Prototyping an Interactive Electronic Book System Using an Object-Oriented Approach

Jacques Pasquier-Boltuck, Ed Grossman, Gérald Collaud
Institute For Automation and Operations Research (IAUF)
University of Fribourg
1700 Fribourg Switzerland

Abstract

An Integrated Electronic Book (IEB) represents a complex network of integrated information and "know how" on a given subject. In the design phase of WEBS (Woven Electronic Book System), we soon realized that, because we were describing the IEB in terms of "objects" and "methods," and because we wanted WEBS to be easily expandable and to offer a consistent user interface, we should use an object-oriented development system.
This paper does not fully discuss the concept of electronic book systems, but rather describes our own experiences developing a complex software system with an object-oriented language. The first part explains our choice of an object-oriented language and software framework. The rest describes the software architecture of WEBS, which is a class hierarchy of three layers: a software framework (MacApp) layer; a system-specific layer; and an application-specific layer.

Keywords : Object-oriented programming, application framework, user interface consistency, hypertext, electronic book.

1. Introduction

1.1 Background

At the beginning of 1984, the research project: *The Integrated Electronic Book* [Kohlas,1984] was launched at the Institute for Automation and Operations Research of the University of Fribourg, Switzerland. The achievements made by our small group of researchers, during the project's first four years, are briefly summarized below:

1. We first designed EBOOK3, a software environment or shell for both creating and consulting electronic books on IBM-PC compatibles [Savoy,1987a, b and c]. We were conscious from the beginning of the limitations of such machines (for example, very limited graphics capabilities), but since we wanted our system to be used by as large a group of people as possible, our choice was motivated by market availability and cost.

2. We then used the EBOOK3 shell to author a small set of electronic books on selected subjects in the areas of operations research, programming, and economic theory. Once two fully functional electronic books[1] existed, we set out to gain experience by analyzing the feedback offered by a group of endusers. We presented the books at several formal and informal meetings and used them as a supplement to our course support material.

3. Based on the experience gained with the EBOOK3 system, we created the first design document for WEBS (Woven Electronic Book System), the eventual successor to EBOOK3 [Pasquier-Boltuck and Collaud, 1987].

4. We soon realized that we were describing the IEB in terms of "objects" and "methods" (see Section 2). It seemed only natural that as we set to work programming the first prototype of WEBS in September 1987, we should choose an object-oriented environment; it reflected the basic principles of our conceptual design. In addition, it provided us with modularity, data abstraction, and reusability of code through inheritance. We therefore selected MacApp, an object-oriented software framework for the Macintosh™ [Schmucker, 1986a and b], and Object Pascal, the first available language that could be used with MacApp.

5. Four months after writing the first line of code, a beta version of the WEBS prototype is operational. This working prototype allows us to test and solidify our design ideas, rather than merely describing them.

1.2 Goal and Outline of this Paper

The goal of this paper is not to fully discuss the design issues related to the concept of electronic book systems[2], but rather to describe our own experiences developing a complex software system with an object-oriented language. With this in mind, the following issues arising from our research will be discussed:

- Why we decided to develop WEBS with the help of an object-oriented language.

- How we proceeded with this task.

- What conclusions can be drawn at the present stage of development.

---

[1]The titles of these two books are "Linear Optimization" and "The Application of Markov Chains in Reliability Theory".

[2]The interested reader is referred to the papers of [Savoy, 1987a, b and c], [Pasquier-Boltuck and Collaud, 1987], [Yankelovich and al.,1985] and [Conklin, 1987].

This paper will thus be organized as follows :

- Section 2 briefly describes the fundamental components of an integrated electronic book system.

- Section 3 presents the technological objectives we set while designing WEBS, and the reasoning behind our choice of MacApp and Object Pascal to attain them.

- Section 4 describes the software architecture of the WEBS prototype by sketching the three layers of its class hierarchy.

- Finally, Section 5 enumerates some of the conclusions we have drawn from our work, and contains some suggestions for improvement at various levels.

2. The IEB Concept

An Integrated Electronic Book (**IEB**) represents a complex network of integrated information and "know-how" on a given subject. For example, an IEB on Markov chains might be composed of:

- A set of **text objects** embodying the hierarchy of chapters, sections, and paragraphs which comprises the bulk of any textbook.

- A set of **graphical objects** containing the illustrations, figures, and other pictorial information of the book.

- A set of **modelling objects** including the data necessary to specify various kinds of Markov models.

Each of these objects includes a set of procedures or **methods** which allow for its creation, computation and management. Some, such as the text and graphical objects and their associated methods, are the basic components of any IEB. Others are specific to an IEB's subject; an IEB on linear optimization, for instance, would contain objects that modelled the tools used to perform a sensibility analysis.

A complete IEB is not merely a collection of objects and methods. We believe that an IEB management shell, such as WEBS, should also incorporate at least the following set of capabilities:

1. The IEB shell should allow the user to **distinguish** and **protect** the objects it manages. In the case of WEBS, objects belong either to an author, in which case they are public objects that can be consulted by all users but altered only by the author, or to a reader, in which case they are private objects with private access rights.

2. In order for the objects of an IEB to constitute a manageable and useful "knowledge base," the shell should provide tools for creating index objects[1]. These index objects should allow the user to **find and access** the piece of information in which s/he is interested. They can be compared to the tables of contents and indices of paper textbooks.

3. Finally, the shell should allow its users to **navigate efficiently throughout the various components of an IEB**. In order to achieve this, WEBS uses a hypertext [Conklin, 1987; Yankelovich et al., 1985] construct which we call a **web**. A web imposes a context in which connections can be made between various documents; each new web is a different context. When reading an IEB, two webs can be open at any given time. One belongs to the author, and contains connections that s/he wanted contained in the book. The other is where a reader, presumably studying the material in the text, can make his/her own connections, perhaps generating new insights beyond what the author had considered. A web is composed of a set of **links**, each of which represents a connection between two **blocks**. A block is simply a selection within an object of an IEB; a string of characters within a text document would be one example. These links allow for direct jumps between different parts of an IEB (and indeed, between different IEBs). Figures 1a and 1b illustrate WEBS's implementation of this fundamental notion by showing the effect of the command **Follow Link**. More than one link can connect to a given block; in such a case the user is given a choice of which one to follow. Blocks and links also contain other information, notably the **explainer**, a short user-defined string intended to give a user an idea as to why the block or link exists.

3. Technological Objectives

The EBOOK3 IEB shell was developed by one person in a reasonable amount of time. This was achieved by adopting a very restrictive user interface paradigm, and by totally separating the creation and the consultation processes, with only the latter being wholly interactive [Savoy, 1987a, b and c]. When confronted with the difficulties of extending EBOOK3 into an entirely interactive system with a powerful modern user interface, we decided that a completely new software design would best suit our new requirements. Our present strategy is based on the following non-exhaustive list of technological objectives:

- **Modularity** and **Prototyping**. The IEB is a set of interconnected objects, for which the shell provides creation, access and management methods. It was clearly necessary to design a basic structure for WEBS which we could later expand when new objects or functionality were needed. Section 4 will detail the design of this structure. One consideration, however, imposed itself from the beginning; the

[1]The integration of index objects and methods within the WEBS prototype is still in its development phase and will not be further discussed in this paper.

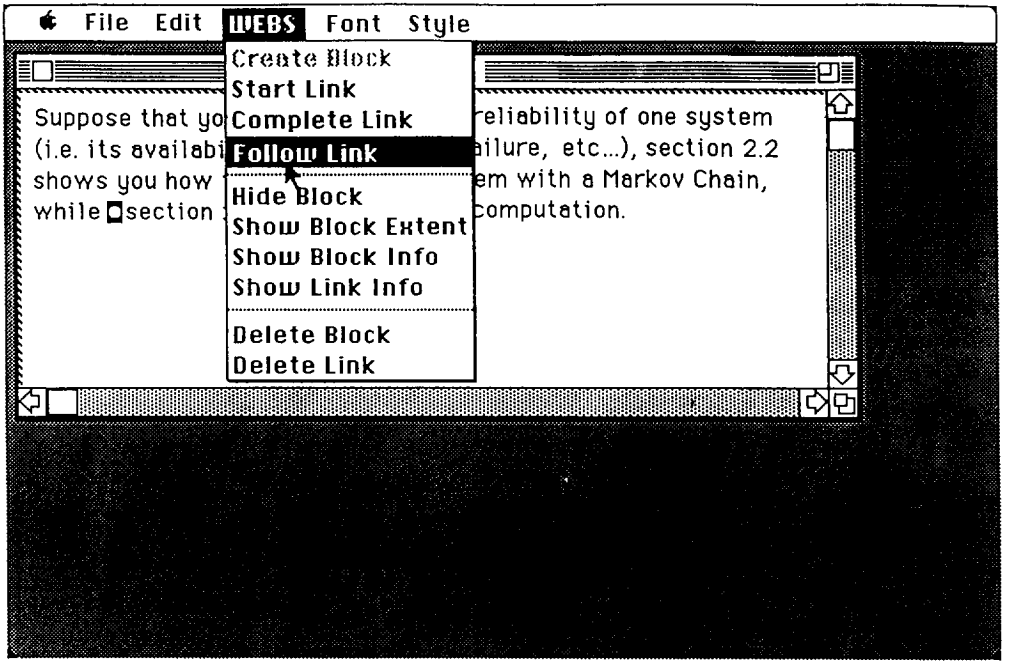

Figure 1a A Snapshot of WEBS before the Follow Link Command

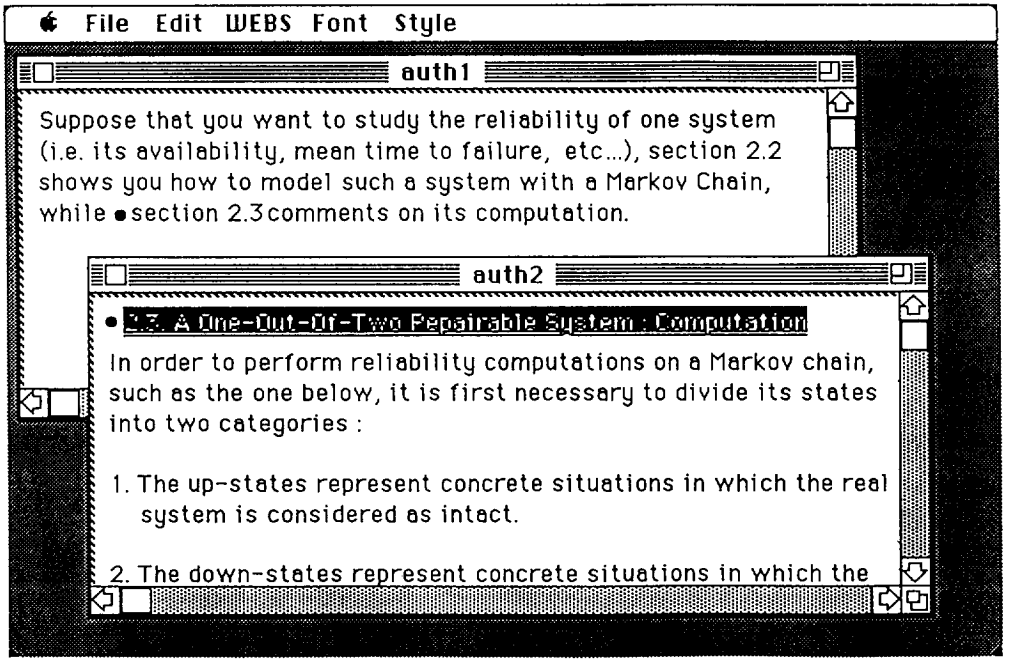

Figure 1b A snapshot of WEBS after the Follow Link Command

system had to be decomposed into modules in such a way that it would be simple both to extend its basic structure and to have several developers working in parallel. For example, while one developer used the WEBS prototype to add modules for creating and manipulating text objects, a second could do the same for modules related to graphical or modelling objects. When both developers have completed their respective tasks, the structure of WEBS should facilitate the aggregation of their modules into a single more powerful tool.

- **Understandability** and **Reusability**. When adopting a modular design, with several developers working in parallel, one must find a way of encapsulating the data and the procedures of the system into meaningful units. Thus there are essentially two goals. First, the design should facilitate the maintenance and expansion of WEBS. Second, developers should be encouraged to extensively reuse each others' code, instead of "reinventing the wheel" at each stage. The inherent modularity of Object Pascal, as well as the facility which with it allows inheritance of methods and data structures, make it an ideal choice in view of these objectives.

- **Consistency**. We felt it essential that WEBS be grounded on a sophisticated and directly manipulable user interface based upon well accepted principles such as those stated in Chapter 2 of *Inside Macintosh™* [Apple, 1985]. Particularly in a multi-developer environment, it is difficult to make such an interface consistent, so that similar commands operate similarly in all modules. We solved this crucial problem by including the user interface functionality of WEBS within a set of basic modules in such a way that a new module could inherit its appropriate behavior with a minimum of additional work on its developer's part.

- **Minimum Effort**. Since the implementation of the WEBS prototype from scratch would take too much time, we wanted a software environment which would provide us with a rich initial framework. MacApp provides such a framework.

- **Targeted Equipment** . The hardware for an IEB must provide an agreeable environment with which the user may interact. Such an environment is particularly necessary because the targeted users of IEBs are not computer specialists, and would soon become frustrated and unreceptive if their interaction with the system were hampered by the machine. The system should include a bitmapped, high resolution screen and an input device other than a keyboard, such as a mouse. This hardware would support the direct manipulation interface described above, which is easy to learn and use. Of the equipment actually available on the market, those which best respond to the above requirements are the workstations. Unfortunately, real workstations are still too expensive to be readily available to the majority of potential WEB users. We therefore selected the Macintosh™ SE as our first target machine, intending to migrate later to the more powerful Macintosh™ II.

4. The Software Architecture of WEBS

The actual objects which are manipulated by an object-oriented program are always instances of abstract objects called **classes**. For example, WEBS can instantiate and control dozens of text documents. These objects, however, all belong to the same *TTextDocument* class, which defines both the data structure and the methods associated with them. In an object-oriented design, then, the unit of modularity is the class. The WEBS prototype is based upon three layers of such classes:

1. the MacApp layer,

2. the WEBS general layer, and

3. the WEBS model layer.

This section provides a high level description of WEBS software architecture. It is divided into three subsections, each describing one of these layers.

4.1 The MacApp Layer

The hierarchy of the MacApp layer's main classes is shown in the upper part of Figure 2.

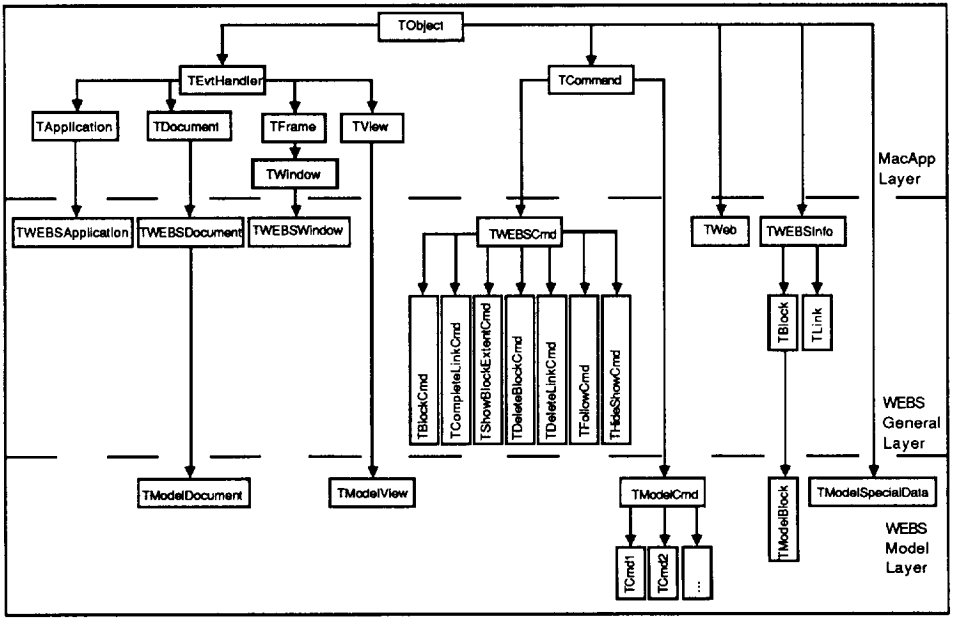

Figure 2 Class Hierarchy of WEBS Prototype

The MacApp layer provides an "expandable" application framework that encapsulates the basic behavior of the standard Macintosh™ user interface, (i.e. pull-down menus; moveable, resizeable, scrollable windows; data storage, retrieval, and printing; and the capability of reacting directly to user generated events such as keyboard or mouse input). The next paragraphs describe the most important classes that MacApp provides:

- The *TObject* class is the ancestor of all other classes. It provides utility methods for cloning and freeing objects.

- The *TEvtHandler* class defines a template method[1] and some default methods to handle several types of events arising in a Macintosh™ application (e.g. key, mouse and menu events). Since most classes must be able to react appropriately to such events, the TEvtHandler class is the ancestor of most of MacApp's other classes.

- Methods for launching an application and displaying its menu bar, and template methods for creating and initializing appropriate document objects, are contained within the *TApplication* class. This class has few fields of its own; MacApp globals, however, are generally treated as fields of the TApplication object. The TApplication class also has the responsibility of managing the main event loop. This loop, which is fully encapsulated within the MacApp layer, receives incoming events from the event queue and dispatches them to their appropriate objects; in the case of most mouse events, for example, the event is sent to the window in which the mouse pointer was located.

- The *TDocument* class is responsible for documents that belong to the application. Furthermore, the TDocument objects contain all the basic information for managing the frames, the windows and the views (see below), which allow the user to inspect and modify a document. Naturally, the methods of TDocument which allow for saving and restoring data must be overridden within the subclasses of TDocument that implement the specific types of documents that the application manages.

- The *TView* class manages the display of the data proper to a document and passes various mouse events to the appropriate objects within the view.

- The user, however, only sees a portion of a view on the screen. The *TFrame* class provides the mechanism for showing a portion of the view in a window, and for scrolling so that different parts of the view appear.

- The *TWindow* class provides methods for opening, closing, moving, resizing, activating, and deactivating windows. A TWindow object is a special type of frame, and may act as a container for other frame objects.

---

[1]Template methods basically do nothing as long as they are not overridden in a subclass of their class.

- Finally, the *TCommand* class manages most actions generated by user input that affect the data or appearance of the active document, i.e. menu commands, mouse commands, and keyboard input. A TCommand object has template methods which provide for executing a command, for undoing its effect, and for redoing its effect. Any subclass of TCommand must override these methods; its methods will act on the application documents, windows or views in order to implement the changes needed to reflect the user's action.

4.2 The WEBS General Layer

The WEBS general layer classes are illustrated in the middle part of Figure 2. The classes of this layer encapsulate the "electronic book" functionality of WEBS. This functionality includes assigning ownership to and protecting document objects, as well as a data structure and methods for dealing with the creation and use of webs, links, and blocks. Later this layer will be further extended with classes allowing for the management of index objects. The following paragraphs describe the main classes of this layer:

- The *TWEBSApplication* class is responsible both for the management of all the documents in the system and for keeping track of the hypertext objects. As such, it includes a method which allocates and initalizes a WEBS document of some requested type[1]. It also finds and installs the appropriate author's and reader's web objects.

- The *TWEBSDocument* class provides template methods which allow WEBS document objects (i.e. any object which inherits from the TWEBSDocument class) to act appropriately in a hypertext environment. A WEBS document object might, for example, be asked to hide all of its blocks. Furthermore, this class contains fields that allow it to be identified by its owner, and methods that allow it to save and restore those fields.

- Hypertext-related commands are supported by the *TWEBSCmd* class. These are basically those commands which appear in the WEBS pull-down menu presented in Figure 1. An object of this class keeps track of the web and document to which the command pertains. Any command object which implements a hypertext-related command should inherit from this object.

- Our notion of a web is implemented in the *TWeb* class. Each object of this class contains three lists: one is a list of all the links in the web; another is a list of sublists, in which each sublist is a list of blocks contained within one document; the third simply maps documents' internal identifiers to their external names and directories. The TWeb class bears the brunt of the responsibility of finding blocks and links and opening their accompanying documents; its methods reflect this.

- The *TWEBSInfo* class is the ancestor of both the block and the link classes. It contains information

[1]Presently, only text documents are supported by the WEBS prototype.

such as the date of creation and user-defined explanatory texts which is common to both. Its methods initialize and display this information.

- The *TBlock* class contains the fields and methods which implement WEBS block objects, providing for their creation, initialization, selection, and deletion. Since we expect that blocks will be implemented differently in each type of document (a sentence cannot be treated in the same way as a group of shapes), many of the methods are templates that will be overridden by a specific document block. All types of block objects, however, contain a list of the links attached to them; this list is therefore contained in the TBlock class description.

- A link knows its starting and ending blocks; the *TLink* class contains the fields which support this knowledge, as well as the fields which are inherited from TWEBSInfo. Most of the TLink class' power is in its methods; they allow link objects to be added to and deleted from webs, disconnected from blocks, and followed from one end to the other.

4.3 The WEBS Model Layer

The WEBS model layer contains all of the classes specific to any of the particular models which make up an IEB. These include the general text and graphic models[1] as well as individual models, such as a Markov chain simulation, which apply solely to a given electronic book. In other words, the model layer of WEBS implements the individual model objects which comprise an IEB, while the general layer of WEBS manages the classification and the connection of these objects within an IEB. Since the number of such model objects within our prototype will steadily increase, we first present a generic strategy for implementing any type of model object within the WEBS model layer and only thereafter discuss the particular case of text model objects.

The lower part of Figure 2 shows the classes which must be created to support a new type of model object. Since a model object's data are contained within or pointed to by a MacApp document, they can be saved on disk in the same way as any other MacApp document:

- The *TModelDocument* class must be a subclass of TWEBSDocument. In addition to incorporating much of the functionality that any WEBS model needs to manage its own duties (being in many ways no different than any stand-alone application), it overrides methods of its parent class that concern themselves with document-specific block operations. Amongst other things, TModelDocument objects must know how to hide and show their blocks, as well as to report whether or not a block has been selected.

- A subclass of MacApp's TView class, the *TModelView* class is a model-specific view that displays a document's data to the user. Sometimes it is instructive to show different displays of the same data;

[1] We call all of these objects models, thus justifying the name given to this layer.

sales figures might be exhibited as both a printed table and a bar graph. Then several different TModelViews might be defined, each being responsible for a different sort of display.

- If a new model is to do anything, command objects, subclasses of MacApp's TCommand class, will have to be defined for most of its operations. These are illustrated in Figure 2 by the *TModelCmd*, *TCmd1* and *TCmd2* classes.

- Any document-specific capabilities of a block must be implemented in the *TModelBlock* class. This class overrides those methods of its TBlock parent class that are related to the block's display; each model will have its own way to mark the existence of a block and to delimit its contents.

- Since any new model will presumably have some special data of its own, it will often be necessary to implement special data objects. Such objects will probably inherit directly from the MacApp class TObject, and are represented in Figure 2 by the *TModelSpecialData* class.

- Finally, it should be noted that the WEBS TLink object is not represented by a child class in this layer. A link only needs to know the identity of the blocks at each end. Since it does not need to know any model-specific information, it does not need to be represented in the model layer; a generic link, that connects equally to any kind of block, is all that is necessary.

Figure 3 shows what the TModelView and the TModelCmd classes look like when the model objects are text objects. In this case, the new text objects follow the class hierarchy described for the generic case above, except for the addition of several new MacApp classes which constitute a unit that supports text editing.

5. Conclusion

5.1 Achievements

Our experience with Object Pascal and MacApp has been very positive thus far. Within a four month period we have developed a prototype which integrates the WEBS general layer described in Subsection 4.2 and which fully supports text model objects. Such quick progress has been possible largely because of MacApp's support of the Macintosh™ basic user interface features. Furthermore, we have written highly modular code which, with the expandable class structure of WEBS and the inheritance capability inherent in an object-oriented language such as Object Pascal, will greatly facilitate the concurrent integration of several new model objects into our prototype.

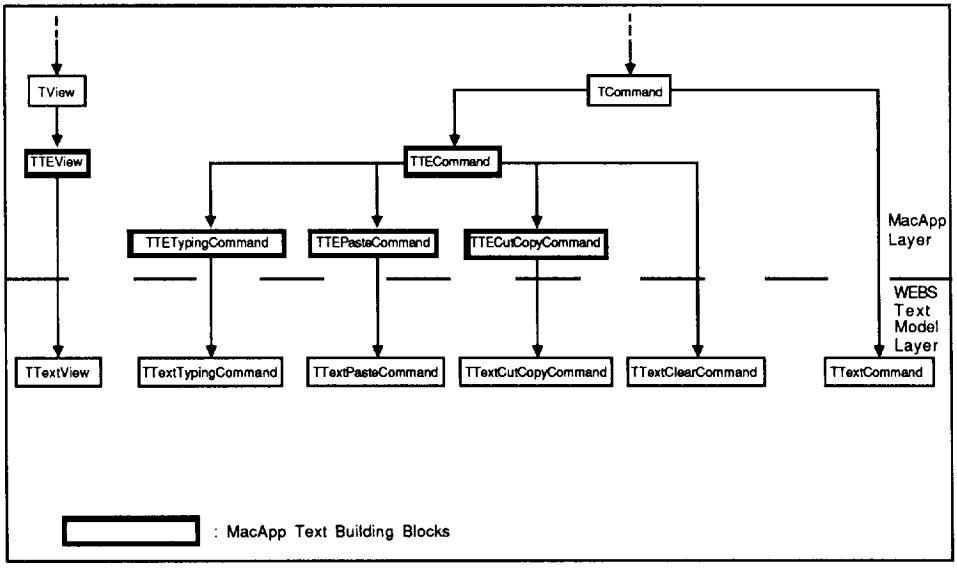

Figure 3 Partial Hierarchy of WEBS Text Model Layer

5.2 Learning Curve

Learning the syntax of Object Pascal is a matter of hours for an experienced Pascal programmer. To master the object-oriented programming style, however, is a more difficult task, which can only be achieved through ample experience, and some good examples to follow. We feel that an object-oriented language is really efficient only if it is used in conjunction with an extensive set of basic classes, such as those which make up MacApp, so that new classes developed for new applications can inherit certain fundamental behavior rather than always reimplementing it. The learning curve for a system such as MacApp is steep, and several months of experimenting with the system appear to be necessary before starting a major project. Nevertheless, the time and consistency gained by using MacApp make learning it a worthwhile effort for the serious developer.

5.3 Additional Comments

Some of the MacApp units do not offer sufficient functionality for our ultimate goals. For example, the TTEView and TTECommand classes offer only simple text editing facilities (no multiple fonts within the same document, no ruler as in MacWrite™) and MacApp does not offer a framework for organizing the display of simple graphic objects such as circles and rectangles. Although it is possible that future releases of MacApp will remedy some of these problems, we may want to develop such facilities ourselves. A major drawback to such a plan is that because of the lack of multiple inheritance capability in Object Pascal, some of this functionality might have to be included in MacApp's own classes. Such a solution would risk

incompatibility with future versions of MacApp. We have already run into this problem with descendants of the TTECommand class. TTextTypingCommand, TTextPasteCommand, TTextClearCommand, and TTextCutCopyCommand are all descendants of the corresponding TTECommand classes (see Figure 3). In addition, they all have to perform certain operations on blocks. Because of the lack of multiple inheritance, and because we did not want to rewrite the parent classes, we were forced to duplicate block-handling code in each of the child classes, rather than have one copy of the code from which each of those classes could inherit.

We also believe that the MPW/MacApp system would be a better prototyping system if it were an interactive system rather than a compiled one. We have spent a lot of time over the past few months waiting for the program to compile and link each time a change is made. For a 5000 line program, compiling and linking with the MacApp units can take over 5 minutes on the Macintosh™ SE. An interactive system, or at the very least an incremental compiler, could cut out a lot of this wasted time.

6. Acknowledgements

Our research on "The Integrated Electronic Book" originated with the paper by [Kohlas, 1984] and has been and continues to be sponsored, since December 1985, by the Swiss National Science Foundation under grant number 1.018-0.84.

References

Apple *Inside Macintosh™,* Volumes I, II and III
Apple Computer, Addison-Wesley, 1985.

Conklin J. *Hypertext: An Introduction and Survey*
Computer, September, 1987.

Cox B. *Message/Object Programming: An Evolutionary Change in Programming Technology*
IEEE Software, Vol. 33, No. 1, pp. 50-61, January, 1984.

Cox B. *Software-ICs*
BYTE, June, 1985.

Cox B. *Object-Oriented Programming: An Evolutionary Approach*
Addison-Wesley, 1986.

Doyle K., Haynes B., Lentczner M. and Rosenstein L. *An Object-Oriented Approach to Macintosh™ Application Development*
Proceedings of the 3rd Working Session on Object-Oriented Languages, Paris, France, January 8-10, 1986.

Garret L. and Smith K. *Building a Timeline Editor from Prefab Parts: The Architecture of an Object-Oriented Application*
OOPSLA '86 Proceedings, Portland, Oregon, September, 1986

Goldberg A. and Robson D. *Smalltalk-80: The Language and its Implementation*
Addison-Wesley, 1983.

Goodman D. *The Two Faces of Hypercard*
Macworld, pp. 123-129, October, 1987

Haan B., Drucker S. and Yankelovich N. *An Object-Oriented Approach to Developing Consistent Integrated Applications*
IRIS Report, Institute for Research in Information and Scholarship, Providence, RI, September, 1985.

Kohlas J. *Das Integrierte Buch (eine Projektidee)*
Working Paper No 78, IAUF , April, 1984.

Meyrowitz N. *Intermedia: The Architecture and Construction of an Object-Oriented Hypermedia System and Applications Framework*
OOPSLA '86 Proceedings, Portland, Oregon, September, 1986

Pasquier-Boltuck J. and Collaud G. *The Woven Electronic Book System, (WEBS): The Enduser Model and Interface*
Working Paper No 129, IAUF, February, 1987.
A shorter version of this paper has been submitted for publication in the International Journal of Man-Machines Studies.

Savoy-a J. *Le livre électronique EBOOK3*
Proceedings of the EAO-87 congress, Cap d'Agde-France, March 23-25, 1987.

Savoy-b J. *The Electronic Book EBOOK3*
Working Paper No 137, IAUF*, July, 1987.
This paper has been submitted for publication in ACM Transactions on Office Information Systems.

Savoy-c J. *Le livre électronique EBOOK3*
Diss., Peter Lang S.A. publishers, Berne, Switzerland, 1987.
ISBN 3-2 61-03772-5.

Schmucker-a K. *Object-Oriented Programming for the Macintosh*
Hayden Book Company, Hasbrouck Heights, NJ, 1986.
ISBN 0-8104-6565-5.

Schmucker-b K. *MacApp: An Application Framework*
BYTE, pp. 189-193, August, 1986.

Tesler L. *Object-Oriented Languages: Programming Experiences*
BYTE, pp. 195-206, August, 1986.

Yankelovich N., Meyrowitz N. and van Dam A. *Reading and Writing the Electronic Book*
Computer, October, 1985.