The eText Engine: An Extensible, Object-Oriented Hypermedia Publishing System

Rohit Khare and the eText Group
The eText Project at Caltech
MSC 256-80, Computer Science
Pasadena, CA, 91125
fax : (818) 792-4257
e-mail: khare@cs.caltech.edu

April 11, 1994

ABSTRACT

The eText Engine is a user-friendly, object-oriented, compound document editor for creating interactive hypermedia documents, specifically interactive textbooks. The Engine supports documents that: can contain arbitrary objects instantiated from dynamically bound classes at run-time; are stored using open-standards component data formats, such as Microsoft RTF; and can be automatically converted to HTML for deployment on the World-Wide-Web. This openness and extensibility is integral to the design of eText, afforded by the introduction of Objective-C "protocols" and a flexible application kernel. The paper presents the design architecture and implementation of the system.

KEYWORDS: Hypermedia, Publishing, Object-Oriented Design, User Interface Design

1 INTRODUCTION

In our efforts towards publishing electronic textbooks, described in [9], we designed and implemented the eText Engine, a new system for authoring, editing, and publishing hypermedia documents. Reflecting on the content requirements of our textbooks, we identified three central design concerns:

- *Documents.* Our ideas about a textbook do not fit into the model of "cards," "screens," or "timelines" dictated by the vast majority of current hypermedia systems.

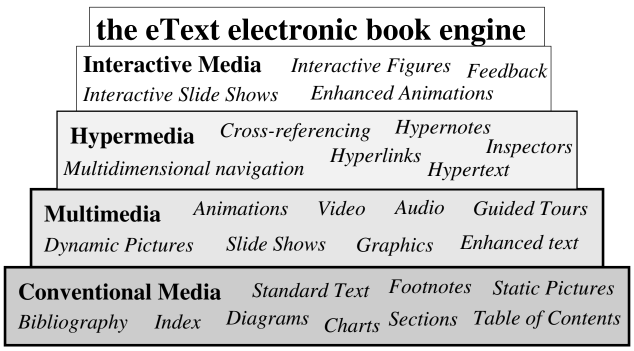

Figure 1: *The Media Pyramid*

- *Objects.* Our goal of creating *interactive figures* requires the ability to "plug in" custom coded simulations, visualizations, and other interactive demonstrations, also not afforded by most systems.

- *Open Publishing.* Even while pursuing a custom solution to the other two goals, we must be able to publish a subset of our documents to widely used, open standards such as Hypertext Markup Language (HTML) [3] or Microsoft Windows [12] .

In addition, the system was designed with ease-of-use in mind: it is user-friendly *and* powerful. Our goal was to make the authoring of complex, *interactive* hypermedia documents as easy and natural as modern WYSIWIG word processing. In the following section we will present the architectural design and implementation of the eText Engine.

2 ARCHITECTURAL DESIGN

2.1 Requirements Model

The specific "textbook" we are constructing teaches the Archetype [15] method of sequential and parallel software development. From a hypermedia perspective, the relevant insights are that this domain is

- *Coarse-Grained.* The finest unit of reference in our reference-cum-textbook was a *document*, from one to ten pages long.

- *Hierarchical.* The Archetypal method is based on self-similarity at all levels of design. At each level of the hierarchy we have five similar aspects to consider, and this simplifies the nature of the "navigation problem."

- *Multimodal.* Documents contain several different kinds of media, and media presentation should be personalized according to use preferences. Our model of a media hierarchy is presented in Figure 1.

These three observations helped us focus on systems that 1) were document-centric, 2) did not need extensive navigational visualization tools, and 3) allowed flexible extensibility to drop code in for Interactive media.

2.2 Previous Work

Why does the world need Yet Another Hypermedia System? When deciding to build the Engine, we compared our requirements to a wide variety of existing systems. We found that the nascent hypermedia marketplace has so far only yielded narrow, "multimedia" systems based on temporal metaphors and sophisticated hypertext systems. None were extensible enough to incorporate custom code (Interactive media)[1] or particularly adept at linking and navigation (Hypermedia) . In the research literature, we found previous work with compound-documents such as *Andrew* [4], document-centric hypermedia such as *IRIS* [11], and object-oriented customizability such as *MediaView* [13], but none that combined these threads together.

During the development of the eText Project, Mosaic [2] exploded onto the hypermedia scene, and we rapidly adopted HTML as our *lingua franca*. However, the limited ability of Mosaic to incorporate interactive simulations validates our design philosophy of creating an maintaining a fully interactive document tree with eText that can automatically be distilled to a "static" format for use in a completely different context (see Section 2.5).

[1] Some commercial systems, notable Macromind Director, Claris HyperCard, and Microsoft Multimedia Viewer can call functions in external dynamically linked libraries, but provide marginal support for developing complete objects with complex user interfaces

2.3 Documents

Most hypertext/hypermedia systems and models have predicated the use of databases as the fundamental expression of storage and manipulation. This paradigm encourages the creation of small-scale "nodes," as evinced by the field's experience with "cards" [8], "stacks" [5], and "frames" [1].

As we pursued the design of our textbook, we saw that these small-scale metaphors were inappropriate for our applications. The natural quanta of a reference book or a personal hypermedia reference system [14] appears to be document of one to ten pages. Furthermore, documents, as far as possible, should be independent. A document should exist as its own corpus of data files, not in a shared database, and links, anchors, and indexing should be adapted to the scope of individual documents. We derived the following design requirements for our model of a Document:

- *Atomic.* Each document is complete. It can be moved whole. It has a fixed set of keys that describe it for indexing purposes (author, title, comments).
- *Associative.* Each document can be associated with others at a macro-level. Hierarchy is created by defining a parent link for each document; additionally, any number of documents can be associated as Peers.
- *Component Data Model.* Each document must be decomposed into component data for each kind of annotation or media supplement; more generally, the storage model must allow storage on a per-component basis.

2.4 Objects

One of our group's inspirations was the interactive graphics paper presented in [13] and the system in which it was created, *MediaView*. The example of placing an interactive, "live" simulation of a lighting model into a paper on lighting models directly motivated our design requirement of extensibility. Dynamically binding custom objects into the system affords the creation of truly interactive media, as mentioned in Figure 1.

In fact, we can and did implement not just the top layer, but the top three layers as custom objects within the Engine. Displaying pictures and providing hyperlink buttons, for instance, are not compiled into the Engine, but are loadable classes that exploit several well-defined interfaces. The Engine can incorporate any kind of object into its model, so long as that object fulfills the following responsibilities:

- *Drawing.* Objects must respond to queries about their size in the text layout and provide a method that the system can call to have it render itself

- *Event-Handling.* The Object must handle any human-interface events in its specified rectangle.

- *Representation and Storage.* Each Object must be able to write a reference to itself out to the text stream (which may be RTF, HTML, ...), and is offered the opportunity to store component data (images, video, etc) within the corpus.

2.5 Open Publishing

Embarking on the creation of an ambitious, large-scale hypermedia content base necessarily incorporates the risk of being tied to a fixed software platform. While Standard Generalized Markup Language (SGML) and emerging standards for multimedia data have reduced the risk for the actual data, few formats yet exist for preserving the encoding of links and formatting. (see [10] and [7] for a discussion of these issues). For our work, we are again tied to a niche platform, but we have designed it to be capable of emitting documents in many different formats.

A consequence of the first and second goals of our Document design, Atomicity and maintenance of discrete Component Data files, suggested a Compound Document Architecture (CDA) of a formatted text stream with objects embedded within (as opposed to, say, a frame-oriented CDA, where objects are bound to areas of a 2D "page") ETFD, the native Enhanced Text Format Directory creates a corpus within a Unix directory with files within for navigational information, the formatted text stream (derived from RTF), and any data maintained by objects within the document. An HTML emitter is designed similarly, where the formatted text stream is converted to analogous HTML codes and component objects are responsible for providing conversion routines for their own data. The latter is yet another benefit of the encapsulation of media behavior into external objects.

2.6 User Interface

To make the object system work seamlessly, the system will have to offer some way for an object to optionally express its own unique interface and controls. In other GUIs, an embedded object might reflect control by replacing the menu bar with its own menu, but can offer little else. The key insight here is the NeXTSTEP paradigm of inspectors. An Inspector panel always reflects controls applicable to the user's current selected object. When the user selects an external Object, the system should offer it the opportunity to provide a panel of controls reflecting the current state of the Object. An example is in Figure 2 Furthermore, the system needs to provide the "glue" between these objects and the power of the underlying operating system and GUI facilities. Primary examples include:

- *Cut-and-Paste, Drag-and-Drop.* The system should arbitrate the encoding, decoding, and data format conversion of objects onto the user's pasteboard.

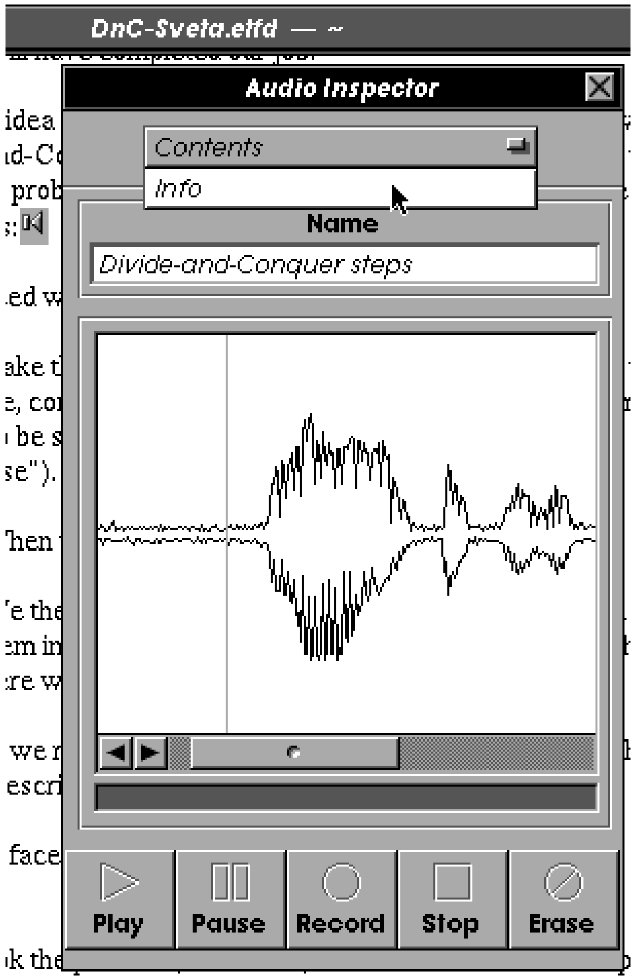

Figure 2: *The user has selected an audio icon in the text, and the inspector reflects the current audio selection. The pop-up menu is a list of the different control panels the Audio object has published to the Engine.*

- *Search and Replace.* The system should offer component objects the ability to respond to user queries about search strings and indexing.

- *Undo and Redo.* The system should offer an architecture for objects to provide change histories to the user, to enable multiple-level undo and redo.

Other system support requirements follow from OS-level considerations, such as the filesystem. For example, the Engine can read documents that are read-only without requiring write permission for lock files, etc. Document saving, retrieval, and reversion also need to work correctly with the filesystem; edits to the document must be kept in memory until a "save" command.

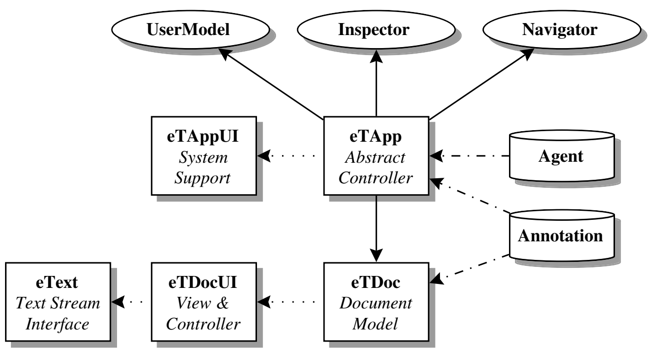

Figure 3: *The eText Kernel Wiring Diagram*

3 IMPLEMENTATION

The actual implementation of the architecture set forth in Section 2, shown in Figure 3, was influenced by three factors:

Microkernel Architecture. One lesson learned from the last decade of operating system development is the trend toward microkernels, of building a reliable, efficient, compact set of core services and cleanly separating user and system facilities layered on top. Similarly, eText was designed as a document-processing kernel that provides a Document metaphor and certain services (e.g., `registerAnnotation:`). All of the multimedia, indexing and searching, and hypermedia linking was implemented separately.

NeXTSTEP Application Kit. The structure of the application and division of responsibilities between the Application and the Document was heavily influenced by the NeXTSTEP API. The concept of First Responder, for example, determines that the command to `open` a document is routed to the `eTAppUI` but to `save` is routed to the `eTDocumentUI` in the foremost window.

Objective-C and "Protocols". Objective-C affords dynamic binding, delegation, and protocols, all of which are reflected in the design. Loadable classes, and the code newly-loaded Annotations use to register their presence are critically dependent on all three. A protocol is a list of methods alone, not a specific implementation or position in the class hierarchy; thus the Engine can check if a supplied object `conformsTo:` the `<Annotation>` protocol before loading it.

3.1 eText Kernel

In Figure 3, the Kernel consists of the square classes. The dotted arrows indicate private interfaces; communication between objects is not an $N \times N$ proposition, but is carefully delimited within logical modules. Application-wide services are depicted in the ovals. Finally, external loadable classes like Annotations and Agents negotiate with the Application and Document to interact indirectly with the rest of the application.

Application. The application is responsible for opening, printing, and managing several Documents. It also maintains the interface with the OS, the windowing system, and the menus. The application can also load new classes and publishes several registration hooks for such classes to add menu items and declare which data formats each can process. In turn, the Application works with the document to create and manage Annotations so that, for example, a message to paste audio waveform data is converted into a new eTAudio annotation initialized against that waveform.

Document. The Document is responsible for maintaining the storage model, as well as lower-level user-interface issues like scrolling. The Document initiates and manages the reading and writing processes. Before writing out a document, it notifies all of its component object, prepares a text stream, then asks each component to write out any component data. Upon completion, the Document garbage collects the corpus directory to remove undeclared component files. This process works with the model of ETFD, HTML, and Microsoft Multimedia Viewer.

3.2 eText Services

Navigation. This service object provides mappings between Document objects being read and on disk and 32-bit unique IDs. In addition to that, it maintains its own user interface and panels for navigating a user's collection of eText documents. Since the Link button Annotation chooses to notify the Navigator when links are followed, the Navigator can also maintain a user history, and backtrack by sending `openID:` requests to the application and `highlight:` requests to individual bookmark Annotations.

The Navigator presents a multi-column browser to the user. Generally, a query is executed against the indexed fields of the documents (authors, titles, keywords, comments). Out of the general network formed by the Parent - and Peer-links, the selected documents are collected under a root node, and the graph is projected onto a tree; the tree is shown in the browser. This method appears to be quite compact and efficient; a similar system was developed at Xerox [6].

User Model. The user model service is currently quite simple. It maintains a per-user hash table of associations between queries and responses. An Bookmark discussing Fortran code in a document can ask the user model to return the user's preference for `FORTRAN` and if the return is `NO`, it will collapse the entry. When the model encounters a new query, it asks the user for an appropriate response.

3.3 eText Annotations & Agents

As implied earlier, Annotations and Agents are bound at run-time and can implement arbitrarily complex interfaces and behaviors. Both derive their power from the NeXTSTEP Bundle class, which defines a protocol for loading binary object code and localized interface resources. The various protocols for both have been described elsewhere.

While an Annotation can only be instantiated within a text stream and only deals with events when the mouse is tracking its visible rectangle, an Agent is bound to the entire document and can potentially observe all events. An Agent is displayed in a pane within the document window and has the special power to edit the entire document on the user's behalf. Some sample applications include: autoformatting, glossary lookup (computed links), and project management.

4 CONCLUSIONS

The eText Engine brings together several different threads from contemporary hypermedia system interfaces. It integrates the four kinds of media we identified into an extensible compound-document architecture. The design of the system is also flexible with respect to document formats, allowing interoperablity between different document-based systems.

Experience. Early experience with the engine has indicated that it has met the test of matching state-of-the art in UI design. Experience with creating components, from simple (lines bouncing in a box in a separate processor thread and a slide-show) to complex (bookmarks, links) has validated our decomposition of tasks into Kernel and Annotations. Development has also accelerated as experience and reuse increased. The Annotation protocol has succeeded in minimizing the amount of glue code required compared with the order-of magnitude larger APIs of OpenDoc, Object Linking and Embedding 2.0, or NeXT Object Links.

Future Work. We look forward to exploiting the Agent facility to construct a programming environment to support Archetypal development, PEN. The potential synergy of hypermedia and agent technology seems very promising. In addition, we are working on improving the "plumbing" by building additional document representations (HTML, Microsoft Multimedia Viewer) and making Annotation development simpler still, by providing default event-handling and storage mechanisms.

5 ACKNOWLEDGEMENTS

Special thanks go to Adam Rifkin, for many productive discussions, and to other members of the eText project group: Paul Ainsworth, Svetlana Kryukova, and Rajit Manohar. Last, but not least, thanks to K. Mani Chandy for establishing, leading, and supporting the eText Project.

This research is sponsored in part by the Air Force Office of Scientific Research grant AFOSR-91-0070 and CRPC support for education and parallel scientific applications under cooperative agreement CCR-9120008.

References

[1] Robert M. *et al.* Acksyn. Kms: A distributed hypermedia system for managing knowledge in organizations. *Communications of the ACM*, July 1988.

[2] Marc Andreesen and Eric Bina. *Mosaic User's Guide and Technical Documentation*. Availible on WWW at www.ncsa.uiuc.edu, 1994.

[3] R. Cailliau Berners-Lee, T.J and J.-F. Groff. The world-wide web, computer networks and isdn systems. In *Proceedings of the 1992 Joint European Networking Conference*, number 25, pages 454–459. North-Holland, 1992.

[4] Nathaniel S. Borenstein. *Multimedia Applications Development with the Andrew Toolkit*. Prentice-Hall, 1990.

[5] Claris Software. *HyperCard User's Guide*, 1993.

[6] Paul M. Eng. Xerox draws a map to find lost files. *Business Week*, page 104A, April 11 1994.

[7] Frank Halasz and Mayer Schwartz. The Dexter hypertext reference model. *Communications of the ACM*, February 1994.

[8] Frank G. Halasz. *NoteCards: A Multimedia Idea Processing Environment*. Microsoft Press, 1988.

[9] Rohit Khare and the eText Group. The eText project: Creating electronic textbooks. In *Submitted to Proceedings of the 1994 European Conference on Hypermedia Technology*, 1994.

[10] Gary Marchionini and Gregory Crane. Evaluating hypermedia and learning: Methods and results from the Perseus project. *ACM Transactions on Information Systems*, January 1994.

[11] Norman K. Meyrowitz. Intermedia: The architecture and construction of an object-oriented hypermedia system and applications framework. In *OOPSLA '86 Proceedings*, 1986.

[12] Microsoft, One Microsoft Way, Redmond, WA. *Multimedia Viewer Technical Reference*, 2.0 edition, 1993.

[13] Richard L. Phillips. MediaView: a general multimedia digital publication system. *Communications of the ACM*, July 1991.

[14] Larry Press. Emerging dynabase tools. *Communications of the ACM*, March 1994.

[15] Adam Rifkin. Teaching archetypical design with an electronic textbook. *Proceedings of the 25th ACM SIG CSE Conference*, March 1994.