CS 501- Software Engineering

Legal Data Markup Software
Program Design Document

Version 1.1

Document Revision History

Date	Version	Description	Author
11/26/00	1.0	Draft for Delivery	LDMS Team
11/30/00	1.1	Added final function prototypes.	LDMS Team

Table of Contents

Program Design Document

1. Introduction

The intent of this project is to create a software tool that will convert the US Code of law from its distribution ASCII format into well-formed, valid XML. The XML output would subsequently be utilized by our client, the Legal Information Institute, in next-generation applications that will make the U.S. Code available in a variety of different formats to the general public. Examples of such use include the electronic publication of the code on the Internet and downloadable versions in Folio Views format.

1.1 Purpose

This purpose of this document is to elucidate several design and implementation points with respect to the finished product. The following are questions which are best addressed by this document:

- Why Perl?
- Why is the source split up into modules?
- What does the LDMS::XMLOut module do?

1.2 Scope

This document applies only to the main script file (LDMS.pl) and the modules in the LDMS namespace.

1.3 Definitions, Acronyms and Abbreviations

DTD	Document Type Definition
HTML	Hypertext Markup Language

LDMS	Legal Data Markup Software
Leda	Name of server running development environment.
LLI	Legal Information Institute
US	United States
W3C	World Wide Web Consortium
XML	Extensible Markup Language
XSL	Extensible Stylesheet Language
PDD	Program Design Document
DDD	DTD Design Document

1.4 References

Developers may find the following documents useful:

- http://uscode.house.gov/download.htm – U.S. Code related input formats.
- http://www.w3.org/TR/REC-xml – The W3C XML 1.0 Draft Specification (2nd Edition).

1.5 Overview

This document is aimed primarily at developers working directly on the LDMS source code. To that end, it shall document the functions provided by each module, and will provide a high-level overview of the structure of the LDMS source code. However, there is, perhaps, a secondary audience: developers who wish to attempt something similar on different data, or using different tools. For them, this document points out reasons behind specific design and implementation decisions, and in some cases, explains why alternative approaches were discarded. This document makes very little use of actual code examples. For those, developers are advised to examine the source code directly, as

there are plenty of comments there.

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. The Problem

The LII wishes to be able to convert the existing US Code from the House of Representatives into XML. However, there are several crucial requirements for a system that would do the conversion:

- Accurate translation.
- Hooks for functionality present in current HTML version.
- Maintainable code.
- Graceful failure.

These four requirements, above all, have proven to be the driving factors behind many of the decisions made during the design and implementation phases of the project.

3. Commentary on Design and Implementation

3.1 Design

Below are some explanations of major decisions made during the design phase.

3.1.1 Development Language

Why Perl? Perl has wonderful support for pattern matching via regular expressions, as well as support for strings as primitives. This makes it the *de facto* language of choice for extended text processing.

3.1.2 Design Model

We used the Iterative Model for our design because the cues that our system would have to use to determine how to tag up certain sections are variable across the entire US Code. This means investing a lot of time in trial and error approaches in order to bring the recognition up to levels consistent with the requirements. Obviously, a strict Waterfall Model would not afford us the flexibility required to adapt the system to the data.

3.1.3 Modularity

There were several factors that figured into our decision to make the system modular. One is that dividing the script up into modules allows accountability: if a certain part of the system fails, it is relatively easy to trace it back to a specific module (and/or developer). Another is that the system can be upgraded in parts—if someone comes up with a better way of recognizing cross-references, it can be written into a new module that uses the interface of the old module.

3.2 Implementation

Below are some explanations of decisions made during the implementation phase, including development decisions.

3.2.1 Version Control

One might argue that since the program was designed to be modular, there would be no need for version control, as long as each module had a single developer assigned to it. However, due to the short schedule, it was not feasible to stop all development while modules were being integrated into the system proper: during integration, individual modules were still under continual development. This meant that changes required to integrate each module with the system had to be merged with individual developer changes. For our purposes, CVS proved to be equal to the task at hand, as it allows changes made concurrently to be merged into a single revision.

4. Modules

4.1 LDMS::DataText

This module processes the <DATATEXT> and <DIVDATATEXT> divisions within each title.

- sub initDataText(@) – Initializes DataText module with a DATATEXT block.
- sub getDataTextBody – Gets the next body of the current DATATEXT.
- sub getDataTextName – Gets the next DATATEXTNAME of the current DATATEXT.
- sub getDataTextLines – Gets all lines remaining in the current DATATEXT block.

- sub processDataText(@) – One-stop processing for DATATEXT blocks.
- sub processDivDataText(@) – Internal processor for processing individual DIVDATATEXT blocks.
- sub startDataText($$) – Begins a DATATEXT block.
- sub endDataText() – Ends a DATATEXT block.

4.2 LDMS::Date

This module produces a timestamp for each title, based on when the document was released by the House of Representatives.

- sub tagDate – Tags up the date from the -CITE- field.

4.3 LDMS::DivSource

This module divides the <SOURCE> field up into a list of sources, assuming each source is delimited by a semicolon.

- sub tagDivSource(@) – Tags up any SOURCE divisions in an array of lines passed into it.

4.4 LDMS::Error

This module handles error messages for the LDMS system.

- sub initError() – Initialize the error message handler to default values.
- sub printErrMsg($arg1_ErrMsg) – Prints the error message passed in.
- sub setErr($arg1_ErrorCode) – Sets the current error code.
- sub getErr() – Returns the current error code.

4.5 LDMS::Footnote

This module tags up actual footnotes in the text.

- sub tagFootnotes(@) – Accepts a block of text and processes any footnotes contained within.
- sub processFootnote() – Processes individual footnotes in the current block of text.

4.6 LDMS::FootRef

- sub markupFootRefs(@) – Accepts an array of lines and marks up any references to footnotes contained within.

4.7 LDMS::Input

This module opens the raw ASCII file from the House of Representatives and splits it up into blocks, delimited by the '-CITE-' field tag.

- sub openFile – Opens a file for reading.
- sub readLine – Reads a single line from the input file.
- sub readBlock – Reads an entire CITE block from the input file.
- sub closeFile – Closes the input file.
- sub newFileName – Replaces a file's extension with ".xml".

4.8 LDMS::LP

This module performs lexical parsing on the text.

- sub initCite(@) – Initializes the module with a block of CITE text.
- sub getCiteTitleNumber(@) – Returns the title number from the current block.

- sub initExpcite(@) – Initializes the EXPCITE parser.
- sub getNextExpciteEntry() – Returns the next EXPCITE entry.
- sub getExpciteLevel() – Returns the number of EXPCITE levels.

4.9 LDMS::Pre

This module implements a <PRE> tag, used to tag text which is, for any reason, unable to be processed by normal means.

- sub tagPre() – Tags up unrecognized text.

4.10 LDMS::State

This module implements a state machine for the LDMS system. This is used to determine the hierarchical state of the document at any given time.

- sub initState() – Initializes the state machine to default values.
- sub ascendLevel() – Ascends one level of hierarchy.
- sub descendLevel($$) – Descends into a new level of hierarchy.
- sub getHLevel() – Returns the sequential position of the current structural division.
- sub getVLevel() – Returns the hierarchical position (depth) of the current structural division.
- sub getLevelLabel() – Returns the hierarchical label of the current structural division.
- sub getSequence() – Returns the extended sequential position of the current structural division.

- sub getSequenceLabel() – Returns the sequential label (I, ii, 3, D, &c.) of the current structural division.
- sub setSequenceLabel($) – Sets the sequential label of the current structural division.
- sub advanceLevel() – Increments the sequence level of the current structural division. Does not update the sequence label.
- sub labelInPath($) – Checks to see if a certain hierarchical label has already been used.
- sub initFootnoteNumbers() – Sets the footnote mini-state machine to default values.
- sub incrFootnoteNumber($) – Increments the current footnote number.
- sub getFootnoteNumber($) – Returns the current footnote number.

4.11 LDMS::Status

This module implements status messages for the LDMS system.

- sub initStatus($) – Initializes the module to default values.
- sub defineIntervalStatus($) – Sets the status interval, in lines….
- sub updateStatus() – Updates the current status message.
- sub updateAndPrintStatus() – Prints the current status message and updates it.
- sub printStatus() – Just prints the current status message.
- sub printProcessingLevel() – Prints the current level that is being processed.

4.12 LDMS::Table

This module handles all tabular data passed into the LDMS system.

- sub markUpTable(@) – Marks up a table passed in as an array of lines. Passes any unprocessed text on to other relevant functions.
- sub getFieldHeaders() – Finds headers in the current table.
- sub markUpFieldData() – Marks up data within individual table fields.

4.13 LDMS::XMLOut

This module does the output work for the LDMS system, buffering output in preparation for commitment to disk by XML::Writer.

- sub Initialize($outputFilename) – Sets up the writer to buffer output text.
- sub BeginTag($tagName) – Begins a new tag.
- sub AddAttribute($attributeName$attributeKey) – Adds an attribute and contents to the currently-open tag.
- sub EndTag() – Ends the currently open tag.
- sub WriteString($stringToWrite) – Writes a given string to the output buffer.
- sub Finalize – Dumps the output buffer to XML::Writer.
- sub DumpStack() – Debug function, displays the contents of the internal stack.

4.14 LDMS::XREFS

This module identifies and tags up cross-references to the US code and Public Law documents. There is a lot of variety in how cross-references are written up in the US code. This module therefore has several functions to extract target information from different representations of cross-references.

- sub handleXrefs – Top level exported subroutine that calls the functions below.
- sub markXrefs – Marks the starts and ends of all known cross reference patterns.
- sub processXrefs – Examines the syntax of each cross reference and assigns it to the appropriate function for the extraction of target information.
- sub uscTargets – Extracts information from targets of the form 1 USC 112 which refers to section 112 of Title 1.
- sub pubTargets – Extracts information from public law references.
- singleTarget – Extracts target information that refer to only one section of the US code.
- sub multiTarget1-5 – Extracts target information from references to multiple targets in the US code.
- sub writeToXML. – Sends output to the FootRef module and XML buffer.