1.0 Introduction
1.1 Purpose
This document provides an outline of the Large Document Search Engine as a whole and describes the contextual analysis engine utilized in this system. This document is intended for future developers in the following groups:
Project Managers
Software Developers
Linguistic Scientists

1.2 Scope

The Large Document Search Engine or LDoc is a system designed to perform contextual analysis on large documents for the purpose of comparing and cataloging documents. The process of research would be greatly enhanced if passages and works could be identified and weighted based upon similarity to other documents or keyword requests. The LDoc evaluates documents by exploiting two types of nouns; standard and proper nouns. Standard nouns are expanded using a thesaurus to extract meaning. Proper nouns are identified by removing all other sentence elements (IE: verbs, adjectives, ...). Future improvement of this technique could allow analysis of an author's whole body of work. Permitting a user to make a complex request regarding the author's view of a topic. The result of this request would provide validation and supporting evidence. Furthermore, If multiple bodies of work were compared theories could be evaluated for an entire movement.

1.3 Terms
1.3.1 Definitions
Lower Order Words - These words are the top weighted contextual items associated with any noun defined by a weighting process of passing a word into a thesaurus and then its resulting synonyms are passed back into the thesaurus The most common terms become a word's Lower Order Words.

Document Element - These consist of Whole Document, Paragraph, Sentence, and Word and define regions for comparison testing

1.3.2 Abbreviations
LDoc - Large Document Search Engine
LOWords - Lower Order Words

1.4 Reference
1.5 Overview
This document is divided into 4 sections including:
Overall Description
Specification Requirements
Supporting information

2.0 Overall Description
2.1 Product perspective
The contextual analysis engine described in this document is intended to be included in a search engine which can store processed documents for meaningful future analysis. Examples of this include the combined works of academic journals like JSTOR or the works of an author from a source like Project Gutenberg. These documents will be stored in a database so users may look for research materials in a manner beyond full-text searches. Additional interfaces would allow for context keywords to be used in existing web search engines to return and on demand analyze websites for matching materials.

2.2 Product Function Overview
2.2.1 Document Processing
The main purpose of this engine is to read a large volume of plain text and return a complex object which contains all the words of the original document in relation to its location. The location of a word is in terms of

the sentence and paragraph in which it existed. Words are categorized based upon their linguistic type. The hypothesis is that the context carrying members of a document are the nouns. By identifying and expressing a document in terms of its nouns it should be possible to identify the context of the whole document.

2.2.1.1 Document Structure Identification

Sentences are identified by a string of words followed by a (.)period.
Paragraphs are identified by tab preceded by a (.)period.

2.2.1.2 Document Canonization

Before the document is processed for its individual parts it is canonized. This results in a string of the whole document with only single spaces between text objects, tabs, (.)periods, and CR LF characters. This process helps to eliminate words being confused with special characters.

2.2.1.3 Term Identification

Externally acquired dictionaries from dict.org and word.net were used to provide a means of identifying a words purpose in the sentence. Currently words are identified as Adjective, Connectives, Adverb, Verb, Noun, and Keyword. Adjectives, Connectives, Adverbs, Verbs, and Nouns are identified by comparing words to their respective dictionary lists. Keywords are the words which fall through this identification process. Keywords are expected to be proper nouns and abstract or foreign words. Keywords are the primary level of contextual comparison as they are the most unique type of words in a document. The secondary comparison items are the nouns and their weighted LOWord values. These are stored along with the word in relation to their location in a Vector. This Vector is a stacked series of objects.

2.2.1.4 Term Weighting

The process of term weighting occurs in both the case of a LOWord or a keyword. LOWords are weighted a number of times before they are even affixed to their respective word. To generate the LOWords for a particular noun the noun is first compared against a thesaurus which returns a long series of synonyms. To consider the best context this list is individually compared against the thesaurus and the resulting aggregated list of words is tested for the most commonly occurring words. These most common synonyms are the LOWords of the noun. This list is limited to 5 or 10 items uniformly across the document. At each junction where a sentence, paragraph, or the document ends the aggregated LOWords for that section are once again tested for commonality and the top set is passed on to represent that document element.
Keywords are tested for commonality the same way as nouns are with the caveat that they are not passed into the thesaurus at any time. So at each junction of document elements the keywords themselves are tested for commonality and the top most common values are assigned to that element.

2.2.2 Document Comparison

The truly experimental process of this engine is in how to evaluate the data between the objects. Generally when comparing two documents each element is tested against all of the same elements in another document these results are evaluated based upon the aggregation of all of the test results.

Eg. when paragraphs are tested against each other a sample output would be this:
Document Comparison = 100.00%
Paragraph Comparison = 60.00%
Sentence Comparison = 52.86%
Paragraph Flat Comparison
The Quality for Comparison of Paragraphs 0 to 0 is 100.0%
The Quality for Comparison of Paragraphs 0 to 1 is 19.999999999999996%
The Quality for Comparison of Paragraphs 0 to 2 is 40.0%
The Quality for Comparison of Paragraphs 0 to 3 is 40.0%
The Quality for Comparison of Paragraphs 0 to 4 is 0.0%
The Quality for Comparison of Paragraphs 0 to 5 is 0.0%
The Quality for Comparison of Paragraphs 0 to 6 is 40.0%
The Quality for Comparison of Paragraphs 0 to 7 is 40.0%
The Quality for Comparison of Paragraphs 0 to 8 is 0.0%
The Quality for Comparison of Paragraphs 1 to 0 is 19.999999999999996%
The Quality for Comparison of Paragraphs 1 to 1 is 100.0%
The Quality for Comparison of Paragraphs 1 to 2 is 40.0%
The Quality for Comparison of Paragraphs 1 to 3 is 0.0%

.
.
.
The Quality for Comparison of Paragraphs 6 to 5 is 0.0%
The Quality for Comparison of Paragraphs 6 to 6 is 100.0%
The Quality for Comparison of Paragraphs 6 to 7 is 40.0%
The Quality for Comparison of Paragraphs 6 to 8 is 0.0%
The Quality for Comparison of Paragraphs 7 to 0 is 40.0%
The Quality for Comparison of Paragraphs 7 to 1 is 0.0%
The Quality for Comparison of Paragraphs 7 to 2 is 0.0%
The Quality for Comparison of Paragraphs 7 to 3 is 40.0%
The Quality for Comparison of Paragraphs 7 to 4 is 0.0%
The Quality for Comparison of Paragraphs 7 to 5 is 40.0%
The Quality for Comparison of Paragraphs 7 to 6 is 40.0%
The Quality for Comparison of Paragraphs 7 to 7 is 100.0%
The Quality for Comparison of Paragraphs 7 to 8 is 0.0%
The Quality for Comparison of Paragraphs 8 to 0 is 0.0%
The Quality for Comparison of Paragraphs 8 to 1 is 0.0%
The Quality for Comparison of Paragraphs 8 to 2 is 0.0%
The Quality for Comparison of Paragraphs 8 to 3 is 0.0%
The Quality for Comparison of Paragraphs 8 to 4 is 19.999999999999996%
The Quality for Comparison of Paragraphs 8 to 5 is 40.0%
The Quality for Comparison of Paragraphs 8 to 6 is 0.0%
The Quality for Comparison of Paragraphs 8 to 7 is 0.0%
The Quality for Comparison of Paragraphs 8 to 8 is 100.0%
0.23950617283950618

These documents are in fact identical. This is evident if we look at the flat document comparison and see that when the paragraph numbers match the quality is 100% While other paragraphs match partially. Currently identical document all score approximately 24% on a flat comparison with reasonable consistency no matter the number of paragraphs.

2.2.3 Available Data Objects

2.2.3.1 Processed Document

This is an example of the Document object hierarchy and associated properties.

```
public class Document
{
	public Vector<String> topLOWords = new Vector<String> ();
	public Vector<String> topKeywords = new Vector<String> ();
	public Document(){}
	public Vector<Paragraphs> Block = new Vector<Paragraphs> ();
	{
		public Vector<String> topLOWords = new Vector<String> ();
		public Vector<String> topKeywords = new Vector<String> ();
		public int length;
		public Vector<Sentences> Paragraph = new Vector<Sentences> ();
		{
			public Vector<Words> Sentence = new Vector<Words> ();
			public Vector<String> topLOWords = new Vector<String> ();
			public Vector<String> topKeywords = new Vector<String> ();
			public int length;
			public boolean contextSentence;
			public Vector<Words> Sentence = new Vector<Words> ();
			{
				public String Word;
				public Vector<String> LOWords = new Vector<String> ();
				public int length;
				public boolean isKeyword;
				public String WordType;
				public Words(){}
			}
		}
```

```
        }
}
```

2.2.3.2 Reconstructed Document

For additional testing the processed document is also converted into a flat array of objects which instead of being nested by document element contain an ID number for the elements they belong to.

Each element of this object is as such:

```
public class ReconstructDocument
{
        public String wordcontent;
        public int paragraphID;
        public int SentenceID;
        public int wordID;
        public int wordLength;
}
```

2.3 User Characteristics

As a standalone engine for comparison testing users would required general programming skills and experience with Java programming in Eclipse to make positive use of this system.

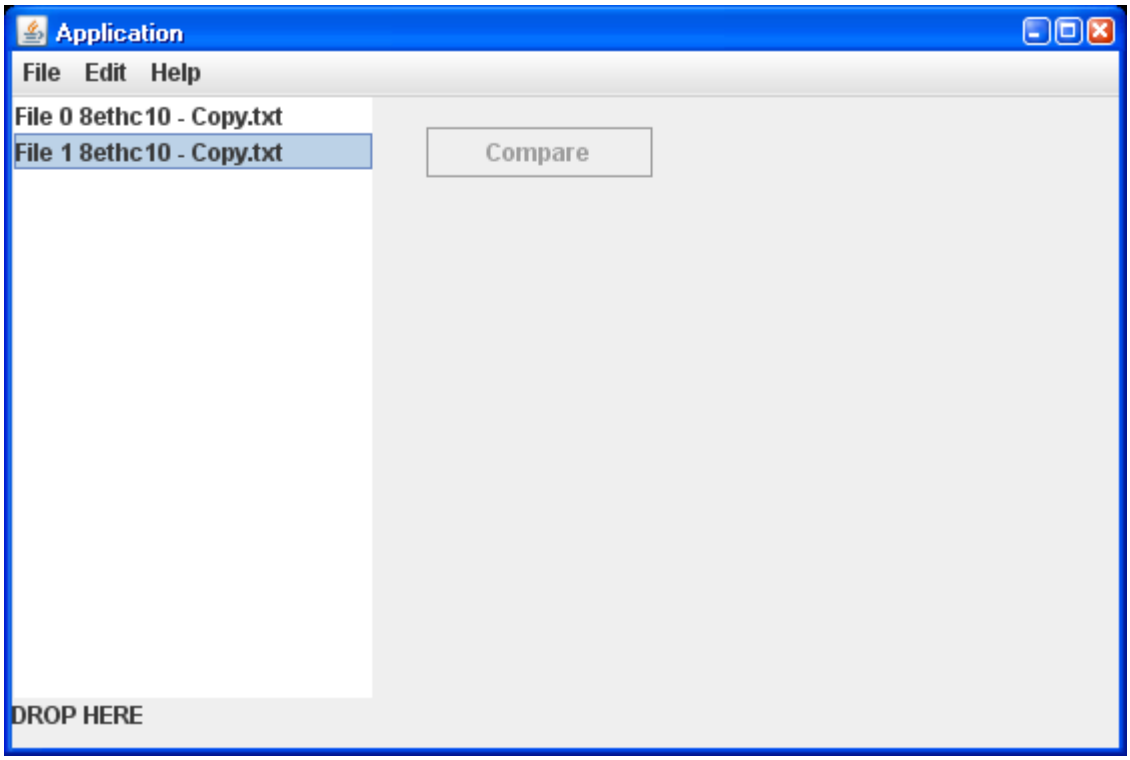
2.4 Constraints

The application is restricted to running in Eclipse for testing and reporting in this developmental phase and requires that arguments be set for Java to properly execute document comparisons. To avoid out of memory errors use the arguments -Xms1024M -Xmx1024M or greater values.

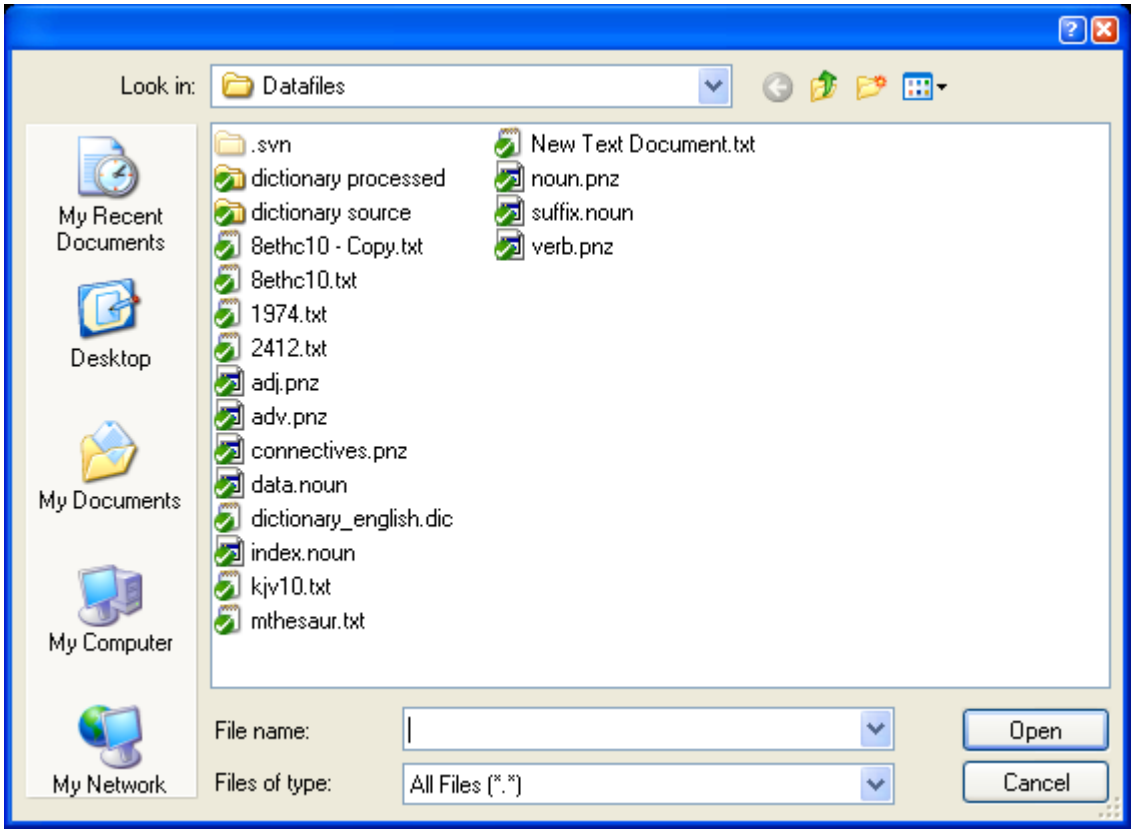
3.0 Specification Requirements

3.1 External Interfaces

Gui is currently for testing comparison only and will change when user comparisons are implemented. The main window allows for documents to be loaded for processing using a drop-down File->Open option.



The open dialog is the OS default dialog and is cross platform compatible.



3.2 Classes

The program has two entry points. First, is the App.java class which is the driver to handle creation of the processed document objects. Second, is the MainApplication.java class which generates a GUI for testing documents and concurrent document processing and calls the App.java driver for each file loaded.

3.2.1 MainApplication

MainApplication handles the generation of the main testing GUI and handles concurrent file processing by threading each new File->Open and Comparison separately.

CRC

Properties

logger : Logger
application : MainApplication
listModel : DefaultListModel
selected : Object[]
jFrame : JFrame
jContentPane : JPanel
jJMenuBar : JMenuBar
fileMenu : JMenu
editMenu : JMenu
helpMenu : JMenu
exitMenuItem : JMenuItem
openMenuItem : JMenuItem
aboutMenuItem : JMenuItem
cutMenuItem : JMenuItem
copyMenuItem : JMenuItem
pasteMenuItem : JMenuItem

saveMenuItem : JMenuItem
aboutDialog : JDialog
aboutContentPane : JPanel
aboutVersionLabel : JLabel
jTextPane : JTextPane
jTextPane1 : JTextPane
jLabel : JLabel
analyzingObjects : Vector<App>
analyzedObjects : Vector<App>
jList : JList
jButton : JButton

Methods:

getJTextPane()
getJTextPane1()
updateFileList()
getJList()
getJList().new ListSelectionListener() {...}
getJButton()
getJButton().new ActionListener() {...}
main(String[])
main(...).new Runnable() {...}
main(...).new Runnable() {...}
getJFrame()
getJContentPane()
getJJMenuBar()
getFileMenu()
getOpenMenuItem()
getOpenMenuItem().new ActionListener() {...}
getEditMenu()
getHelpMenu()
getExitMenuItem()
getExitMenuItem().new ActionListener() {...}
getAboutMenuItem()
getAboutMenuItem().new ActionListener() {...}
getAboutDialog()
getAboutContentPane()
getAboutVersionLabel()
getCutMenuItem()
getCopyMenuItem()
getPasteMenuItem()
getSaveMenuItem()

3.2.2 App

App is the main driver for all document processing. Not only does negotiate the reading of the document to be processed but also creates the nested document object and the flattened sequence object. Please refer to the minimalist sequence diagram. It excludes trivial functions like the logger from the classes description.

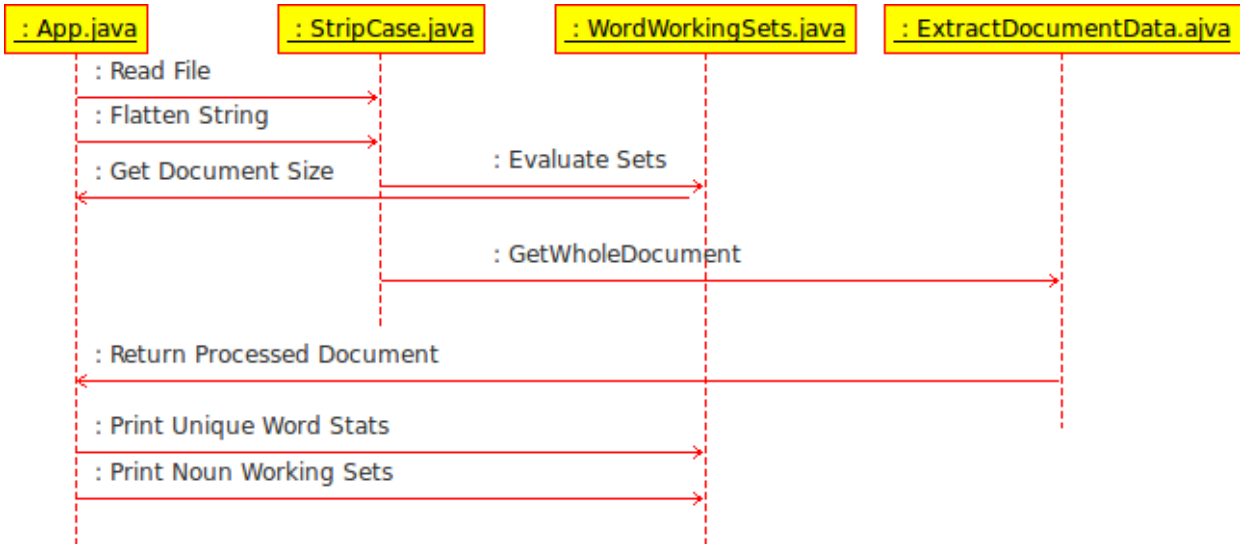
CRC

Properties:
_done : boolean
logger : Logger
Filename : String
EvaluateSets : WordWorkingSets
file : StripCase
SequencedDocument : ArrayList<ReconstructDocument>
ProcessedDocument : Document
threadID : int
FullFileName : String

Methods:

getProcessedDocument()
App()
App(String)
getSequencedDocument()
isDone()
getFullFileName()
getFileName()
setThreadID(int)
getThreadID()
setisDone(boolean)
ProcessFile(String)

Sequence Diagram



3.2.3 ExtractDocumentData

By utilizing the WordManagement class to provide word lists this class breaks the entire document into its core components, Words. It sequentially identifies each word by type and handles term weighting on nouns and keywords. strings are initially delimited by (.)period and stored as sentences followed by tab delimiting for paragraph identification. This information is stored as a nested Document object as described in section 2.2.3.1

CRC

Properties:
loggerDocument : Logger
loggerParagraph : Logger
loggerSentence : Logger
FinalDocument : Document
WholeDocument : String
NounList : WordManagement
ConnectivesList : WordManagement
AdverbList : WordManagement
AdjectiveList : WordManagement
VerbList : WordManagement
DocSize : double
CurrentWord : double

Methods:
ExtractDocumentData()
ExtractDocumentData(String, double)
setDocSize(double)
getFinalDocument()

setWholeDocument(String)
BreakOnDocument()

3.2.4 SequenceDocument

This class does a reversal of the work done in the ExtractDocumentData class except this storage is flat and word location is identified by IDs for each paragraph, sentence, and word. This is accomplished by traversal of the previously created object.

CRC

Property:
SequencedDocument : ArrayList<ReconstructDocument>

Methods:
getSequencedDocument()
SequenceDocument(Document)

3.2.5 StripCase

StripCase is actually more multifunction then its title would let on. This class is responsible for reading the file provided to App.java and creates a string that is all lowercase with punctuation limited to the (.)period and formatting limited to CLRF and tabs for identifying paragraphs. White space is also normalized so that no non tab whitespace is followed by additional whitespace. This eliminates null words being identified later by ExtractDocumentData.

CRC

```
Properties:
logger : Logger
WholeDocument : String
NormalizeWhitespaceFlag : boolean

Methods:
StripCase()
GetWholeDocument()
FlattenString()
NormalizeWhitespace(boolean)
ReadFile(String)
```

3.2.6 WordManagement

A general class to handle dict.org format wordlists and place them into a String Key null contents hashmap for rapid comparison to the tests in ExtractDocumentData. The wordlist format is a single word per line with no spaces or special characters. These lists are processed ahead of time and handle in a static manner.

CRC

Properties:
WordList : HashMap<String, Object>
file : File

Methods:
WordManagement(String)
LoadList(String)
isInList(String)

3.2.7 WordWorkingSets

This class handles some statistical needs for optimizing ExtractDocumentData and to provide progress feedback for the user when processing very large files.

The generated statistics are:
Number of words in document
Number of unique words in the document
Number of nouns in the document
Number of unique nouns in the document.

Since the number of nouns in a document is about 10% of the total unique words the identification process is optimized by preprocessing the LOWord terms and assigning them to their respective nouns as opposed to processing the LOWords each time the noun occurs.

CRC

Properties:
uniquesize : int
start : int
stop : int
WholeDocumentSize : double
wholedoc : String
uniqueworkingSet : Vector<String>
nounworkingSet : Vector<String>

Methods:
getNounWorkinSet()
getUniqueSize()
setThreadStartStop(int, int)
WordWorkingSets(String)
getDocumentSize()
BuildNounWorkingSet()
BuildNounWorkingSet(int, int)
BuildUniqueWordWorkingSet()
PrintNounWorkingSet()
PrintUniqueWordStats()

3.2.8 ThesaurusHandler

In a similar way this class acts very similarly to the WordManagement class except the format of the files it handles is different. The thesaurus used is Grady's Moby Thesaurus. The format of this document is a word on each line followed by its synonyms comma delimited. Unlike WordManagement the hashmap created uses the first word on a line as the key value and places its synonyms in an arraylist. This class also handles reducing the list of synonyms returned from the thesaurus lookup to a constant length.

CRC

Properties:
file : File
Thesarus : HashMap<String, ArrayList<String>>
LOWordSize : int

Methods:
ThesaurusHandler()
ThesaurusLookup(String)
ReduceSynonyms(Vector<String>)
CorrelateThesaurusItems(String)

3.2.9 WordListTools

This is a general purpose static class with functions derived from the WordManagement method which identifies the most commonly occurring items in a list of words. Primarily used to handle the keyword filtering that occurs at each junction of a document element in ExtractDocumentData.

CRC

Properties:
ListLimit : int

Methods:
WordListTools(int)
TopItems(Vector<String>, int)
TopItems(Vector<String>)

3.2.10 CompareDocuments

This is a temporary class for testing methods of evaluating the results of two or more processed files. When fluent methods of identifying documents is discovered this class will be depricated in lue of individual classes which will handle each comparison method.

CRC

Properties:
Selection : ArrayList<SelectionItems>
logger : Logger
AnaylzedDocuments : Vector<App>
Comparables : ArrayList<Document>
DocumentComparison : ArrayList<Comparisons>

Methods:
CompareDocuments(ArrayList<SelectionItems>, Vector<App>)
compareDocuments()
CmpLOWords(Vector<String>, Vector<String>)
LOWordComparison(Vector<Vector<String>>)
gatherParagraphsLOWords()
gatherSentencesLOWords()
retrieveDocumentLOWords(Document)
retrieveParagraphLOWords(Paragraphs)
retrieveSentenceLOWords(Sentences)
retrieveWordLOWords(Document, int, int, int)
extractComparisonList()

3.3 Logical Database Requirements

None

3.4 Software System Attributes

3.4.1 Availability

As an end user research platform the only factor defining system availability is the amount of system memory and the presence of the Java runtime. This application requires a heap size of at least 1024 for both the main process and spawned threads.

3.4.2 Security

This application does not currently allow for any privileged access to the system which hosts the applications process.

3.4.3 Maintainability

Modification to the runtime of the document processor is housed within a single Class allowing for future changes to each document process event to be introduced with minimal modification. Document objects are kept at the top level of the end user runtime and can be passed from the GUI or the main document processor without conflict with the rest of the system.

3.4.4 Portability

LDoc is completely cross platform under the standards that Java provides for its runtime to exist in multiple platforms. The only system specific issues are CR LF items in document encoding, directory seperators, and OS specific dialogues which are handled nativly by Java. The system has been tested to be fully portable across Windows and Linux builds which contain a current Java runtime.

Specific Requirements

3.5.1 System Mode

No special System modes are defined for LDoc

3.5.2 User Class

All classes of objects within this system are defined for the same purpose of evaluating the contextual analysis engine.

3.5.3 Objects

3.5.3.1 Document

This object contains a complex nested series of objects as described in section 2.2.3.1 and is available from App.java, MainApplication.java and ExtractDocumentData.java. This is the primary object used for testing and includes all of the documents words which have been identified by the document processor along with LOWords and keywords.

3.5.3.2 Sequenced Document

The Sequenced document is a flattened version as described in Section 2.2.3.2 and lacks the identification information for each word but maintains the words position in the document for distance calculation and document regeneration.

3.5.3.3 Selection Items

This class defined structure is used to pass completed document objects from the MainApplication.java GUI to the associated comparison classes. This is used in conjunction with the analyzed documents collection which is defined by all of the App.java objects containing the objects from Sections 3.5.3.1 and 3.5.3.2

Object Definintion:

```
public class SelectionItems {
	public String Filename;
	public int ID;
}
```

3.5.4 Feature

Current user interaction is limited while testing the contextual comparison results. Three primary elements exist with which a user can interact.

3.5.4.1 File->Open Dialog

This event draws the OS defined open dialog and passes the resulting value to App.java for processing.

Event Handler

From MainApplication.java

```
private JMenuItem getOpenMenuItem() {
if (openMenuItem == null) {
	openMenuItem = new JMenuItem();
	openMenuItem.setText("Open");
	openMenuItem.addActionListener(new ActionListener() {
		public void actionPerformed(ActionEvent e) {
		FileDialog fd = new FileDialog(jFrame, null, FileDialog.LOAD);
		fd.setLocation(50, 50);
		fd.setVisible(true);
		final FileDialog fd2 = fd;
		new Thread(new Runnable() {
				@Override
				public void run() {
					try {
						logger.trace("File Dialog: "+fd2.getDirectory() + fd2.getFile());
						App element = new App(fd2.getDirectory() + fd2.getFile());
						analyzingObjects.add(element);
						}catch (Exception e) {
							logger.trace("Cannot Create New Thread");
						}
				}
			}).start();
		}
	});
}
return openMenuItem;
}
```

3.5.4.2 Compare Button

When two or more items in the processed document list box are highlighted this button is accessible to the user. It fires the comparedocuments process which evaluates the selection for contextual similarity using a multitude of experimental tests and weights.

Event Handler

```
private JButton getJButton() {
if (jButton == null) {
	jButton = new JButton();
	jButton.setSize(113, 25);
	jButton.setLocation(new Point(207, 15));
	jButton.setText("Compare");
	jButton.setEnabled(false);
	jButton.addActionListener(new ActionListener() {

		@Override
		public void actionPerformed(ActionEvent e) {
			new Thread(new Runnable() {
				private SelectionItems SelItems = null;
				@Override
				public void run() {
					String filename = "";
					ArrayList<SelectionItems> Selection = new
						ArrayList<SelectionItems>();
					for(Object item : selected) {
						String[] splititem = item.toString().split(" ");
						for(int i = 2; i<splititem.length;i++) {
							filename = filename+ " " +splititem[i];
						}
						filename = filename.substring(1);
						SelItems = new SelectionItems();
						SelItems.ID = Integer.parseInt(splititem[1]);
						SelItems.Filename = filename;
						Selection.add(SelItems);
						filename = "";
					}
					@SuppressWarnings("unused")
					CompareDocuments Comparison = new
						CompareDocuments(Selection, analyzedObjects);
				}
			}).start();
		}
	});
}
return jButton;
}
```

3.5.4.3 Processed Documents List

This control is shown to the user as a multi-selection file list. Each successfully processed document is placed in this list and can be highlighted by the user. When two or more objects are highlighted this control activates the compare button so that the selected objects can be passed to the CompareDocuments.java class. Selections are passed into a list which identifies the filename and the file ID of each selection so that only the required items are transmitted to CompareDocuments.java.

Event Handler

```
private JList getJList() {
	if (jList == null) {
		jList = new JList();
		jList.setBounds(new Rectangle(1, 0, 179, 300));
		jList.addListSelectionListener(new ListSelectionListener() {
		public void valueChanged(ListSelectionEvent evt) {
```

```
                        if (!evt.getValueIsAdjusting())
                        {
                                JList list = (JList) evt.getSource();
                                selected = list.getSelectedValues();
                                if (selected.length >= 2){
                                        jButton.setEnabled(true);
                                }
                                else {
                                        jButton.setEnabled(false);
                                }
                         }
                    }
                   });
        }
        return jList;
}
```

3.5.5 Stimulus

This application takes no automated inputs for processing.

3.5.6 Response

System responses are logged for each document processed by Log4j in the following files:
CompareDocumentThread.txt:
Shows the results of the document comparison for the last comparison

DataProcessThread.txt
Shows the term identification for the last document processed

Document.txt
Shows the words and LOWords of the last document processed for each document

Paragraph.txt
Shows the words and LOWords of the last document processed for each paragraph

Sentence.txt
Shows the words and LOWords of the last document processed for each sentence

3.5.7 Functional Hierarchy

please refer to the Sequence Diagram attached to section 3.2.2 App.

3.6 Additional Comments

4.0 Supporting Information

4.1 Table of Contents and Index

4.2 Appendixes