/ MANUAL

5.1 Markup: A General Overview

A markup language is a system for marking or tagging a document to define the structure of the document. You may add tags to your document to define which parts of your document are paragraphs, titles, sections, glossary items (the list goes on!). There are many markup languages in use today. XHTML and HTML will be familiar to those who author web documents. The LDP uses a markup language known as DocBook. Each of these markup languages uses its own "controlled vocabulary" to describe documents. For example: in XHTML a paragraph would be marked up with the tagset while in DocBook a paragraph would be marked up with <para></para>. The tagsets are defined in a quasi dictionary known as a Document Type Definition (DTD).

Markup languages also follow a set of rules on how a document can be assembled. The rules are either SGML (Standard Generalized Markup Language) or XML (eXtensible Markup Language). These rules are essentially the "grammar" of a document's markup. SGML and XML are very similiar. XML is a sub-set of SGML, but XML requires more precise use of the tags when marking up a document. The LDP accepts both SGML and XML documents, but prefers XML.

There are three components to an XML/SGML document which is read by a person.

* Content. It is good to remember that this is the most important piece. Many authors will write the content first and add their markup later. Content may

include both plain text and graphics.

* Markup. To describe the structure of a document a controlled vocabulary is added on top of the content. It is used to distinguish different kinds of content: paragraphs, lists, tables, warnings (and so on). The markup must also conform to either SGML or XML rules.

* Transformation. Finally the document is transformed from DocBook to PDF, HTML, PostScript for display in digital or paper form. This transformation is controlled through the Document Style Semantics and Specification Language (DSSSL). The DSSSL tells the program doing the transformation how to convert the raw markup into something that a human can read.

A schematic diagram of how the aforementioned components function in totality

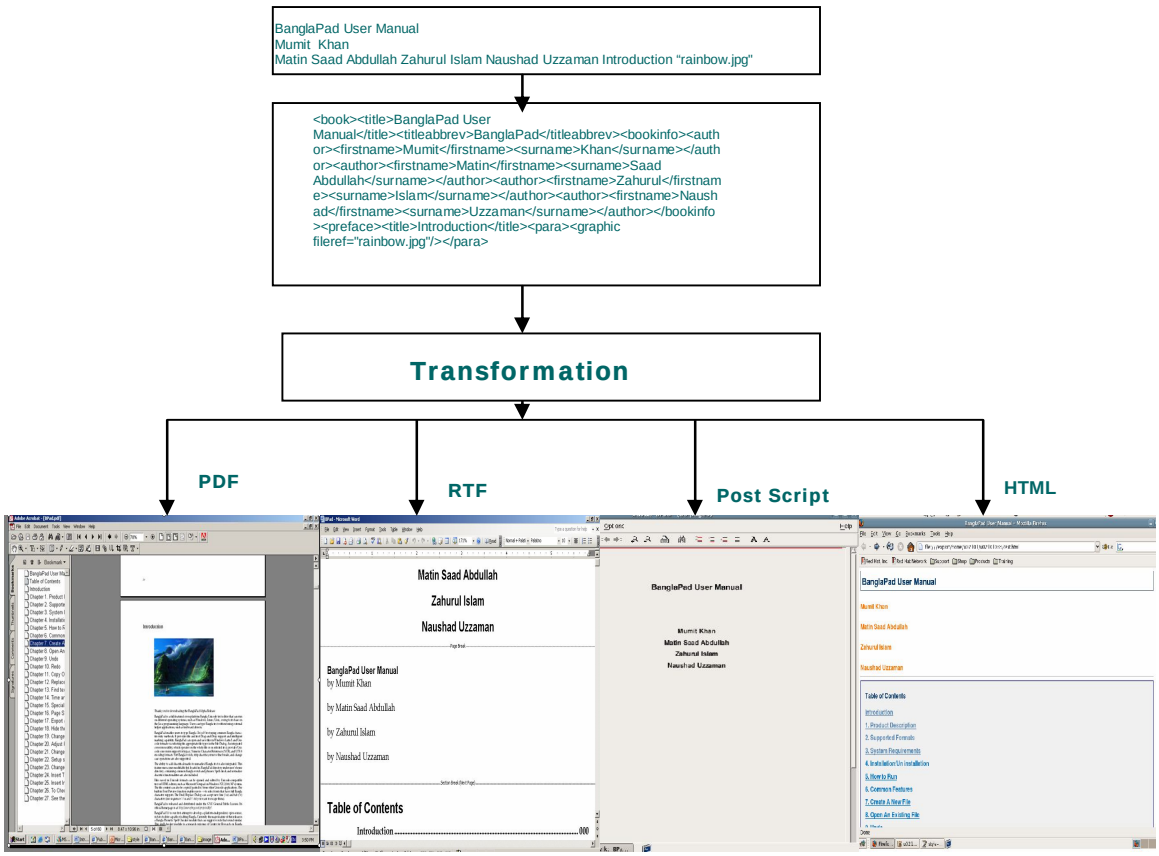

Figure 5.1: General Overview of DocBook components

5.2 What is Docbook

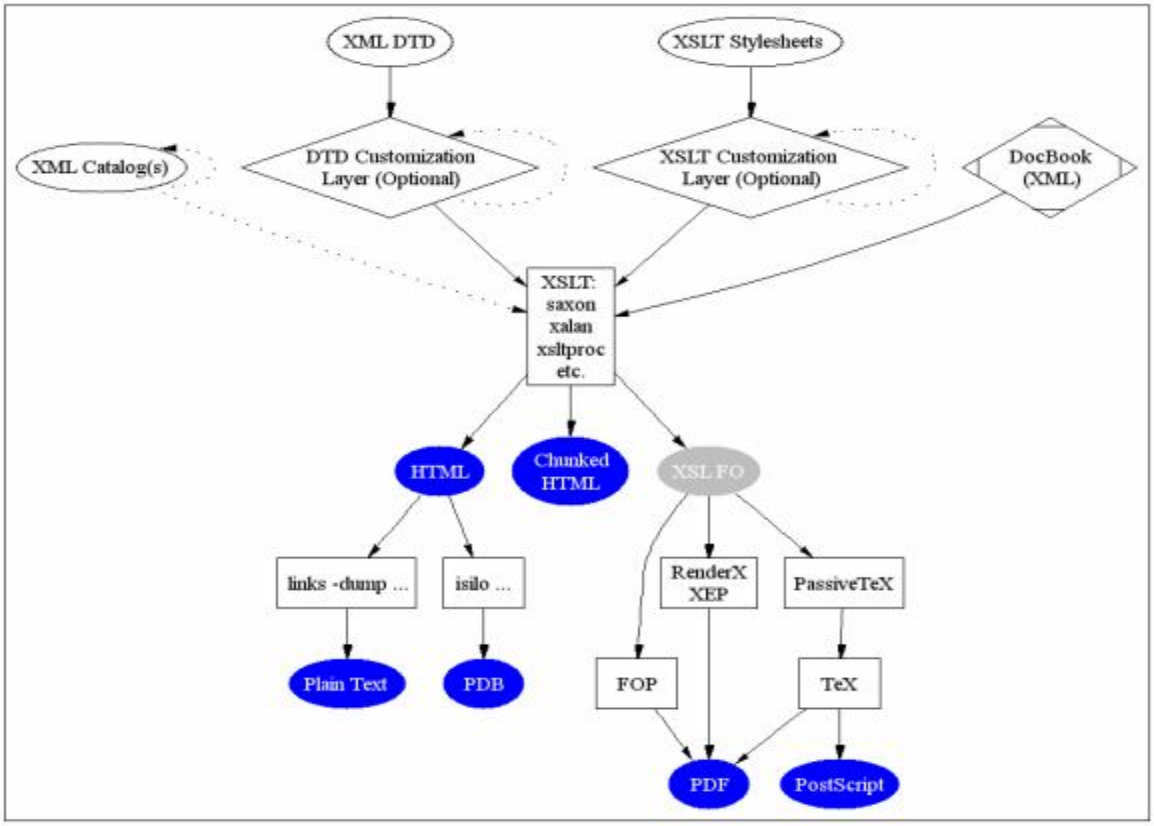

Figure 5.2: Docbook XML Publishing (XSLT)

DocBook is a markup language for technical documentation, originally intended for authoring technical documents related to computer hardware and software but which can be used for any other sort of documentation. It is maintained and standardized by the DocBook Technical Committee at OASIS (originally SGML Open).

DocBook is a schema (available in several languages including RELAX NG, SGML and XML DTDs, and W3C XML Schema) maintained by the Docbook Technical; Committee of OASIS It is particularly well suited to books and papers about computer hardware and software (though it is by no means limited to these applications).

Because it is a large and robust schema, and because its main structures correspond to the general notion of what constitutes a “book,” DocBook has been adopted by a large and growing community of authors writing books of all kinds. DocBook is supported “out of the box” by a number of commercial tools, and there is rapidly expanding support for it in a number of free software environments. These features have combined to make DocBook a generally easy to understand, widely useful, and very popular schema. Dozens of organizations are using DocBook for millions of pages of documentation, in various print and online formats, worldwide.

DocBook is a collection of standards and tools for technical publishing. DocBook was originally created by a consortium of software companies as a standard for computer documentation. But the basic "book" features of DocBook can be used for other kinds of content, so it has been adapted to many purposes.

Processing steps used by Docbook

- DocBook follows the syntax of XML, just like XHTML does.
- It is converted to other formats by XSLT style sheets.
- PDF layout being done through a XSL-FO engine.

The tools used to process DocBook files have to keep up with this evolution.

The elements of a publishing system based on DocBook include:

* DocBook Document Type Definition (DTD).
* XML writing tools.
* DocBook XSL stylesheets.
* Processing tools.

The DocBook DTD

The core DocBook standard is the DocBook Document Type Definition (DTD) maintained by the DocBook Technical Committee in OASIS. The DTD defines the vocabulary of content elements that an author can use and how they relate to each other. For example, a book element can contain a title element, any number of para elements for paragraphs, and any number of chapter elements. The DTD is available in both XML and SGML versions.

5.3. DocBook Modules

DocBook is composed of seven primary modules. These modules decompose the DTD into large, related chunks. Most modifications are restricted to a single chunk.

Figure shows the module structure of DocBook as a flowchart.

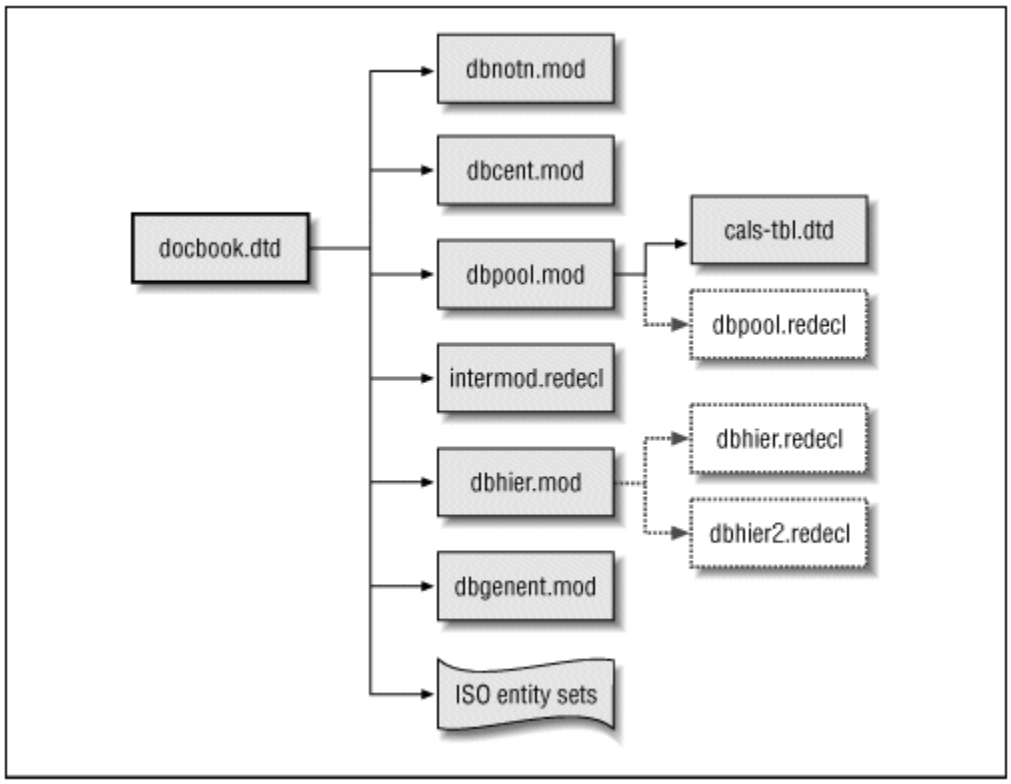

Figure 5.3: Structure of the DocBook DTD

The modules are:

docbook.dtd

The main driver file. This module declares and references the other top-level modules.

dbhier.mod

The hierarchy. This module declares the elements that provide the hierarchical structure of DocBook (sets, books, chapters, articles, and so on).

Changes to this module alter the top-level structure of the DTD. If you want to write a DocBook-derived DTD with a different structure (something other than a book), but with the same paragraph and inline-level elements, you make most of your changes in this module.

dbpool.mod

The information pool. This module declares the elements that describe content (inline elements, bibliographic data, block quotes, sidebars, and so on) but are not part of the large-scale hierarchy of a document. You can incorporate these elements into an entirely different element hierarchy.

The most common reason for changing this module is to add or remove inline elements.

dbnotn.mod

The notation declarations. This module declares the notations used by DocBook.
This module can be changed to add or remove notations.

dbcent.mod

The character entities. This module declares and references the ISO entity sets used by DocBook.
Changes to this module can add or remove entity sets.

dbgenent.mod

The general entities. This is a place where you can customize the general entities available in DocBook instances.

This is the place to add, for example, boiler plate text, logos for institutional identity, or additional notations understood by your local processing system.

cals-tbl.dtd

The CALS Table Model. CALS is an initiative by the United States Department of Defense to standardize the document types used across branches of the military. The CALS table model, published in MIL-HDBK-28001, was for a long time the most widely supported SGML table model (one might now argue that the HTML table model is more widely supported by some definitions of "widely supported"). In any event, it is the table model used by DocBook.

DocBook predates the publication of the OASIS Technical Resolution TR 9503:1995, which defines an industry standard exchange table model and thus incorporates the full CALS Table Model.

Most changes to the CALS table model can be accomplished by modifying parameter entities in dbpool.mod; changing this DTD fragment is strongly discouraged. If you want to use a different table model, remove this one and add your own.

***.gml**

The ISO standard character entity sets. These entity sets are not actually part of the official DocBook distribution, but are referenced by default.

XML writing tools

Using the DTD and XML syntax, authors mark up their text content with tag names enclosed in angle brackets like <chapter>. The markup is similar to HTML, but with more tags and tighter rules. As with HTML, you can use a plain text editor to write HTML. But it helps to use a writing program that keeps track of the XML tag names and rules. See the DocBook Wiki website for more information on writing tools.

DocBook XSL stylesheets

A DocBook file contains no information about how to format it. That information is kept in a separate stylesheet file. The most powerful and flexible stylesheets for DocBook are the DocBook XSL stylesheets written by Norman Walsh. These free stylesheets can produce HTML and print output from the same DocBook files. The Sagehill Enterprises' book DocBook XSL: the Complete Guide provides more information.

Processing tools

To apply an XSL stylesheet to a DocBook file, you use an XSLT processor such as Saxon or xsltproc. The XSLT processor can generate HTML directly. For print, it generates an intermediate XSL-FO (formatting objects) file. That file is then processed with an XSL-FO processor to product PDF or PostScript output for printing.

XSLT Processors

There are a variety of XSLT processors. The table below provides some detail in one location on how to download, install, and use seven processors. It also supplies some basic information on working the Java programming environment, which is essential to using several of the processors.

The table lists and describes a dozen readily available XSLT processors. This is by no means a complete list of what's available, but it provides you with a wide variety of choices from among the most commonly used processors. All of the processors support only version 1.0 of XSLT and XPath, unless otherwise noted. A much longer list of processors exists at http://xml.coverpages.org/xslSoftware.html

Table-1. XSLT processors

XSLT processor	URL	Notes
Cocoon	http://cocoon.apache.org/	Apache's XML publishing environment with central XSLT support.
Cooktop	http://www.xmlcooktop.com	Victor Pavlov's free XML editor that includes support for XSLT transformations.
Instant Saxon	http://saxon.sourceforge.net	Michael Kay's Windows-executable XSLT processor.
jd.xslt	http://www.aztecrider.com/xslt/	Written and maintained by Johannes Döbler. Supports the now withdrawn XSLT 1.1 draft.
MSXSL	http://msdn.microsoft.com/downloads	Microsoft's command-line XSLT processor, based on MSXML 4.0.
Saxon	http://saxon.sourceforge.net	Michael Kay's full Java version of Saxon that offers partial support for XSLT 2.0

Table-1. XSLT processors

XSLT processor	URL	Notes
		and XPath 2.0.
Stylus Studio	http://www.sonicsoftware.com	An XML development environment with an XSLT editor and debugger.
Xalan	http://xml.apache.org	Apache's open source processor available in C++ and Java versions.
xmlspy	http://www.xmlspy.com	Altova's popular and well-featured XML development environment, which includes, among may other things, a built-in XSLT processor and debugger.
xRay2	http://architag.com/xray/	An XML editing environment that supports XSLT.
xsltproc	http://xmlsoft.org/XSLT/	Daniel Veillard's XSLT processor based on his libxml/libxslt libraries.
XT	http://www.blnz.com/xt/index.html	Originally written by James Clark, XT is now maintained by Bill Lindsey.

However I have tried to use to FOP engine to generate PDF file from the docbook file. The main objective of FOP is to deliver an XSL-FO to PDF formatter that is compliant to at least the Basic conformance level described in

the W3C Recommendation from 15 October 2001, and that complies with the 11 March 1999 Portable Document Format Specification (Version 1.3) from Adobe Systems.

5.4 Software Packages requisite for Docbook

Since the XML-based tools are much easier to use than the SGML-based tools, there is no software wrapper like the DocBook-tools around them. The basic utilities can be used directly.

On a RPM-based system, you will need the following software packages:

docbook-dtdXX-xml
DocBook's XML DTD (there's one package per version of the DTD)
docbook-style-xsl
Norman Walsh's XSL stylesheets for DocBook
libxml2
Daniel Veillard's XML library, needed by libxslt
libxslt
Daniel Veillard's XSLT conversion engine
java-1_4_2-sun
... or any other Java Runtime Environment, needed by fop
fop

XSL-FO processor

Most of these packages are usually preinstalled with your Linux distribution. For the other ones, use **rpm -ih packagename**.
You are now ready to edit XML/DocBook documents, then convert them to other formats. To do that, you will need the following commands:

- **xsltproc**: DocBook to HTML or XSL-FO

- **fop**: XSL-FO to PDF, PS, etc.

5.5 Users of DocBook

* IBM Linux Technology Center
* Sun Microsystems, Inc.
* Linux Documentation Project
* Free BSD
* GNOME
* PHP
* KDE
* Hewlett Packard

Also can be used for

- Maintaining websites
- FAQ websites
- Computer documentation, including xfree86
- Producing presentation slides
- Producing generated documentation from code comments (GNOME, Linux kernel!)
- For training materials. From a single document.
- Embedded documentation in code: XSLT Standard Library
- Formal specifications. RELAX NG XML schema language official committee specification

5.6 When is DocBook appropriate?

* Multiple output formats.

* Multiple releases over time.
* Large documentation sets.
* Batch processing environment.
* Shared authoring.

5.7 When is DocBook not appropriate?

* Highly formatted documents like magazines.
* Short documents.
* Authors who know nothing of XML.
* One-off documents.

CHAPTER 6: DISTINCTIVE FEATURES OF DOCBOOK

The DocBook format was designed by OASIS consortium specifically for technical documentation. It provides a rich set of tags to describe the content of your document.

Here is a number of key points that help understand what DocBook does:

Docbook is a rich markup language

It is very similar to HTML in this respect. The tags give some structure to your document, and appear intermixed with the informational text. A rich markup language like DocBook is a good idea from many points of view, but it can also be difficult to use. DocBook has hundreds of tags (as opposed to just a few in HTML), so you might find the learning curve steep.

This pecular point makes it a revolution with respect to documentation translation, because the DTP phase (making the text look nice) is done once for all indirectly by tagging the original text. The translators only have to translate "in between the tags" and by pressing a single keystroke the translated output is generated.

Docbook is a rich markup language

DocBook is perfectly suited for car engine parts documentation. However, it is strongly biased towards computer programs documentation.

It is maintained by an independent consortium

The OASIS consortium is in charge of maintaining and making this standard evolve through the DocBook Technical Committee. This is a guarantee of independence in front of proprietary software and standards.
Major actors of the industry like Boeing or IBM are members of OASIS.

Technically, DocBook is a SGML or XML DTD

This means that one can take profit of the many SGML and XML aware tools. While DocBook as an XML implementation is quite recent, it has a long history as a SGML implementation.

DocBook is not a presentation language

DocBook carefully cares about not specifying how the final documentation looks like. This allows the writer to concentrate on the organization and meaning of the document he or she writes. All the presentation issues are devolved to style sheets.

This ensures all your documents have a consistent appearance, whoever should be the technical writer.

DocBook is customizable

It is quite easy to customize the DTD to meet one's need thanks to its modular organization. But one must be aware that this must be done with respect to SGML/XML conventions and that it might introduce incompatibilities.

If DocBook is used in conjunction with Norman Walsh's modular stylesheets, it is also possible to customize the way a DocBook file can be printed or put online too.

DocBook is comprehensive

The large number of tags defined in DocBook guarantees that it can accommodate a wide range of situations and of processing expectations.
This in turn makes it a bit difficult to learn, but one can manage writing documentation knowing only a limited set of tags and referring to the reference documentation when needed.

DocBook uses long and understable tags

Example of such tags are <itemizedlist> or <literallayout>. This makes a DocBook text much easier to read than an HTML source for example. As a drawback, it can also become a bit tedious to type those long tags, but specialized modes in usual editors (like Emacs' psgml mode) can help out of this. One can use authboring tools as well.

DocBook does not ensure ascending compatibility between major releases

While this might seem a drawback, in fact it is not, because it ensures a clean design even if wrong choices have been made previously by the DocBook Commitee at OASIS, and because documents written with different DTDs can coexist on a same computer system.

6.1 Output formats

- HTML
- PDF
- MIF
- PCL
- PS
- SVG

RTF is planned by the experimental releases of FOP.

The following diagram illustrates the entire procedure of how each of these output formats can be achieved from the Docbook file.

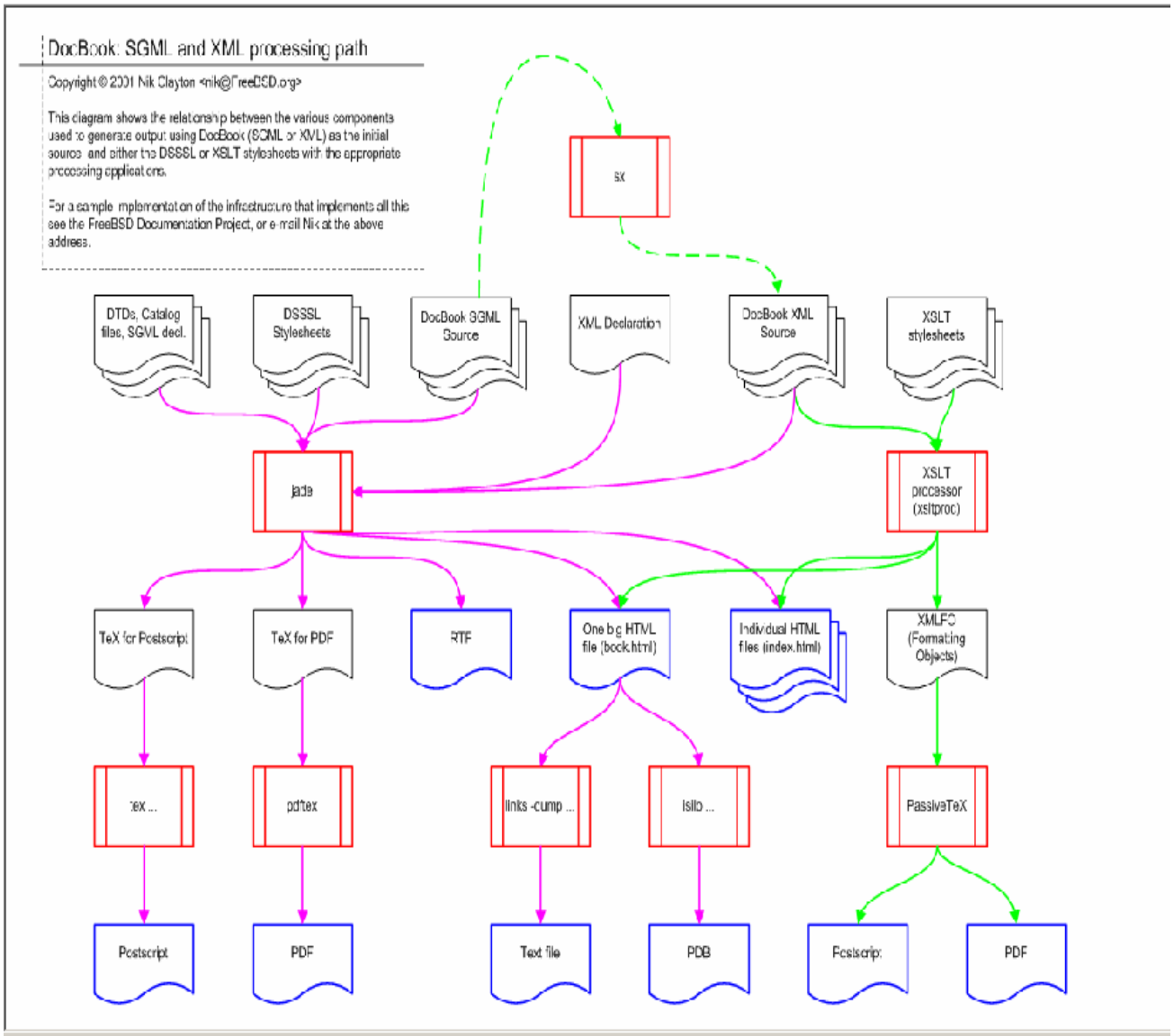

Figure 6.1: Docbook to other output formats

CHAPTER 7: COMMANDS USED

Commands used are as followed:

jw

NAME

jw, docbook2dvi, docbook2html, docbook2man, docbook2pdf, docbook2ps,
docbook2rtf, docbook2tex, docbook2texi, docbook2txt - (Jade Wrapper) converts SGML files to other formats

SYNOPSIS

jw [**-f** frontend | **--frontend** frontend]
[**-b** backend | **--backend** backend]
[**-c** file | **--cat** file]
[**-n** | **--nostd**]
[**-d** file**|default|none** | **--dsl** file**|default|none**]
[**-l** file | **--dcl** file]
[**-s** path | **--sgmlbase** path]
[**-p** program | **--parser** program]
[**-o** directory | **--output** directory]
[**-V** variable**[=**value**]**]
[**-u** | **--nochunks**] [**-i** section | **--include** section]
[**-w** type**|list** | **--warning** type**|list**]
[**-e** type**|list** | **--error** type**|list**]
[**-h** | **--help**] [**-v** | **--version**]
SGML-file

docbook2dvi SGML-file
docbook2html SGML-file
docbook2man SGML-file
docbook2pdf SGML-file

docbook2ps SGML-file
docbook2rtf SGML-file
docbook2tex SGML-file
docbook2texi SGML-file
docbook2txt SGML-file

DESCRIPTION

The **jw** shell script allows to convert a DocBook file (or some other SGML-based format) to other formats (including HTML, RTF, PS and PDF) with an easy-to-understand syntax. It hides most of Jade's or OpenJade complexity and adds comfortable features.

Other scripts like **docbook2html**, **docbook2rtf** or **docbook2ps** provide different ways of calling **jw** that might be easier to remember. For the moment, jw does not handle XML, but only SGML.

This utility assumes that several other components are installed. The list includes:

- the ISO character entities for SGML
- James Clark's DSSSL engine, jade, or an equivalent parser like Open-Jade
- the DocBook DTD from the OASIS consortium
- Norman Walsh's DocBook modular style sheets (or some other set of DSSSL style sheets)
- Sebastian Rahtz's jadetex set of TeX macros for jade (for backends

intended to "printing" formats like PDF, RTF or PostScript)

· A perl interpreter (for backends that use perl)

· SGMLSpm from CPAN (for backends that use sgmls)

· Lynx HTML browser (for the txt backend)

The jw script is basically called like this:

jw mydoc.sgml

where mydoc.sgml is a SGML file.

The command line above uses default options: it converts from DocBook (the default frontend) to HTML (the default backend), does not put the result in a subdirectory (unless specified otherwise in the style sheets), etc.

In this example, the "mydoc" file name as well as the ".sgml" extension can be replaced by anything else. Current extensions for SGML DocBook files include ".sgml", ".sgm", ".docbook", and ".db". The processed file mydoc.sgml can be in any other directory than the current one. Here we have chosen to generate HTML output. In fact we can use any of the backends stored in the backends/ subdirectory of the DocBook-utils distribution directory (usually /usr/share/sgml/docbook/utils-0.6.14).

Similarly, you can use any frontend defined in the frontends/ subdirectory to convert from another input format.

This sample command creates one or many HTML files with arbitrary file names in the current directory. This default behavior can be changed through command line options and/or customization style sheets.

OPTIONS

The following options apply to the conversion script:

-f frontend **| --frontend** frontend

Allows to specify another frontend than default docbook. The list of currently available frontends is:

docbook

Converts docbook with Norman Walsh's style sheets. This frontend searches in the subdirectories of the base SGML directory for a file named html/docbook.dsl or print/docbook.dsl (depending on the backend's type: html or print).

-b backend **| --backend** backend

Allows to specify another backend than default HTML. The list of currently available backends is:

dvi Converts to DVI (DeVice Independant files) by calling **Jade** or **OpenJade**.

html Converts to HTML (HyperText Markup Language) by calling **Jade** or **OpenJade**.

man Converts a refentry to a Unix manual page by calling docbook2man. Does not work with other SGML document types than DocBook.

pdf Converts to PDF (Portable Document Format) by calling **Jade** or **OpenJade**.

ps Converts to PostScript by calling **Jade** or **OpenJade**.

rtf Converts to RTF (Rich Text Format) by calling **Jade** or **OpenJade**. The resulting file can then be inported into **MS Word** or one of its Linux replacement programs.

tex Converts to TeX by calling **Jade** or **OpenJade**.

texi Converts to GNU TeXinfo pages by calling docbook2texi. Does not work with other SGML document types than DocBook.

txt Converts to a bare text file by calling **Jade** or **OpenJade**, then **Lynx**.

-c file **| --cat** file

Allows to use an extra SGML Open Catalog that will list other files like customization style sheets, adaptations to the DocBook Document Type Definition, special character entities, etc. This catalog is added to the list of catalogs determined by the script (see option **--nostd** below)

-n | --nostd

Do not use the standard SGML Open Catalogs. Normally, the standard catalogs list is determined like this:

· if the centralized catalog exists, then use it. The centralized catalog is a list of all catalogs that might be necessary that usually resides in /etc/sgml. Its name is provided by the frontend, for example the docbook frontend returns

/etc/sgml/sgml-docbook.cat.

· Otherwise, take all the files named catalog from the subdirectories of the SGML base directory (usually /usr/share/sgml).

This option is useful in conjunction with the **--cat** option to use only the catalogs that are specified on the command line.

-d file**|default|none | --dsl** file**|default|none**

Allows to use a customized style sheet instead of the default one.

A "target" starting with a hash mark "#" can be appended to the file name. As a result, only the corresponding part of the style sheet is executed (the "style specification" whose "identificator" is equal to the target's name). A common use of this mechanism is to define "#html" and "#print" targets to trigger the corresponding part of a replacement style sheet which is common for both HTML and printout conversion.

By replacing the file name with "default", the default style sheet provided with the front end is used. For example, the docbook front end returns ./docbook.dsl#html (or ./doc book.dsl#print) in the SGML base directory.

By replacing the file name with "none", no replacement style sheet is used, not even the default style sheet. The style sheet which is used is also determined by the front end. For example, the docbook frontend returns Norman Walsh's html/docbook.dsl (or print/docbook.dsl) found somewhere below the SGML base directory.
If no --dsl option is specified, then "--dsl default" is used.

-l file **| --dcl** file

Allows to use a customized SGML declaration instead of the default one. The file name of the default SGML declaration is not set for SGML files, and is set to xml.dcl in the SGML base directory for XML files.

-s path **| --sgmlbase** path

Allows to use another location for the SGML base directory. This is the directory below which all SGML DTDs, style sheets, entities, etc are installed. The default value is /usr/share/sgml.

-p program **| --parser** program

Specify the parser to use (**Jade** or **OpenJade**) if several are installed. If this option is not specified, the script first tries to use Jade, then it tries **OpenJade**.

-o directory **| --output** directory

Set output directory where all the resulting files will be stored. If the style sheets define a subdirectory where to store the resulting files too, the subdirectory defined by the style sheets will be placed below the subdirectory defined by this option.

-V variable**=[**value**]**

Set a variable (to a value, if one is specified).

-u | --nochunks

Output only one big file. This option is useful only when generating HTML, because the output can be split into several files. This option overrides the setting that may be done in the style sheets.

-i section **| --include** section

Declare a SGML marked section as "include". A SGML marked section is a kind of conditional part of a document. If it is declared "ignore", it will be left ignored, otherwise it will be processed. An example of such a marked section would be:

```
<DOCTYPE mydoc [
  <!ENTITY % confidential "ignore">
]>
<mydoc>
  ...
  <![ %confidential [ Some confidential text... ]]>
  ...
</mydoc>
```

-w type**|list | --warning** type**|list**

Enables or disables the display of given types of warnings. Several -w options might be entered on the command line. Warning types that start with "no-" disable the corresponding warnings, the other types enable them.

If the warning type is replaced with "list", then a list of allowed warning types is displayed.

-e type**|list | --error** type**|list**

Disables given types of errors. Several -e options might be entered on the command line. All error types start with "no-".

If the error type is replaced with "list", then a list of allowed error types is displayed.

-h | --help

Print a short help message and exit

-v | --version

Print the version identifier and exit

FILES

/etc/sgml/sgml-docbook.cat

Centralized SGML open catalog. This file name might vary if another frontend than docbook is used.

/usr/share/sgml/docbook/utils-0.6.14/backends

The various backends

/usr/share/sgml/docbook/utils-0.6.14/frontends

The various frontends

/usr/share/sgml/docbook/utils-0.6.14/helpers

The various helper scripts like docbook2man or docbook2texi

CHAPTER 8: CUSTOMIZING DOCBOOK DOCUMENT USING STYLE SHEETS

Style sheets can tune the conversion in a way the resulting files have more clever names. Change the <book> and <chapter> tags in the above example as follows:

Example 1. The minimal DocBook file, with some attributes

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE book PUBLIC "-//OASIS//DTD DocBook XML V4.4//EN">

<book lang="en">
<!-- Please remark the "lang" attribute here -->

<bookinfo>
<title>Hello, world</title>
</bookinfo>

<chapter id="introduction">
<title>Hello, world</title>

<para>This is my first DocBook file.</para>

</chapter>
</book>
```

The text between <!-- and --> above is a comment; use it to attract the attention of someone reading the DocBook source. It will never get processed.

Now use the style sheet provided with the docbook-utils:

```
$ cp /usr/share/sgml/docbook/utils*/docbook-utils.dsl .
$ docbook2html -d docbook-utils.dsl#html myfile.docbook
```

Now the files should go to a HTML directory, and be named index.html and introduction.html, instead of having names like book1.htm. The main file will always be named index.html and the chapters like <chapter id="introduction"> will go to files named after the id attribute. This change has been accomplished through style sheet magic.

Use #print instead of #html to specify the right part of the style sheet to use if you try them with some command like **docbook2pdf** instead of **docbook2html**.

In fact the style sheets are a very powerful tool. They enable you to get rid of problems like "I want it to look like this". If you come to such questions while writing a DocBook file, then it means that something is going wrong in your approach of the things.

If you get a look to the style sheet file named docbook-utils.dsl, you'll see that it is written in a cryptic language named DSSSL ,that looks really like some LISP. This unfortunately means that some good programming knowledge is often required to tune the style sheets.

There are a number of other style sheets that could also be used too.

FOSIs

First, the U.S. Department of Defense, in an attempt to standardize stylesheets across military branches, created the Output Specification, which is defined in MIL-PRF-28001C, Markup Requirements and Generic Style Specification for Electronic Printed Output and Exchange of Text.[14]

Commonly called FOSIs (for Formatting Output Specification Instances), they are supported by a few products including ADEPT Publisher by Arbortext and DL Composer by Datalogics.

DSSSL

Next, the International Organization for Standardization (ISO) created DSSSL, the Document Style Semantics and Specification Language. Subsets of DSSSL are supported by Jade and a few other tools, but it never achieved widespread support.

CSS

The W3C CSS Working Group created CSS as a style attachment language for HTML, and, more recently, XML.

XSL

Most recently, the XML effort has identified a standard Extensible Style Language (XSL) as a requirement. The W3C XSL Working Group is currently pursuing that effort.

In fact the whole use of style sheets in DocBook is all about how to transform documents from DocBook to other formats.

If you would like to transform your documents for proofreading purposes, please use the XML to HTML on-line converter. You will need to upload your XML file(s) to a web site. Then simply drop the URL into the form and click the submit button. Your document will be magically transformed into a beautiful (and legible) HTML document. External files are supported. You may use either absolute or relative URIs.

Another easy-to-use package is xmlto. It is a front-end for xsltproc. It is available as a RedHat, Debian (etc) package or can be downloaded from http://cyberelk.net/tim/xmlto/. You can use it to convert documents with:

```
bash$ xmlto html mydoc.xml
bash$ xmlto txt mydoc.xml
```

Transformations are a pretty basic requirement to get what you've written from a messy tag-soup into something that can be read. This section will help you get your system set up and ready to transform your latest document into other formats. This is very useful is you want to see your document before you release it to the world.

There are currently two ways to transform your document: Document Style Semantics and Specification Language (DSSSL); and XML Style sheets (XSLT). Although the LDP web site uses DSSSL to convert documents you may use XSLT if you want.

DSSSL

There are three basic requirements to transform a document using DSSSL:

* The Document Style and Semantics Specification Language files (these are plain text files).

* The Document Type Definition file which matches the DOCTYPE of your document (this is a plain text file).

* A processor to do the actual work.

The Document Style and Semantics Specification Language files (these are plain text files).

There are two versions of the Document Style Semantics and Specification Language used by the LDP to transform documents from your raw DocBook files into other formats (which are then published on the Web). The LDP version of the style sheets requires the Norman Walsh version--which basically means if you're

using DSSSL the Norman Walsh version can be considered a requirement for system setup.

Norman Walsh DSSSL http://docbook.sourceforge.net/projects/dsssl/. The Document Style Semantics and Specification Language tells Jade how to render a DocBook document into print or on-line form. The DSSSL is what converts a title tag into an <h1> HTML tag, or to 14 point bold Times Roman for RTF, for example. Documentation for DSSSL is located at the same site. Note that modifying the DSSSL doesn't modify DocBook itself. It merely changes the way the rendered text looks. The LDP uses a modified DSSSL

LDP DSSSL http://www.tldp.org/authors/tools/ldp.dsl. The LDP DSSSL requires the Norman Walsh version but is a slightly modified DSSSL to provide things like a table of contents.

Installing" DSSSL style sheets

Create a base directory to store everything such as /usr/share/sgml/. Copy the DSSSL style sheets into a sub-directory named dsssl.

DSSSL Processors

There are two versions of the Jade processor: the original version by James Clark; and an open-source version of approximately the same program, OpenJade. You only need one of these programs. It should be installed after the DTD and DSSSL have been "installed."

DSSSL Transformation Tools

Jade

ftp://ftp.jclark.com/pub/jade/

Currently, the latest version of the package is jade-1.2.1.tar.gz.

Jade is the front-end processor for SGML and XML. It uses the DSSSL and DocBook DTD to perform the verification and rendering from SGML and XML into the target format.

OpenJade

http://openjade.sourceforge.net/

It is an extension of Jade written by the DSSSL community. Some applications require jade, but are being updated to support either software package.

System Setup for DSSSL Transformations

1 Tell your system where to find the SGML_CATALOG_FILES (yes, even if you are using XML).

2. Download the DSSSL and DTD files and copy them into your working directory.

Transformations with DSSSL

Once your system is configured (see the previous section), you should be able to start using jade to transform your files from XML to XHTML.

To create individual HTML files, point jade at the correct DSL (style sheet). The following example uses the LDP style sheet.

```
bash$ jade -t xml -i html \
-d /usr/local/sgml/dsssl/docbook/html/ldp.dsl#html \  HOWTO.xml
```

If you would like to produce a single-file HTML page, add the -V nochunks parameter. You can specify the name of the final HTML file by appending the command with > output.html.

```
bash$ jade -t xml -i html -V nochunks \
        -d /usr/local/sgml/dsssl/stylesheets/ldp.dsl#html \
        HOWTO.sgml > output.html
```

Note Not a function name errors

If you get an error about "is not a function name", you will need to add a pointer to xml.dcl. It has to be listed immediately before the pointer to your XML document. Try one of the following locations: /usr/lib/sgml/declaration/xml.dcl, or /usr/share/sgml/declaration/xml.dcl. Use locate to find the file if it is not in either of those two places. The modified command would be as follows:

```
bash$ jade -t xml -i html \
        -d /usr/local/sgml/dsssl/docbook/html/ldp.dsl#html \
        /usr/lib/sgml/declaration/xml.dcl HOWTO.xml
```

If you would like to create print-friendly files instead of HTML files, simply change the style sheet that you are using. In the file name above, note "html/ldp.dsl" at the end. Change this to "print/docbook.dsl", or if you want XHTML output, instead of HTML, change the file name to "xhtml/docbook.dsl".

Changing CSS Files

If you want your HTML files to use a specific CSS stylesheet, you will need to edit ldp.dsl. Immediately after; End of $verbatim-display$ redefinition add the following lines:

```
(define %stylesheet-type%
  ;; The type of the stylesheet to use
  "text/css")

(define %stylesheet%
  ;; Name of the css stylesheet to use, use value #f if you don't want to
  ;; use css stylesheets
  "base.css")
```

Replace base.css with the name of the CSS file you would like to use.

The docbook-utils Package

The docbook-utils provide commands like db2html, db2pdf and db2ps, based on the jw scripts, that is a front-end to Jade. These tools ease the everyday management of documentation and add comfortable features.

The package, originally created by RedHat and available from http://sources.redhat.com/docbook-tools/ can be installed on most systems.

Example creating HTML output

After validating your document, simply issue the command db2html mydoc.xml to create (a) HTML file(s). You can also use the docbook-utils as validation tools. .

Using CSS and DSL for pretty output

You can define your own additional DSL instructions, which can include a pointer to a personalized CSS file.

The sample DSL file will create a table of contents, and have all HTML files start with the prefix intro2linux- and end with a suffix of .html. The %stylesheet% variable points to the CSS file which should be called by your HTML file.

To use a specific DSL style sheet the following command should be used:

db2html -d mystyle.dsl mydoc.xml

You can compare the result here: http://tille.xalasys.com/training/unix/ is a book formatted with the standard tools; http://tille.xalasys.com/training/tldp/ is one using personalized DSL and CSS files. Soft tones and special effects, for instance in buttons, were used to achieve maximum effect.

XSL

There are alternatives to DSSSL and Jade/OpenJade.

When working with DocBook XML, the LDP offers a series of XSL style sheets to process documents into HTML. These style sheets create output files using the XML tool set that are similar to those produced by the SGML tools using ldp.dsl.

The major difference between using ldp.dsl and the XSL style sheets is the way that the generation of multiple files is handled, such as the creation of a separate file for each chapter, section and appendix. With the SGML tools, such as jade or openjade, the tool itself was responsible for generating the separate files. Because of this, only a single file, ldp.dsl was necessary as a customization layer for the standard DocBook DSSSL style sheets.

With the DocBook XSL style sheets, generation of multiple files is controlled by the style sheet. If you want to generate a single file, you call one style sheet. If you want to generate multiple files, you call a different style sheet. For that reason the LDP XSL style sheet distribution is comprised of four files:

1 tldp-html.xsl - style sheet called to generate a single file.

2. tldp-html-chunk.xsl[2] - style sheet called to generate multiple files based on chapter, section and appendix elements.

3. tldp-html-common.xsl - style sheet containing the actual XSLT transformations. It is called by the other two HTML style sheets and is never directly called.

4. tldp-print.xsl - style sheet for generation of XSL Formatting Objects for print output.

You can find the latest copy of the files at http://my.core.com/~dhorton/docbook/tldp-xsl/. The package includes installation instructions which are duplicated at http://my.core.com/~dhorton/docbook/tldp-xsl/doc/tldp-xsl-howto.html. The short version of the install instructions is as follows: Download and unzip the latest package from the web site. Take the files from the html directory of TLDP-XSL and put them in the html directory of Norman Walsh's stylesheets. Take the file from the TLDP-XSL fo directory and put it in the Norman Walsh fo directory.
Once you have installed these files you can use xsltproc to generate HTML files from your XML documents. To transform your XML file(s) into a single-page HTML document use the following command:

```
bash$ xsltproc -o test.html /usr/local/sgml/stylesheets/tldp-html.xsl test.xml
```

To generate a set of linked HTML pages, with a separate page for each chapter, sect1 or appendix, use the following command:

bash$ xsltproc /usr/share/sgml/stylesheets/tldp-html-chunk.xsl HOWTO.xml

Note that you never directly call the style sheet tldp-html-common.xsl. It is called by both of the other two style sheets.

Changing CSS Files

If you want your HTML files to use a specific CSS stylesheet, you will need to edit tldp-html-common.xsl. Look for a line that ressembles <xsl:param name="html.stylesheet" select="'style.css'"/>.

Replace style.css with the name of the CSS file you would like to use.

In truth, "XSL" is actually comprised of three components: the XSLT transformation language, the XPath expression language (used by XSLT), and XSL Formatting Objects (FO) that are used for describing a page. The style sheets are actually written in XSLT and generate either HTML or (for print output) FO. The FO file is then run through a FO processor to create the actual print (PDF or PostScript) output. See the W3C web site for more information.

Summary: Transformation From DocBook To Other formats

Type	Components needed	Chunking	Style sheet used	CSS Use	UFT Support
DSSSL	• **DSSSL file---there are**	**Not**	**ldp.dsl**	**Possible**	**Yes**

	two basic versions of these file • **DTD file** • **DSSSL Processor--- there are two versions of Jade Processor namely Jade and Open Jade (Open source Jade) The front end to Jade is jw**	**Possible**			
XSLT	• **XPath Expression language** • **XSL Formatting Object file** • **FO Processor —FOP Engine, Xalan ,Saxon**	**Possible**	**XSL stylesheet**	**Possible**	**No**

CHAPTER 9: TEST CASE: BANGLAPAD USER MANUAL USING DOCBOOK

9.1 Background

BanglaPad is a full-featured cross-platform Bangla Unicode text editor that can run on different operating systems, such as Windows, Linux/Unix, owing to its base on the Java programming language. Users can type Bangla text without using external helper applications, such as keyboard drivers. This has been developed by Dr. Mumit Khan, Mr.Matin Saad Abdullah, Naushad Uzzaman and Zahurul Islam.

9.2 Application (HTML Version)

The command at the terminal is: docbook2html <sgml input file>

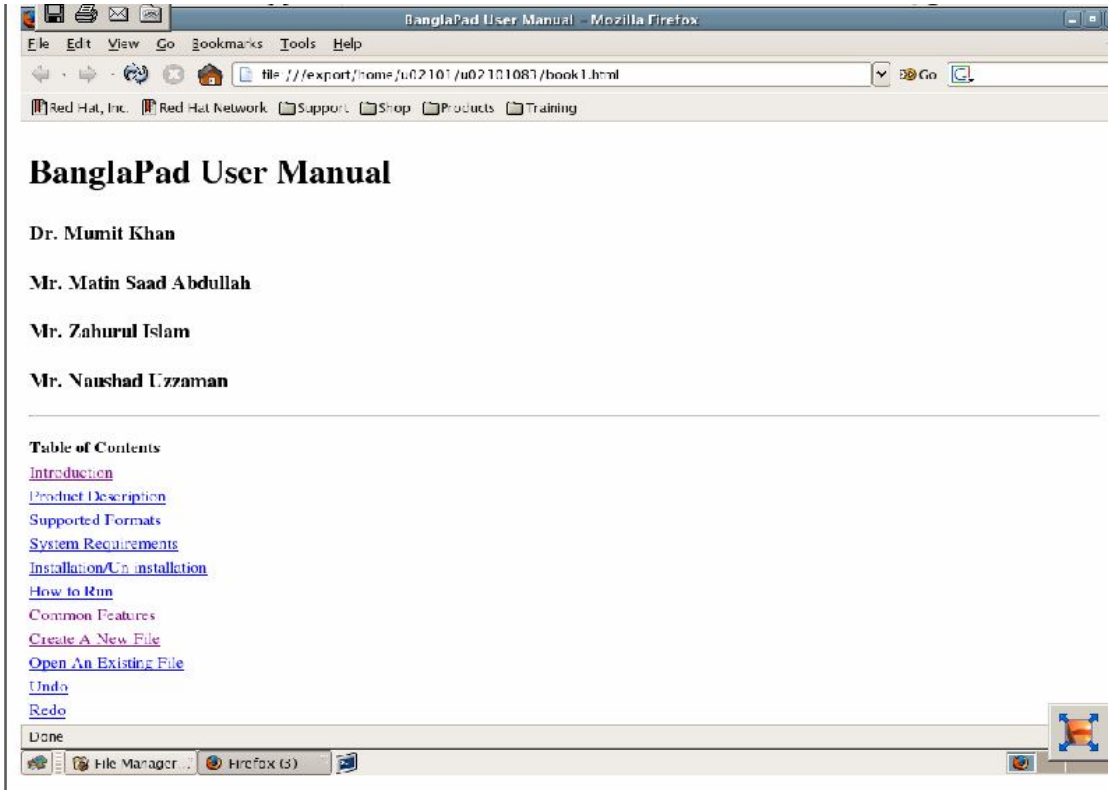

Figure 9.1: Front page (html version)

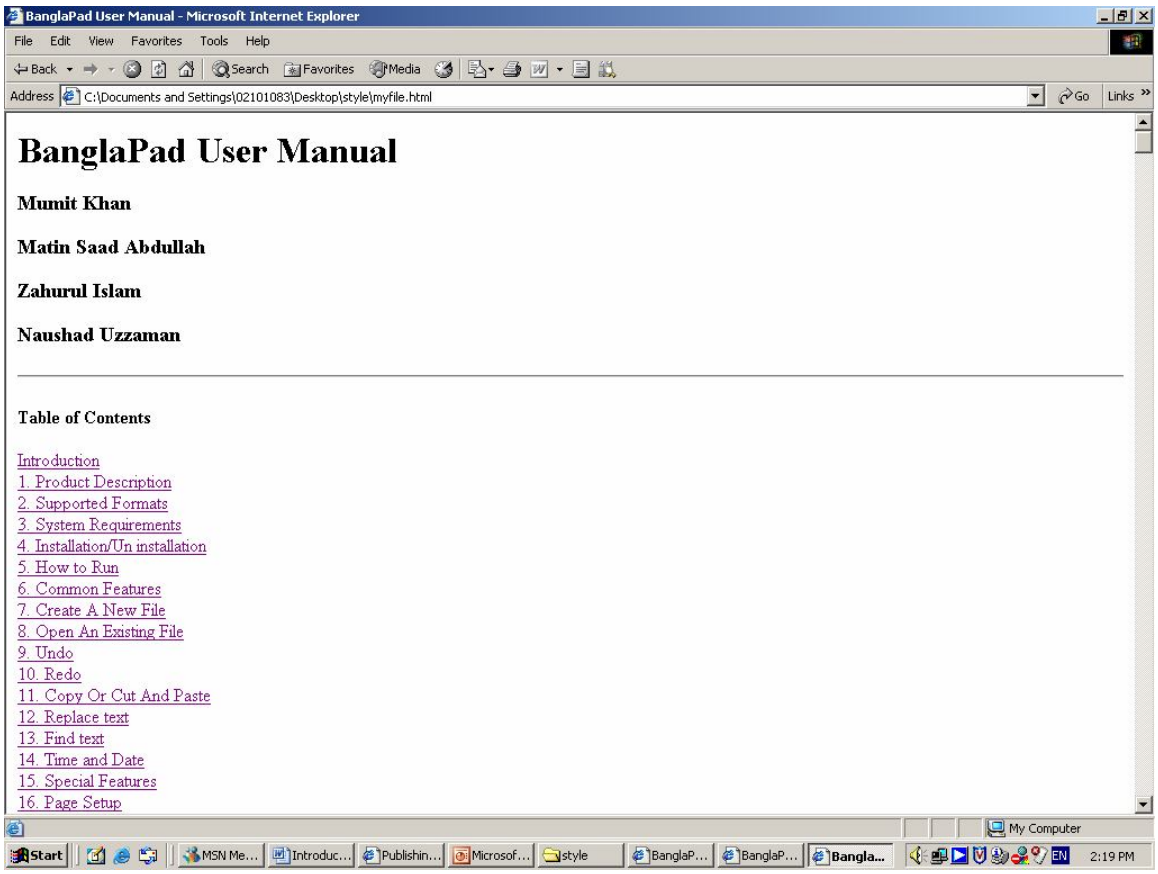

Figure 9.2: Front page of the User Manual

This is the front page of the HTML file that has been generated from Dockbook file. The tags used here are pretty simple yet very effective. They are <book>, <bookinfo>, <author>, <firstname> ,<preface> etc.

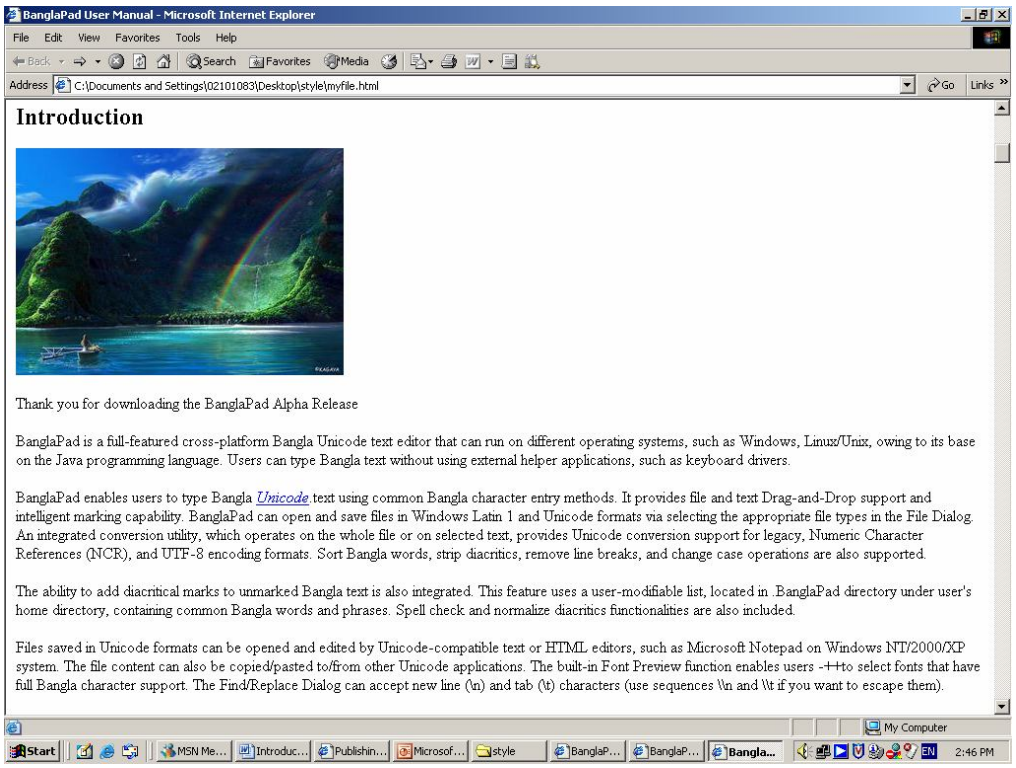

Figure 9.3: The second page

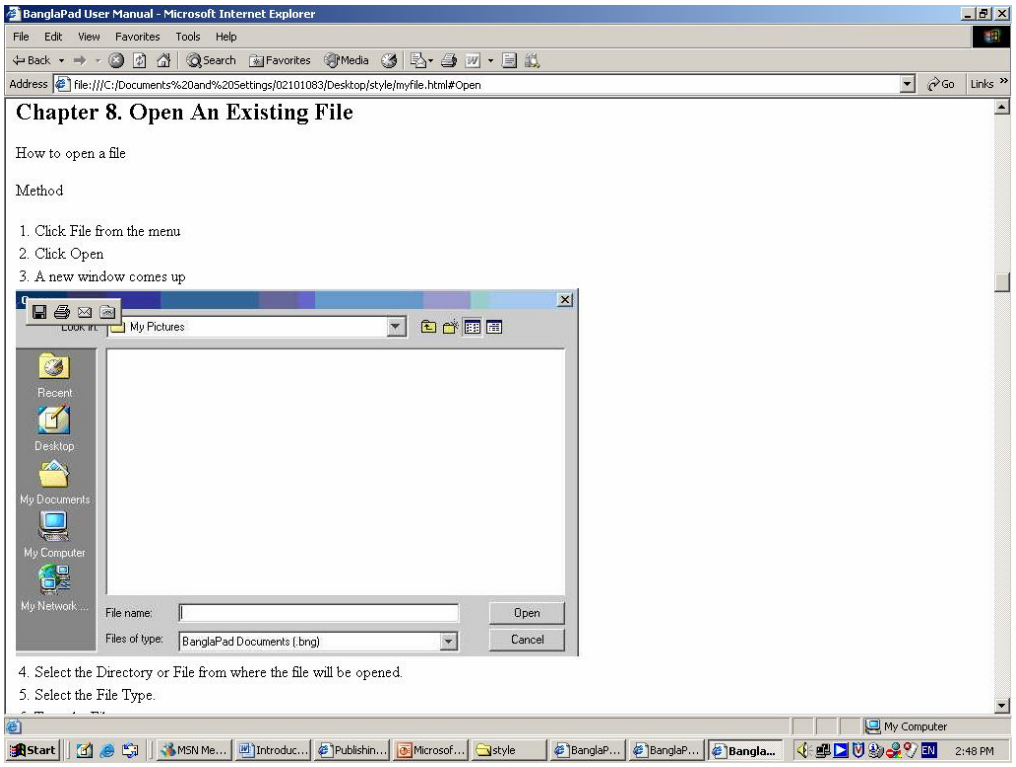

Figure 9.4: “How to open a file” page of the User Manual

9.3 The Manual in RTF

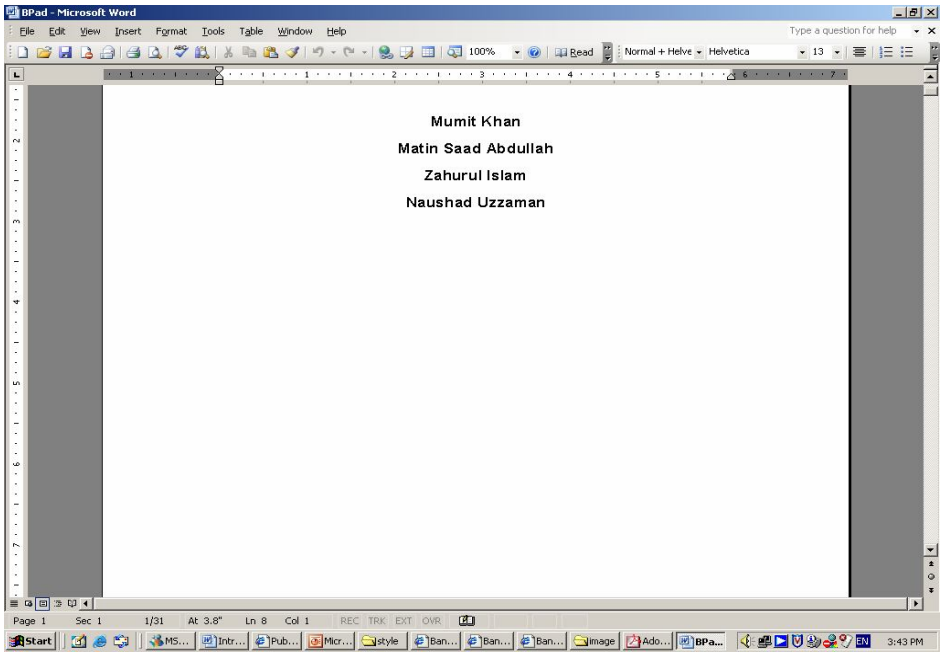

Figure 9.5: Cover Page

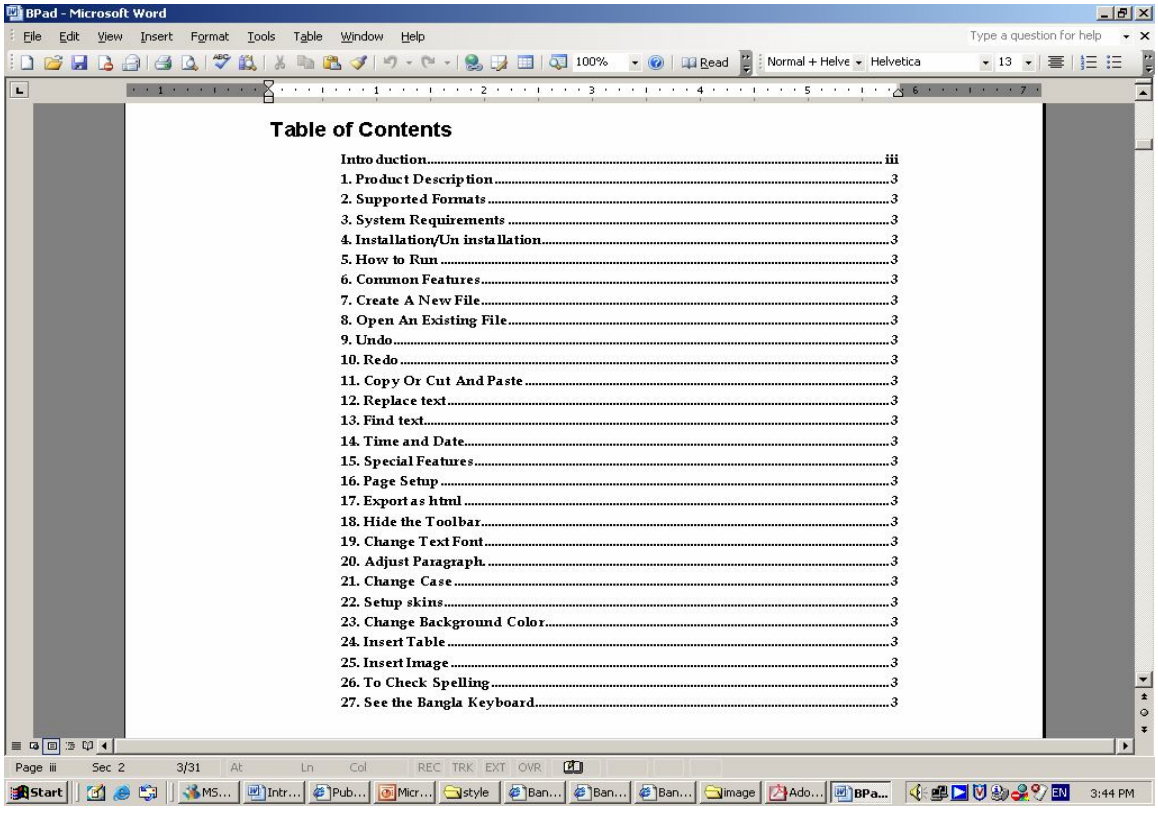

Figure 9.6: TOC

9.4 The Manual in PDF

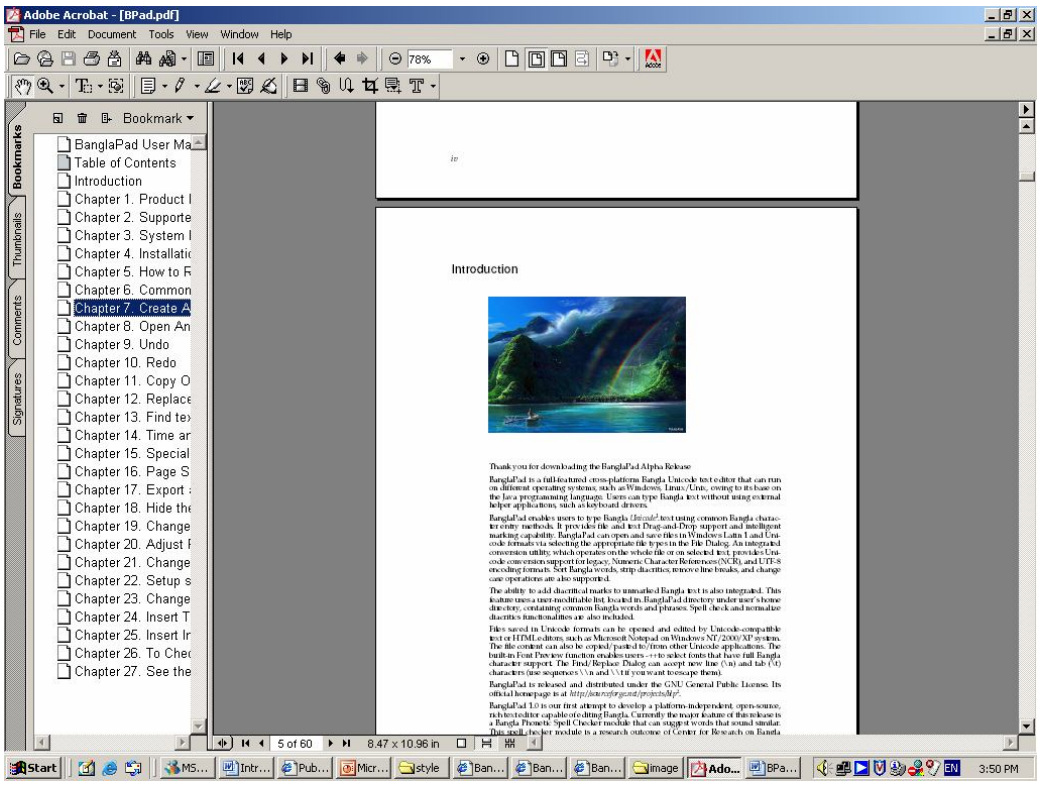

Figure 9.7: PDF version of User manual

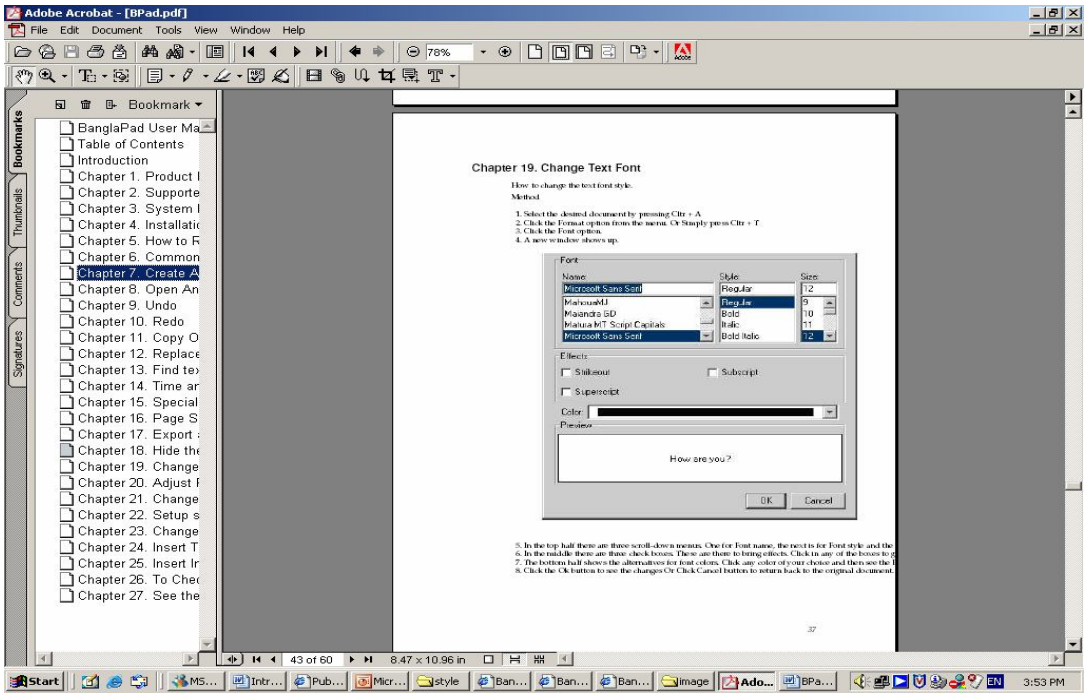

Figure 9.8: A page of "Change font"

9.5 The Manual in Post Script

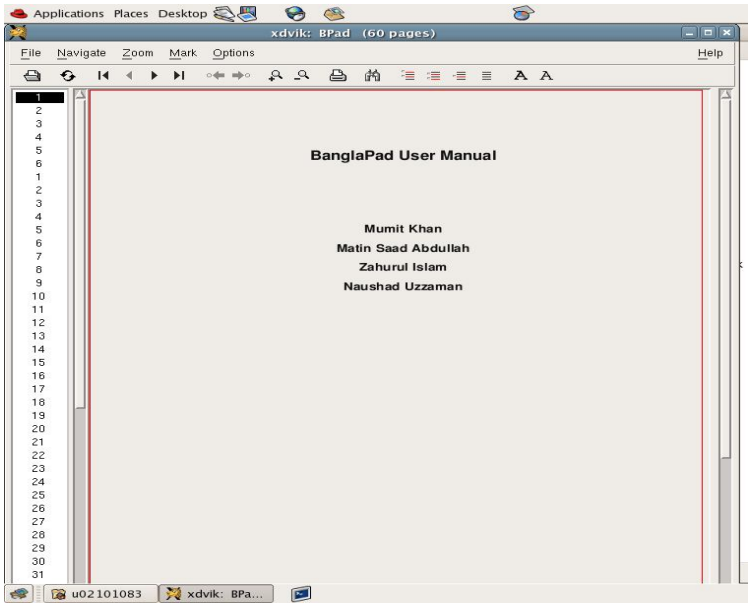

Figure 9.9: Cover page

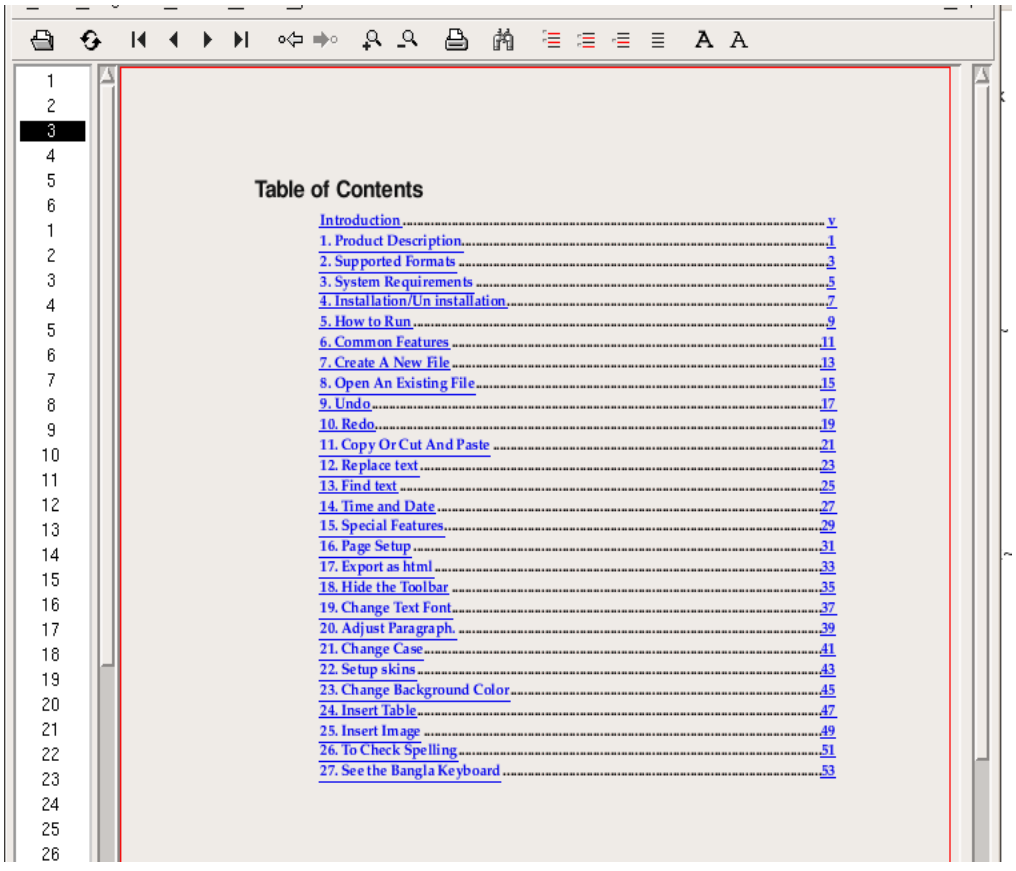

Figure 9.10: TOC

Chapter 10: After Customization Using Style Sheets

Figure 10.1: Customized HTML Version

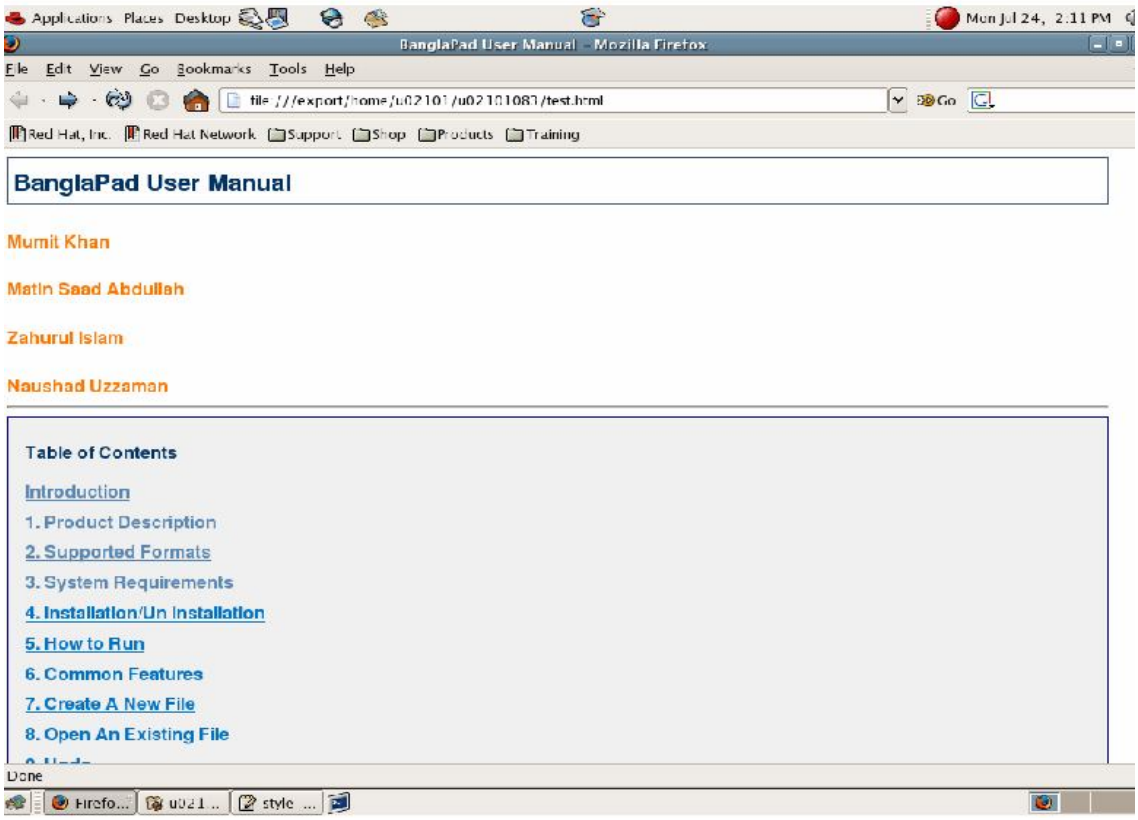

Figure 10.2: Customized First Page

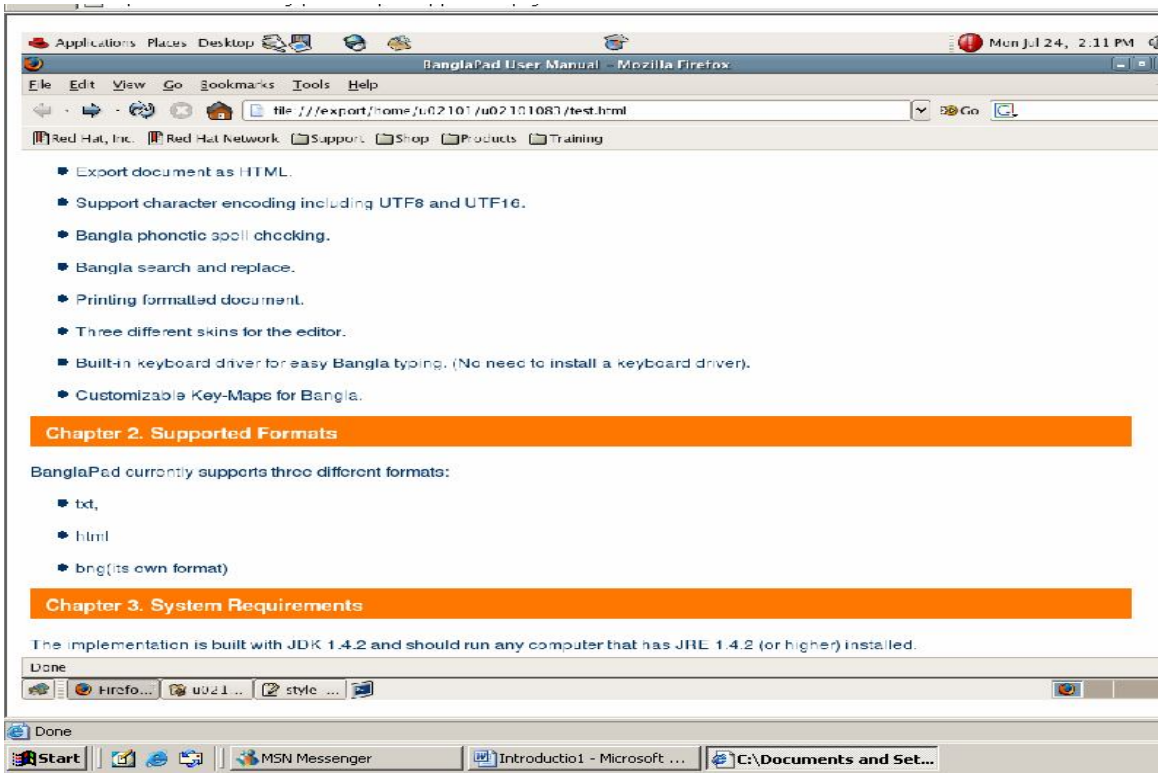

Figure 10.3: Customized Middle Page

CHAPTER 11: CONCLUSION AND FUTURE WORK

There are a number of documentations types and approaches each of which caters to different stakeholders of a software development team. In first few chapters of this thesis paper I have put some light on the some basic conventions of how to write documentations, next I have focused on the best practice methods for making architectural documentation, even though my focus area is User Documentation. In my research I have tried to focus the best practice method to generate user manual i.e. by the use of DocBook. However there are some other approaches too which I could not cover due to time constraint and lack of adequate resources. Nevertheless, my analysis has put appreciable degree of insight into the user manual generation technique, extensively showing all the possible steps in using DocBook. Starting from installation of packages requisite to use DocBook and covering different commands at the terminal in Linux operating system platform, I have religiously and elaborately mentioned all the intermediate steps to obtain the final output. The main objective of using DocBook is to segregate main content of user manual from the presentation format. Or more precisely, the ultimate purpose of DocBook usage is to have a simple and single content with different mark ups and a comprehensive yet effective way from the same content to achieve different outputs each in different formats. These transformations have been made possible only for the tools provided by the DocBook utilities. Mark ups assist to treat the variations in content meaning eg. Para should look different from table, similarly cross reference should have a different look than a numbered list. These features are taken care of by the DocBook DTD, which is found in XML as well as in SGML type.

I have also used different style sheets to customize and alter the look of the final outputs. Sample snippets have been enclosed for better understanding. I could

cover only the HTML output format customization. Nevertheless it is quite likely that other format's output could also be altered as per our desire.

Future Work

The use of DocBook is one possible technique that keeps the content aloof from presentation, definitely there are other similar frameworks that essentially does close task. Thus there lies a provision for further scrutiny in this field. Certainly similar initiative has been taken which demands actual use by potential users. Moreover, as aforementioned customization of the other output formats like pdf, post script or may be rtf could be tried out later too. Consequently it can be stated that sky is the limit only tool that can enhance the desire to evolve is the drive to discover new arenas and update one self every now and then.

References:

[1] http://www-128.ibm.com/developerworks/wireless/library/wi-arch11/

[2] http://www.sei.cmu.edu/publications/documents/01.reports/01tn010/01tn010.html

[3] http://www.win.tue.nl/~mchaudro/sa2004/Kruchten4+1.pdf

[4] http://standards.ieee.org/reading/ieee/std_public/description/se/1471-2000_desc.htm

[5] http://www.linuxcommand.org/man_pages/jw1.html

[6] http://www-128.ibm.com/developerworks/wireless/library/wi-arch11/l

[7] http://en.wikipedia.org/wiki/Software_documentation#User_Documentation

[8] http://www.docbook.org/tdg/en/html/part1.html

[9] http://www.docbook.org/tdg/en/html/ch04.html#d0e9242