2010

Document Builder

Thien Tran
San Jose State University

Document Builder

Thien Tran
Department of Computer Science
San Jose State University
thien_t_tran@hotmail.com

Dr. Mark Stamp
Department of Computer Science
San Jose State University
stamp@cs.sjsu.edu

Dr. Jon Pearce
Department of Computer Science
San Jose State University
pearce@cs.sjsu.edu

Dr. Robert Chun
Department of Computer Science
San Jose State University
rchun@cs.sjsu.edu

ABSTRACT

In this paper, we consider problems related to on-demand content publishing and maintenance. Specifically, we are concerned with the recent concept of structural Content Management Systems (CMS) and its design principles. We focus on Apache Ant, a popular document generator tool for the Java development industry. However, Ant has not been widely extended beyond its capacity to deal with computer programs, which limits its utility.

We analyze the Ant build script structure, study its usage, and implement an on-demand document generator for Ant. The focus is to provide a better document build model based on Ant, which can provide document workflows and templates enabling people to work together more efficiently.

61 pages

Table of Contents

1 Introduction

1.1 Problem Overview

The CMS market is huge, and it has been driven by problems related to the amount of information created by people in this world. These problems still remain and they are keeping us from delivering the right information which we have available, so that our customers can obtain only the needed information and be more satisfied. Let's review some of the problems and acknowledge the focus needs which drive CMS to this day.

The abundance of information created for many product components, which increases from day to day, creates the problem of too much information. It won't be very meaningful if we deliver too much information to the user. Therefore, it is important that we focus on delivering the right information to the right user. When there is an efficient process for a specific user to obtain the right information, that process will help the user to achieve his or her specific goals more efficiently and quickly.

Over time, when a product has been through several release versions, and the product components have been updated and evolved, the problem of obsolete information arises. In this case, the customized information set which has been delivered to specific users will also need to be updated and evolved along with the product. Hence, the smaller the set of units of a component which an engineer needs to update, the more efficiently and frequently the writer can update. This also implies that smaller sets of digital online solutions are much better, whereas delivering hard copies of books to customers as a whole is not an efficient method.

Cost is another problem if we print a book and ship it to every customer when only a small change or patch has occurred in the product components, and it is very common for products to have new changes or patches on a monthly basis.

Last but not least, technical support problems also need to be considered. It would not be an efficient way to do business if we had many people to support all problems and answer all questions for all products and components a company has delivered. This will not address specific problems, because it will not help out within the context of customer works. Also, it will not keep the customer satisfied if the problem is only solved after the customer has already become frustrated from reading almost the entire user's manual.

1.2 Purpose

From the problems set described, it would be wise for CMS not to deliver the entire set of information for a product, as an isolated set of information which can only be used independently. Instead, CMS should allow the customer to specify what information should be delivered, and the CMS to deliver just that.

Further, it is important that information delivery be designed as an integrated component solution, which can be used together whenever it is requested. This means that along with the component integration providing information about the featured functionalities, CMS also needs to expose the meta-data which can describe the integration path to acquire that information. If we are going to allow users to specify what information they want to know for their goals, we need to allow the user to describe the component path from the solution.

In the first chapter section of this research paper, we will review the CMS history and how CMS has evolved. We will look at the differences between the concepts and technologies of the traditional CMS model and those of the new CMS model.

In the following chapters, we will review some of today's well known CMS products. The purpose is to understand whether the product was built to support the new CMS model. Hence the reviews will only discuss their CMS strengths in general, and put much focus on their common weaknesses related to this paper topic, if there are any.

From the products review list, we will select Apache Ant and try to expand it's content delivery support ability. We will examine an Ant build file and try to propose an on-demand document system solution which can be used as a collaborative document builder tool between engineers, technical writers, users, and clients. We will call our proposed system the Document Builder (DB) system.

To help the reader to better understand the DB design and implementation in this paper, there will also be some technical backgrounds review sections on XSLT, XPath, XPointer, and XSL-FO. The paper will also describe some of DB's important user scenarios, as well as results of the application runs.

1.3 Audience

Researchers, developers, or users can refer to this paper to gain experience or further develop any related topics.

2 Conceptual Background

2.1 History

From the below history timeline of CMS, you can see that there has been a long history problem of trying to revolve from a book-centric publishing approach into providing information by adapting publishing solutions for specific areas.

- Books in the beginning.
- Adapting to the exponential growth in knowledge resources, 1960s
- Leveraging knowledge through automation, 1970
- Still book-centric, 1970+
- The Internet, 1990+
- Standard Generalized Markup Language, SGML/HTML
- Data Interchange Problem, 1995+
- XML, offered simplicity, 2000+
- XML, an incomplete revolution?
- Web 2.0 – The social Web, 2003+
- Entering the age of Full Spectrum Publishing, XML -> multi publish formats, 2004+
- Information specialization, on-demand content distribution, adapting in each areas to provide flexibility when building adaption solutions, 2006+

2.2 Traditional CMS

In the early 2000s, CMS vendors began to realize that delivering a giant book is not a feasible solution in the distributed environment, but instead having many smaller documents. There were many factors which have brought CMS to this day, let's review some of those problems which existed in the traditional CMS.

2.2.1 Common Functionalities

Traditional CMS was defined very broadly and it was commonly considered to be a system whose purpose was providing key functionalities such as content storage, authoring, editing and versioning. At that time, there were other coexisting CMS

functionalities such as publishing, collaborating and maintaining, but they were often being implemented as separate processes.

By having separate processes, collaborative tasks between the communities of IT, engineering, and group or individual technical writers were difficult. There was no supported workflow for teams to perform content publishing, collaboration and maintenance. Team members strongly desired CMS to have these functionalities integrated.

Because of these processes' separation from the traditional CMS model of separating, earlier CMS had some key weakness in how content structuring should be designed and implemented. In the next chapters, we will try to exploit the traditional CMS content structure and trying to understand how it should be designed, integrated and represented.

There were also other weaknesses and strengths in the traditional CMS model, but we will not discuss them in this research paper.

2.2.2 Structural Weaknesses

In a traditional CMS, the term *content object* was defined as the minimal component within the CMS, and archived contents were stored into content objects.

To provide some level of organization for storing these content objects, most typical CMS systems used a tree-like file system structure. But as the number of users begin to grow, this structure becomes more and more difficult to use and maintain, because when the content size is large, it is hard to fit into a tree structure, and the purpose of each folder becomes unclear.

To resolve such problem, most CMS systems started to introduce several kinds of meta data surrounding the content object, and these meta data were grouped into sets which varied from one CMS system to another. The meta data sets were all introduced for similar purposes of categorizing and searching the content objects within a CMS system. This meta data feature approach has had some success in making content navigation easier, and queries can be used as the alternative navigation method. However the meta data feature approach can still be considered weak because it depends heavily on the meta data scheme.

Some advanced CMS introduced links between content objects to express their relationship. These links are often hyper-links which connect the content of content objects among one to another, thus creating some conceptual relationship between them. This general link approach is quite useful for supporting meta data querying or navigating through links, but because there is no strong integration of the link into the CMS system, its power has been restrained from limitations.

2.3 Future CMS

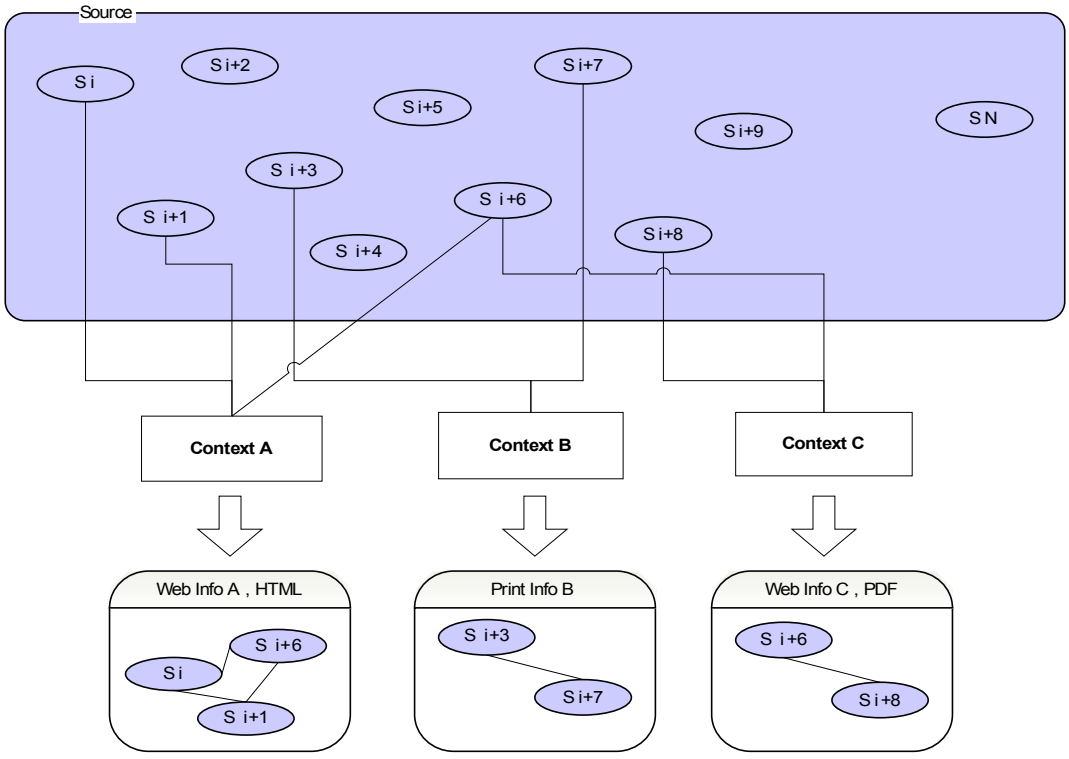
Based on the lessons learned from problems, many CMS vendors have begun to think that the key features of a better structural CMS must follow this organization principle:

A content object should be a free-floating object, and meta data should be the only information used for locating and managing these content objects.

This principle is essentially describing what a topic map, is. The content objects are free-floating free objects, and a topic map is created on top to organize them for managing and searching. There are two separate entities, the topic map and the content object. The topic map is made up of a set of meta data pointing to the content object, and the content object is to hold the actual content.

Each topic will then be collected in one context. Each context will describe a different set of functionalities, which groups the information for different products, different sets of components, or different media types, such as Web or print, or possibly different audiences.

The topic could be reusable. The separation of context from content is really what can give a more scalable, high-quality, and reusable solution.



In the above diagram, content objects (Si, .., Si+n) are free-floating objects in the Source area. Topics are collected into Contexts (A, B, C, ..) and point to sets of content objects. Each Context has different targets (Web, Print, etc.).

2.3.1 Design Principles

Starting from the principle of the topic map, we can build a better CMS by designing a system to support the following concepts.

Topic Orientation

The unit of information use which covers a specific subject with a specific purpose. The idea is that if we start by writing around topics, such as answering a specific question distinctly and directly, this is the basic principle of technical information communication and instructional design.

If we chunk the way we communicate, people tends to absorb the information much better. If we don't chunk it, people will chunk it on their own, but it will be more work for them because they will have to figure out for themselves where the boundaries are. But if we can provide the boundaries for them, organize it around a discrete subject when we write and deliver the information, it will make it much easier for the reader and user to understand and move on.

This concept applies whether the topic appears in a book, online, or a help set on the screen.

Topic Granularity

This is a unit of reuse. Once we have the idea of a topic of use, we should also look at it as a topic of reuse. If a unit was designed to stand on its own or make sense to the reader, then the reader will be able to reuse it as well. This is the core design of a reusable and scalable system which needs to come out of the system architecture.

Strong Typing

This is driven by the fact that we cannot support a general purpose document type. We have to have a set of specific document types, and the ability to create more and more specific document types. The idea is that the closer we get to the author understanding the material, the more efficiently the documents can be used.

If we give the readers the html version and ask them to parse/scan it to another format, they can do it but because there are lots of different choices in the markup, the result will be that there will be lots of inconsistencies. Thus, we should have a requirement task to

think about the context, think about the format, and follow with the results or examples. This is very descriptive and it provides checkpoints or templates that the writer can work with. They can also go beyond the templates, sub-verse the templates, or jump back to the topic level if they object to it. But it provides the code changes, the starting points for authoring that a generic document type cannot do.

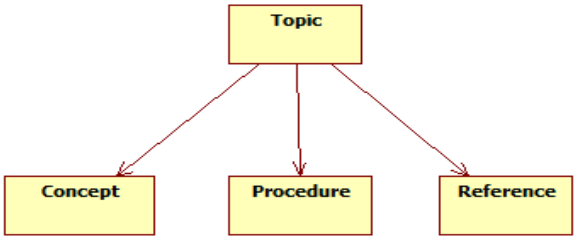
Specialization

This is how we enable strong typing. If any specific groups have specific concerns, specific definition tasks, or a variety of references on information that they want to provide for specific support and consistency, specialization provides them with specific document types to perform those tasks without losing interoperability with the overall publishing infrastructure.

2.3.2 Topic Elements

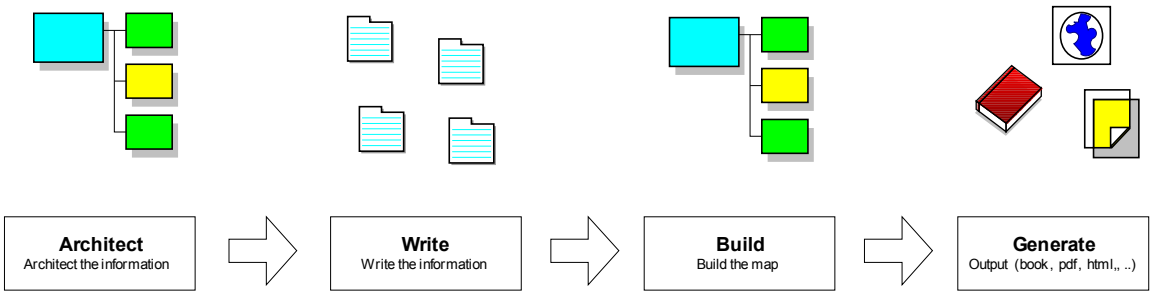
This is a generic topic which has some information and it still has some meaning when it stands on its own. Structurally, it could be a title with a body which may have some sections in it, but there should be nothing else other than that.

More specifically, we have a Concept which provides background information that the user needs to know, a Procedure which provides procedural details such as step-by-step instructions, and a Reference provides quick access to facts.



2.3.3 Map

The idea of having the topic and map is to provide a simple and flexible approach which can be applied to all sources of area. This diagram provides a case of specific and concrete uses.



Architect

We start by architecting the information to identify things like text flow, conceptual hierarchy, and a map which supports the relationships between topics. Once we have the architecture model without the content, it is already a map.

Write

We then start altering the content; it may be that already existing content is being reused. However, we will be reusing it in a model, which means that we should not say, “What do we have?”, or “What do we need?”. We need to say “What does the user need?”, then we will be able to reuse the content to fit that model. We do not start by saying, “We have a giant repository, what we can get out of it?”. Instead, we should say, “What is the user's need, and what is missing from the repository?”. Thus we focus on the user needs first.

Build

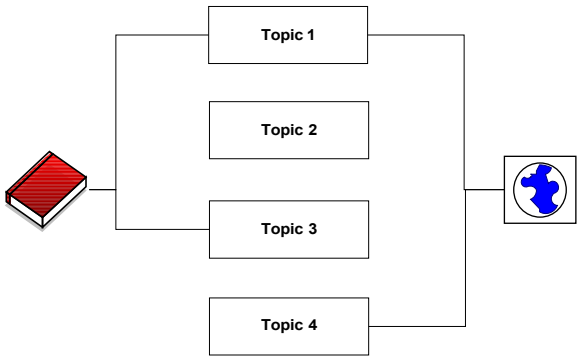
Once we have that topic completed either by authoring or reuse, we will use the map to build the content out into various media types. We can also then start to customize the map for different media types, or create different versions of it for different outputs to different audiences.

So the idea is that you start with the information model and you end up with the information model. The information model is what you work with to author the interface from start to finish, which means we are focusing on the complete picture for our audience and for our deliverables. In other words, we will be focusing on the audience, focusing on the big picture, but at the same focusing on the individual topics, as individual topics as well. We don’t lose focus on the big picture, we move that focus into its own document where it actually becomes more important.

It’s not a design fact that you start and will throw away; it’s a living piece of your contents and your workflow, and all the way through the day we build and deliver it to the customer.

2.3.4 Reused Content

When topics are defined, we can start reusing them by pulling different sets for different deliverables. We can also leave out those topics we don't want, and that is an important feature.



For example in this diagram, a book used topic 1 and topic 3, a Web site used topic 1 and topic 4, and topic 2 is not used at all.

If there a large document from which we would like to generate different versions from it, then we must explicitly say that we want one topic but not another topic, which can be considered a filtering activity.

Once we have the idea of topic and map, it is no longer a filtering activity, but instead we will have a collection of selection activities. By using a map, we can say that we have this topic or need that topic. Someone else can create another map by expressing his or her own criteria, which results in another selection set for having a different set of topics.

When a user needs a different criteria set, it doesn't affect us in the leap, nor does it affect other map creators. The problem of having a large document or set of conditions is that it will start to overlap and increase, and when we have many re-users, the process will tend to collapse, or break easily during the build process. That problem will disappear, and instead when we have a new user creating a new map, the map embodies the selection criteria which describe the topics they want and ignore the topics they don't want. When someone else adds another topic, they don't care. When someone edits some of the topics that they shared, then they care, but that is an isolated case. It is not the case that every time anyone touches something, everyone cares.

It is the case that only the parts which are shared matter; the part which they are not shared, do not matter. So with the scalable reuse, there is no limit number of maps that we can have across the system. The maps do not conflict with each other and are not aware of each other, unless we make them aware.

3 Product Review

In this chapter, we will briefly review some of today's CMS products. Based on what we have said in the previous chapters, CMS Visions and Design Principles, we will review each product's strengths and weaknesses. The review will be not between one CMS product and another, but rather we will focus on understanding their content structure and on-demand publishing features if there are any.

The purpose is to see whether each CMS has followed the topic map principle, therefore, we will not so much document details on their strengths, but rather we will explore their structure content weaknesses.

Before reading further, it is important to know that there are four main categories of CMS:

1. An Enterprise Content Management System (ECMS) is to manage documents, records and details which are related to an organization processes in an enterprise. The purpose is to provide the benefits of reducing paper use, faster access to the information, and security.
2. A Web Content Management System (WCMS) is to allow the user to create and manage HTML content without needing to understand HTML coding. Web Content Distribution and Publishing are also supported by WCMS.
3. A Document Management System (DMS) is an electronic filing system for capturing, storing and tracking document images. Access to content in a DMS should be quick and secured. Retention and Traceability should also be parts of DMS.
4. A Component Content Management System (CCMS) deals with content or component at a granular level rather than finished document. Each component can be represented as a topic, with a concept and a potential set of tasks defined. A component can be a chapter or a single word within a chapter. Further each component can have it's own lifecycle and be tracked individually or as part of the whole document. CCMS can also be integrated with ECMS, WCMS or DMS.

3.1 Dreamweaver

3.1.1 Description

Dreamweaver is a Web Content Management System, a product of Macromedia which was introduced to the market in November 1999. It has grown in popularity and become a powerful Web authoring tool.

Similar to other tools in it's technology space, Dreamweaver provides a visual way to create and manage Web pages within a Web site or across many Web sites. The purpose is to help developers speed up the process of building a Web site with point-click and drag-drop friendly features.

Also, Dreamweaver's graphical editor, content import and workflow engine were built to help developers working collaboratively in a team, and enable integration to and from support applications such as Microsoft Office, Photoshop and Fireworks.

3.1.2 Strengths

Dreamweaver has stood out from it's class. It is known as one of the most friendly tools for creating neutral Web site, which can be done quickly by using point-click and drag-drop features, and without any manual coding.

Dreamweaver can allow developers to create a central document storage location for a Web site project. From this central location, the developer can organize the content into directory structure and share it with the team. Individuals can work collaboratively with other by pull and push content changes through an FTP connection.

3.1.3 Weaknesses

Building a Web site locally with Dreamweaver is very convenient. However maintaining or deploying small changes to published Web pages is not that easy or user friendly.

Because the nature of the Web has changed over the years, Web 2.0 has been a fundamental technology break through and has dramatically gained popularity. It requires Web sites to have their content change more often than previously. For this reason, it is now a fundamental problem for Dreamweaver to remain a static Web site builder, which means Dreamweaver lies on the wrong side of the Web 2.0 growth path.

This is not Dreamweaver's fault, but rather it is because it was based on the older model of a static Web site builder from the beginning. It was meant for a central Web developer to develop all sites and content, as well as adding the navigation paths to every searchable item. This is not scalable, and it is in opposition to Web 2.0 fundamental concepts.

This implies that Dreamweaver was not built with the topic map model under the hook, where meta data of the content object can be easily modified and be reused for collaborative dynamic site updates or reconstructions.

The future of a site content builder is that it's content will need to be posted by many content distributors. These can be developers, technical writers, or even site visitors. The Web site will need to be constructed around content on the fly, where Web pages can also be ripped out, reconstructed or replaced. [13]

3.2 Drupal

3.2.1 Description

Drupal is an open source Enterprise Content Management System distributed under GPL (GNU General Public License). It allows an individual user or a group of users to easily organize, manage and publish different type of content on Web sites. It is often used for building web portals; however, because of its richness in features, Drupal can be used for almost any kind of web site.

3.2.2 Strengths

When you understand how to use Drupal, you will find it is very flexible and powerful. Because there are many components and features in Drupal, it is a great tool for building complex web sites. Almost whatever you think that your site will need, there is a great chance that the component is available for you to put on your Web site. In addition, Drupal also allows the user to create custom content types.

Drupal supports different types of site structures, and within each site structure, a user can specify very detailed rules on how the content should be displayed on the web site.

Drupal has very strong Web 2.0 support; this includes functionalities, group blogs, and user submitted content. Last but not least, Drupal makes things easy to find and update.

This implies that Drupal has followed the principle of the topic map in one way or another, but Drupal’s complexity has opened it up to some other kinds of weakness.

3.2.3 Weaknesses

The downside of having many features and flexibilities is that Drupal's administration user interface can be very confusing for a new administrator. The site configuration screens have a large number of settings and options, making them less intuitive and clunky to use. The installation process is little complex, which may require some learning curve.

Because of the flexibility of the system, it is important that user plan ahead for what they are going to build. It is recommended that people not start out by installing and using Drupal alone; they should hire a consultant to help out with the initial process.

This also implies that too many product features can easily create complexities which can lead to new problems and confusion. A successful CMS application should have it's content structure well enough defined for simple tasks. Feature overloading can create confusion which may require more time for a team to be trained and start collaborating.

Finally, Drupal's workflow is not strong, therefore it is probably not the best content management choice for an organization where there are different roles and ownerships needed for content. [15]

3.3 Velocity

3.3.1 Description

Velocity is a Java-based open source project, directed by the Apache Software Foundation. Velocity is not a CMS application; it is a template engine which was created to ensure a clean separation between the presentation tier and the business tier according to the MVC project model. This means that Java codes can be separated from the web pages, allowing Web designers to stay focused solely on designing the Web site, and programmers to focus solely on writing the code.

Here are some common application types for which Velocity can be used:

- For Web applications, by Web designers can use Velocity to create HTML dynamic information pages. According to MVC design pattern, these pages are processed by the VelocityViewServlet supported by Velocity.
- For generating source code, developer can use Velocity templates to generate Java source code, PostScript, or SQL.
- For email, developer can store email templates as a text file, and use Velocity to generate automatic emails for sending reports, account signup or password policy reminders.
- For document transformation, Velocity provides an Ant task for reading XML data file and feeds to Velocity templates. A typical example would be to convert a Microsoft Word document into an HTML page.

3.3.2 Strengths

These are some of Velocity strengths which many people found: [16]

- Velocity leads view designers and back-end developers with clean separation between the presentation tier and model/control tiers.
- Velocity macros can help view designer to reuse their code.
- Velocity templates can be used to generate several output formats such as HTML, XML, SQL, PostScript, and ASCII.
- Velocity is easy to learn.

Velocity is documented in this paper for it's success in solving JSP code separation problems, which allows different teams to work effectively.

3.3.3 Weaknesses

There were some complains between regarding Velocity error handling, templates and Hibernate tools. People have logged many issues about Hibernate typo errors which were not captured, nor can they be reported clearly by Velocity. It didn't fail when it should, nor did it tell where in the template the error occurred. [17]

3.4 JavaDoc

3.4.1 Description

JavaDoc is not a CMS application, it is a standard HTML documentation generator for Java, provided by Sun Microsystems. It generates API documentation from Java source code to HTML format.

3.4.2 Strengths

- JavaDoc is a necessary tool for Java, and it is simple and easy to use.
- JavaDoc can provide doclets and taglets API, which help to describe Java application structure.

3.4.3 Weaknesses

- JavaDoc cannot be easily extended to tasks other than documenting the programming Java code.
- JavaDoc is designed only for Java applications; it cannot be used for other programming languages, nor can it be applied to other content types.
- JavaDoc is not based on anything anywhere near the topic map principle.

That means JavaDoc is not a choice to serve as the tool for collaborative or on-demand content publishing.

4 Technology Background

This section is to review the technologies which needed to support our Document Builder. It will give some background information and a brief analysis on how we can relate to it in this project.

4.1 Apache Ant

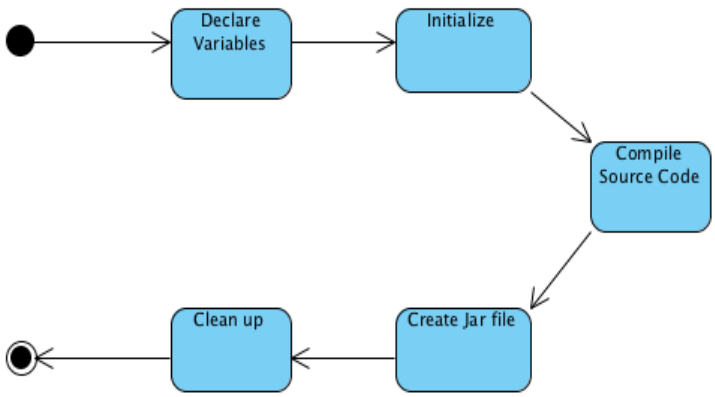
Apache Ant is an automated Java-based build tool. It has a similar purpose to the traditional Make tools which build C and C++ projects, Ant is designed to build Java projects.

The build file

By default, an Ant build file is named build.xml. One noticeable difference between Make and Ant tools is that the Ant build file is in XML format, whereas the Make build file has its own format. The simplicity and flexibility of XML allows new elements to be defined for extending Ant's build capabilities.

A simple build.xml

Ant tool can be used to automate typical build tasks such as compiling the code into Java byte code, creating the jar file for code distribution, and clean up.



These build steps can be specified in the build.xml as follow:

```
<?xml version="1.0"?>
```

```
<project name="odm" default="dist" basedir=".">
        <property name="src" value="."/>
        <property name="car" location="scr/car"/>
        <property name="build" value="build"/>
        <property name="dist" value="dist"/>

         <target name="init">
                <mkdir dir="${build}"/>
        </target>

        <target name="compile" depends="init" >
                <!-- Compile the java code -->
                <javac srcdir="${car}" destdir="${build}"/>
        </target>
        <target name="dist" depends="compile">
                <!-- Create the distribution directory and pack all built files into .jar file -->
                <mkdir dir="${dist}/lib"/>
                <jar jarfile="${dist}/lib/ODM-${DSTAMP}.jar" basedir="${build}"/>
        </target>

        <target name="clean" >
                 <!-- Delete the ${build} and ${dist} directory trees -->
                <delete dir="${build}"/>
                <delete dir="${dist}"/>
        </target>
</project>
```

```
<project name="odm" default="dist" basedir=".">
```

The *project* is the root element which has three attributes:

- *Name* is an optional attribute which is the name of the project that you would like to build.
- *Default* is the mandatory attribute, a default target for the build.
- *Basedir* is an optional attribute which specifies the base directory from which the Ant build file can be referenced. If this attribute value is not specified, the parent folder will be used as the default base directory.

```
<property name="src" value="."/>
```

With the *property* element, the user can define the variables for the Ant build file. There are two attributes one can define for the proper element.

- *Name* is a mandatory attribute which is the name of the variable.
- *Value* is a mandatory attribute which is the value of the variable.

In the example above, the variable src has a value of ".".

```
<target name="init">
```

After declaring the variables for the build file, the user will need to define a set of target actions for the Ant build. This is when the Ant *target* element should be used.

It would make sense to have an *init* section as the first target actions set for the Ant build initialization step. In the above initialization example, we have the mkdir ${build} as the action step of creating the build directory.

```
<target name="compile" depends="init" >
```

The next target section *compile* is to indicate what code will be compiled. It has a dependency on the init section executed before this.

In the above example, we have the command <javac srcdir="${car}" destdir="${build}"/>, which is to compile the source code from the directory "car" into the destination directory "build".

```
<target name="dist" depends="compile">
```

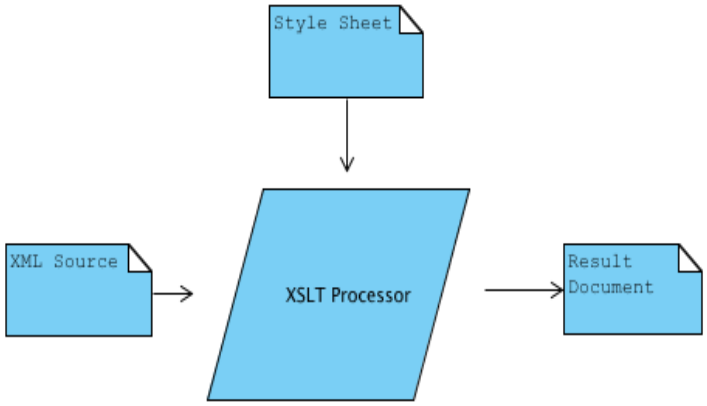
Next is to bundle up the compiled code for distribution. From the above, the first command in this section <mkdir dir="${dist}/lib"/> is to create the "dist/lib". The second command <jar jarfile="${dist}/lib/ODM-${DSTAMP}.jar" basedir="${build}"/> is to create the jar file from compiled code into the newly created directory.

```
<target name="clean" >
```

The last target step is to *clean* up. For this, we need to delete the "build" and "dist" directories.

4.2 XSLT, Xpath

XSLT stands for Extensive Stylesheet Language Transformation. It is used to transform XML document source into other document formats, such as HTML, PDF, TXT, or XML.



The transformation process occurs as in the above diagram. First, the XSLT processor parser reads the input XML source and parses it into a tree. With the style sheet as the second input, the engine transforms the tree by using the transformation template rule which is defined in the style sheet. Finally, the serializer writes the result tree out to the result document.

During the transformation process, XSLT uses Xpath to find matching elements from the source xml. When a matched element is found, the matched part in the source xml will be transformed into the result document. The translation rule in the template instructs the processor to create nodes in the result tree.

Template Rule

The XSLT style sheet is XML based. That means the XSLT style sheet is an XML tree, which defines a set of transformation rules, and is used to transform other XML trees.

The transformation rules are referred to as templates within XSLT. Here is a simple example of XSLT and template definitions.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
        <xsl:output method="xml" indent="yes"/>
```

```
    <xsl:variable name = "car"
            select = "document('../data/car.xml')"/>

    <xsl:template match="/project">
            <cs298>
                    <xsl:apply-templates select="property"/>
            </cs298>
    </xsl:template>

    <xsl:template match="property">
            <item name="{@name}">
                    <xsl:value-of select="document($car)" />
            </item>
    </xsl:template>

</xsl:stylesheet>
```

```
<?xml version="1.0" encoding="UTF-8"?>
```

Because XSLT is XML based, it needs to begin with the XML declaration.

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
```

Since this is a stylesheet document, we need to declare it with <xsl:stylesheet>

```
<xsl:output method="xml" indent="yes"/>
```

We need to indicate the output format, it is XML in this case.

```
<xsl:variable name = "car" select = "document('../data/car.xml')"/>
```

We declare a variable name “car”, which has the value of select statement.

```
<xsl:template match="/project">
```

The <xsl:template> element is to defined a template. In this case, we have a template which matches "/project".

The section inside the <xsl:template> can contain html tags.

```
<xsl:apply-templates select="property"/>
```

Since we have many properties inside the "/project" element, we need to call the property template.

```
<xsl:value-of select="document($car)" />
```

<xsl:value-of> is used to select the source content value of an xml element. In this example, we select all content from the car variable.

4.3 XSLT Processor

One of the XSLT processors is Apache Java Xalan. It is used to transform the source XML document into other document format such as HTML, text, or to XML format.

Xalan can be run from the command line. For example:

```
java org.apache.xalan.xslt.Process -in build.xml -xsl trans.xsl -out out.xml
```

or Xalan can be used through the Java package javax.xml.transform. For example:

```
import java.io.*;
import javax.xml.transform.*;
import javax.xml.transform.stream.*;

public void transform(String xslFile, String xmlFile, String outFile)
   throws TransformerException, TransformerConfigurationException, FileNotFoundException, IOException
 {
        TransformerFactory tFactory = TransformerFactory.newInstance();
        Transformer transformer = tFactory.newTransformer(new StreamSource(xslFile));
        transformer.transform(new StreamSource(xmlFile), new StreamResult(new FileOutputStream(outFile)));
}
```

4.4 Xlink, XPointer

XLink

Similar to HTML hyperlink, Xlink allows the user to create hyperlinks to specific content in the XML document. It is an XML element which represents a URI linking to a Web resource, including another element within an XML document.

Consider the following Xlink elements within an XML document.

```
<?xml version=”1.0”?>
<cs298 xmlns:xlink="http://www.w3.org/1999/xlink">
        <topics>
                <topic name="truck" index="1" xlink:type="simple" xlink:href="../data/truck.xml" />
                <topic name="car" index="2" xlink:type="simple" xlink:href="../data/car.xml" />
        </topics>
</cs298>
```

```
<cs298 xmlns:xlink="http://www.w3.org/1999/xlink">
```

The Xlink namespace needs to be declared.

```
<topic name="car" index="2" xlink:type="simple" xlink:href="../data/car.xml" />
```

The above Xlink element has the name “car”, xlink type is simple, and the hyperlink is to car.xml. The simple xlink type is a two-ended link, which means that when it's clicked, it goes to where xlink:href said. There are also multi-ended links; please refer to Xlink documents for more details.

XPointer

There are cases when we need to point to specific content in the XML document. For this purpose we can use Xpoint, which is a combination of URL and Xpath.

For example,

```
<?xml version=”1.0”?>
<cs298 xmlns:xlink="http://www.w3.org/1999/xlink">
```

```
        <topics>
                <topic name="truck" index="1" xlink:type="simple" xlink:href="../data/truck.xml" />
                        <level name="beginner" xlink:type="simple"
        xlink:href="truck.xml#xpointer(//../level[@name='beginner']"/>
                        <level name="intermediate" xlink:type="simple"
        xlink:href="truck.xml#xpointer(//../level[@name='intermediate']"/>
                        <level name="advance" xlink:type="simple"
        xlink:href="truck.xml#xpointer(//../level[@name='advanced']"/>
        </topics>
</cs298>
```

```
<level name="beginner" xlink:type="simple" xlink:href="truck.xml#xpointer(//../level[@name='beginner']"/>
```

This example xlink has a hyper link to the truck element with documentation level for beginner.

4.5 XSL-FO

XSL-FO stands for Extensible StyleSheet Formatting Objects; it is an XML based document, a language for formatting XML data out to other media. XSL-FO allows more complex visual presentation than HTML, and is more suitable for creating professional presentation layout. It is often used to generate PDF documents.

Document Structure

There are two steps in generating a PDF document with XSL-FO. The first is to create the .fo or .fob file; the second is to generate the PDF from the .fo file. Here is an example of the .fo file.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">
<fo:layout-master-set>
  <fo:simple-page-master master-name="Letter" page-height=”11in” page-width=”8.5in”>
    <!-- Page template goes here -->
  </fo:simple-page-master>
</fo:layout-master-set>
<fo:page-sequence master-reference="Letter">
  <!-- Page content goes here -->
</fo:page-sequence>
```

```
</fo:root>
```

When the .fo is parsed, the page is created based on the page master. The page content is then filled with content from the page sequence. This process is repeated until there is no more content.

```
<fo:layout-master-set>
  <!-- All page templates go here -->
</fo:layout-master-set>
```

The <fo:layout-master-set> element can contains one or many templates.

```
<fo:simple-page-master master-name="Letter" page-height=”11in” page-width=”8.5in”>
   <!-- Page template goes here -->
</fo:simple-page-master>
```

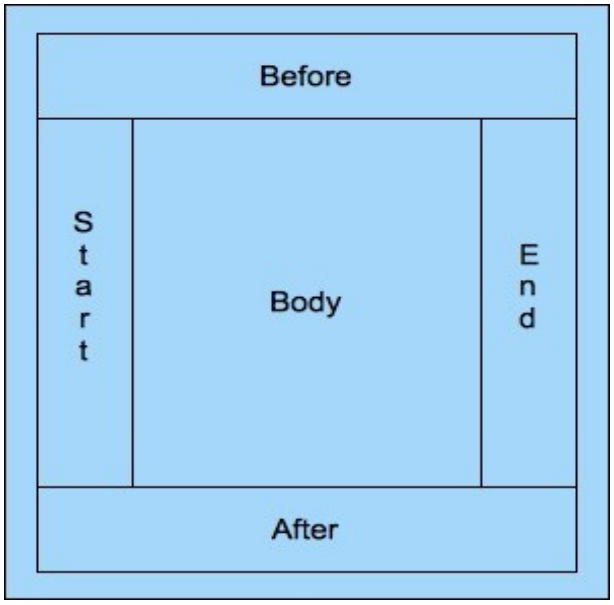
Each <fo:simple-page-master> element has a single page template with a unique template name.

```
<fo:page-sequence master-reference="Letter">
  <!-- Page content goes here -->
</fo:page-sequence>
```

To describe the page content, the page-master references to one or more <fo:page-sequence> elements. Each has a unique name; “Letter” is used as example, but one can use any other unique name.

To display the output, there can be any number of areas (boxes). Output can be text, paragraphs, pictures, etc. Each will need to be formatted within the areas. There are several types of areas, such as Page, Region, Block, Line, and Inline.

The Page layout can be viewed as in the below diagram. The body Region has elements such as Before, After, Start, and End. Please note the Margin is the outer section between the body Region and the edge of the Page.



Here is the sample code for constructing the Page.

```
<fo:simple-page-master master-name="Letter" page-width="297mm"
page-height="210mm" margin-top="1cm" margin-bottom="1cm"
margin-left="1cm" margin-right="1cm">
  <fo:region-body margin="3cm"/>
  <fo:region-before extent="2cm"/>
  <fo:region-after extent="2cm"/>
  <fo:region-start extent="2cm"/>
  <fo:region-end extent="2cm"/>
</fo:simple-page-master>
```

This has given a brief introduction to XSL-FO structure. Please refer to XSL-FO online documentation for more details on Block, Line, and Inline as well as other XSL-FO elements types.

Rendering Processors

There are many XSL-FO processors available; some are free, such as FOP and PassiveTeX, and some are commercial products, such as XEP and XSL Formatter. The major differences between the processors are often mostly in the output qualities.

In this project we used XEP desktop version. We selected XEP just because it has better quality and the vendor offers a free desktop version for non-commercial products.

5 System Requirement

Based on the concepts introduced in previous chapters; topic map, structured content, and design principles; we will attempt to implement a document builder system. Starting with Ant build.xml, our system will document the components individually and allow the user to assemble them as needed for delivery. Our system design should also facilitate the concepts of content reuse, different levels of content, different channels of publishing, and different forms of customization.

We shall call our system Document Builder (DB), and it should be designed to support the following feature set.

Topics about Object

- DB design should be topic based.
- DB build script should be object based.
- Each object can be documented by one or more topics.
- The user manual should be built as a collection of topics.

Topic Granularity

- The user manual should be built to target difference audience levels, hence DB topic should be specialized into levels of basic, intermediate, or advanced.

Map

- The links between DB object and DB topic, levels should be maintainable in an XML table format.

Specialization

- New objects, topics and topic levels should be easy to introduce.

Customization

- It should be easy to customize the layout presentation of the user manual.

Different Publishing Channels

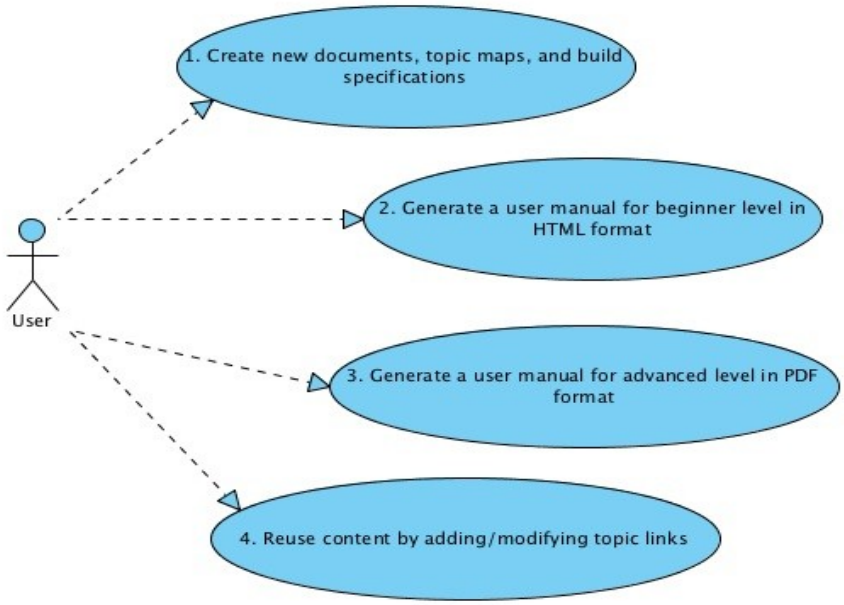
- The DB system should be able generate the user manual in HTML and PDF formats.

Reused of Content

- In the DB build xml, we should be able to indicate different sets for different deliverables. We should be able to open an existing DB build xml, and remove those objects we don't want, or add new objects we do want.
- In the table of the map, we should be able to associate different DB objects to different DB topic(s)/level(s) .

5.1 Use Cases

We want to build our system to support the following use cases:



1. To be able to deliver the right content to the client, DB should allow the user to specify how the content of a user manual should be built. The user should be able to open the DB build script and indicate which components should be documented, and how the documentation should be assembled.
2. One publishing channel of the user manual is in HTML, and this is typically the default. DB topics can be documented to target different audiences, such as beginner, intermediate, or advanced, where beginner is the default target.
3. Another publishing channel is in PDF. We should be able to generate an advanced user manual in PDF format.
4. DB topics should be able to be reused and linked to different DB objects and publishing channels. It should easy to configure how this works.

5.2 UI Prototype

The below screenshot shows how a user manual will look in HTML form. It has a general description of the user manual and a table of contents, followed by the chapters and their topics.

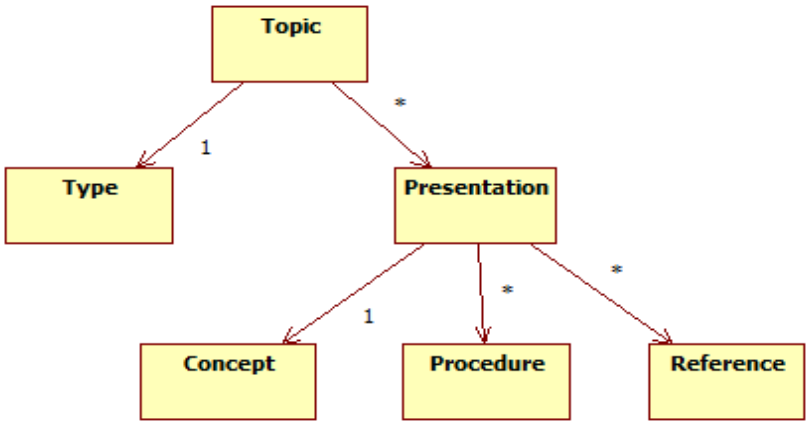
6 System Design

This chapter is to describe the DB system design and it's elements. The sections to be covered are:

- Topic - What is a DB topic?
- Document Structure - What is the user manual document structure like?
- Document Views – What are the possible presentation views of a user manual?
- Map – How does DB map an object to a topic?
- TOC – What is a TOC in the DB system and how will it be generated?
- Build Process – An overview of the stages in the build process.
- File Naming Convention and Usage – An agreement on how the system files should be named and used.

6.1 Topic Elements

A topic has a specific type, and a specific set of information which can be organized to be a partial or complete documentation around a specific object. For example, a specific compiled Java class can be considered as an object, or a compiled Java folder can be an object, or an entire Java package can be an object. These objects are specified in the Apache Ant build xml file, as the specification of the user manual to be built.
The Type element is to describe what the topic type is, whether it's a title name, a chapter, a paragraph, a header or footer note, a glossary, an appendix, etc.



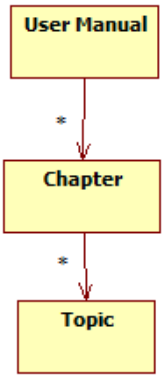
A topic also has different presentations; each presentation is targeted for a specific group of different audience experience level: beginner, intermediate, or advanced. Within a presentation, we have three elements: Concept, Procedure, and Reference.
The Concept element is to describe the concept of the topic presentation. For example, if the topic is a car, it should have information about the car. Please note that for some

topics, the description of the concept can be generalized for all levels or it can be specialized to that presentation.
The Procedure element is to describe the process or task on how to perform the topic at the given presentation. If the topic is a car, it should have information on how to operate the car at the given presentation.
The Reference element contains relevant links to more information about the topic presentation. For example, relevant links for a topic about a car could be a Toyota dealer website, the California driver's license test, etc.

DB topic structure supports the reused of content. New user manual structure arrangement can link to an existing topic from an existing user manual, to form a new different user manual.

6.2 Document Structure

The user manual is a document, which is the root element in the document tree. Just like a book, the user manual can have many chapters, and each chapter can have many topics.

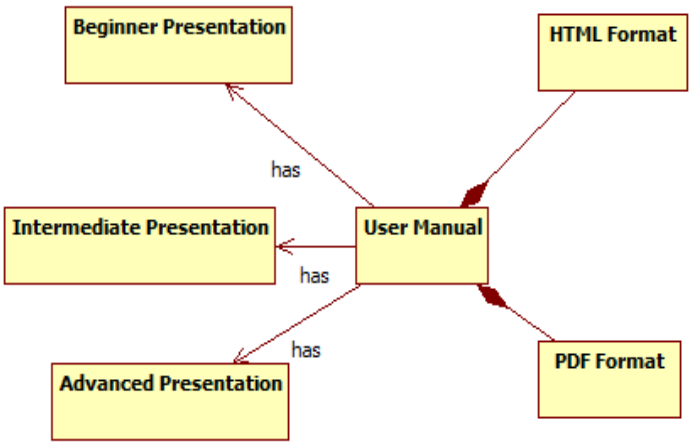


Depending on who the audience is, each topic in the user manual can be built with a specific presentation of information for that audience. If the user manual is for Beginner, the topic should be built with a Beginner presentation of information. The same presentation rule applies for Intermediate and Advanced.

6.3 Document Presentations

The user manual's publishing format can be either in HTML or PDF.

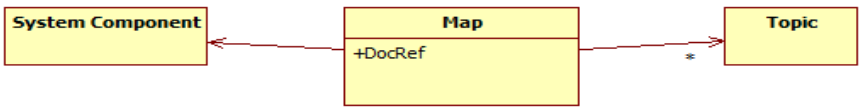
Since the user manual can be built to target difference audience levels; basic, intermediate, or advanced; it probably makes sense to have different template layouts for the user manual. In practice, it is often better for a basic user manual to have a much simpler layout presentation than an advanced user manual, where an advanced user manual may include much more information and more diagrams.



The layout presentation can also be customized. If the user wishes to do so, s/he will need to create his/her own stylesheet and specify it's use in the system. If there is no custom stylesheet created, the system will use the default stylesheet.

6.4 Map

A system component can be a Java class object which has been built by Apache Ant. To generate the documentation in the user manual for the system components, a map table (DB Map) can be used to locate the topic document files. A system component can be mapped to one or more topics.



A map entry in the map table, can have one or more DocRef attributes, which link to the topics which have been documented in a specific location. This location can be a file, a Web resource, or any other information repository.

To reuse content, user can modify the map to link to an existing topic or a new topic.

6.4.1 DocRef

In the DB Map XML file, the DocRef (document reference) attribute is an Xlink which points to the source of the topic information stored in a remote XML file. We should recall that an Xlink is similar an HTML link; it allows an XML author to link from one XML document to another.

For example, in the DB Map file, we have a DB Map with topic name “car” as follows:

```
<topic name="car" index="1" xlink:type="simple" XLink:href="../data/car.xml">
```

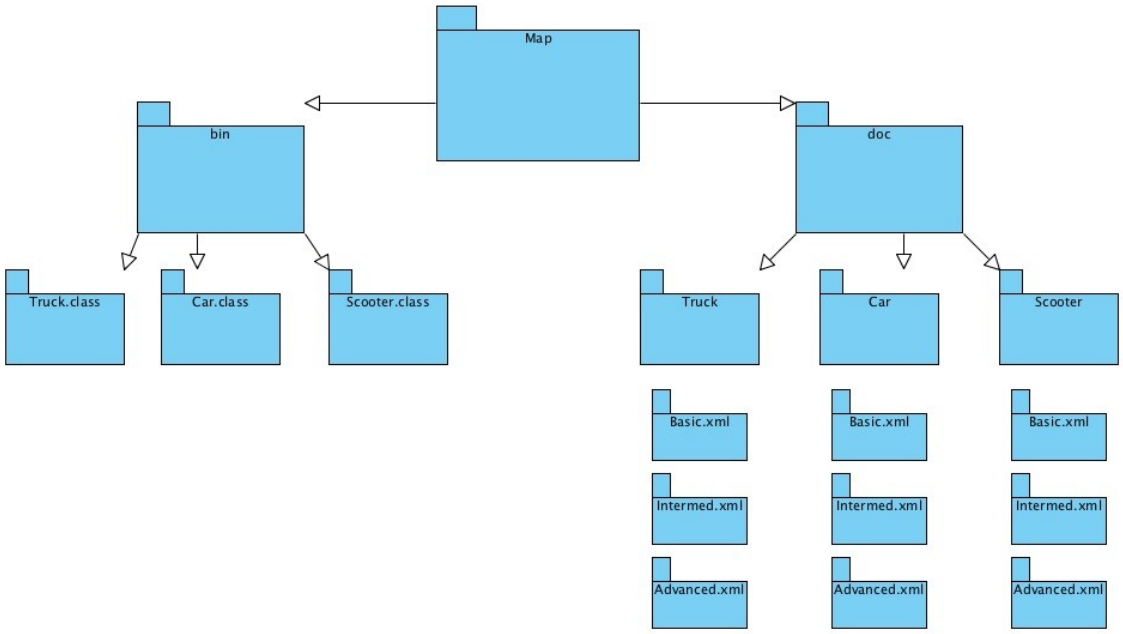
In the case that we need a DB Map to remotely reference to a specific element in another XML file, we can use Xpointer in conjunction with Xlink to provide the remote path to it. We should recall that Xpointer is a combination between a URL and an XPATH expression, and is used to reference a specific element in the remote XML document.

For example, in the remote car.xml file, we have different sections documented for different audience levels, and each section is defined as a sub-element of the topic “car”. To extract a specific element section for a specific audience level, called “beginner” in that topic, we can use Xpointer as follows:

```
<level name="beginner" xlink:type="simple"
xlink:href="../data/car.xml#XPointer(//cs298/content/levels/level[@name='beginner']"/>
```

6.4.2 Create Doc Files

DB document files are in XML format, and they should be created and kept in a folder name. The folder name should be the same as the topic name, and each topic folder should be created just underneath the doc root folder. This is to allow the system to map correctly between the object files and their associated document files.



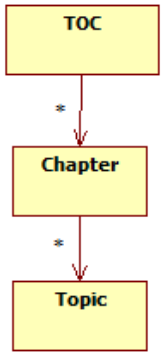
The above diagram shows an example of how folder structure should be created. The example is to have three Java objects compiled out to the bin folders, Truck.java, Car.java, and Scooter.java, and we want to create document files and map them to these class objects.

On the right hand side, under the Doc folder, we need to create three folders Truck, Car, and Scooter. For each of these folders, we create different documents for different topic levels, Basic.xml, Intermediate.xml, and Advanced.xml.

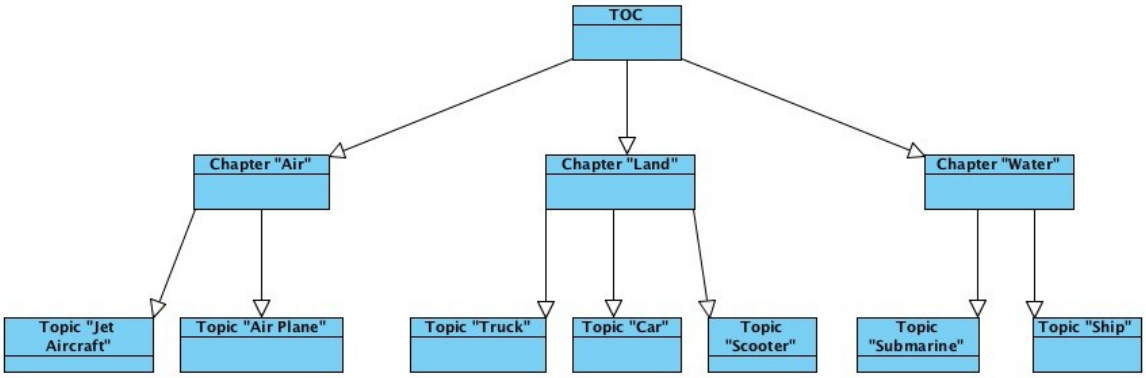
It is recommended that one plan the document directory structure in advance. It could be a good practice to perform these steps: create the directories, create the document files, then update the map.xml to link between the object and its topic document file.

6.5 TOC

One nice thing for a user manual to have is a table of contents (TOC). Logically, a TOC should be constructed as in the below tree structure diagram. A TOC should have at least one chapter, and each chapter should have at least one topic.



An example of a TOC for our Transportation User Manual would be as follows:

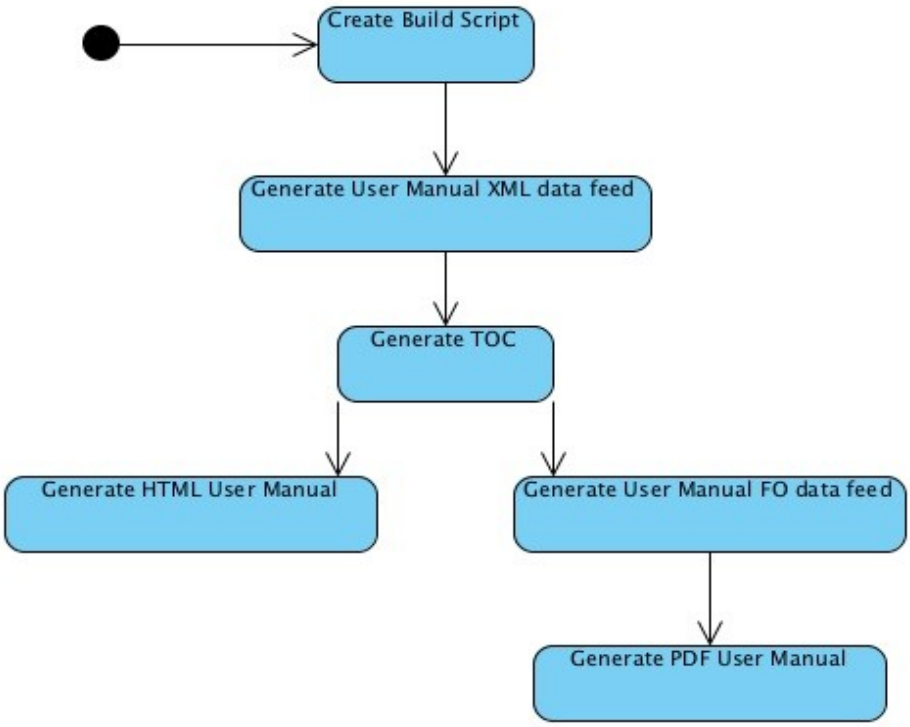


The Transportation User Manual has different chapters: Air, Land, and Water. Each chapter describes the vehicle topics for the chapter.

To generate the TOC for the user manual, the system will parse the xml data feed and generate a TOC xml file. During the final build of the user manual, the TOC xml file will be loaded into the user manual. In the next section, we will look in more detail at the entire system workflow.

6.6 Build Process

This section will give an overview of the stages in the build process, from the beginning process of constructing the basic elements to the end process of building the user manual.



At first, the user needs to create the build script, which should indicate which DB objects will be documented and how.

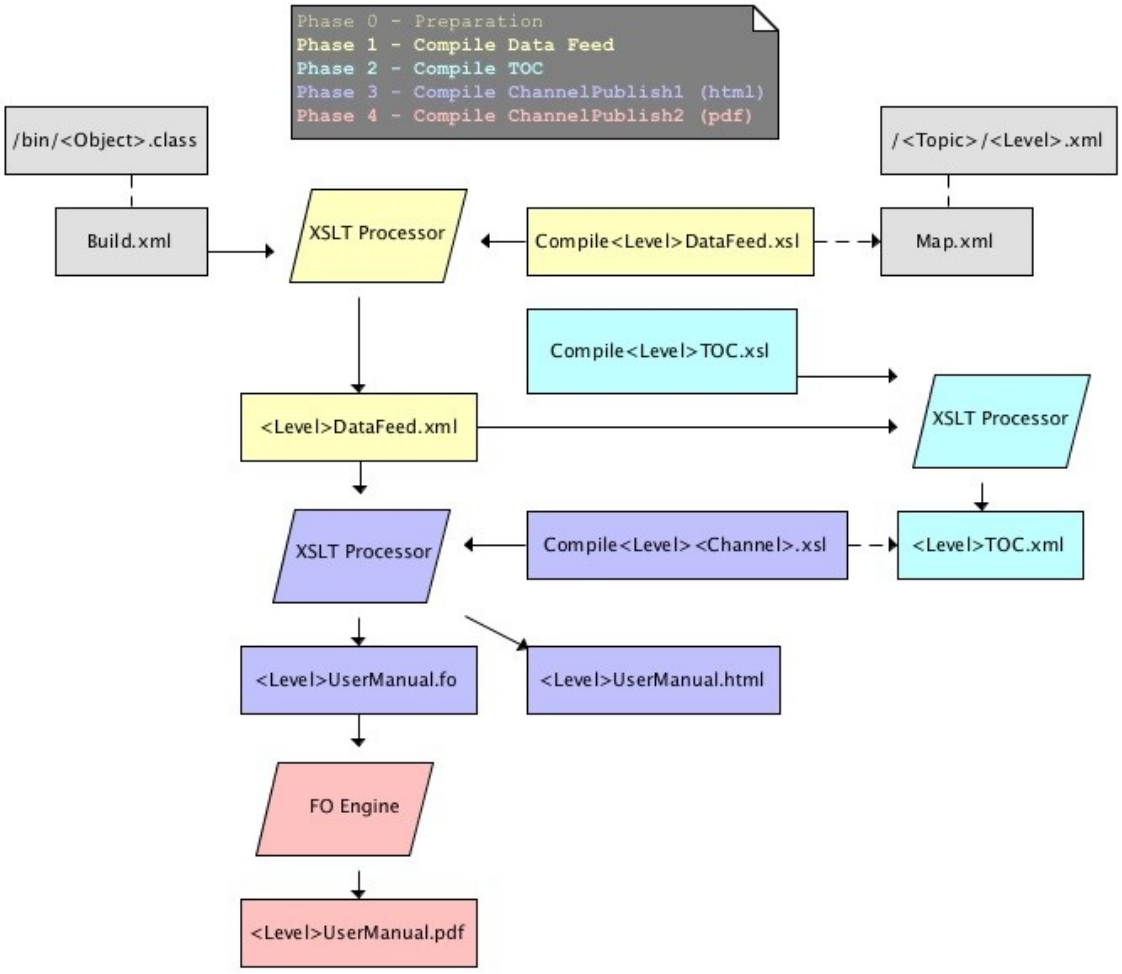
Once the build script has been created, the user will execute the build process. There should be a master script to automate the entire build process. The system will parse the build script to generate the intermediate data feed file. The format of the data feed will be in universal XML format. From the XML data feed, the system can generate the TOC file. It will later be merged during the process of constructing the user manual.

Depending on which publishing channel has been selected, the necessary auto scripts will be run to render the output into that channel data format. If there was a request to build the user manual to HTML, there will be XSLT to parse the XML data feed and render it into HTML presentation. If there was a request to build the user manual to PDF format, there will be an intermediate step to translate XML data feed into FO format before rendering it into PDF.

6.7 File Naming Convention and Usage

The following diagram shows how we should name our files and folders. This will not only help the system to locate the file correctly, but also help us to organize and identify it's purpose easily.

In the diagram, please note that we use the color scheme to identify different phases for different set of files.



This is how each file is used:

Before the system can run the build process, user will need to create document files, create topic map, and configure the build.xml.

In phase 1, the system generates the xml data feed file. The CompileDataFeed style sheet parses the Ant build.xml to figure out what are the topics that need to be included in the user manual. For each topic that it found, the style sheet will look up in the Map file for the content file. The content will be merged into the xml data feed file.

In phase 2, the system generates the TOC. The Compile<Level>TOC style sheets are used to parse the <Level>DataFeed xml files. By default, there will be three levels: beginner, intermediate, and advanced. For each of the levels there will be a specific style sheet file and data feed file working together to generate the TOC.

In phase 3, the system generates the media channel file for the user manual. If the publishing media channel is HTML, the Compile<Level><Channel> style sheet will produce the HTML user manual. It will parse the <Level>DataFeed xml file and merge with the <Level>TOC xml to produce the user manual in HTML.

If the publish channel is PDF, phase 3 is to generate the FO (Format Object) document files. As with HTML, the Compile<Level><Channel> style sheet will parse the <Level>DataFeed xml file and merge it with the <Level>TOC xml to produce the FO files.

In phase 4, the system generates the PDF for the user manual. The <Level>UserManual FO file contains the formatting rules, which will be fed into the FO engine to render nicely printable PDF files.

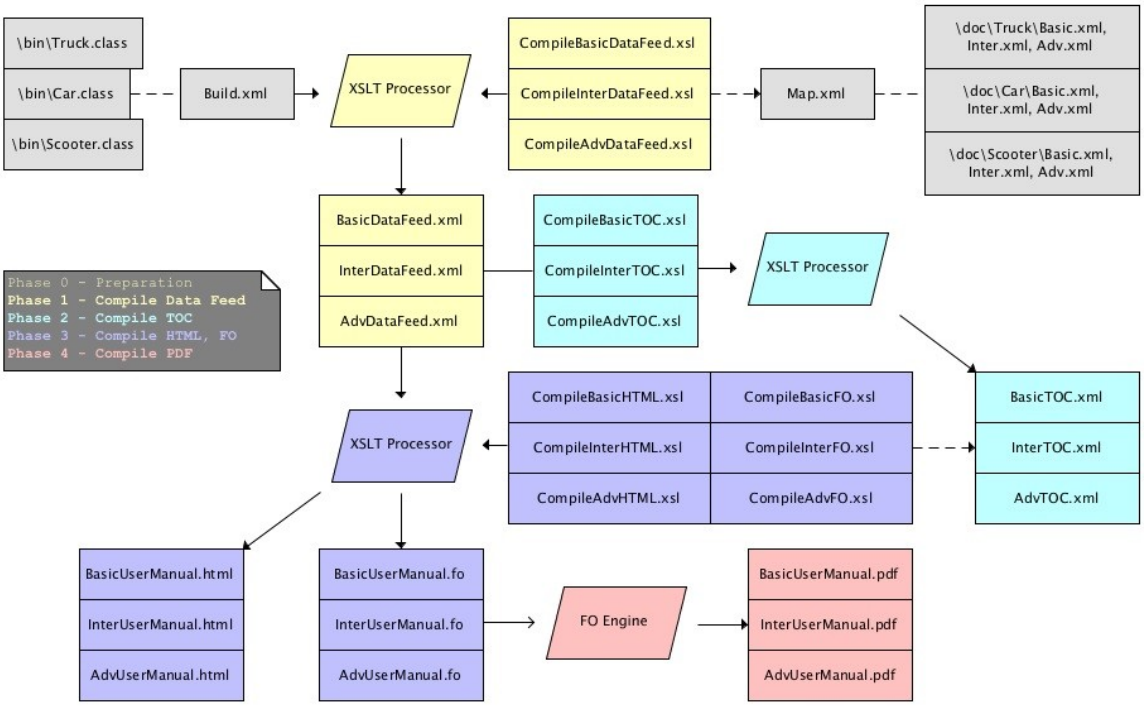
In the next chapter on Implementation, we will look into the details of the style sheets, as well as their input/output files.

7 Implementation

To describe the system implementation, we will use the example of compiling Transportation User Manuals for Truck, Car, and Scooter.

7.1 Sample Files Overview

Below is the summary view of the files for the given example, and how they are connected from one phase to another.

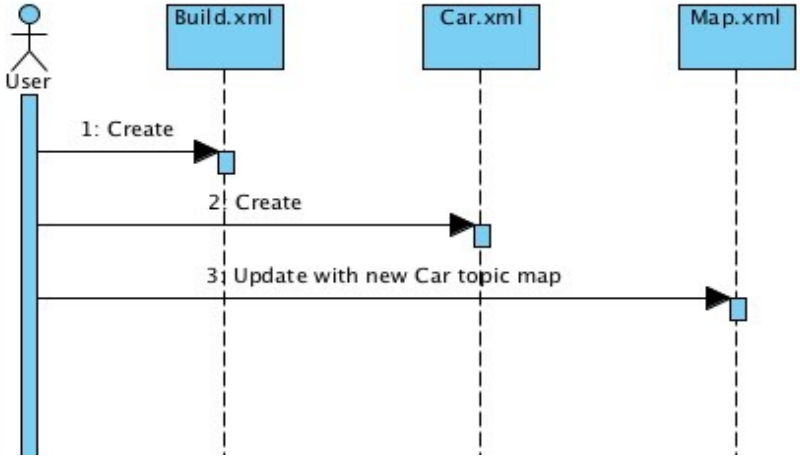


In the following sub chapters, we will explain in details how each file is actually implemented and operated. This will help us to understand DBS system implementation in more detail.

7.2 Components in Detail

7.2.1 Phase 0 – Preparation

Creating the build specification, preparing the content file, and updating the map table are the first steps in the process of compiling a user manual. In this section, we look into the detail of their steps.



7.2.1.1 Build.xml

To provide the build specifications for the user manual, the user will need to open the build.xml, and specify the object and it's build specifications.

For example in the following build.xml, the build specification is to generate a user manual for the topic truck with a basic documentation level, topic car with an intermediate documentation level, and topic scooter with an advanced documentation level.

```
<?xml version="1.0" encoding="UTF-8"?>

<project name="Transporting" default="dist" basedir=".">
   <description>
      simple example build file
   </description>
```

```
<!-- set global properties for this build -->
<property name="truck" doclevel="basic" location="src/truck"/>
<property name="car" doclevel="intermediate" location="src/car"/>
<property name="scooter" doclevel="advanced" location="src/scooter"/>
<property name="build" location="build"/>
<property name="dist"  location="dist"/>

....
```

When parsing the build.xml, the DB program will look for a property element which has a doclevel attribute specified with values of either basic, intermediate, or advanced. According to the given doclevel, the program will load the right content into the user manual. If there is no doclevel value , the program will not load the documentation for that property element. Please also note that the property name is the topic name.

7.2.1.2 Content File

For each of the DB topics, the user will need to create content file(s) for it and place them under the topic name folder. By default for any topic, there can be three kinds of content files: basic, intermediate, and advanced. Here is a sample of a basic content file for the topic *car*.

```
<DB>
	<topic name="car"/>
		<type name="text body"/>
		<content>
			<level name="beginner">
				<concept>A car is a small four wheels automobile which run on gas.</concept>
				<procedure>Turn on the ignition key, release brake, step on gas.</procedure>
				<reference>http://en.wikipedia.org/wiki/Car</reference>
			</level>
		</content>
	</topic>
</DB>
```

As you can see in the XML tree, the topic node has name and content. There are three content levels, and each has a concept, procedure and reference.

7.2.1.3 Map File

The purpose of the map file is to keep track of which topic belongs to which chapter, as well as the index order of it.

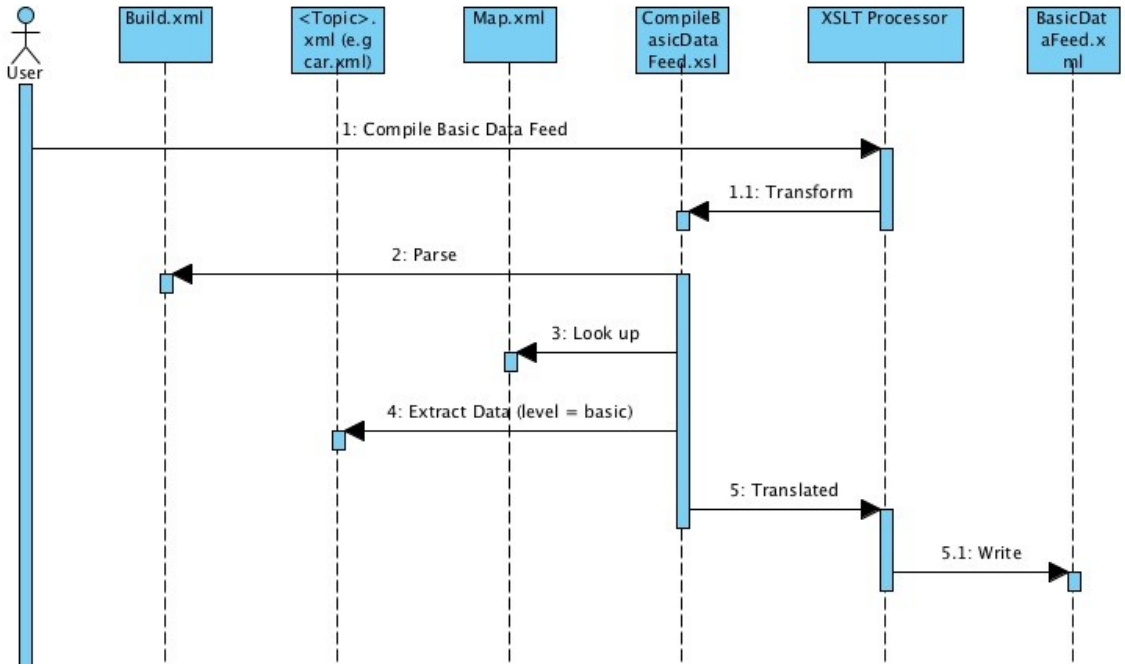
For a topic, the user can also use an Xlink to point to some specific element in a remote content XML file. By default, the content file name is the same as the topic name, and it is located under the folder name (which has same name as the topic name). If a user wants to customize the content file name and location, he or she will need to update the link information in this map file.

When the program is run, the map file will be used for looking up a specific topic and its chapter and topic information. The match link information will be used to generate the TOC for the user manual. Therefore, it is important that the user update this map file when there is a change, such as when a new topic introduced or when a topic index, file name, or file location has new value.

```
<DB xmlns:xlink="http://www.w3.org/1999/xlink">
        <docs>
                <doc name="transport">
                        <chapters>
                                <chapter name="fly" index="1">
                                        ...
                                </chapter>
                                <chapter name="land" index="2">
                                        <topics>
                                                <topic name="truck" index="1"
                                                        xlink:type="simple" xlink:href="../data/truck">
                                                </topic>
                                                <topic name="car" index="2"
                                                        xlink:type="simple" xlink:href="../data/car">
                                                </topic>
                                                <topic name="scooter" index="3"
                                                        xlink:type="simple" xlink:href="../data/scooter">
                                                </topic>
                                        </topics>
                                </chapter>
                                <chapter name="water" index="3">
                                        ...
```

7.2.2 Phase 1 - Compiling Data Feed

After the user has completed the preparation phase, the program is ready to compile the data feed XML for the user manual. Below are the program sequential steps for this compilation phase.



7.2.2.1 CompileBeginnerDataFeed.xsl

In this section, we examine the sample stylesheet which compiles the XML data feed for the beginner audience level. There are three main code sections in the stylesheet supporting our data feed compiling operation.

1. Variables - the first section is the variables section, which has two variables declared, map and dlevel. The *map* string value is pointing to the map file. The *dlevel* string value is to indicate this stylesheet is for a beginner audience level.
2. Template *project* – this template section is invoked each time the XSLT processor sees a project element in the build.xml file. It expects to see an attribute property in the element, and call the next template.
3. Template *property* – this template section is invoked for every project's property attribute, which is a topic to be documented in the user manual. The template code

is to extract the content documented in the content file for the topic, the result will be written out to an XML file. The content of the XML out file is documented in the next section.

Please note the header code section are highlighted in bold in the below code:

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
…

<!-- 1. variables  -->
<xsl:variable name = "map" select = "document('../in/map.xml')"/>
<xsl:variable name = "_dlevel" select = "'beginner'"/>

<!-- 2. template for match=”project” -->
<xsl:template match="project">
        …
        <xsl:apply-templates select="property"/>
        ...
</xsl:template>

<!-- 3. template for match=”property” -->
<xsl:template match="property">

        <xsl:variable name = "bname" select = "@name"/>
        <xsl:variable name = "tname" select =
        "$map//DB/docs/doc[@name='transport']/chapters/chapter/topics/topic/@name"/>
        <xsl:variable name = "dname" select =
        "$map//DB/docs/doc[@name='transport']/chapters/chapter/topics/topic[@name=$bname]/@xlink:href"/>
        <xsl:variable name = "fulldname" select = "$dname"/>
        <xsl:variable name = "dlevel" select = "@doclevel"/>

        <topic>
        <!-- Module name -->
        <name><xsl:value-of select="@name"/></name>

        <xsl:choose>
```

```
                <xsl:when test="$bname=$tname and $dlevel=$_dlevel">
                        ...
                        <!-- Concept -->
                        <concept>
                        <xsl:value-of
        select="document($fulldname)//DB/content/levels/level[@name=$dlevel]/concept"/>
                        </concept>
                        …
                </xsl:when>
        </xsl:choose>
        </topic>
</xsl:template>
</xsl:stylesheet>
```

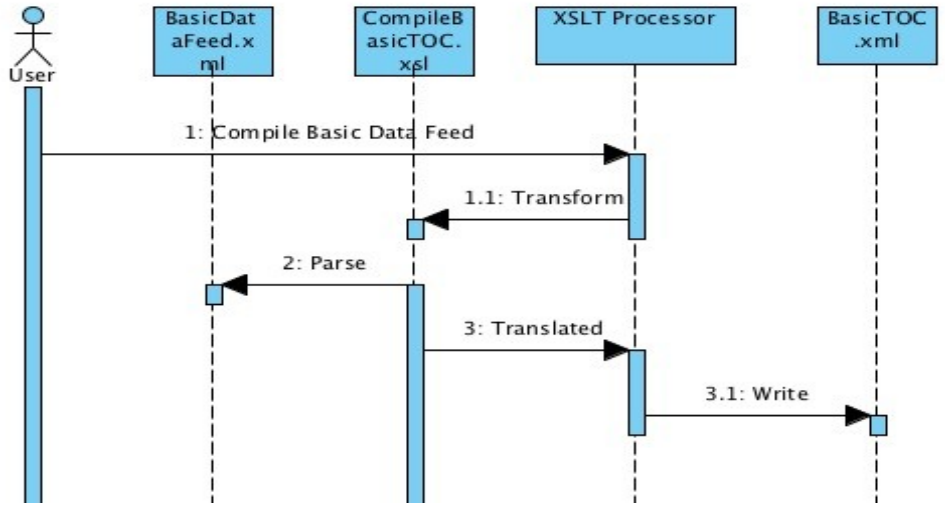
7.2.2.2 BeginnerDataFeed.xml

Below is the result XML which is generated from the CompileBasicDataFeed.xsl. It includes topic information such as name, index, chapter, and content. These data are required to build the user manual.

```
<?xml version="1.0" encoding="UTF-8"?>
<DB     xmlns:xlink="http://www.w3.org/1999/xlink"
            xmlns:xptrf="http://www.isogen.com/functions/xpointer">

    <topic name="truck" index="1"/>
            <chapter name="land" index="2"/>
            <type name="text body"/>
            <content>
                <concept>A truck is a large automobile which run on gas.</concept>
                <procedure>Turn on the ignition key, release brake, step on gas.</procedure>
                <reference>http://en.wikipedia.org/wiki/Truck</reference>
            </content>
    </topic>

    <topic name="car" index="2"/>
            ...
    </topic>

</DB>
```

7.2.3 Phase 2 - CompileBasicTOC

In phase 2, the system parses the data feed file and construct the TOC xml file.



To understand the details of this process phase, we will examine two files in this section: the style sheet CompileBasicTOC, which is used to parse the data feed file, and the BasicTOC, which is a result file of the parsing.
For both files, we will document the TOC process for Basic audience level so as to demonstrate how the TOC xml may look in general. Other TOCs for different audience levels, such as intermediate or advanced, share the same process and file structure as basic.

7.2.3.1 CompileBasicTOC.xsl

This style sheet parses the xml data feed file and generates the TOC content into a separate xml file. The main template in this style sheet is invoked when a topic element is parsed. The code body of this template will parse the child nodes to find information such as parent node name, index, and type. This information will be constructed into the TOC, and be written out to the BasicTOC.xml as in the next section sample file.

7.2.3.2 BasicTOC.xml

The basic TOC file is to outline the topics with basic audience level, and their parent chapter, topic name, and index should also be documented. It is a skeleton sketch of the Basic User Manual, which will be merged at the final Basic User Manual construction phase.

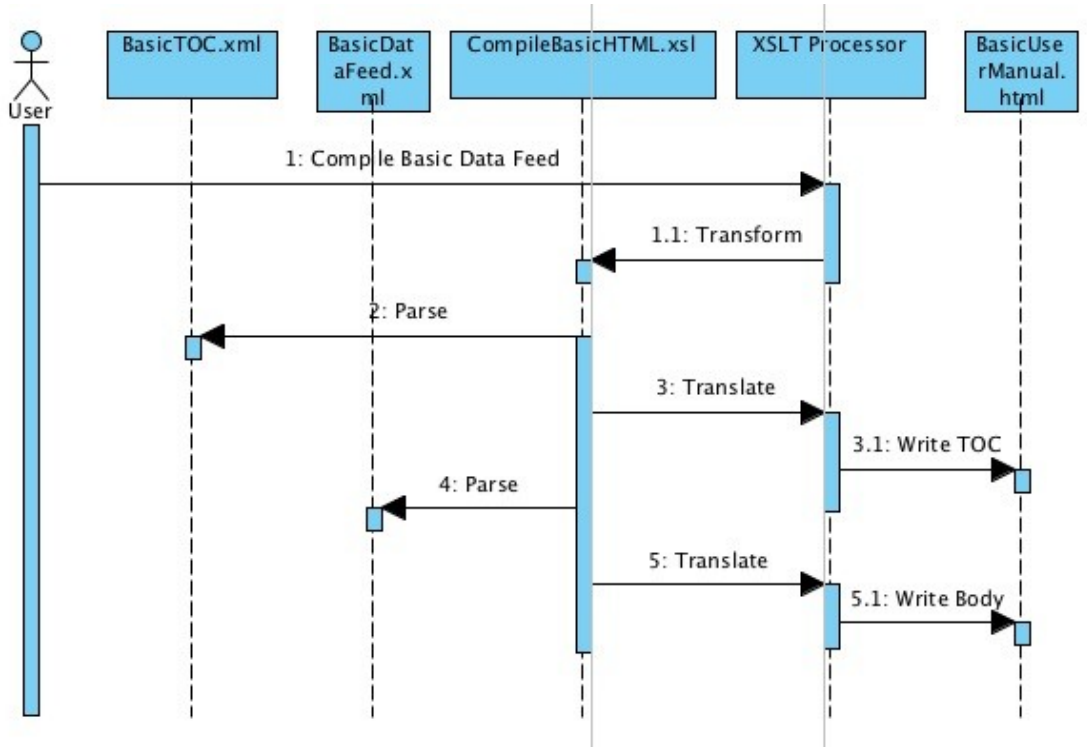
```
<?xml version="1.0" encoding="UTF-8"?>
<DB     xmlns:xlink="http://www.w3.org/1999/xlink"
                xmlns:xptrf="http://www.isogen.com/functions/xpointer">
        <chapters>
                <chapter name="air" index="1"/>
                        <topics>
                                <topic name="jet aircraft" index="1"/>
                                ...
                        </topics>
                </chapter>
                <chapter name="land" index="2"/>
                        <topics>
                                <topic name="truck" index="1"/>
                                ...
                        </topics>
                </chapter>
                ...
        </chapters
</DB>
```

7.2.4 Phase 3 - Compile HTML, FO

In phase 3, the system generates the user manual either in HTML, in FO, or in both.

7.2.4.1 BasicUserManualHTML.xsl

To print out the user manual in HTML code, the stylesheet parses the data feed XML, pre-append with TOC contents from the TOC XML file, and translates them to HTML presentation format.



In the below code, the key sections are highlighted in bold. They include the *variables* section, and a set of templates which translate the topic node content into HTML.

<?xml version="1.0"?>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"

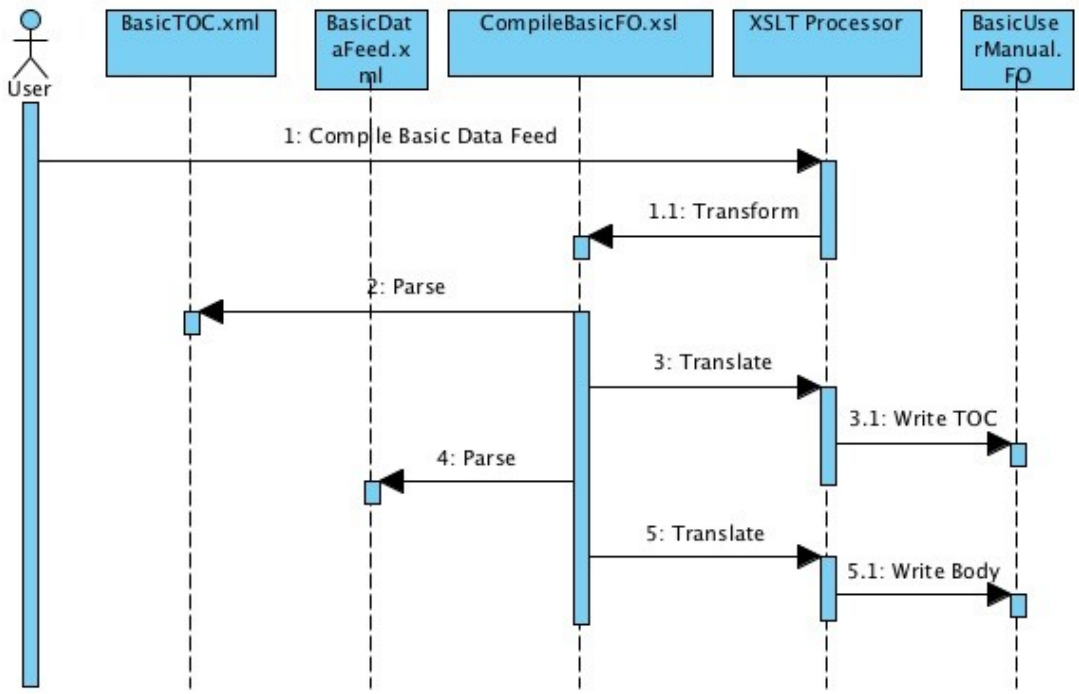
...

```
<!-- variables -->
<xsl:variable name = "_dlevel" select = "'basic'"/>
<xsl:variable name = "tocxmlfile" select = "document('basic_toc.xml')"/>

<xsl:template match="DB">
<html>
        <title>Document Builder</title>
        <body>
        <table width='100%'>
        ...

        <xsl:apply-templates select="docs"/>
        </body>
</html>
</xsl:template>

<xsl:template match="docs">
        <xsl:apply-templates select="doc"/>
</xsl:template>

<xsl:template match="doc">
        <xsl:apply-templates select="topic"/>
</xsl:template>

<xsl:template match="topic">

        <xsl:variable name = "tname" select = "./name"/>
        <xsl:variable name = "cname" select = "./concept"/>
        <xsl:variable name = "pname" select = "./procedure"/>
        <xsl:variable name = "rname" select = "./reference"/>
        …

        <tr>
        <xsl:choose>
                <xsl:when test="$tname=$tname and $dlevel=$_dlevel">
                        <table border="1" bgcolor="grey">
                        <tr><td>
```

```
                    <!-- Module name →
                    ...
                    </td></tr>
                    <tr><td>
                    <dl>
                        <dt><b>Concept</b></dt>
                        <dd><xsl:value-of select="$cname"/></dd>
                        <dt><b>Procedure</b></dt>
                        <dd><xsl:value-of select="$pname"/></dd>
                        <dt><b>More info</b></dt>
                        <dd><xsl:value-of select="$rname"/></dd>
                    </dl>
                    </td></tr></table>
                </xsl:when>
        </xsl:choose>
        </tr>
</xsl:template>
</xsl:stylesheet>
```

7.2.4.2 BasicUserManualFO.xsl

To produce XSL-FO for formatting user data feed to PDF or other media, the style sheet parses the data feed XML, merges it with TOC contents from the TOC XML file, and translates them to XSL-FO language format.



In the below code, the key sections are highlighted in bold. They include the *variables* section, and a set of templates which translate the topic node content into XSL-FO format.

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
 ...
<!-- variables -->
<xsl:variable name = "_dlevel" select = "'basic'"/>
<xsl:variable name = "xmlfile" select = "document('basic_usermanual.xml')"/>

<xsl:template match="DB">
```

```
        <fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">
        <fo:layout-master-set>

        <!-- first page of chapter -->
        <fo:simple-page-master  master-name="first" page-width="8.5in"  ... >
                ...
        <!-- subsequent chapter pages -->
        <fo:simple-page-master  master-name="subsequent" page-width="8.5in" ...>
                ...
        <!-- sequence master -->
        <fo:page-sequence-master master-name="contents">
          ...
        <xsl:apply-templates select="docs"/>
        </fo:root>
</xsl:template>

<xsl:template match="docs">
        <xsl:apply-templates select="doc"/>
</xsl:template>

<xsl:template match="doc">
  <fo:page-sequence master-reference="contents">
  <!-- put chapter #, title in header -->
  <fo:static-content flow-name="xsl-region-before">
        ...

  <!-- put chp title, page, chp # in footer -->
  <fo:static-content flow-name="xsl-region-after">
        ...
      page-<fo:page-number/>
        ...
  <fo:flow flow-name="xsl-region-body">
    <xsl:apply-templates select="topic"/>
  </fo:flow>
</fo:page-sequence>
</xsl:template>
```

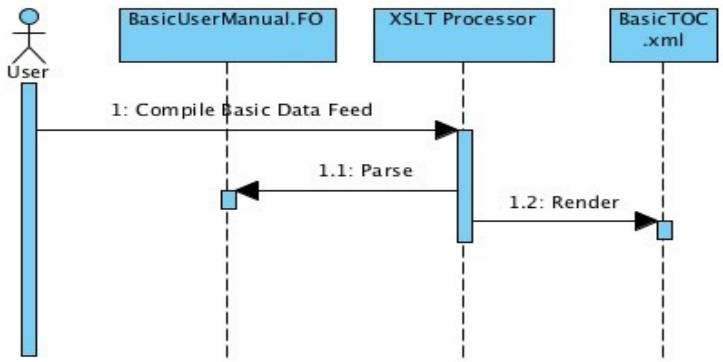
```
<xsl:template match="topic">
        <xsl:variable name = "tname" select = "./name"/>
        <xsl:variable name = "dlevel" select = "./doc_level"/>
        <xsl:variable name = "cname" select = "./concept"/>
        <xsl:variable name = "pname" select = "./procedure"/>
        <xsl:variable name = "rname" select = "./reference"/>
        ...
        <xsl:choose>
                <xsl:when test="$tname=$tname and $dlevel=$_dlevel">

    <!-- output chapter title & # -->
    <fo:block font-weight="bold" font-size="20pt"
        …
      <xsl:value-of select = " $tname"/>
        ...
    <!-- output chapter paragraphs -->
        …
```

7.2.4.3 PDF Rendering

Once the XSL-FO file has been created, it can be fed into an XSL-FO engine to render the PDF file. In this project, we use RenderX as the XSL-FO engine, and we can launch the engine through the command line as follows.

```
/XEP/xep-fo        /Users/thientran/workspace/java/DB/xml/basic_usermanual.fob
```

where xep is the RenderX application, and basic_usermanual.fob is the XSL-FO file.



8 Conclusion and Future Work

8.1 Document Builder

Our Document Builder system prototype has shown how we can create an effective document system by simply modifying the Ant build script, creating the topic map, and putting together some effective style sheets. Style sheets are powerful, and one can use them for many good things.

I hope you have enjoyed reading this paper and have found it helpful. There are many more ideas which we can explore and develop to enhance our Document Builder, some of which are:

- **Administration UI**: To make the system more user friendly, we can create Web pages for users to manage, configure, and run the build process. This will give some benefits for a non-technical person using the system, or s/he can operate remotely via a Web browser.
- **Import/Export for Reuse**: The topic map file (Map.xml) is important; it defines which topic is mapped to which XML document file, as well as the hierarchy build structure between chapters and topics of the user manual. Therefore, it is probably a good idea for the user to pick and choose which sections in an existing Map.xml to import into or export from the system.
- **Search**: We can make use of style sheets, Xpath, and XPointer to provide a Search feature. Based on a given topic name, we can locate the XML document from the topic map file. To search within an XML document, we can use Xpath and Xpointer to parse for some given text or attribute.
- **Email**: At the end of the build process, the user manual will often be sent to the user via email. It's probably a good idea to incorporate email sending as one of the last steps in our Document Builder.
- **Access Control**: A specific Document Build task may be assigned only to specific team members, plus the documents may be sensitive; therefore, it would be good if we could enhance our Document Builder with role-based access control.
- **Secure Distribution**: Since a document may be targeted for a specific audience, we can improve our Document Builder to deliver the user manual more securely. This can be done with PDF and DRM technologies, or some other way could be devised.

8.2 Potential Weaknesses for Future CMS

Topic Maps pave the way to add structural content into CMS, and there will be many tools which will be available for users to add new topic maps easily. However Topic Maps do not put any restrictions on users creating their own content structures, which can promote too much flexibility and potentially introduce new problems, such as different people creating the same or similar topic maps in many places for the same subject, leading to redundancy or even conflicts. When more than one person working on the same field have disagreements, they will tend to create different content and topic maps for the same subject.

The research has shown there is little effort in standardizing between the two areas of Topic Maps and Semantic Web. The growing amount of people participating in these two areas can potentially become a critical issue to the success of Semantic Web. More work will need to be done to make the World Wide Web mature in providing reliable information.

9 References

[1] Mark Shuttlework (2006), Writing a book collaboratively?
http://www.markshuttleworth.com/archives/59

[2] Bill Trippe (2005), Component Content Management in Practice
http://gilbane.com/whitepapers/X-Hive/Xhive_Gilbane_Whitepaper_CCMS6-final.pdf

[3] By Green Is Good (2009), What is Content Management Software - source content management systems
http://hubpages.com/hub/What-is-Content-Management-Software-source-content-management-systems

[4] Dr. Jon Pearce, Document Management Project Topic
http://www.cs.sjsu.edu/faculty/pearce/cs251a/cms/hints.htm

[5] Dr. Jon Pearce, Computer Science Lecture Notes
http://www.cs.sjsu.edu/faculty/pearce/modules/lectures/web/xml/index.htm

[6] James E. Harvey (2002), Topic Maps & The Semantic Web
http://www.media4theworld.com/Papers/Topic_Maps.pdf

[7] Lars Marius Garshol (2004), Topic maps in content management
http://www.ontopia.net/topicmaps/materials/itms.html

[8] David Rugge (2006), Writing Documentation Using DocBook
http://opensource.bureau-cornavin.com/crash-course/en/index.html

[9] Michael Priestley and Amber Swope (2008), DITA Maturity Model
http://na.justsystems.com/files/Whitepaper-DITA_MM.pdf

[10] Apache (2010), Apache Ant 1.8.0 Manual
http://ant.apache.org/manual/index.html

[11] Wikipedia (2010), Apache Ant

http://en.wikipedia.org/wiki/Apache_Ant

[12] ptgmedia.pearsoncmg.com (2008), Macromedia Dreamweaver

http://ptgmedia.pearsoncmg.com/images/0789728958/webresources/0789728958_web04/web04_que_0789728958_02.html

[13] Tom Arah (2009), I'm sorry but Dreamweaver is dying

http://www.pcpro.co.uk/blogs/2009/03/05/dreamweaver-is-dying/

[14] Drupal (2010), Understanding Drupal

http://drupal.org/getting-started/before

[15] CMS Design Resource (2010), Drupal Review

http://www.cmsdesignresource.com/cms-list/drupal/

[16] Apache (2008), Velocity Design

http://velocity.apache.org/engine/releases/velocity-1.4/design.html

[17] Max Andersen (2006),The problems with Velocity

http://blog.hibernate.org/Bloggers/Everyone/Year/2006/Month/02/Day/03

[18] Sun Microsystem (2010), JavaDoc tool

http://java.sun.com/j2se/javadoc/