



NOVeLLA: A Multi-Modal Electronic-Book Reader With Visual and Auditory Interfaces

JOSHUA S. HODAS

Department of Computer Science, Harvey Mudd College, 1250 North Dartmouth Ave, Claremont, CA 91711
hodas@cs.hmc.edu

NEEL SUNDARESAN

NehaNet Corp., 2355 Paragon Drive, Suite E, San Jose, CA 95131

JARED JACKSON

Almaden Research Center, IBM Corporation, 650 Harry Road, San Jose, CA 95120
jjared@almaden.ibm.com

BRANDON L. DUNCAN

Lead Systems Engineer, Common Technology and Services, Gap Inc., 900 Cherry Avenue, San Bruno, CA 94066
brandon_duncan@gap.com

WALTER I. NISSEN*

L-039, 7000 East Ave., Livermore, CA 94550-9234, 925-423-0947
nissen5@llnl.gov

JUDE BATTISTA

207 West 31st Ave, Spokane, WA 99203, (509) 624-1459

Received August 15, 2000; Revised April 6, 2001

Abstract. While significant growth in the popularity of electronic books seems likely over the next few years, current systems and near-term proposals from major vendors have shown little foresight in leveraging the equally inevitable increase in processing power available to such systems. In this note we describe a prototype electronic-book reader, NOVeLLA, whose design takes as its starting point the assumption that computing power equivalent to today's high-end desktop systems will be available in hand-held formats within just a few years. It proposes interesting ways in which that power might be utilized, in particular by delivering a powerful speech-based user interface. A secondary goal of the project was to implement the system using off-the-shelf software technologies and Application Programming Interfaces (APIs) available freely or cheaply today, testing whether those technologies and related standards were adequate to the task. Finally, NOVeLLA is the first generally available reader compatible with the unencoded version of the new Open Ebook Document Structure.

*I work for the Lawrence Livermore National Laboratory, Computation Directorate, Computer Applications Organization, Scientific Computing Applications Division, A Support Group. Or, to simplify, LLNL, CD, CAO, SCAD, ASG.

Keywords: electronic book, user interface, audio interface, disabled users, speech recognition, text-to-text speech

I. Scenario

A young executive sits on the train reading a draft of a company report on the screen of an electronic tablet. She highlights an occasional passage by dragging her finger across the screen. Once, she taps on a word and types a comment into a dialog box using a keyboard that appears on screen.

As the train pulls into her stop, she gets off and walks to her car parked in the lot. Tossing the tablet on the seat, she starts the car, and, as she pulls out of the lot, she says,

“Begin speaking.”

The tablet obliges, reciting the report from the beginning of the page she was viewing.

“Stop,” she says, at one point.

“Begin dictation,” and she dictates a brief comment about some dubious sounding numbers.

“Resume speaking,” she says when she is done with her note, and the tablet does so.

The next morning, in her office, she scans back through the report. At each point that she had dictated or typed a comment an annotation mark appears. Tapping on the mark causes the text of the annotation to appear in a dialog box, regardless of whether it was originally spoken or typed. As she pages through the document she adjusts the base font size to show more text on the screen. The text reflows smoothly, maintaining its overall formatting; all the highlights and annotations she added remain faithfully positioned on the text to which they relate.

She selects a few of the notes that she wants her assistant to see, and marks them as such. At the click of a button in the document chooser, she uploads those comments to the department server over the wireless network. The next time her assistant opens her copy of the report, her boss’ comments appear, marked with red highlights.

II. Introduction

The advantages of digital information storage naturally lead to a desire to access information at any time, using an intuitive interface, and in a convenient form factor. The replacement of paper books with electronic devices is a common goal of numerous companies and researchers working in this area. Given present

limitations of screen design in terms of contrast and resolution, and the reduction in quality of user experience that this implies, it may seem unrealistic to expect anyone other than the traditional early adopters to use these devices. However, even before advances in display technology overcome these limitations, such systems would seem to offer numerous benefits that at least balance the negatives, and should therefore lead to more broad-based adoption. These benefits include, but are not limited to:

- The ability to hold many books at once, limited only by storage capacity rather than by physical bulk
- The ability to provide in-the-moment content-purchase opportunities, unencumbered by the delays of traditional book delivery systems
- The ability to provide a back-up mechanism for documents, so that physical damage does not inevitably correspond to content loss
- The ability to easily exchange documents between readers (subject to some license-management scheme) and to view user-generated documents as well
- The ability to arbitrarily resize text to provide users with visual impairments access to documents without their having to wait for special large-print editions
- The ability to render text as speech for visually-disabled users, and as a convenience feature for other users
- The ability to accept audible commands in navigating and annotating book content
- The ability to store and retrieve user annotations, and to allow users to share those annotations with others flexibly and securely without redistributing the entire document

Unfortunately, commercially available devices to date have delivered on only a small fraction of this promise; typically, in fact, they deliver only the first three or four features, which are innate in essentially any electronic device.

Existing commercially available systems, in particular the Softbook Press Softbook (Softbook Press, 2000), the Rocket Ebook (NuvoMedia, 2000), and the Glassbook Reader (Glassbook, 2000), provide almost

no flexibility in content presentation. The Softbook and Ebook appear to use a very simple markup language internally, and do not support even such simple mechanisms as right-justified text. In addition they allow the user to view the text in only two font sizes, apparently storing preformatted versions of the document in each size. The Glassbook Reader, which exists simply as a software system at this point, is designed primarily to view documents in a variant of Adobe's PDF language (Adobe Systems, Inc., 2000). While this provides for rich document formatting, text can be scaled only in the sense that the user can zoom in on part of the page and then scroll around to view the rest. While proposals for future designs demonstrate a few more features, they still seem startlingly limited, given the computational power that will be available in appropriate form factors in just a short time.

The eNovelties project was undertaken by students at Harvey Mudd College in the fall of 1999 under the direction of researchers at the IBM Almaden Research Center. The initial goals of the project were simply to provide a reader for the then new Open Ebook (OEB) Publication Structure (OEB Foundation, 1999) and to investigate the use of the Extensible Markup Language (XML) (Graham and Quin, 1999) for storing user annotations. When the OEB specification was published, and a review showed it to be little more than a minor restriction of HTML with Cascading Style Sheets (CSS), the project was modified to create a "blue-sky" reader system that incorporated as many features as could be managed with current hardware. At the same time, the team was to rely as much as possible on published, open APIs and off-the-shelf implementations of those APIs, in order to test the adequacy of the proposed standards and their implementations. In part because of the team's ties to IBM, they focused in large part on the use of speech technologies as embodied in IBM's ViaVoice (IBM Corporation, 2000a,b).

The resulting system, NOVeLLA, is a software-only system, but is intended as a prototype of a hardware interface. While still in prototype stage, NOVeLLA is available freely and was built entirely using open standards. All code was written in Java 2.0 (Sun Microsystems, 2000a), using the Java Speech API (Sun Microsystems, 2000c) and the XML For Java parser API (McLaughlin and Loukidos, 2000 and IBM Corporation, 2000c) as additional support. NOVeLLA now supports many of the actions described in the sample scenario above, and contains the necessary architectural underpinnings to support the rest.

III. Open Ebook Standard

In September 1999, the Open Ebook (OEB) Authoring Group, the working group of the OEB Foundation, released the first version of the OEB Publication Structure specification (OEB Foundation, 1999). In releasing this specification, the foundation, comprised of representatives from publishing and software companies including Microsoft, Adobe and IBM, and current electronic book reader manufacturers, hoped to standardize the way in which electronic books are formatted and shared among readers.

As mentioned earlier, the OEB specification uses XML as its foundation. XML is used in two distinct ways to describe the content of a book. First, a package file is created that stores metadata concerning the content and navigation of the book. This file is described in further detail in Section VI. Second, XML is also used to describe the content to be made available for rendering. In this case, the tag set used is little more than a subset of HTML 4.0 (World Wide Web Consortium, 1999d), with additional restrictions on the attributes associated with some tags.

While the number of contributors to the OEB Foundation grew significantly after the publication of the specification, at the time the NOVeLLA prototype was completed in the spring of 1999 there was still no other electronic book reader on the market that used the standard as its basis of representing the content of its books.

It should be noted that other OEB Foundation groups have focused primarily on standards for document licensing, security, and commerce. In fact, these specifications seem to have received a good deal more attention than the document structure specification. The eNovelties project was not concerned with this aspect of the electronic book experience, and NOVeLLA provides no support for these mechanisms. The OEB documents it reads are in unencoded, raw-text form.

IV. User's View of NOVeLLA

As discussed briefly above, the NOVeLLA user interface provides a near balance between visual and aural/oral interfaces (together referred to as the *audio user interface*). The graphical user interface is fairly traditional, drawing on the design of existing electronic book readers and similar systems. The audio interface is based solely on the designers' inspiration. They conceive of two, somewhat different, target audiences for the audio interface: users who use it as a convenience

feature, and physically or visually impaired users who make use of it out of necessity. The scenario presented in the introduction focuses on the convenience user. The goal is eventually to produce a single, possibly configurable audio interface that can serve both markets.

At present, most actions that can be triggered by clicking on the visual interface can also be triggered by voice command. Much of the information presented visually can also be presented aurally, likely at least enough to satisfy the convenience user. Techniques for presenting the remaining types of information in this manner remain an area of current and future research.

The behaviors of the graphical and audio interfaces are tightly linked, and therefore will be described together, organized by function rather than interface type. As discussed in Section IX, the performance of the off-the-shelf software components was adequate to provide roughly real-time response to commands, whether issued by voice or the GUI.

The Book Chooser

The user's first interaction with NOVeLLA is with the book chooser, from which a book is selected for reading. Rather than simply presenting a list containing the names of book files present on the file system, requiring user familiarity with file naming conventions, the book chooser parses the metadata contained in the book files and presents the user with a list containing book titles and author names. Figure 1 shows a typical book chooser display.

A book is selected from the list using the GUI in the typical manner. Alternatively, the user can navigate the list using speech commands such as "up" and "down" to move the selection highlight, and "select" or "open" to open the highlighted document. The user can also move the highlight simply by speaking the title of the book they wish to select. The audio interface will recognize the name and move the highlight to that title.

Main Window

Once a title has been selected and opened, the book chooser is hidden and the main viewing interface is presented with the first page of the book loaded. Figure 2 shows a typical view of the main window of NOVeLLA. At the top of the page is the name of the book as well as the chapter number and the page number within the chapter. The latter is obviously dependent on the current font size, since different

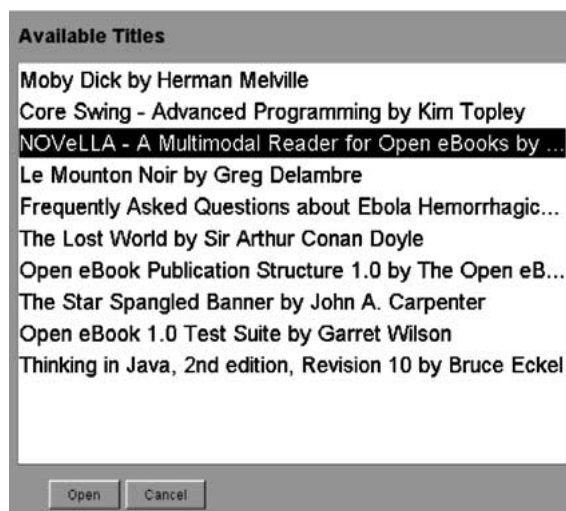


Figure 1. The Book Chooser displays the title and author information of all electronic books stored within the reader. Selecting a title opens a particular book for display.

sizes will cause different amounts of text to fit on a page.

The user interface functions of the main viewer are divided into three principal categories reflected in the grouping of buttons around the display. They are: *navigation*, *presentation*, and *annotation*. In addition there are buttons to initiate a text search, and to reopen the book chooser.

Navigation. The user can navigate forward or backward in the document by either a page or a chapter at a time. The buttons in the navigation cluster are, from top to bottom: *previous chapter*, *previous page*, *next page*, and *next chapter*. The user can navigate by clicking the appropriate button, or by speaking these names to invoke the corresponding action. Additional variations, such as "go to next page" and "page down", are also recognized. (The number of spoken variations accepted for each command is limited, and at present it is necessary to inform the user of the options available. It would be advantageous to expand the grammar to accept as many natural variations of each command as can possibly be foreseen.)

Presentation. The presentation functions are used to control the size of the fonts used to display the text in the display frame, and to direct the system to start and stop reciting the text of the book to the user.

The two left-hand buttons decrease and increase the font size, respectively. Possible font sizes vary

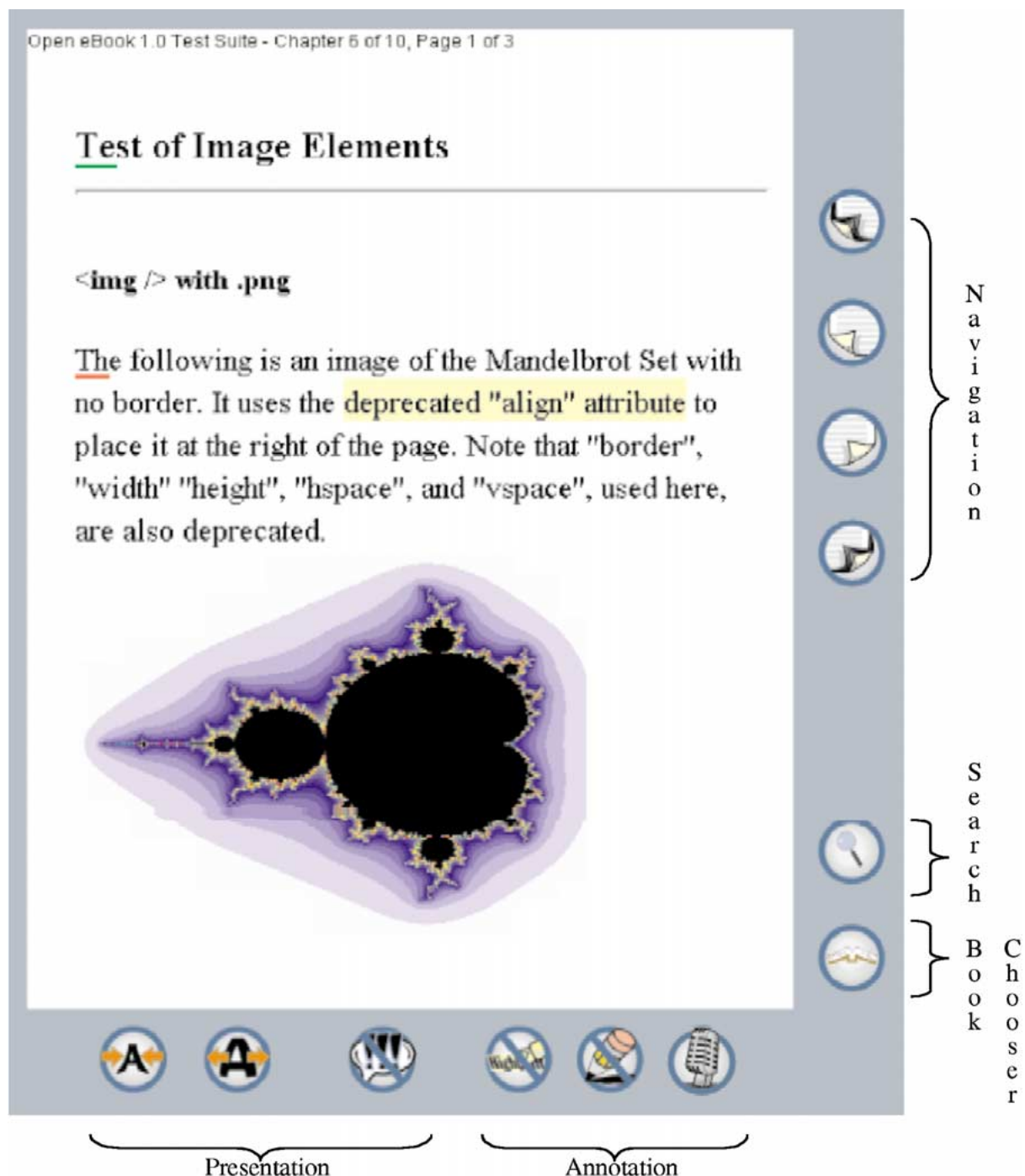


Figure 2. The NOVeLLA main viewer window with a document loaded. The red and green underscores indicate the presence of user annotations. The buttons around the perimeter are grouped by function.

smoothly over the range from eight-point to forty-eight points. There are presently no voice commands to invoke these functions.

The third button is a toggle controlling speech generation. In the state shown, the button indicates that speech is not being generated. Clicking the button toggles it, and signals the system to begin reading the current page from the beginning. Clicking it again pauses the recitation. If the button is clicked a third time while the user is viewing the same page, recitation resumes where it left off. When the system reaches the end of the text of the current page, it automatically advances the viewer to the next page (loading the next chapter, if necessary) and continues speaking. Recitation can also be toggled using voice commands such as "begin speaking" and "stop talking" (or more colorful phrases, such as "shut up"). Whether the toggle is invoked by a click or a voice command, the visual state of the button is always updated appropriately.

Annotation. The book viewer is always ready to accept an annotation from the user. In order to create an annotation the user simply clicks on, or drags over, the text to be annotated. The three annotation buttons control which type of annotation is created by this action. When one annotation method is selected its associated button is highlighted and the other annotation buttons are deselected. Each type of annotation is represented by a different mark in the text window. The sample display in Fig. 2 includes examples of all three types of annotation marks.

In highlight mode, which is the initial setting (and which can be reselected by clicking on the leftmost annotation button), dragging across a block of text highlights it with a light yellow background. The highlight can be removed by clicking anywhere in the highlighted block.

Clicking on the middle button, labeled with a picture of a pencil, puts the system into text-note mode. Clicking on a position in the text causes a dialog box to open into which the user can type some text. When the text box is closed, a green underscore indicates the presence of the annotation. Clicking on the mark reopens the note box, displaying the text of the annotation. The text can be edited, or the note can be deleted, removing the corresponding annotation mark as well. When the text-entry box is displayed, the user can also choose to dictate the annotation rather than type it. Clicking on the record button, or saying "begin recording," causes the system to enter full recognition

mode, in which it will transcribe arbitrary spoken text. Clicking on the stop button, or saying "stop recording," ends transcription.

Clicking on the rightmost button, marked with a microphone, puts the system in audio-note mode. In this mode, clicking on a position in the text opens a box in which an audio annotation (that is, a stream of uninterpreted digitized audio) can be recorded. Recording is started by clicking on the record button, or saying, "begin recording." For reasons explained below in Section VIII, recording can only be terminated by clicking on the stop button; no voice command can be used for this purpose. The presence of an audio annotation is indicated by a red underscore beneath that position in the text. As with text-notes, clicking on the marked spot causes the annotation window to reopen so that the recording can be replayed, replaced, or deleted.

At present the annotation system is largely inaccessible through the speech interface. The difficulty of specifying a location to be annotated or highlighted more precisely than at the page level makes this an area of future research. When the system is dictating the text, the user could pause the recitation at the desired point. But when the user is reading the text, a complex dialogue would seem necessary.

Search. Clicking on the search button, labeled with the magnifying glass, or saying such phrases as "find a word" or "search for a phrase" brings up a text box into which the user can type or speak a phrase to search for. Saying or clicking the button labeled find then invokes a case-insensitive search throughout the current chapter. If the text is found, the page containing the next instance is displayed on screen with the phrase highlighted. Clicking the find next button, or saying the appropriate phrase, causes the process to repeat. When the end of the current chapter is reached, the user is prompted whether to continue the search into the next chapter. Options are available to begin the search from the beginning of the chapter or of the book.

Book Chooser. Clicking on the book-chooser button, labeled with an open book, or saying such phrases as "open a new book," "new book," or "open the book chooser," closes the current book and returns the user to the book chooser where he or she began. When opening a book that the user has previously read, the system returns the viewer to the point at which the user left off reading.

Software Versus Hardware. While NOVeLLA is implemented as a software system, it is intended as a prototype of an eventual hardware device. The most flexible hardware implementation would be a tablet in which the entire interface is implemented in software and the user interacts only by way of a touch-sensitive screen. While this has the advantage of being arbitrarily malleable, physical, tactile buttons offer a more natural interface to most users. Thus parts of the NOVeLLA visual interface are intended to model aspects that will eventually be implemented in hardware, while other parts will remain as software controls.

The white page that fills most of the viewer is presumed to represent the display screen. Any controls and dialog boxes that appear within that area represent features that will display under software control, even in a hardware implementation. The book chooser, search query boxes, and text and audio note boxes fall into this category. In contrast, any feature outside that frame is presumed to represent a tactile control. Of course, in the end it was difficult for the programmers to resist the temptations of the software interface. Many of the supposedly tactile buttons provide visual feedback that would not necessarily be practical or possible with inexpensive plastics. These issues would need to be resolved in the industrial design of a hardware realization of NOVeLLA.

V. System Architecture and Implementation Overview

One of the primary mandates of the NOVeLLA project was to construct the system using open standards and off-the-shelf technologies to the greatest extent possible. The intent was to leverage these technologies to reduce implementation time, while ensuring that the resulting system would not be tied too tightly to any one platform.

Because the system was intended as a test bed for evaluating different user interface ideas, it was also crucial that it be designed extensibly, so that new ideas could be easily integrated into the system at any point. To further this goal, the class structure and method architecture for NOVeLLA were specified and maintained using the Unified Modeling Language (UML) (Page-Jones, 1999). Figure 3 gives a high-level overview of the class structure. In the following sections we describe the most important classes, their relationships, and some details of their implementation. We also discuss some of the difficulties that arose

during implementation, and some of the trade-offs that were made as a result.

VI. User Interface Architecture Overview

The AbstractInterface

Early in the design of the system it became apparent that, if all user interface controls were connected directly to the underlying system, its complexity would grow quickly out of control due to the potentially numerous ways to initiate a given action. Inspired by compiler systems in which the literal, concrete syntax of the language as handled by the parser is distinguished from the abstract syntax used in the compiler internals (McCarthy, 1963; Pfenning and Elliot, 1988), NOVeLLA is built around an *abstract interface* class, which implements the actual actions to be executed, regardless of how the user triggers that action.

An *AbstractInterface* is started automatically when the system is launched. Invisible to the user, it manages the underlying functionality of the system. Any request from the user detected by one of the concrete interfaces is passed on to the abstract interface through a method call. The abstract interface then manages the request as appropriate and signals what changes have been made to all of the concrete interfaces.

Concrete Interfaces

The abstract interface communicates with the user via concrete interfaces. A concrete interface provides means for the user to initiate actions and for the system to provide feedback to the user. The system currently includes two concrete interfaces: the graphical user interface provided by the GUI class, and the audio user interface provided by the AUI class, each of which is created automatically at startup by the abstract interface after the *BookChooser* (described below) is constructed. Aspects of the implementation of the GUI and the AUI are discussed separately later.

Concrete interfaces run simultaneously on separate threads. Communicating with the abstract interface internally allows for seamless and non-conflicting integration of multiple concrete interfaces. It also allows the system to be extended to other interfaces. While NOVeLLA currently offers only two interfaces to the end user, other interfaces could be added easily using this architecture, and would run concurrently with the existing interfaces. For example, one could imagine

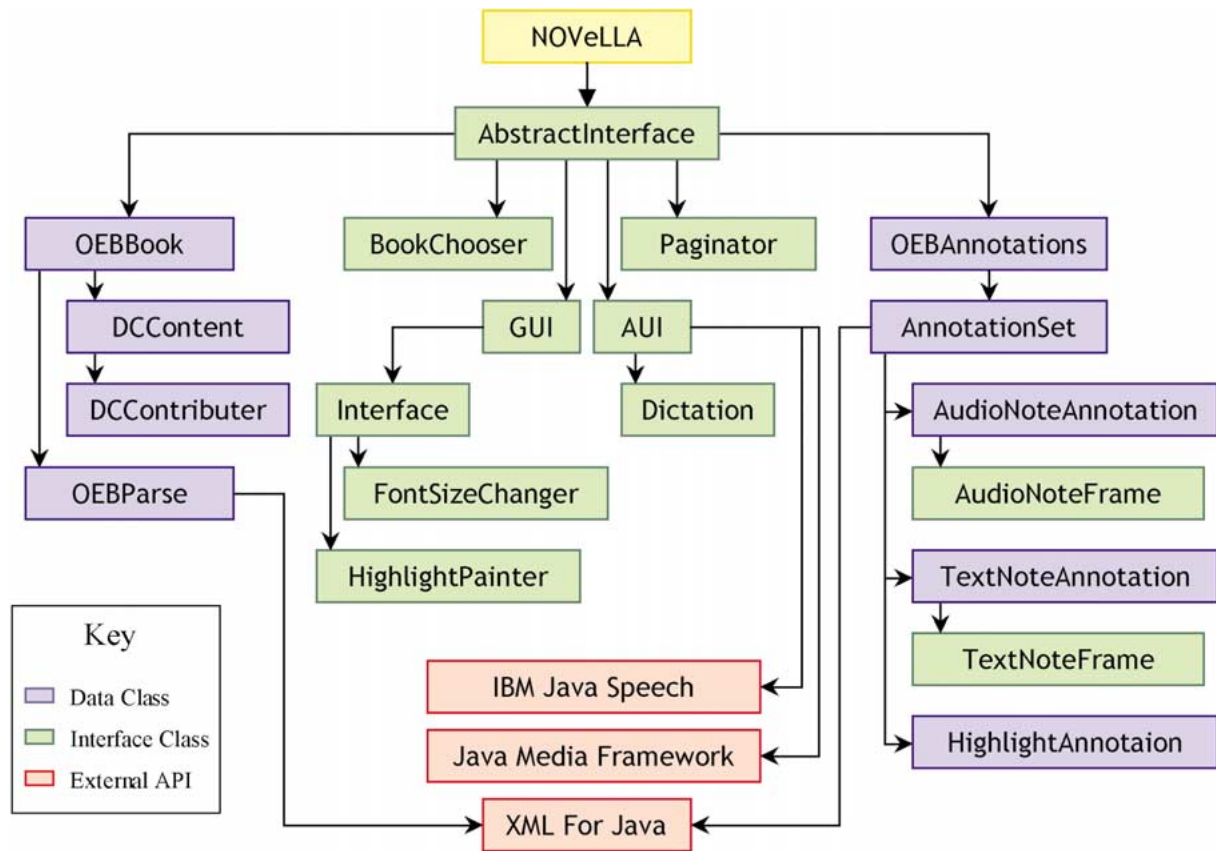


Figure 3. UML Overview of NOVeLLA class architecture, with classes grouped by function.

that visually impaired users might find situations where the use of text-to-speech is impractical or inappropriate. Through the concrete interface API, a Braille output device could be linked to the system for such situations. It would then operate in parallel with the existing concrete interfaces.

The BookChooser and the OEBBook Class

When NOVeLLA is launched, the abstract interface class interrogates the file system as to the presence of book files in designated directories. A single book is typically stored across a number of files. Most of these files contain the actual text content of the book, stored as XML files for each chapter or other unit, while some files hold graphics and other inserts as in typical HTML documents. In addition to these files, each book has a package file, designated with the .opf file extension. This file contains an XML specification of

metadata used by the reader. It contains information such as the name of the author, the publisher, the date of publication, and the type of book. In addition, the package file contains a complete manifest of the files that make up the electronic book. It is also required to list the linear order, called a “spine,” in which the elements of the book should be presented. It optionally defines alternative nonlinear navigation paths through the files of the book, called “tours” and “guides.” Figure 4 shows an excerpt from a sample package file.

After determining the books that are available, based on the presence of .opf files, the BookChooser parses each book’s package file and stores its metadata in an object representing the book. Once the books are loaded into the BookChooser, the abstract interface initializes the GUI and AUI. The GUI presents the user with a list of book titles and authors extracted from the metadata, and both the GUI and the AUI respond to appropriate book selection events. When the user selects a book, the

```

<?xml version="1.0"?>
<!-- Taken from Test Suite for the Open eBook document version 1.0
  Author:    Garret Wilson <garret@globalmentor.com> -->
<!DOCTYPE package PUBLIC "-//ISBN 0-9673008-1-9//DTD OEB 1.0 Package//EN" "oebpkg1.dtd">
<package unique-identifier="oeb10testsuite">
  <metadata>
    <dc-metadata>
      <dc:Title>Open eBook 1.0 Test Suite</dc:Title>
      <dc:Creator role="aut" file-as="Wilson, Garret">Garret Wilson</dc:Creator>
      <dc:Rights>Copyright &copy; 1999 Garret Wilson, GlobalMentor, Inc...</dc:Rights>
      <dc:Description>Test suite for validating the OEB spec.</dc:Description>
      ...
    </dc-metadata>
  </metadata>

  <manifest>
    <!-- OEB Documents-->
    <item id="foreword" href="foreword.html" media-type="text/x-oeb1-document" />
    <item id="headings" href="headings.html" media-type="text/x-oeb1-document" />
    ...
    <!-- PNG Images -->
    <item id="mandelbrot.png" href="mandelbrot.png" media-type="image/png" />
    ...
  </manifest>

  <spine>
    <itemref idref="foreword" />
    <itemref idref="headings" />
    ...
  </spine>

  <tours>
    <tour id="alphatour" title="Alphabetical Tour of Element">
      <site title="Element &lt;a>" href="links.html#a" />
      ...
    </tour>
  </tours>

  <guide>
    <reference type="toc" title="Table of Contents" href="toc.html" />
    <reference type="foreword" title="Foreword" href="foreword.html" />
  </guide>
</package>

```

Figure 4. Sample package file for an OEB-standard book. It specifies the files comprising the book as well as different orders in which the material may be presented.

abstract interface checks the metadata of the selected book and loads and parses the document indicated as first in the “spine” of the book. This document is then passed in turn to the concrete interfaces, and the book chooser is closed.

The package files (and all other XML files handled by the system) are parsed using the Document Object Model (DOM) (World Wide Web Consortium, 1999b) and Simple API for XML (SAX) (Megginson Technologies, 2000) standards. The current implementation of the reader uses IBM’s XML4J 2.0.15 parser (IBM Corporation, 2000c; Maruyama et al., 1999) to accomplish this.

VII. Graphical User Interface (GUI) Architecture and Implementation

JEditorPane

As was mentioned earlier, the actual content of an OEB-standard document is represented as XML files. However, this is a bit misleading. In reality the tag set used is essentially that of HTML 4.0 (World Wide Web Consortium, 1999d) and level-one Cascading Style Sheets (CSS) (World Wide Web Consortium, 1999a). Since the project guidelines called for leveraging off-the-shelf software as much as possible, and since

writing a new HTML renderer would be a daunting and unrewarding task, one of the earliest decisions was to use an existing rendering engine. Several renderers were considered.

Because the rest of the system was to be implemented in Java, renderers written in other languages, such as Amaya and Gecko (the rendering engine for Mozilla), which would have required writing an interface to cross the language boundary every time the system required access to the renderer, were ruled out.

The Java JEditorPane Swing component (Sun Microsystems, 2000a; Eckstein et al., 1998) supports HTML 3.2 (World Wide Web Consortium, 1999c) and a subset of level one of the CSS standard. While it provides suboptimal HTML rendering, its ease of integration and use, and the hope that its wide adoption is likely to lead to its being developed into a more full-fledged renderer, led the team to adopt it for NOVELLA. Set in read-only mode (as its name implies, this component is actually intended for providing editor functions), it provides an adequate, if limited, core for the reader.

The Paginator

Unfortunately, using an HTML renderer presented a serious problem. Web browsers have adopted the paradigm of a continuous scrolling frame for HTML files that will not fit on screen. In contrast, electronic book readers have been designed almost universally, to present documents in the familiar page-by-page manner, which was assumed to be the paradigm for this system as well. Thus it was necessary to devise a method of controlling the presentation of text to reproduce the sensation of pages.

One might suppose that it would be enough to hide the scroll bar and have the page-forward and page-backward buttons send page-down and page-up events to the JEditorPane. However, when an HTML renderer displays a document, it generally makes no attempt to avoid clipping a line of text at the top or bottom of the viewing pane; nor does it make any concessions for a bottom margin. The left-hand pane of Figure 5 depicts this unacceptable situation. Clearly, a more robust solution was necessary.

The Paginator class is responsible for pre-rendering a chapter file, determining where to place page breaks, and dividing the document at these points. Since changing the font size used in the viewer changes the amount of text that can appear on screen at one time, this process may occur dynamically at any time. One reason sys-

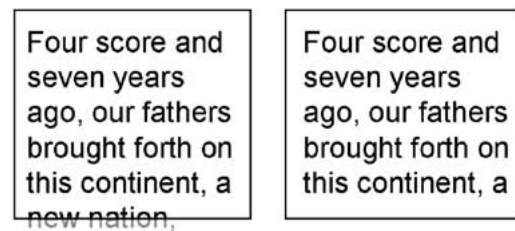


Figure 5. The box on the left shows an example of a scrolling pane of text before pagination, with the clipped region shown in gray. The box on the right shows the result of pagination.

tems such as the SoftPress SoftBook limit the number of usable font sizes is that they appear to use documents that have been pre-paginated at the available sizes and do no on-the-fly formatting at all.

The Paginator works by first rendering the chapter off-screen in a single scrolling JEditorPane sized to the dimensions of the displayed pages. The Paginator steps through the content of the chapter finding the cutoff position for page breaks. This is accomplished by finding the character offset of the final character displayed in full in the off-screen buffer, then advancing the following line to the top of the buffer and repeating the process until the entire chapter has been examined. It stores the resulting character offsets in an array.

Once the page breaks have been determined, the system fragments the scrolling, off-screen pane into a series of page panes. This in itself proved to be a significant challenge, as the JEditorPane component does not implement the clonable interface used to produce a deep copy of a Java object. The workaround is unfortunate: a new JEditorPane is created, and the entire chapter file re-parsed, for each page of the chapter. Once the file is paginated, the JEditorPane holding the first page is displayed in the viewer.

It is important to note that pagination occurs after the HTML has already been rendered to styled text. If the Paginator were to operate on raw HTML it would be forced to consider what tags were open at the point that it cut off a page, close them for the current page and reopen them on the next. In the case of complex tag structures, like tables, this could inappropriately change the appearance of the rendered output.

The FontSizeChanger

Another unfortunate result of using the JEditorPane component as the renderer is that the seemingly simple task of changing the size of the fonts used for display is unduly complex. The natural method for changing font sizes would have been to rewrite the StyleContext

class in `JEditorPane` so that its `getFont` method would return a `Font` object modified by a scaling factor set according to the currently selected size, thus changing the size at which every font in the document is rendered. Unfortunately this technique fails for two reasons. First, the class structure that does the rendering is several levels deep, necessitating either a change in a large number of class files or else overwriting the existing default classes in order to use the new `StyleContext`. Second, the design of the `JEditorPane` appears to contain a design loop, making it difficult to determine at which level the font size is actually stored, and thus which class's `getFont` method would have to be overwritten.

In the end, the technique used was to apply a new default CSS style sheet to the `JEditorPane` setting a new base font size. When changing the font size, the new style sheet is applied, and the `Paginator` is called to reflow the text and redisplay the pages. Since a new style sheet is generated on the fly for each font-size changing event, this technique does not rely on some fixed number of predefined style sheets. Unfortunately, besides being somewhat awkward, this technique does not work very well, due to another bug in the `JEditorPane` class. The method `setBaseFontSize` in the class `StyleSheet`, which is the container for the style rules, does not work properly. Tags whose sizes are calculated relative to the base font are not properly recalculated with respect to the new base font. Text contained in heading tags, for instance, does not change size as it should. Hopefully, however, these problems will be addressed in a future release of the swing libraries.

Annotations

Providing a flexible, extensible, annotation scheme was one of the principal challenges of the NOVeLLA project. The first problem was how to tie a location in

the text on screen to a location in the document when storing an annotation, and then how to reverse the process when retrieving the annotation. Java uses character offsets to indicate positions in the (rendered) text contained in a `JEditorPane`, and it became apparent that this was the most straightforward measure to use as annotation positions. Because these offsets are the values returned from `JEditorPane` text-selection events, they can be used directly as returned when the user selects a block of text to annotate. They are used in the same way to tell the `JEditorPane` where to put the highlights and annotation marks when displaying annotated text. Most importantly, these offsets, which are relative to the beginning of the HTML file being rendered, are stable with respect to changes of font size and other rendering characteristics. Any scheme relying on page-relative references would fail in this regard. (It is interesting to note that with some of the current systems, annotations made at one point size are not available when the document is viewed at another size.)

A second problem was how to store annotations separately from the actual book files. To that end, a custom XML tag set is used to store annotations on the file system. An annotation file is broken into three subsections corresponding to the three types of annotations. A highlight is represented by its starting and ending offset in the chapter text. A text note is stored as an offset location together with a string containing the text of the note. These strings are Java-style, and so can contain arbitrary Unicode. An audio note consists of the location of the note and the name of the file containing the digitized audio. Figure 6 shows a sample annotation file.

One annotation file is stored for each chapter file in an OEB package. The annotation file's name is formed by appending the extension `.annotation` to the underlying chapter file name. Whenever a chapter file is read in, the corresponding annotation file, if it exists, is read as well.

```
<?xml version="1.0"?>
<annotation_set>
  <highlight_set>
    <highlight start="1301" end="2984"/>
  </highlight_set>
  <text_note_set>
    <text_note offset="66" text="Implementation details?"/>
  </text_note_set>
  <audio_note_set>
    <audio_note offset="13" path="file:/C:/oeb/RIAO2000/riao0.html"/>
  </audio_note_set>
</annotation_set>
```

Figure 6. A sample annotation file. Highlights are denoted by beginning and ending offsets within the chapter, while text and audio notes are represented by an offset position together with the text of the note or the name of the file containing the audio stream.

The original intention had been to indicate text and audio notes using an overlaid mark or icon at the appropriate position in the text. However, this requires the ability to obtain the pixel location (position in the *view*) for a given offset location (position in the *model*). Unfortunately, this was ultimately unsuccessful due to an apparent bug in the `JEditorPane modelToView` method. The coordinates returned by this method appear to be offset several pixels to the left, with the magnitude of the offset varying inconsistently with the font size.

In the current version of the system, highlights and annotations are displayed using the `HighlightPainter` provided by `JEditorPane`. While text and audio notes are denoted with green and red underline markings, these are actually implemented as narrow highlights, generated by a customized version of `HighlightPainter`. Ordinary highlights are presented using the `DefaultHighlightPainter` with a light yellow color setting.

VIII. Audio User Interface (AUI) Architecture and Implementation

The Audio User Interface encompasses the automatic speech recognition-based control interface, the speech synthesis-based ability to read the book aloud, and the ability to store digitized audio annotations. Speech recognition and synthesis are implemented using the Java Speech API (JSAPI) from Sun Microsystems (Sun Microsystems, 2000c). The digitized audio notes are implemented using the Java Media Framework (JMF) (Sun Microsystems, 2000b; deCarmo, 1999). As with the implementation of the GUI, the decision to use off-the-shelf components made it possible to build a powerful system in very little time. However, it again led to unforeseen complications as the components were stretched beyond their designers' visions, or simply failed to perform to specification.

Speech Recognition

As stated above, the speech-recognition features of NOVeLLA were implemented using the Java Speech API (JSAPI) (Sun Microsystems, 2000c). Within this API, the recognition capabilities are provided by the class `javax.speech.recognition.Recognizer` and its supporting classes. While the API is specified by Sun, it does not provide an implementation. NOVeLLA was built around IBM's Speech for Java (IBM Corporation, 2000a). This implementation is in the form of a

set of Java objects which interface with a set of DLLs which in turn drive IBM's ViaVoice Millennium Edition, which must be installed on the system. While this particular implementation restricts the reader to running only under Microsoft Windows, the code in NOVeLLA is written purely in terms of the JSAPI, and therefore should be directly compatible with any implementation of the API on any platform.

Though it is possible to use the recognizer directly in full-recognition mode, it is also possible to restrict the range of recognized terms to a pre-specified grammar. NOVeLLA takes advantage of this feature as it improves speaker-independent recognition considerably. Grammars, which are specified in the Java Speech Grammar Format (JSGF), are BNF grammars (Naur, 1963) augmented with rule names and public/private flags. They are typically read from a text file, though they also can be modified directly using the JSAPI. Figure 7 shows part of NOVeLLA's main grammar file, including some of the rules that are in place when the main viewer window is active.

Once a grammar is built, it can be dynamically enabled or disabled, as can individual rules. This makes it possible to use different grammars in different contexts. NOVeLLA makes use of this feature in several ways. For instance, the main viewer and the book chooser use different grammars, each of which contains navigation commands appropriate to the context. For example, the main grammar contains commands such as "next page" and "begin recording" that would be inappropriate in the chooser. Similarly, when the user begins dictating a text note, the full ViaVoice grammar is enabled (since the note is free-form text) while all rules in the NOVeLLA grammar are disabled, except for the rule used to stop recording.

In addition to the method to load a grammar from a file, the JSAPI provides methods to add rules programmatically, one by one. NOVeLLA uses this method to advantage when the book chooser is activated. After parsing the metadata of all the available books, the system not only displays the title and author in the chooser, but also adds a rule to the grammar for each title. Thus the user can select a book by saying the title without the system having to switch into general recognition mode with its inherent loss of precision.

Text-To-Speech

In addition to speech recognition facilities, the Java Speech API also provides text-to-speech facilities,

```

grammar bookbrowse;

public <prevpage> = Page Up           {prevpage}
                  | Previous Page     {prevpage};

public <nextpage> = Page Down          {nextpage}
                  | Next Page         {nextpage};

private <reading> = (talking | speaking | reading);

public <startspeech> = (Start | Begin) <reading> {startspeech};

public <stopspeech> = Stop <reading> | Shut up {stopspeech};

public <openbook> = Open [another | new | a new] book {openbook}
                  | Change [books | book]           {openbook}
                  | New book                         {openbook};

public <find> = Find (a word | a passage | a phrase) {find}
               | Search the (page | document | chapter) {find};

```

Figure 7. Fragment of a sample Java Speech API grammar file, specifying some of the verbal commands accepted by the main viewer window.

encapsulated in the class `javax.speech.synthesis.Synthesizer`. The basic synthesis routines are very simple. The principal method used is `speak`, which takes a string to be spoken. For brief prompts and responses, simply calling this method is adequate. However, when reciting the contents of the book, using this direct approach introduces problems. If speech is initiated in the same thread as the main viewer, the system will not respond to any commands, spoken or clicked, while it is speaking.

Therefore, in order to enable the user to issue commands during synthesis, it was necessary to wrap the synthesizer in a separate thread implemented in the Dictation class. When this thread is created, it is provided with the text content of the current page as seen on the screen, extracted from the active `JEditorPane`. When the entire page has been dictated, unless stopped by user command (issued from the GUI or by voice command), the dictation thread will direct the abstract interface to advance the viewer to the next page, retrieve the text for that page, and continue speaking.

Because of the structure of the `Synthesizer` interface, it proved difficult to pause the dictation effectively. When the user issues a command to stop speaking, and then prompts the system to begin speaking without first changing pages or chapters, the desired behavior is to pick up where the synthesizer left off. Since it is not possible to query the synthesizer for its current location in the text when it is stopped, it is necessary to pause the synthesizer and place the dictation thread containing it in a state that will keep it alive. When the user directs the system to begin speaking again, if

the page has not changed, the dictation thread simply resumes the paused synthesizer.

If the user changes pages or chapters while the system is speaking a page, the synthesizer is paused, all data on the speech queue is flushed, and the dictation thread is directed to exit gracefully. A new speech thread is then created with the new data placed in its queue. If the navigation command is issued while speaking is paused, then the same process is followed except that the new dictation thread remains paused at the end. Because in either case the speech has been paused at the point that the new dictation thread is started, it is necessary to keep track of whether that pause was user or system driven.

Digitized Audio

As was the case with font-size changing in the GUI, audio notes, which were expected to be a relatively simple feature to implement, proved to be one of the major challenges of the project. This was due to limitations in the underlying component APIs and implementations.

The IBM implementation of the JSAPI recognizer (IBM Corporation, 2000a) implements a method that returns the digitized audio stream as an `AudioClip` object. It seemed natural, therefore, to record an audio note in essentially the same way a spoken text-note was captured. The system would, however, load an empty grammar (except for the “stop recording” command) to disable actual recognition, grab the resulting `AudioClip`, and discard the (empty) recognition result.

Unfortunately, it turned out that an AudioClip is actually a complete applet, together with its own user interface for starting and stopping play of the audio stream. This in itself would not have been a problem, as that user interface was adequate. However, this particular subclass of the Applet class is not serializable, and it does not implement any method to convert the audio stream to some other, serializable format, such as those provided by the Java Media Framework. This means that there is no way to store the clips on the file system, making them useless for capturing audio notes, which must be stored between user sessions.

The solution was to use the Java Media Framework (JMF) directly. This library provides classes that manipulate many types of media data, including a straightforward framework to capture an audio stream and save it to disk. Using the JMF, however, led to another major problem. The Java Virtual Machine treats the computer's microphone as an exclusive resource, meaning that only one object can take input from it at a time. When the microphone is already allocated to the ViaVoice JSAPI recognizer, as it is when NOVeLLA starts up, JMF is unable to gain access to the audio stream. Unfortunately, there is no method to force ViaVoice to release the microphone temporarily. In order for audio notes to be recorded, therefore, it is instead necessary to temporarily de-allocate ViaVoice completely while JMF objects capture the audio stream. A new recognizer is started after the capture of the audio note is completed. Unfortunately, because the recognizer is shut down entirely while the audio note is being captured, this precludes the user from giving a voice command such as "stop recording" to halt the capture of an audio annotation. This is an unfortunate gap in the voice command interface.

A better solution will require the Java Speech API to be rewritten. It should specify a method for acquiring the audio stream of a recognized passage in a format compatible with the JMF conversion and serialization routines. In this way the audio note capture could be accomplished within the recognizer framework, as desired, while allowing the resultant streams to be stored for future retrieval.

IX. Conclusions and Future Research

As discussed above, NOVeLLA is the first of its kind: an electronic book reader that is poised to take advantage of technologies on the immediate horizon, built

entirely with off-the-shelf software technologies and using open standards. It provides a number of features not found in current, proprietary, commercial systems including:

- Support for the new Open Ebook Publication Structure Standard.
- Flexible, on-the-fly reformatting of text, allowing the user to change font size at will.
- User annotations stored in a flexible, extensible, divisible format. This format will soon allow the user to share annotations easily with others reading the same document.
- A rich audio user interface utilizing both automatic speech recognition and text-to-speech technology. This interface provides functionality to both the convenience user and to visually and physically impaired users.

Performance

One obvious question to ask is, how well did the off-the-shelf speech technologies perform? No formal testing was performed, so all conclusions are anecdotal. The general answer lies somewhere on the spectrum between "terrible" and "startlingly well," depending on circumstances and the user's expectations. The automatic speech recognition (ASR) and text-to-speech (TTS) technologies, both embodied in IBM ViaVoice, are of the typical level now available for desktop computers. Response time was quite good. In general, the time needed for command recognition and response by the system was negligible, providing essentially real-time responsiveness (i.e., rarely was there a delay of as much as half a second). The quality of generated speech was also adequate for general comprehension.

However, the requirement of a high-quality headset-mounted microphone makes the system fall short of the open-environment model proposed in the scenario. Further, while ASR accuracy was generally adequate, even without training, it was far more reliable for some users than for others. Repeated failed recognitions were generally frustrating enough to get the user quickly reaching for the graphical interface, and would be more frustrating for the impaired user. Certainly the usability of the application will scale directly with improvements of the ASR and TTS modules. The advantage of the design of NOVeLLA is that any improved system compliant with the Java Speech API could be incorporated directly.

Future Work

The current software implementation is a fully functional prototype for reading electronic books formatted according to the OEB specification, and has been designed from the outset with extension of its capabilities in mind. Some paths of development are already well defined, and have been described in the fictional user scenario in the introduction. In many cases the architectural infrastructure for such extensions is already in place. Other paths will require significant additional thought, research, and user testing. We describe some extensions in each category below.

Enriched, More Natural, Command Grammar. As is often the case with such projects, the largest part of the effort was expended on implementation and integration of the various components of the system. In the end relatively little time was spent on the development of the command grammar. Therefore, the variations available for many of the commands are somewhat limited, and the syntax fairly stilted and artificial. It would be useful, and not terribly difficult, to enhance the command grammar to make the communications seem more natural to the user. This would also reduce the need to train users in the available commands, which currently requires providing them with some listing of those commands supported in each mode. Since the core grammars are stored in text files external to the actual software, it is trivial to make such extensions, which could even be customized on a per-user basis.

Extension of the Audio User Interface for Users with Impairments. One goal of the project was to create a system in which every action possible using the graphical user interface would also be possible using voice commands. While enough actions meet this criterion to satisfy the convenience user, there are still enough actions that have no audio equivalent to limit the usefulness of the system for people with visual and physical impairments. Many questions remain about how to extend the audio user interface, including: How should the book chooser present the library to a user with visual impairment? How can user with a physical impairment designate a location in the text to place an annotation? And, how should the presence and content of annotations be presented during recitation of the text? While the sorts of dialogues necessary to solve these problems seem cumbersome, it is important to note that they are

in use in practice in many current systems for people with physical disabilities.

Enhanced Book Chooser Interface. The current book chooser represents a significant improvement over those systems that provide, essentially, a file browser, and it functions well in the present environment. However, it is not hard to conceive of a device with a large storage capacity, or a wireless connection to a large repository, in which the user will have to choose among dozens, or even hundreds, of books. This will certainly require hierarchical presentation, ideally organized over several possible axes. How to present and navigate such a hierarchy in the audio user interface raises many questions.

Multi-User Documents and Shared and Searchable Annotations. In the introductory scenario, the fictional executive selected some of her annotations to share with her assistant. This capability is not currently provided by NOVeLLA. However, because annotations are stored external to the documents in a simple, easily divisible XML structure, it would be relatively straightforward to add mechanisms to support this feature. Similarly, while the current search function looks at only document content, it is easy to imagine extending the search function to allow the user to search through annotations as well.

Alternative Types of Annotations. One notable feature of the annotation system is the ease with which new annotation types can be added. The XML schema, as mentioned above, is extensible, and it would not require substantial rearchitecting to support the display and entry of new annotation types.

One obvious new type of annotation would include freehand writing and drawing on the book. Handwritten notes could be rendered into text via a handwriting recognizer. This would not involve any substantial change to the current model, once a suitable recognizer was identified; the process of input and retrieval would be essentially the same as for spoken text notes. However, in the case of notes or diagrams that the user wishes to store in their original form, significant complications arise. When the font size of the book is changed, how should the annotation be rendered? In the case of handwritten text there is not too much of an issue. However, a figure may include arrows or other pointers to the underlying text. Determining how to position and scale the figure to keep these pointers properly aligned is a

daunting challenge. While some current commercial e-books allow freehand input, such annotations are linked to one particular rendering of the text, and disappear when the text is viewed in another font size.

It is even possible to imagine a system in which annotation types could be added without further modification to the software. Such a system would require specifying a plug-in style API (in the form of a Java interface) that any new annotation class would have to implement.

Support for Tours and Guides and Hyperlinks. As discussed above, in addition to the linear spine, the OEB standard allows for non-linear navigation paths through a book in the form of “tours” and “guides.” NOVeLLA could easily support changing from the default spine to a non-linear path, but there are questions about what button/command behavior would be suitable. This is one of several instances where the weakness of the OEB standard, which says nothing about intended behaviors, makes it difficult to take full advantage of document structure. Support for embedded hyperlinks (internal to the document), also allowed by the standard, raises similar issues.

Support for Pre-recorded Audio (Books-on-Tape). Every electronic book can be read aloud to a reader using the TTS engine in Speech for Java. Unfortunately, speech synthesis still produces speech that is not always easy to understand, and rarely sounds completely natural. The NOVeLLA annotation mechanism already allows for digitized audio notes to be tied to particular blocks of text. By associating an MP3 stream (which is supported by the JMF) with each paragraph of a document, it would be straightforward to implement a book-on-tape mode in which the spoken text could be synchronized with the presentation of text. With little additional effort, this feature also could be used to implement children’s read-along books and follow-the-bouncing-ball type songbooks.

Acknowledgments

This project was undertaken as part of the Harvey Mudd College Computer Science Clinic Program, an industrial collaboration program, while the last four authors were students at the college. They, the first author (who acted as faculty advisor to the team), and the college, would like to thank IBM Almaden Research Center

(ARC) for sponsoring the project, and for providing the time of the second author, then an employee of the ARC, as project liaison.

References

- Adobe Systems, Inc. (Ed.) (2000). *PDF Reference*. 2nd ed., Version 1.3. New York: Addison-Wesley.
- DeCarmo, L. (1999). *Core Java Media Framework*. Prentice Hall.
- Eckstein, R., Loy, M., and Wood, D. (1998). *Java Swing*. Sebastopol, California: O’Reilly and Associates.
- GlassBook, Inc. (2000). Glassbook Home Page. <http://www.glassbook.com>
- Graham, I.S. and Quin, L. (1999). *XML Specification Guide*. New York: John Wiley & Sons.
- IBM Corporation (2000a). Speech For Java. <http://www.alphaworks.ibm.com/tech/speech>
- IBM Corporation (2000b). ViaVoice Home Page. <http://www.ibm.com/software/speech/>
- IBM Corporation (2000c). XML For Java. <http://www.alphaworks.ibm.com/tech/xml4j>
- McCarthy, J. (1963). *Towards a Mathematical Science of Computation. Information Processing (1962)*. Amsterdam: North Holland, pp. 21–28.
- McLaughlin B. and Loukides, M. (2000). *Java and XML*. O’Reilly and Associates.
- Mariyama, H., Tamura, K., and Uramoto, N. (1999). *XML and Java: Developing Web Applications*. Addison Wesley.
- Meggison Technologies (2000). Simple API for XML 2.0. <http://www.meggison.com/SAX/>
- Naur, P. (Ed.) (1963). Revised report on the algorithmic language Algol. *Communications of the ACM*, New York, 6(1):1–17.
- NuvoMedia, Inc. (2000). Rocket E-Book Home page. <http://www.rocketebook.com>
- Open Ebook Foundation (OEB) (1999). OEB Publication Structure Specification: <http://www.openebook.org/OEB1.html>
- Page-Jones, M. and Constantine, L.L. (1999). *Fundamentals of Object-Oriented Design in UML*. Addison-Wesley.
- Pfenning, F. and Elliot, C. (1988). Higher-order abstract syntax. In *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*. New York: ACM Press, pp. 199–208.
- Softbook Press, Inc. (2000) Softbook Home Page. <http://www.softbook.com>
- Sun Microsystems, Inc. (2000a). Java 1.3. <http://java.sun.com/j2se/1.3/>
- Sun Microsystems, Inc. (2000b). Java Media Framework. <http://java.sun.com/products/java-media/jmf/index.html>
- Sun Microsystems, Inc. (2000c). Java Speech API. <http://java.sun.com/products/java-media/speech/>
- World Wide Web Consortium (W3C) (1999a). Cascading Style Sheets, Level 1: Specification. Cambridge, MA: To Excel.
- World Wide Web Consortium (W3C) (1999b). Document Object Model (DOM): Level 1 Specification. To Excel.
- World Wide Web Consortium (W3C) (1999c). HTML 3.2: Reference Specification. To Excel.
- World Wide Web Consortium (W3C) (1999d). HTML 4.0: Specification. To Excel.