

Thoughts on NEH Implementation Strategies

Introduction.....	1
What The NEH Project Is About.....	1
Sources For Works.....	3
OCR.....	3
Correcting Optical Character Recognition Errors.....	3
Text to XML.....	4
Gutenberg.....	5
Existing Works in TEI Format.....	5
Preparing A Work For Import Into The Library.....	5
Proofreading.....	7
Correcting Morphological Tagging.....	8
What versions of a work document reside in the library?.....	9
Training data.....	10
Metadata For Works.....	10
Main and Paratext.....	10
Word and Sentence-level Metadata.....	11
Verticalized Text (WordBase).....	11
Implementation.....	13
Implementing the Content Manager.....	13
Implementing the Work Editor.....	14
Implementing Work Flow.....	15
Time estimates.....	15
Conclusion.....	17

Introduction

In this memo I describe the NEH project in terms suitable for defining the design and programming labor involved. I offer some suggestions for implementation approaches, and rough estimates of the development time involved. These notes are fuzzy at times since much of the project is not yet well defined. The contents are necessarily technical.

What The NEH Project Is About

The NEH project seeks to create a digital library of printed English language works from the Early Modern English period (~1470) to about 1930. Many of the works will be newly scanned from original sources. Each source work will be converted to a canonical legible text format using a specialized variant of the the Text Encoding Initiative (TEI) Extensible Markup Language (XML) version P5 encoding standard. Called TEI Analytics, this version of the TEI XML standard was developed for use in the Monk project. The original page images, when available, will be kept along with the corresponding XML representation. The TEI Analytics representation of each work will be canonical. Other representations may also be stored, such as a Document Object Model representation, to improve the efficiency of access by the computer programs which manage and manipulate the texts.

Each work in the library will be "marked up" with XML tags to encode traditional divisions such as volumes, chapters, acts, poems, letters, paragraphs, and sentences -- reflecting the original printed form of the work. Other types of divisions, not based upon the original print representation, may also appear. For example, non-English words and sections of text will be marked. The mark-up process will be partly automatic and partly manual. Software will be created to assist with manual mark-up.

Each word in every text will be morphologically adorned with information such as lemma and part of speech. Every effort will be made to distinguish between text written by the original author(s) and text added by others, such as notes from editors or printers. Narrative and spoken text will be distinguished. Other distinctions (assigning words to specific speakers in plays, marking stage directions, and so on) may also be noted.

Extra-textual bibliographic information or "metadata" will be added to each work. The format and contents of this metadata will be standardized to allow scholars to select works by criteria such as author, period, and genre.

The process of preparing a work for inclusion in the library requires a variety of textual emendations. This results in multiple versions of each work, at different stages in the editorial process. The library maintains a complete history of all changes to each work. Changes may be effected by many different persons or programs, sometimes simultaneously. The library will be able to track who made what changes, when, and where. Some changes will unavoidably collide. The library will offer some assistance in resolving such collisions. Changes can be rescinded, and any previous state of a work (and all its associated documents and extracts) can be restored.

Once a work has been stored in the library, it may be subject to further editorial correction and annotation. Additionally, associated documents and annotations for a work, under version control, can be stored in the library along with the document. Examples of such additional documents might be glossaries, maps of obsolete to modern spellings, scholia, bibliographies, biographies, authorial images, and databases of derived counts of morphological phenomena. All of these can also be placed under version control.

The functions of the digital library can be separated into two major pieces of computer programming.

1. The **Content Manager** manages the storage and retrieval of all the different data associated with each work, including the history of all emendations to each work and its associated documents. Access to the Content Manager is mediated through the Work Editor.
2. The **Work Editor** centralizes all aspects of editing a work and its associated documents. The work editor operates in a social environment to allow multiple people (and programs) to edit a work simultaneously. Examples of editing processes include textual proofreading activities (correction of orthographic scanning errors, correction of structural markup, correction of morphological adornment errors), defining values for work metadata, and creating workflows for processes such as adding a work to the content library or updating training data.

During the work preparation process the Work Editor invokes two existing software

suites which will nevertheless require refinements for use in NEH.

3. Abbot is the TEI Analytics transformer which mutates TEI XML variants into the canonical TEI Analytics format. Abbot is under development at the University of Nebraska.
4. MorphAdorner is a program suite which segments texts into tokens and sentences, and adorns each token with morphological information. MorphAdorner's home is Northwestern University. See

<http://morphadorner.northwestern.edu/>

for details.

Sources For Works

There are three primary sources for the texts which reside in the digital library.

1. Printed volumes, newly scanned and converted to text format using Optical Character Recognition technology.
2. Existing works in text format or a non-TEI form of markup such as texts from Project Gutenberg.

<http://www.gutenberg.org/>

3. Existing works in TEI format such as texts from the Wright Archive.

<http://www.letts.indiana.edu/web/w/wrightmrc/>

OCR

The Open Content Alliance is producing new electronic versions of texts from printed materials using modern Optical Character Recognition technology. The scanned text must be edited for errors, then converted to TEI XML format.

Correcting Optical Character Recognition Errors

I am not familiar with the latest generation of high-end text scanners, so some of what I say here may be obsolete. The last time I worked extensively with original scanned texts was back in 2000 as part of a volunteer project. I helped transcribe a small number of 19th and early 20th century books, monographs, and journals on the subject of astronomical mythology.

Common mistakes made by text scanners and the associated optical character recognition software include the following.

1. Introducing extraneous whitespace in words. Example: the word “bottom” is read as two words, “bot” and “tom”.
2. Deleting whitespace between words. Example: the words “bad manners” are read as the single word “badmanners”.
3. Substituting characters or runs of characters for other characters. Examples: “cl”

is read as “d”, “iii” is read an “m”, the letter “l” as the digit “1”, “t” as “c”, the obsolete long “s” as “f”, “rn” as “m”, and so on.

4. Inserting a default character such as “~” for an unrecognized character. Example: the word “examine” becomes “exa~ine”.

All of these kinds of errors appear frequently in the TCP texts, and to a lesser extent, in the NCF texts. Multiple types of errors can occur in the same word. I presume we can expect more of the same for any newly scanned works. All such errors can be corrected manually. As this is a tedious job, it would helpful to correct as many as possible without manual intervention.

A simple approach which corrects about 40% of OCR errors is to scan the work text with at least three different brands of scanners, and compare the resulting output using a simple voting scheme. A specific reading is considered “correct” when the majority of the scanners “vote” by choosing that reading. This is expensive in terms of scanning time and equipment, but in the past, has been less expensive than the time and cost of human labor.

Statistical methods based upon observed frequencies of character ngrams (sequences of characters) and approximate string-matching methods may help, as may edit-distance algorithms. (An edit distance algorithm determines the number of single character insertions, deletions, replacements, or transpositions required to transform one string to another. Fred Damerau demonstrated many years ago that 80% of ordinary spelling errors correspond to one of these types of changes.)

The algorithms used in the OCRSpell spelling correction system of Taghva and Stofsky

<http://www.isri.unlv.edu/ISRI/Software>

may be useful in implementing an OCR spelling corrector in the Work Editor.

Text to XML

Once the text of a work is considered sufficiently cleaned of OCR defects, the next step is to convert the cleaned text to XML format. The same is true for works which arrive in plain text format. Converting plain text to XML is not easy. There is no general algorithm for doing this. Each individual work requires separate treatment. This was hammered home when I worked on the astronomical mythology project. Each book and journal had a different format, and each required separate handling.

It may be possible to devise a cascade of algorithms to jumpstart the markup process so that a human editor can complete the job in reasonable time. GLOSS implements some of these algorithms:

<http://gloss.bham.ac.uk/>

GLOSS mechanizes a part of of the process of converting plain text to XML by extracting structural information from an input text file and generating a well-formed XML output file. GLOSS is not a magic bullet. Originally it was designed to allow writing a well-structured plain text file for conversion to LaTeX. Nevertheless GLOSS may prove useful when designing the plain text to XML transformation process of the

Work Editor, even should none of GLOSS's code be used.

The documentation for ATOX helpfully references a number of other programs which try to convert plain text to some kind of structured text, often HTML.

<http://atox.sourceforge.net/>

The text-to-XML conversion portion of the Work Editor will likely undergo continuous refinement as new works are added to the digital library. It is unlikely that this process can be fully automated.

Gutenberg

Many Gutenberg Project texts were prepared under the aegis of the Distributed Proofreaders group.

<http://www.pgdp.net/c/>

These texts are encoded with a custom tagging scheme, starting with scanned page images converted to text using an Optical Character Recognition process. An overview of the tagging scheme appears on the formatting guidelines page at:

<http://www.pgdp.net/c/faq/document.php>

There are already over 12,000 texts encoded in this format. Since there is an existing alliance between the Digital Proofreaders Group and the Open Content Alliance, it might be worth creating a program to convert the Gutenberg texts into an XML format which is close enough to “standard” TEI to be converted into TEI-Analytics form by Abbott. While the resulting texts might not pass scholarly muster, they would provide a large set of texts for populating the digital library during the creation and testing phases of the Content Manager and Work Editor. And they might be turn out to be good enough after all.

Existing Works in TEI Format

When a collection of English language works is encoded in some variant of TEI XML, it should be possible to adjust Abbot and MorphAdorner to process those texts. This assumes an existing training data set pertains to the collection. It may be necessary to prepare new training data using texts within the new collection to enable MorphAdorner to adorn the texts accurately. Abbott may also require modifications to handle variances in the XML markup in the new collection.

Preparing A Work For Import Into The Library

The following work flow used by the Distributed Proofreaders suggests how a work might wend its way through the various stages of initial preparation and editing. The NEH project stores far more information, so the Distributed Proofreaders work flow is incomplete for our purposes, but it serves as a useful basis for thinking about how portions of the NEH work flow show operate.

One major difference is that there are extra levels of proofreading in NEH. Proofreading

Preparation

(mostly offline)

Stage	Requirements	What goes on
CP Content Provision	none, or a scanner	find a book to scan, or find images online; obtain a copyright clearance for it
OCR Optical character recognition	OCR software	convert image to text using software
PM Project Manager	familiarity with DP PM status (given upon request)	pre-process text, load project onto DP, write Project Comments, answer questions later on

The Rounds

(at the DP website)

P1 Proofreading round 1	DP account	check OCR output, match text to image
P2 Proofreading round 2	300 P1+R pages pass proofing quiz 21 days at DP	check P1 output, examine image and text closely, spell check
P3 (optional) Proofreading round 3	400 P1+P2 pages 50 F1 pages, 42 days review of P2 pages	check P2 output, examine image and text closely, spell check
F1 Formatting round 1	300 P1+R pages pass formatting quiz 21 days at DP	check/add formatting markup
F2 Formatting round 2	400 F1 pages 91 days review of F1 pages	check F1 output, examine image and text closely for formatting

Final Assembly

(mostly offline)

PP Post-processing	400 F1 pages	final proofing and formatting checks, assemble text, create other versions (html, LaTeX, etc.)
SR (optional) Smooth reading	none to read DP account to upload	read through the final work, note any queries or problems
PPV PP verification	peer review	check PP output, correct any problems, mentor new PPs
PG Project Gutenberg		text is available at PG for download

is controlled by the Work Editor. Another is that NEH keeps a complete journal of all changes to each work's text and regenerates associated files as needed. Yet another is that the number of roles for participants is more varied. There are different types of proofreaders, various types of curators, editors, and administrators. The set of roles in NEH is not yet defined.

Proofreading

There are four main types of errors the Work Editor proofreading system should assist in correcting.

1. Optical Character Recognition errors.
2. XML Structural Tagging errors.
3. Spelling errors.
4. Morphological Adornment errors.

Some errors can be corrected automatically using statistical methods and algorithms. Other require manual correction. When feasible, the Work Editor running in an interactive mode should offer suggestions to allow for simple “click and select” corrections, much in the style of traditional spelling correctors.

Each type of error may require a different style of display. A suitable display might be the original page image on the left, the formatted text on the right (generated from the XML or DOM version of the work), and a panel below highlighting suspect words. For editing the structural markup, a similar display could show the original page image on the left and the marked up text on the right, but only with as much of the markup as needed for the specific task at hand. For example, when adding XML tags to mark non-English sections of text, there is no reason to show word-level tags.

The Work Editor could implement a "smart" XML editor that allows turning on and off the display of specific tags and tag levels. Or the Work Editor could use a “what you see is what you get” type of editor that operates like a typical word processor, using TEI Analytics XML underneath. Both options require a way to edit large XML documents with very little memory, so that multiple users can access the Work Editor server simultaneously. A typical XML file takes about ten times the memory of its disk-based character representation. Adorned XML files can be several hundred megabytes in size. It is impractical (or even impossible) to load such large files for editing in a shared online environment.

One approach is to implement a “persistent DOM” which maps a document object model representation of the XML to a disk file, maintaining only a small portion of the file in memory at any given time. Writing a persistent DOM implementation is quite a bit of work, but something like it will be needed throughout the NEH project.

Regrettably there does not appear to be any free or open source version of a persistent DOM for Java. The expensive commercial product from InfonYTE is based upon the PDOM which used to be free from Fraunhofer:

http://www.infonyte.com/en/prod_pdom.html

This is a read-only implementation and so not very useful for our purposes. It also doesn't work with Java 1.6 .

eXist is an XML database system that implements a persistent DOM at its core:

<http://exist-db.org/>

My experiences with eXist have never been particularly successful. eXist has never handled large files well. A recent rewrite of portions of the core promise to improve performance and stability for large files.

There are many commercial XML-based databases and indexing tools available. How useful any of these might be in NEH is unknown at this time.

A third option is to avoid online editing altogether by allowing people to “check out” the XML text and edit it offline. A check-in process then validates the updated XML and determines the changes by comparing the updated XML with the existing XML. A fourth option, which is probably the best (and at the same time the most difficult to implement), combines online and offline editing in a seamless manner. Small sections of the text to be edited are sent to the client system, where editing proceeds in a specialized local editor. Changes are sent back to the server at specific intervals or when manually specified by a “save” operation.

Regardless of the editing method used, the Work Editor works with the Content Manager to handle editing conflicts. Changes are never made directly to the current version of a document, but instead are kept in the form of instructions telling how to change the old version to the new version.

The experiences of the Distributed Proofreaders group may be helpful in thinking about how to design the proofreading components of the Work Editor:

http://www.pgdp.net/c/faq/proofreading_guidelines.php

Also useful are the Wright 19th century archive editing instructions:

<http://www.letts.indiana.edu/web/w/wrightmrc/guidelines.html>

Hopefully some of these operations can be automated by the Work Editor.

Correcting Morphological Tagging

The most difficult type of proofreading, requiring the most expertise, is the correction of morphological tagging. MorphAdorner, like other part of speech taggers, makes mistakes when assigning parts of speech to individual words. MorphAdorner can also make mistakes in assigning lemmata and standard spellings.

Morphological proofreading can proceed page by page. Usually however this type of proofreading targets specific problems, such as correcting the part of speech for the genitive in older English texts.

In English before 1700 the apostrophe never indicates the genitive, and "her mother's daughter" is written "her mothers daughter". An even more problematic example is "her majesty's daughter" which appears in early texts as "her majesties daughter." The use of the apostrophe as a genitive marker gained ground during the eighteenth century, and has

been used as it is today since the early nineteenth century. In the eighteenth century, the apostrophe is sometimes used as a plural marker in certain character combinations. Thus "canoe's" is much more likely to be a plural than a possessive form.

MorphAdorner often assigns the correct possessive part of speech for these older types of spellings, but not always. A specific Word Editor procedure might be to display all plural and possessive nouns in context, and allow the proofreader to select the correct part of speech if the wrong one was supplied by MorphAdorner (or a previous proofreader).

Another approach is to display a “wordbase” of counts of tuples of the form (spelling, lemma, part of speech, standard spelling, count). Allowing the rows to be sorted by each column often reveals obvious errors in these forms, especially when they occur only once. When an error is found and corrected in one work, it might be advisable for the Work Editor to remember the correction, and automatically apply it to other texts as an option.

What versions of a work document reside in the library?

A work proceeds through many stages starting with its original entry in the digital library under control of the Content Manager and Work Editor. Typically, the XML representation for a work exists in the following variants, representing successive stages in the editing process under control of the Work Editor. The decision to move a work from one stage to the next will generally be made by a human curator.

- Original OCR text (one).
- Corrected OCR text (many).
- Initial XML text (one).
- Corrected XML text (many).
- Initial TEI-Analytics XML (one).
- Corrected TEI-Analytics XML (many).
- Initial MorphAdorned XML (one).
- Corrected MorphAdorned XML (many).

In addition to the legible XML text versions, there may be other representations stored, such as a Document Object Model. Derived data files may include token lists and counts, tuples of morphological information and counts, verticalized text representations in which each word and its associated information appear as a row in a legible or database file, bibliographic metadata, and so on.

There will also be data compiled across works, such as counts of morphological phenomena, scripts for creating and updating associated database files, and files containing training data.

The size of a MorphAdorned XML file can get quite large, into the hundreds of megabytes. One option for reducing the size is to use the abbreviated MorphAdorner format that avoids storing redundant adornments. A description of the abbreviated MorphAdorner format appears at:

<https://apps.lis.uiuc.edu/wiki/display/MONK/MorphAdorner+XML+Output>

The Work Editor or Content Manager should incorporate the facility of producing the non-abbreviated MorphAdorned version as an option. When displaying word-level morphological attributes, the Work Editor supplies the values for the redundant attributes as needed.

Training data

Some documents achieve special status, serving as sources of training data for MorphAdorner and other statistical or machine-learning based software.

In particular, MorphAdorner requires substantial training data in order to generate the statistical probability matrices and other derivative data used to adorn texts with morphological information. These derived data files are stored in the Content Manager repository, and receive the same version control treatment as their source texts,

As the training texts are edited, the derived training data files and MorphAdorner files change automatically to match.

Metadata For Works

The Monk project offers a list of the work metadata which should be elicited for each work:

<https://apps.lis.uiuc.edu/wiki/display/MONK/Proposal+for+metadata+about+works+%2810-19-2007%29>

<https://apps.lis.uiuc.edu/wiki/display/MONK/Collection+Metadata+Group>

Development of the content manager includes refining the parts of these proposals that are not Monk specific, and defining specific storage formats and retrieval specifications.

For example, the TEI Analytics processing modules could embed the work metadata within the canonical XML representation of a work. Probably the best way to do that is to define a custom XML name space.

The Work Editor provides the user interface for adding and editing work metadata.

Main and Paratext

Words in a work are divided into two classes, **main text** and **paratext**.

Words in main text words are clearly the author's. Words in paratext are ambiguously or clearly not the author's. Paratext may also include words in lists and tables that are not easily parsed as sentences.

For more details, see the Monk project memo at:

<https://apps.lis.uiuc.edu/wiki/display/MONK/Proposal+about+main+text+and+paratext+%2810-17-2007%29>

Word and Sentence-level Metadata

Word level metadata are about each token in a text. Punctuation marks count as separate word tokens. MorphAdorner is responsible for dividing the text of a work into tokens and assigning the word-level metadata.

For more details, see the Monk project memo at:

<https://apps.lis.uiuc.edu/wiki/display/MONK/Proposal+about+word+and+sentence+level+metadata+%2812-1-2007%29>

and also the memo on MorphAdorner XML output at

<https://apps.lis.uiuc.edu/wiki/display/MONK/MorphAdorner+XML+Output>

Verticalized Text (WordBase)

The basic premise of a WordBase is to provide a fully verticalized version of every work in the digital library, in what appears to be a single database-like table. This means there is a row of information in this table for each word in every work. In actuality, this means multiple tables are probably required, since counts of word-level phenomena and bibliographic information for the works should also be available. The intent is to allow any type of query against this database one could imagine, and have it satisfied quickly and efficiently.

This is essentially a superset of the Monk datastore.

John Norstad's memo about scaling to a billion words with MySQL remains germane:

<https://apps.lis.uiuc.edu/wiki/display/MONK/Scaling+to+a+Billion+Words+with+MySQL+%28archive%29>

The problem in the context of NEH is actually much worse. Reasonably fast retrieval time is required if the WordBase is to be used in online proofreading applications. Someone may wait minutes or hours for a "batch" analytical operation, but this is out of the question when nearly instantaneous response is needed for real-time proofreading. Likewise, database build times measured in days or weeks are out of the question when texts are being added and edited on a daily basis.

Generating partial forms of the WordBase -- word-level information on a work-by-work basis, or information concatenated for a small number of works for generating training data -- is feasible. A fully verticalized database, with fully general query potential, with fast update for corrections and new works, appears infeasible using a single non-distributed database.

Is it possible to create a simple-minded distributed WordBase using ordinary MySQL technology?

Suppose we have 1,000 texts with which we wish to populate the giant WordBase (not one of the summary versions). Also suppose we have a sudden influx of cash and we can purchase 10 good PCs.

Divide the 1,000 texts into 10 sets of 100 texts.

On each PC, set up MySQL. In each copy of MySQL, create a database "wordbase" with

100 tables of identical form. Each table is the wordbase for a single work. The name for each table is the name of the work. For example, the wordbase for Emma is named "ancf0204" after its name in the NCF collection.

This results in 1,000 individual "wordbase" tables, distributed across ten databases, one database to a PC. (An alternative is to use 1,000 databases, 100 to a PC, each named for a work, and containing a single table "wordbase".)

Now assume we have a strictly limited set of well-known, pre-specified queries which can be issued. A simple server program -- let's call it QueryMelder -- accepts one of these predefined types of queries, for example, "do query-type-1 with values "help" and "uh". QueryMelder turns this into a SQL query of the general format:

```
select spelling, lemma, pos where spelling='help' and pos='uh'
```

QueryMelder knows the names of the 10 PCs and the names of the databases and tables on each PC. QueryMelder creates a modified version of the SQL query adding relevant database/table names for each PC. QueryMelder issues the query customized for each PC's database names and tables, to each PC's copy of MySQL. QueryMelder waits for all 10 PCs to respond, merges the results, and passes them back to the requester -- not necessarily as a Java ResultSet, but a more specific Java object, as John's current Monk database does.

This is a much more straightforward operation than a generic distributed database query handler because all the tables have identical formats and all the queries are limited to a preselected set. The client programs never use MySQL directly -- everything passes through QueryMelder. So the back end could actually be something completely different from MySQL.

I suspect that a month or two of full-time work by a programmer would reveal whether this scheme (or a minor modification) would offer sufficiently good response time for actual use. Two nodes would suffice to test the QueryMelder to start. We could use texts from the current set of NCF texts for creation of the database tables. We could duplicate texts using different names to get larger numbers of texts. For example, we could create ancf0204-1, ancf0204-2, and ancf0204-3 to get three more "texts" by cloning Emma.

If this scheme works, we would have a way to update the "big" WordBase one work at a time. There would be lots of work to deal with versioning, handling unresponsive PCs, figuring out how to build-in redundancy, and so on. None of that work matters until we know if this distributed database scheme works. If it does, the WordBase project should fall into the one year time line as well (given another person to work on it, so presumably three programmers total so far).

This scheme might prove useful even on one machine for breaking up the aggregated wordbase by work. With that we're dealing with updating tables containing tens of thousands of entries, which MySQL should be able to do very quickly.

In addition to the main "wordbase" table, which contains one row for each word, two much smaller summary versions are useful. The first summarizes counts of morphological tuples by work. The second table contains the counts collapsed across all works.

A variant of this last table, with an added field giving the works in which the tuple appears, is actually all that is needed to perform proofreading. Clever use of bit maps allow saving the list of applicable works without using a helper table. Once the work to proofread is known, the full WordBase for that work can be constructed on the fly, or retrieved from a previously saved copy.

Emerging products such as HyperTable -- the first test release was issued a couple of months ago in February 2008 -- may offer a solution to creating large databases such as the full WordBase.

<http://www.hypertable.org/>

HyperTable runs over Apache Hadoop, which supports data intensive distributed applications running on large clusters of ordinary computers. Hadoop was inspired by [Google's MapReduce](#) and [Google File System](#) (GFS) papers.

HyperTable is not a simple replacement for a database like MySQL. For one thing, HyperTable data is not typed -- it is all uninterpreted byte strings. All revisions of the data are stored in HyperTable, allowing simple access to older versions. This fits in well with the Content Manager.

Implementation

I assume we want all of the software to be as system-independent as possible. I also assume the non-interactive portions of the software will be written in Java.

Implementing the Content Manager

For the Content Manager I suggest we use an implementation of the Java Content Repository (JCR) programming interface. This interface defines a content repository as a hierarchical content store with support for structured and unstructured content, text search, versioning, transactions, observation of data changes, and so on.

Version 1 of the JCR interface is defined in Java Specification Request 170:

<http://jcp.org/en/jsr/detail?id=170>

and version 2 -- in progress- is defined in Java Specification Request 283:

<http://jcp.org/en/jsr/detail?id=283>

The Wikipedia article at

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

offers a useful introduction (in technical terms) to the Java Content Repository API.

There are several available implementations of the Java Content Repository. Apache Jackrabbit

<http://jackrabbit.apache.org/>

has been used in projects such as Sakai, an open-source collaboration and learning environment. Whether it can handle content repositories of the scale planned for NEH is unclear. Alfresco, a commercial open-source implementation, is intended to support large

scale repositories.

<http://www.alfresco.com>

Also see the Wikipedia article about Alfresco at:

http://en.wikipedia.org/wiki/Alfresco_%28software%29

Magnolia is another commercial implementation:

<http://www.magnolia.info/en/magnolia.html>

Implementing the Work Editor

A “bare bones” implementation of the Work Editor could be constructed using a set of dynamic web pages constructed using Java servlets or similar technologies. However, it might be worth considering the extra startup cost of building the Work Editor on top of an existing superstructure for “social collaboration” such as a Wiki or a portal environment.

A portal provides a framework for integrating information, people, and processes in an controlled and secure manner. The open-source JASIG uPortal environment is widely deployed in educational environments and has matured over the past few years, while still receiving constant development from its constituent community.

<http://www.uportal.org/>

There are several Apache portal projects:

<http://portals.apache.org/>

In fact, uPortal integrates Apache Pluto to implements the Java Portlet Standard:

<http://www.jcp.org/en/jsr/detail?id=168>

The Java Portlet Standard enables (theoretically) interoperability among different portals and portlets, and provides several useful programming interfaces for aggregation, personalization, presentation and security. Portlets provide pluggable user-interface components which are managed and displayed under control of the portal. The various proofreader components of the Work Editor could be implemented as portlets.

The open-source Liferay portal

<http://www.liferay.com/web/guest/downloads/portal>

demonstrates how a portal can integrate social networking facilities such as chat, a message base, a wiki, email, and others. Now consider integrating NEH-specific facilities such as proofreading, metadata definition, and workflow editing into this mix. .

While the programming learning curve is steeper when developing in a portal environment, the long term benefits should outway the startup costs. For example, building atop a portal structure using portlets would allow expansion to a distributed environment in the future. Portals are, by their nature, intended to operate in a distributed fashion.

Implementing Work Flow

A **work flow management system** accepts a formal description of business processes as input, then executes and maintains the state of the defined processes, delegating activities among people and programs as needed.

Operating systems usually provide a system-wide but system-specific way of defining a work flow. Windows offers batch files. Unix and Mac OS X offer shell scripts. Scripting languages such as Perl, Python, or Ruby allow for defining work flows in more system independent terms.

Modern programs often provide an embedded scripting system to allow extending the program beyond its original intent, to allow adding new or modified features without changing the program code itself, or to allow interfacing to other programs. For example, WordHoard uses the BeanShell scripting language to provide extensibility:

<http://www.beanshell.org>

BeanShell is essentially an interpreted version of Java with a relaxed syntax. There are other scripting languages available for Java, such as Jython (a Java version of Python) and Groovy.

The last few years have seen the arrival of specialized languages designed solely to define work flows. There are many specialized work flow languages available for use in Java programs.

<http://java-source.net/open-source/workflow-engines>

The advantage of a general scripting language is that it can also provide for user extensions to the Content Manager and Work Editor. On the other hand, defining new work flows is usually easier with a specialized work flow language.

Time estimates

The following are very rough estimates. Assume they represent lower bounds. It might be prudent to think about obtaining a three month planning grant to allow a more detailed definition of many of the project processes, and to allow investigation in to the alternative tools for building the various portions of the project.

The design, programming, documentation, and deployment of a minimal "first edition" of the library Content Manager would take at least a year of a senior programmer's time. The first edition would include a non-distributed, single node digital library content manager with versioning and sufficient work flow automation to allow input and output of works provided in TEI-like XML sources. Both a programmatic and simple web-based interface (possibly web-browser based, but not necessarily) would provide access. Different levels of access would be controlled by assignation of roles to individual users (which could be programs) of the Content Manager.

The design, programming, documentation, and deployment of a minimal "first edition" of the the Work Editor would take at least a year of a senior programmer's time. The first edition would include a non-distributed, single node Work Editor running in a portal environment, interfacing with a single Content Manager through a well-defined

programming interface. The Work Editor would include sufficient work flow automation to allow multiple individuals to add works to the digital library and edit those works either online or offline at different stages. Both a programmatic and simple web-based interface (possibly web-browser based, but not necessarily) would provide access. Different levels of access would be controlled by assignment of roles to individual users (which could be programs) of the Work Editor.

Work on the content manager and work editor can proceed in parallel once an initial programming interface for the Content Manager services is available for use by the Work Editor.

The design, programming, documentation, and deployment of the work flow editor can overlap with the development of both the Content Manager and the Work Editor. The work flow editor will probably change quite a bit as the specific needs of those two parts of the project become clearer. A quarter-time programmer would be helpful in reviewing the available work flow products and assisting both the Content Manager programmer and the Work Editor programmer with integrating work flow methodology. This can be a junior programmer position.

Another quarter-time programmer, who could be the same as the work flow programmer, could usefully assist in designing test scenarios for both the Content Manager and Work Editor, as well as with miscellaneous programming tasks. One of these might be to write the program to convert Gutenberg markup to TEI-like XML.

Thus 2.5 programmers working for a year should be able to produce viable initial versions of the Content Manager and the Work Editor. Since it is more realistic to consider programmers working half time, two years is the better elapsed time estimate.

A user interface designer and developer would be helpful in designing the Work Editor human interface, including the social editing environment. That is probably a half-time position which should run the entire length of the project.

A competent technical writer would be helpful in developing both the technical and the user-level documentation. Few programmers are good prose writers (John Norstad is a rare exception). In any case, programmers should concentrate on programming. A good technical writer can often point out where the program design is weak by recognizing where the documentation cannot be written clearly.

The design, programming, documentation, and deployment of the Gutenberg markup to XML converter will take between one and three months of a programmer's time, depending upon how much documentation is available for the Gutenberg format. This process can overlap development of the rest of the project.

The design, programming, documentation, and deployment of the fully verticalized text (WordBase) is unlikely to be feasible in the near future. This should be a separately funded project. Since the WordBase requires the digital library, a good time to revisit its design would be after a substantial digital library is built and the data needed for the wordbase becomes available. Technological advances over the year or two – particularly with projects like Apache Hadoop and HyperTable -- may allow effective implementation of the full WordBase.

The design, programming, and deployment of the individual work WordBases, and the

“small” WordBase with counts accumulated over works, will take a programmer working somewhere between a quarter and half time over a year.

The design and implementation of distributed versions of the Content Manager and Work Editor should be considered after the initial non-distributed versions are complete.

Conclusion

The creation of the Content Manager and the Work Editor allows for the construction of a large digital library of English works. The availability of these works, stored in a standardized canonical format in a managed repository, allows their use in many different applications.

Constructing the Content Manager and Work Editor to interact seamlessly and effectively will be a difficult task. It will take several dedicated programmers, as noted above, at least a year to produce minimally useful versions of these programs. More realistically, programmers working half time could produce them in two years.