CS 501- Software Engineering

Legal Data Markup Software
Software Design Document

Version 1.1

Revision History

Date	Version	Description	Author
10/31/2000	1.0	Initial Version.	LDMS Group
11-29-2000	1.1	Revision for Delivery	LDMS Group

Table of Contents

1. Introduction

The intent of this project is to create a software tool that will convert the U.S. Code of Law from its distributed ASCII format into well-formed, valid XML. Our client, the Legal Information Institute, will utilize the XML output in next-generation applications that will make the U.S. Code available in a variety of different formats to the general public. Particular examples of such use include the electronic publication of the code on the Internet and downloadable versions in Folio Views format.

1.1 Purpose

The LDMS is a content-based conversion utility designed to facilitate the distribution of the US code of law. The purpose of the LDMS is to allow the client to fulfill its goal of distributing the US code in formats not feasible with their previous conversion utility. The purpose of this document is to outline and explain the details of the design of the LDMS with respect to the following points:

- Construction of the DTD for the XML output
- Development and execution environments
- Reading and parsing of the formatted ASCII input
- Processing of the formatted ASCII input
- Generation of XML output

1.1.1 Note on this Document version 1.01

Please note that this document reflects the LDMS design as of 10/31/2000. For details on the Program and DTD designs as of 11/29/2000, please refer to the Program Design Document and DTD Design Document. This document should be referred to for information concerning our implementation process such as naming standards, coding standards, and packaging.

1.2 Scope

This document applies only to the LDMS XML conversion utility and any associated maintenance tools we provide. The software design is by no means a comprehensive specification of future functionality that may be added by the developers or by the client. Only those design decisions that are apparent at the time of this writing are included in this particular draft of the software design document; future revisions may include extended or additional design specifications that are added if the current design becomes inflexible and needs to be expanded.

1.3 Definitions, Acronyms and Abbreviations

DDD	DTD Design Document
DTD	Document Type Definition
HTML	Hypertext Markup Language
LDMS	Legal Data Markup Software
Leda	Name of server running development and application environment.
LII	Legal Information Institute
MTBF	Mean Time Between Failures
MTTR	Mean Time to Repair

PDD	Program Design Document
SDD	Software Design Document
SRS	Software Requirements Specification
TOC	Table of Contents
US	United States
W3C	World Wide Web Consortium
XML	Extensible Markup Language
XSL	Extensible Stylesheet Language

1.4 References

The following documents are related to the project and have been consulted:

- Current version of source code for conversion from raw ASCII to HTML.
- http://uscode.house.gov/download.htm – U.S. Code related input formats.
- http://www.lexum.umontreal.ca/fr/equipes/technologie/dtd/LOIQ.dtd – Montreal's DTDs used for legislative purposes.
- http://elj.warwick.ac.uk/jilt/00-2/bruce.html – General background on legislative and legal publishing.
- http://elj.warwick.ac.uk/jilt/00-1/arnold.html – Provides history, motivation, and high-level implementation of EnAct, a project done in Tasmania similar to this one.
- http://nwalsh.com/docs/articles/xml/ – An XML tutorial.
- http://developer.irt.org/script/xml.htm – Archive of frequently asked XML questions.
- http://www.w3.org/TR/REC-xml – The W3C XML draft specification.
- *The Perl CD Bookshelf*, 6 best-selling books on CD-ROM, O'Reilly & Associates, Inc., August 1999.

- *The House of Quality*, Hauser, J.R. and D. Clausing, *Harvard Business Review*, May-June 1988, pp. 63-73.
- *Writing Quality Requirements*, Wiegers, K. E., *Software Development*, May 1999.

1.5 Overview

The rest of this document provides a comprehensive specification of all the software design decisions for the project. These specifications include the explanation of the construction of the DTD for the XML output; development and execution environments; reading, parsing, and processing of ASCII input; generation of XML output; and the top level components and modules design. In summary, the specifications outlined by this document will provide a sufficient description of the design that will be used to create the final product.

1.6 Roles and Responsibilities

Name	Department	Responsibility
Thomas Bruce	Legal Information Institute	Project Sponsor
William Arms	Computer Science Department	Project Sponsor
Amy Siu	Computer Science Department	Project Reviewer
Ju Joh	Computer Science Department	Student Developer
Sylvia Kwakye	Computer Science Department	Student Developer
Jason Lee	Computer Science Department	Student Developer
Nidhi Loyalka	Computer Science Department	Student Developer
Omar Mehmood	Computer Science Department	Student Developer
Charles Shagong	Computer Science Department	Student Developer
Brian Williams	Computer Science Department	Student Developer

2. Development and Execution Environment

2.1 Development Environment

2.1.1 Hardware

The primary hardware that we will be using in the development environment is a machine with a 233 MHz Intel Pentium II processor, 128 MB memory, and a 28 GB hard disk. Additionally, we will be using 4 Dell Latitude CPt notebook computers to connect to the primary hardware during our coding sessions, each consisting of a 400 MHz Intel Celeron processor, 96 MB memory, a 4.7 GB hard disk, and an Aironet PC4800 Wireless PCMCIA card.

2.1.2 Software

The software that we will be using in the development environment consists of the following:

- Red Hat Linux 6.2, the operating system.
- Perl 5.6, the coding language.
- SSH Secure Shell 2.3, the remote connection software.
- CVS 1.10.7, the version control software.
- Emacs 20.5.1, a text editor.
- VIM 5.6, a text editor.

2.2 Execution Environment

2.2.1 Hardware

The hardware that will be used in the execution environment is the same hardware that we will be using in the development environment. This fact alleviates almost any potential hardware compatibility threats. Even though the LDMS will presumably function normally after a successful hardware upgrade, the client knows that any hardware upgrade is at his own risk and the software is not guaranteed to work under the new conditions.

2.2.2 Software

The software that will be used in the execution environment consists of the following:

- Red Hat Linux 6.2
- Perl 5.6

Even though the LDMS will presumably function normally after a successful software upgrade of the Linux OS and/or Perl Compiler/Interpreter, the client knows that any software upgrade is at his own risk and the software is not guaranteed to work under the new conditions.

3. Naming Standards

The LDMS team will use the following naming standards for the design and implementation of the software:

- File Name Styles: All file names will begin with a single word in all lower case followed by any number of words with the first letter of each word in upper case followed by a "." and up to three characters (e.g. thePerlFile.pl)
- File Name Length: All files names will be at most 20 characters in length not including the "." and up to three character extension
- Function Names: All function names will begin with a single verb in all lower case followed by any number of words with the first letter of each word in upper case (e.g. initializeErrorModule)
- Variable Names: All variable names will begin with the appropriate qualifier followed by a single word in all lower case followed by any number of words with the first letter of each word in upper case (e.g. $variableOneName). Additionally, local storage variable names will begin with the appropriate qualifier and the name of the module that "owns" the storage variable followed by an underscore and then the standard variable name production (e.g. $error_LastErrorMessage).
- Filehandle Names: All filehandle names will be in all capital letters
- XML Output File Names: The output file names will begin with the first part of the input file name before the first "." followed by ".xml"
- DTD Element Names: All DTD element names will be in all capital letters. All DTD tag names for divisions within other elements will begin with "DIV" followed by the name of the element.

- Version Numbers: Each subsequent prototype of the system will be assigned a version number equal to an increment of 0.1 to the previous version number. Since we are not planning to develop more than 10 prototypes, the system will be set at version 1.0 after the completion of the project. No other major version number changes are planned, e.g. 1.0 to 2.0.

4. Coding Standards

The LDMS team will use the following coding standards for the design and implementation of the software:

- Functions shall not exceed 100 lines.
- Each function shall have a preceding comments section describing the function's precondition, postcondition, and purpose.
- Each variable shall have a comment describing the variable's purpose.
- Each loop shall have begin and end comments.
- A consistent indentation of 3 spaces shall be used for each block.
- All variables that are not in the interface specification shall be declared locally.
- Perl contractions shall not be used.
- Each file shall have a modification history log showing the version number, date, user id, and description of each change.
- Each file shall include a copyright and license notice at the beginning of the file.

5. Software Design (Please refer to the PDD for latest design)

5.1 Overview

The software design consists of three main components: Read and Parse File, Language Parsing, and Output. The raw ASCII file contains the 50 titles provided by the House of Representatives and will be fed as input to the system. The whole file will be read by the Read and Parse File component with the help of its modules. The input module will read either lines of text or blocks of text at a time. As the lines are being read, the StateMachine module provides us state information as to which part of the document we are processing, i.e. which title we are currently at, if we are encountering white spaces or special characters that need to be parsed, external/internal references etc. The WhiteSpacePatternMatching and the WordPatternMatching take care of inconsistencies in white space and tabs, and special characters. In the event of any error, the error is outputted through modules of StoreAndOutputErrors component and stored in an error file. The modules of the Status component periodically display the status, as to which lines of the document have been processed. Finally, a well-formed and valid XML document is created.

5.2 Architecture

The software architecture diagrams summarize the important physical groupings of code of that make up the LDMS.

5.2.1 System Architectural Components

The “Program” represents the top of the system architecture. The LDMS has three major components: Read and Parse File, Language Parsing, and Output. Each of these components has several modules that perform the tasks described by the component.

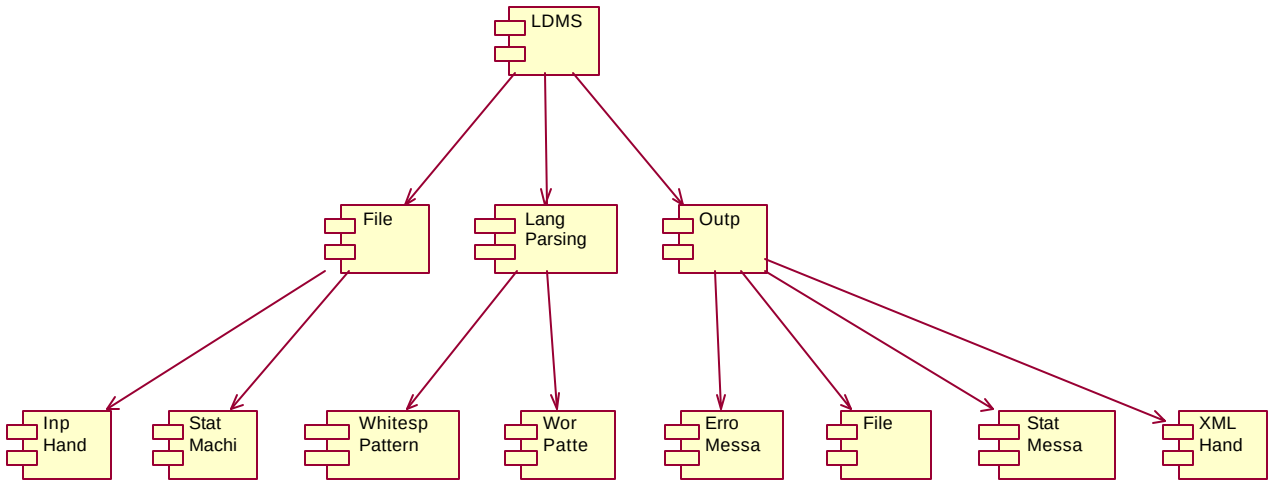


Figure 5.1: Top-level UML component diagram.

5.2.2 Modules in Components

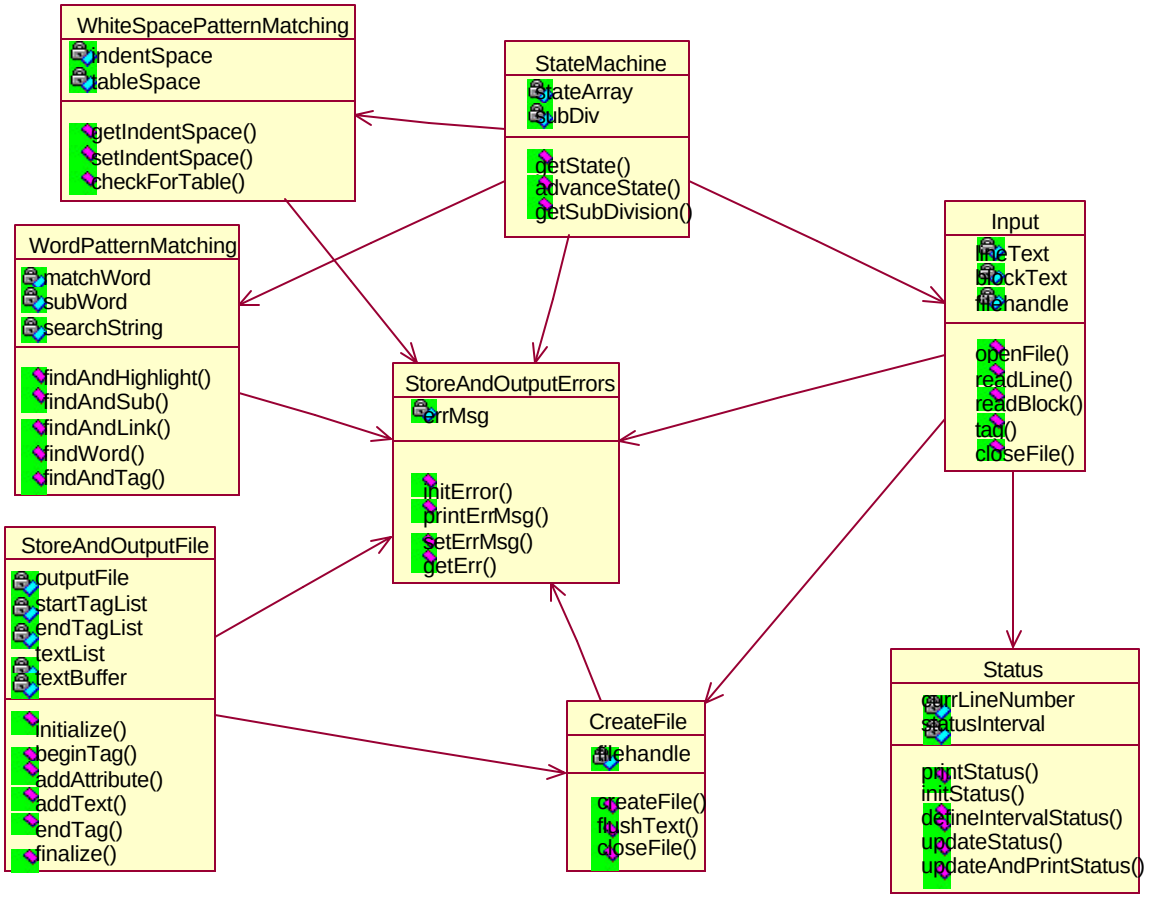


Figure 5.2: UML Class Diagram.

Module Descriptions:

- The **CreateFile** module creates the corresponding XML file for each title processed from the raw ASCII input file. Filehandle points to the current XML file being prepared. CreateFile() creates the XML file, flushText() flushes the processed ASCII text stream into the newly created XML file and closeFile() closes the XML file once the whole file is created.

- The **Input** module supports the reading of the raw ASCII document provided by the House of Representatives. It can either read text line by line in the variable lineText, or read an entire block defined by catchlines stored in the variable blockText, e.g. all lines of text between –CITE– and the line immediately before the next –CITE– would form one block. The variable filehandle is a reference to the current file being read. The function openFile() opens the raw ASCII file for reading. ReadLine() reads lines of text until the end of the document while readBlock() reads blocks of text. Tag() replaces the catchlines with their equivalent DTD tags defined in the DTD document. Finally, closeFile() closes the file when the whole file has been read.
- The **StateMachine** module helps us to know where we currently are in the raw ASCII document. For example, it can tell us whether we are at the beginning of a title, chapter, section, or subsection, what the number of the particular division is, whether we are referring to a footnote or an external/internal reference, etc. Basically, it helps us to perform functions such as white space and word pattern matching, tagging, etc.
- The **Status** module provides a periodic display of the current line of the input file being processed. It contains functions to increment and print the current processing line number as well as define the status display interval.
- The **WhiteSpacePatternMatching** module has 2 attributes, indentSpace and tableSpace. IndentSpace is the number of spaces a line is from the edge while tableSpace refers to the spacing that is used to denote tables in the ASCII text. Functions are provided to get or set the indentSpace. Another function allows one

to check for tables.

- The **StoreAndOutputErrors** module has one attribute, errMsg. ErrMsg is the last error message that was set by setErr() and can be retrieved using getErr(). InitErr() initializes the error module for output to the standard error stream and printErrMsg() writes a message to this stream.
- The attributes for the **WordPatternMatching** module are: matchWord, which is the word we are searching for; subWord, which is the word we can substitute the matchWord for; and searchString, which is the text we are searching through. The functions are straightforward. One can find a word and substitute for it, find and link it to something, find and highlight it, find and tag it, or just find it.
- In the **StoreAndOutputFile** module**,** the attribute startTagList is a list of lists, each of which contains a start tag with its associated attributes, in the order that they will occur in the output. EndTagList is a list containing the end tags in the order they will occur in the output and the number of start tags preceding this end tag. TextList is a list containing the strings that are the text between tags. TextBuffer is a string that collects all the text between the last and the next tag. The function initialize() reads in the output file name (outputFile) and initializes the three lists and the buffer as empty. BeginTag() adds a start tag to the startTagList, putting the textBuffer into the textList. AddAttribute() adds an attribute to the most recent unclosed start tag. EndTag() adds an end tag to the endTagList with the number of start tags (for proper ordering in the output), putting the textBuffer into the textList. Finalize() creates the outputFile and the Perl XMLWriter module uses the three stacks to write to the output file.

5.3 Design Constraints

Our design will be based on the following design constraints and assumptions:

- The U.S. Code input file will be in ASCII format. We note that the division of titles into chapters, sections, and sub-sections, white spaces, special characters, footnotes, references, and tables, may differ significantly across each title.
- The software will be designed to facilitate the unattended operation of the program. This will have implications in error handling. For example, the issues of automatic decision making to re-start execution of the program in case of a server crash and steps taken by the system to handle graceful failures without user intervention, will be addressed in the design.

5.4 Global Data Objects

We will not be using any global data objects other than the input file name. The output file names will not be hard coded as global variables as is common practice, but rather will be determined by the name of the input file with the ".xml" extension appended to the file name.

5.5 Error Handling

Error handling is managed by the main program loop and is stored and output using the StoreAndOutputErrors module (refer to 5.2.1 and 5.2.2).

The following are errors that may be encountered and their associated solutions:

- Improper program call: If the program is executed with unknown parameters or extraneous information, the program will exit and print the usage parameters to the console.
- Output file already exists: If the overwrite flag was set during the program call, then the existing file will be overwritten. If the flag was not set, then the program will write an appropriate error message to the standard error stream and exit the program.
- System errors: If a system error such as "disk is full" is experienced, the program will log the error message to the standard error stream and exit the program.
- Non-critical errors: Any non-critical errors in processing the input data will be handled gracefully by using a tag similar to the HTML <PRE> tag.

5.6 Development Language

The software will be developed using Perl 5.6.

5.7 Preconditions and Postconditions

The following preconditions must be met prior to starting the application:

- The input file(s) must be available in a particular path
- The execution environment including the hardware and software must be setup.
- The required memory and disk space as previously stated in the requirements

document must be available.

The following post conditions shall be guaranteed by the system:

- A well-formed, valid XML document with respect to the DTD we designed will be produced. It will correspond to the input file.

5.8 User Interfaces

5.8.1 Introduction

Since the goal of the system is simply to convert the raw ASCII input files into a well-formed, valid XML document conforming to the DTD we designed, there will be no sophisticated user interactivity or interfaces. The program will use user-specified command line arguments such as the name of the input file to be processed as well as other common parameters. A batch processing approach will be adopted whereby the user inputs all the files that are to be converted and the program will process these files together.

5.8.2 User Interface

The inputs to the program will be entered as command line arguments.

5.8.3 Menu Design

The following parameters will be used to employ particular commands:

- -O <filename>: Output file name

- -F: Overwrite existing file
- -V: Verbose error messages
- -L#: Status message frequency
- -?: Help

In addition, the standard error stream can be redirected to a file for later viewing. The form for the program execution is the program name followed by zero or more command line parameters (with their associated file name, if necessary).

5.8.4 Data Screen Design

Since there will be no sophisticated user interface, the data screen design is not necessary.

5.8.5 Report Formats

The following output will be presented to the user:

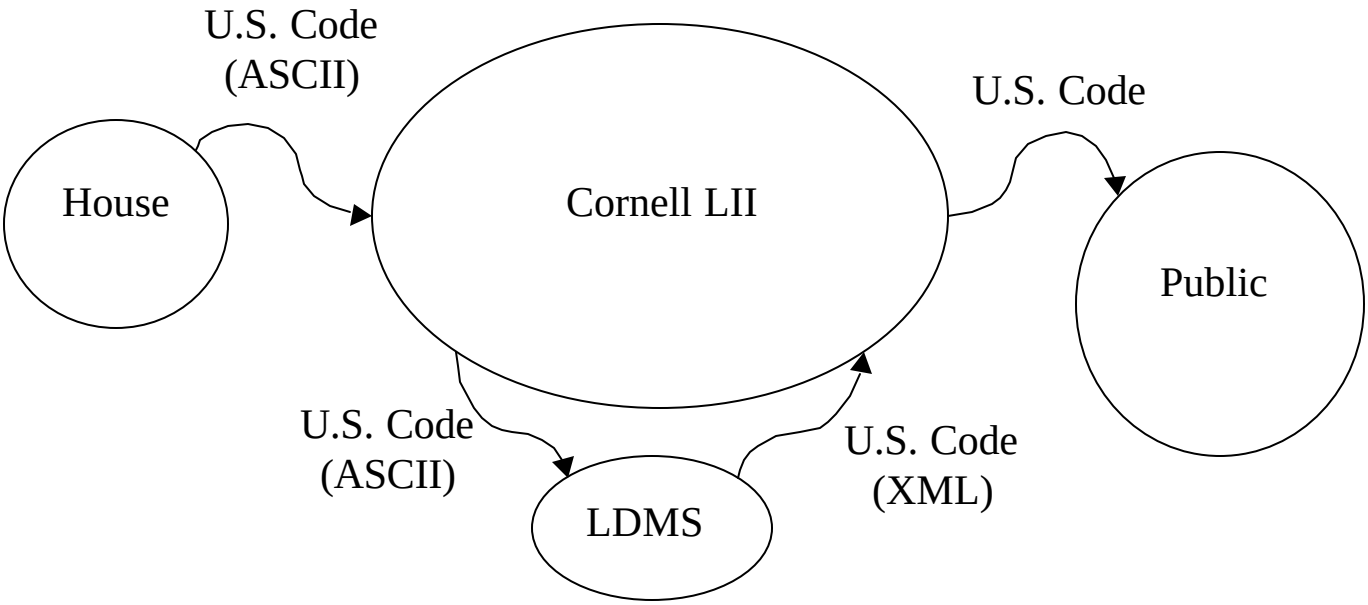
- The program will display simple status messages after a specified number of lines have been processed. The frequency of these messages is set by using the –L command line parameter and if none is specified, then no messages will be output to the screen.
- The program will output the error messages to the standard error stream in the following format:

<date> <time> <input filename> <user id> <line number> <error message>

5.9 Supporting Diagrams

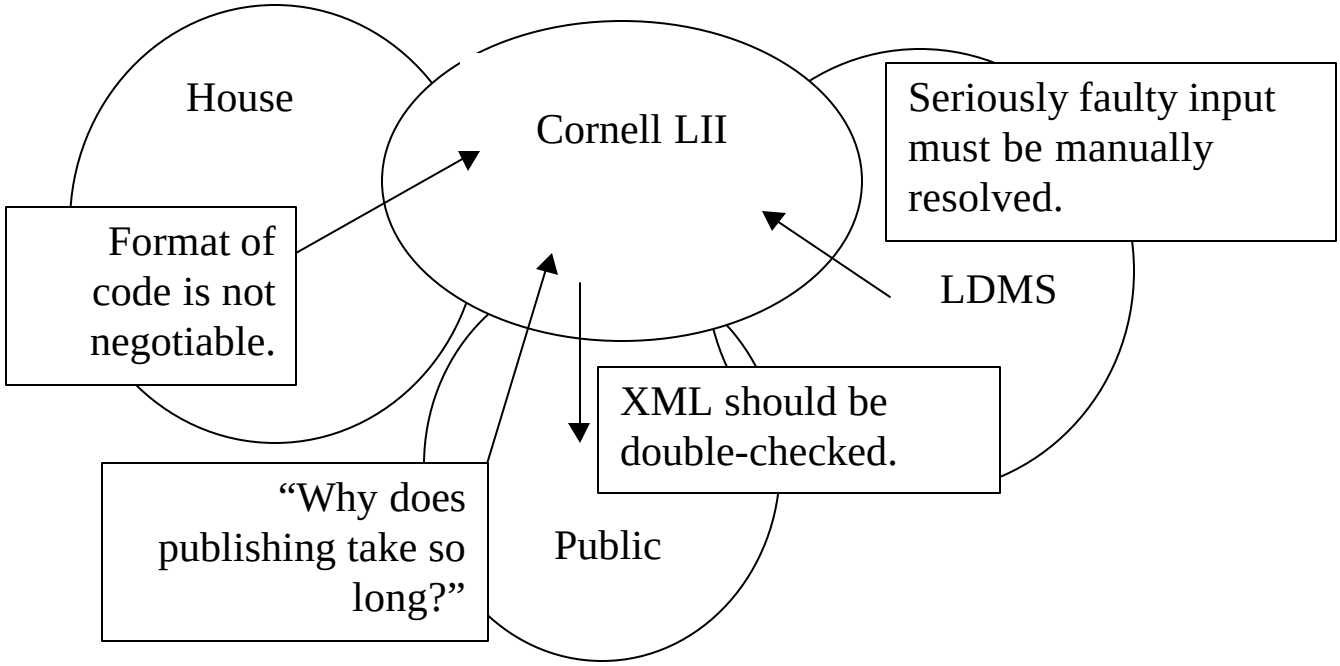
5.9.1 Flow Diagram

This diagram demonstrates how the LDMS fits into the client system. The LII will be the sole user in the current model. The US Code in ASCII format must be downloaded by LII and used as input to the LDMS. The public can access the US code in XML format only through the LII.



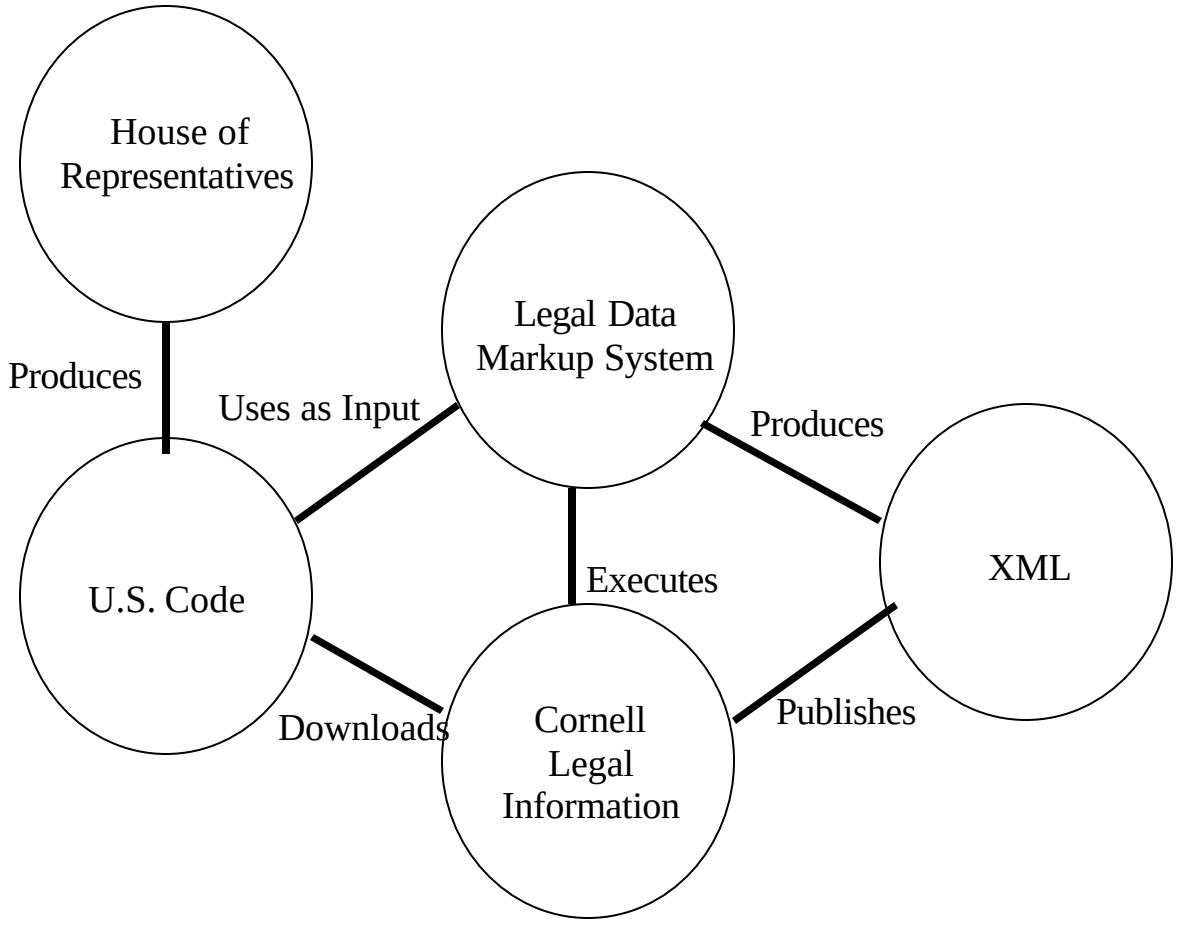
5.9.2 Culture Diagram

This diagram shows particular cultural and situational design caveats. The LII and LDMS have little control over the input due to the fact that the House of Representatives controls the format and availability of the US Code. The lag in the cultural diagram is between the LII and Public and not between the LII and LDMS because the source of the lag stems from the House of Representatives and their several month delay in transcribing the US Code in paper to an ASCII formatted text.



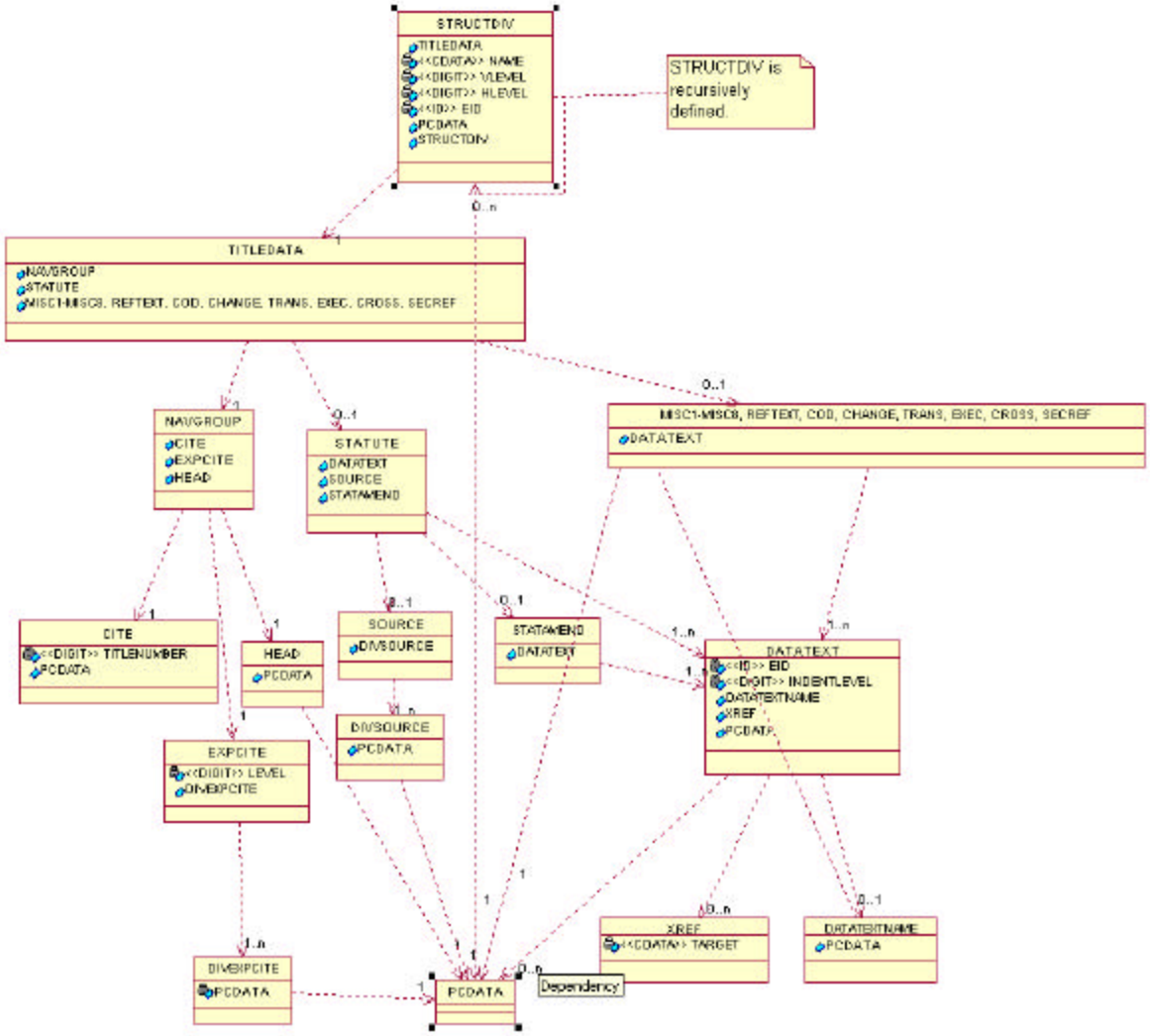
5.9.3 Context Diagram

This diagram represents the behavioral relationship between the various objects, systems, and the LDMS.



6. DTD Design (Please Refer to the DDD for latest Design)

6.1 Conceptual Schema



In the above class diagram, each element in the XML DTD is shown as a class. The attribute list of each class is shown as protected attributes, while elements in each class are shown as public attributes with appropriate dependencies. Each dependency also includes the cardinality requirement.

6.2 Tag Descriptions

- XREF is the element that will mark the cross-references.
    - TARGET is an attribute that contains the EID of the cross-referenced material.
        - Originally ID-Ref was considered as the type for TARGET, but the ID-Ref requires that the ID being referenced is present in the current XML document. Therefore it was too limiting for our cross-title (cross-XML file) referencing purposes.
- DATATEXT is a generic tag designed to be the US Code's equivalent to XML PCDATA.
- STRUCTDIV is a generic tag designed to mark the structural divisions of the US Code, e.g. Titles, Chapters, Subchapters, Sections, etc.
    - NAME is the label name of the division, e.g. "Chapter", "Section", etc.
    - VLEVEL is the depth of the division. For an example if the Table of Contents is organized by Title, Chapter, Subchapter, and Section, then Title will have VLEVEL=0, Chapter=1, Subchapter=2, Section=4.
    - HLEVEL is the sequential order number of the division. For an example, the first chapter in a title will have HLEVEL=0, second chapter will have HLEVEL=1, etc.
    - EID is a globally unique string identifier for the structural division. These will be the values to be referenced from XREF::TARGET's. It is of type ID.

- TITLEDATA is a group of dashline-tag sequences. Dashlines, i.e. tags found in the ASCII formatted US Code, delimit particular portions of the document and all portions of any document are delimited by these dashlines. There are numerous such portions in any given title, each portion forming its own particular mini-group. TITLEDATA is used to preserve that relationship.
    - NAVGROUP is an element that encapsulates the dashline tags that will be used for navigational purposes. The tags in NAVGROUP do not contain any legally useful data, but they are a sort of meta-tag that identifies the subsequent legal data.
        - CITE contains the data from dashline tag –CITE–. It contains the label portion of the catchline, i.e. the label "Chapter," "Section," etc., and the number for the label, e.g. "Section 2b" that labels the current TITLEDATA portion. In addition, it contains the current title number and the identifier USC (for US Code).
        - EXPCITE is an expanded CITE. It notes the hierarchy of catchlines, up to but not including the bottom most level of the TOC.
            - DIVEXPCITE marks the text of each level within the hierarchy.
        - HEAD contains similar information as CITE but with greater local scope. It also contains the name for the current TOC section.
- STATUTE marks up the actual legal data, i.e. the actual words of the law.

- STATAMEND marks up the amendments made to a statute, and therefore can only be found inside a STATUTE tag.
- SOURCE marks the various sources relevant to the current STATUTE.
  - DIVSOURCE marks an individual source with the dashline tag –SOURCE–. These are delimited by semi-colons in the input.

- CROSS contains particularly worthy cross-references relevant to the TITLEDATA section.
- SECREF contains the various other sections that reference the current section.
- MISC1 through MISC8 are simply additional information regarding the particular TITLEDATA portion. The differences between the tags are simply their relative location to other tags.
- REFTEXT, COD, CHANGE, TRANS, EXEC contain further information regarding the TITLEDATA portion. REFTEXT contains further information regarding references previously mentioned. CHANGE notes various changes, e.g. change of name for an organization previously mentioned. TRANS contains the transitive information for the particular portion.

6.3 Document Type Definition (DDD contains latest DTD)

```
<!-- XML DTD LII Ver. 0.51-->

<!--XREF is the tag for cross-reference-->
<!ELEMENT XREF (#PCDATA)>
  <!ATTLIST XREF
    TARGET CDATA #REQUIRED>

<!--DATATEXT is the generic tag for the texts-->
```

```
<!--DATATEXT may include various other elements that can be found within the text,
e.g. crossreferences, tables, additional DATATEXT's etc.-->
<!--DATATEXTNAME marks the preceding header-esque line-->
<!ELEMENT DATATEXT (DATATEXTNAME?, (XREF|#PCDATA)+ )>
  <!--INDENTLEVEL is the count of indentation depth, i.e. how many levels of white-
space delimited texts there exists at the current DATATEXT.-->
  <!ATTLIST DATATEXT
    INDENTLEVEL DIGIT #IMPLIED
    EID ID #IMPLIED>

<!--STRUCTDIV is the generic structural divison tag.-->
<!ELEMENT STRUCTDIV (#PCDATA, TITLEDATA, STRUCTDIV*)>
  <!--NAME denotes the label of the division, e.g. Title, Subtitle, Chapter, etc.-->
  <!--VLEVEL denotes the vertical depth of the tag. Starts at 0.-->
  <!--HLEVEL denotes the sequencial (horizontal) order of the tag.  Starts at 0.-->
  <!ATTLIST STRUCTDIV
    NAME CDATA #REQUIRED
    VLEVEL DIGIT #REQUIRED
    HLEVEL DIGIT #REQUIRED
    EID ID #REQUIRED>


<!--TITLEDATA contains the sequence of ordered dashline tags. Within each title there
will be several sets of TITLEDATA-->
<!ELEMENT TITLEDATA (NAVGROUP, STATUTE?, MISC1?, REFTEXT?,
MISC2?, COD?, MISC3?, CHANGE?, MISC4?, TRANS?, MISC5?, EXEC?, MISC6?,
CROSS?, MISC7?, SECREF?, MISC8?)>

<!--NAVGROUP contains the navigational dashline information of the title, i.e. the
current section within the table of content.  It is required in the beginning of each
TITLEDATA-->
<!ELEMENT NAVGROUP (CITE, EXPCITE, HEAD)>
  <!ELEMENT CITE (#PCDATA)>
  <!--TITLENUMBER is the current title number-->
    <!ATTLIST CITE
      TITLENUMBER DIGIT #REQUIRED>

    <!ELEMENT EXPCITE (DIVEXPCITE+)>
    <!ATTLIST EXPCITE
      LEVEL DIGIT #IMPLIED>
    <!--DIVEXPCITE is a divider within EXPCITE.  Each DIVEXPCITE corresponds to
a catchline.  It is divided by specific pattern, e.g.   TITLE 27 --->
  <!ELEMENT DIVEXPCITE (#PCDATA)>

  <!ELEMENT HEAD (#PCDATA)>
```

```
<!--STATUTE dashline contains the actual law text.  SOURCE and STATAMEND
dashlines must be matched to a statute, i.e. included within the STATUTE element.-->
<!ELEMENT STATUTE (DATATEXT+, SOURCE?, STATAMEND?)>
  <!ELEMENT SOURCE (DIVSOURCE+)>
  <!--DIVSOURCE is separated by semicolon-->
    <!ELEMENT DIVSOURCE (#PCDATA)>

  <!ELEMENT STATAMEND (DATATEXT+)>

<!--MISC1 through MISC8 are identical texts, except in terms of physical location. e.g.
MISC1 is found between REFTEXT and STATAMEND, etc.-->
<!ELEMENT MISC1 (DATATEXT+)>
<!ELEMENT REFTEXT (DATATEXT+)>

<!ELEMENT MISC2 (DATATEXT+)>
<!ELEMENT COD (DATATEXT+)>

<!ELEMENT MISC3 (DATATEXT+)>
<!ELEMENT CHANGE (DATATEXT+)>

<!ELEMENT MISC4 (DATATEXT+)>
<!ELEMENT TRANS (DATATEXT+)>

<!ELEMENT MISC5 (DATATEXT+)>
<!ELEMENT EXEC (DATATEXT+)>

<!ELEMENT MISC6 (DATATEXT+)>

<!--CROSS dashlines denote the various crossreferences from this portion of the
document to other titles and/or sections.-->
<!ELEMENT CROSS (DATATEXT+)>

<!--SECREF lists the various sections that have this section listed in their crossreference-
->
<!ELEMENT MISC7 (DATATEXT+)>
<!ELEMENT SECREF (DATATEXT+)>

<!ELEMENT MISC8 (DATATEXT+)>
```

7. Packaging

7.1 Documentation

7.1.1 Source Level Documentation

It is deemed vitally important for development and maintenance that all source code is thoroughly documented, especially when code segments are written with specific functional requirements in mind. No code shall be approved for inclusion in builds without accompanying source level documentation and peer review of such documentation. This portion of the documentation will be included as comments in the source files, and separate text files in the source code directory tree.

7.1.2 Program Design Document

Development of the LDMS shall be documented by a program design document (PDD) outlining the implementation. It shall be the central reference for developers responsible for understanding, maintaining, and extending the LDMS. The PDD shall contain a high level view of the LDMS processing engine, detail individual processing components, and display all interfaces, within and external to, the system. To aid in supporting the LDMS, no development diverging from the requirements shall occur without peer approval, without modifying requirements, nor without modifying the PDD.

7.1.3 DTD Design Document

Development of an appropriate DTD shall be documented by a DTD design document (DDD). It shall be the sole reference for developers responsible for understanding,

maintaining, and extending the DTD. The DDD shall contain a list of all elements and element attributes with details of their use. To aid in supporting the LDMS, no modifications to the DTD shall occur without peer approval, or without modifying the DDD.

This portion of the documentation will be included as comments in the DTD file, and within this design document.

7.2 Source Code

- None of the source code for the various prototypes will be delivered.
- None of the code for testing purposes will be delivered.
- All of the source code for the final version will be provided.

7.3 Executables

There will be a single executable script file that will be used to start the program.

There will be no other executables in the project.

7.4 Data Files

Document Type Definition (DTD) file will be delivered. This file shall contain the definitions for tags used by the program.

7.5 Installation

Simply copy the project directory into the desired location.

The relative paths within the directory must remain identical.

Also Perl interpreter must have been installed in the system, and the correct path to the interpreter binary must be specified in the head of the executable script.

The process is simple and the likely hood, of additional installations beyond the first, is low; therefore, there will be no additional installation tools.

8. Supporting Information

8.1 Description of Responsibilities

Each developer was charged with the responsibility to oversee a major area of the delivery of this software design document.

Omar Mehmood is responsible for working on the DTD design and high level UML component and module design, writing sections 1-4, 5.5, 5.6, 5.8.3, 5.8.4, 5.8.5, 5.9, and 8 of the SDD, creating the UML diagram for StoreAndOutputErrors, revising the UML class diagram descriptions, and compiling and editing the entire SDD document.

Ju Joh worked on the DTD design, wrote sections 6 and 7 of the SDD, and created the DTD Class diagram, the Context Diagram, the Culture Diagram, and the Flow Diagram.

Sylvia Kwakye is responsible for merging other software design documents with our requirements template to create our SDD template. She also worked on the DTD design, edited the entire SDD document, created UML diagrams for WhiteSpacePatternMatching and WordPatternMatching, and prepared the slides for sections 1-5.2.

Brian Williams is primarily responsible for co-presenting the SDD to the client and the professor. His is also responsible for working on the DTD design, creating the UML diagram and description for StoreAndOutputFile, and presenting our SDD to the clients.

Charles Shagong is principally responsible for co-presenting the SDD to the client and the professor. Additionally, he co-designed the UML class diagram for the StateMachine and created the Status module, defined the relationships between the modules in the class diagram, edited the SDD, and provided input to the DTD design.

Nidhi Loyalka is primarily responsible for writing sections 5.1, 5.3, 5.4, 5.7, 5.8.1, and 5.8.2 of the SDD. She also drew the UML diagram for the Input module, co-created the UML diagram for the StateMachine module, and helped define the relationships between the modules.

Jason Lee is primarily responsible for creating the slides for sections 5.3 to the end of SDD except for section 5.9, which was a collaboration between he and Sylvia Kwakye. He also assisted in the development of the DTD design and created the overall UML component diagram.