ELECTRONIC PUBLISHING, VOL. 7(2), 75–87 (JUNE 1994)

Interleaf active documents

PAUL M. ENGLISH AND RAMAN TENNETI

Interleaf, Inc.
9 Hillside Ave
Waltham, MA 02154, USA

email: pme@ileaf.com, raman@ileaf.com

SUMMARY

A commercial structured document processing system has been built with an extensible object system. This system is an excellent platform for the design, implementation, and delivery of active documents. Examples are discussed.

KEY WORDS Active documents Document-based applications User interfaces Document-object systems Lisp

1 INTRODUCTION

1.1 Motivation for building an active document system

Interleaf 6 (I6) is the document creation component of Interleaf's document management software. It was originally designed for the creation of technical manuals that could be hundreds of thousands of pages in length. Such complex documents influenced the software's requirements, not just in performance and quantity handling, but also in its ability to address several organization and production workflow issues regarding documentation. One of the things done to address such requirements was to build a highly extensible system—one that could be customized to fit the specific needs of a particular site. One goal of this extensibility when it was designed in 1989 was the ability to create *active documents* and document-based applications.

1.2 Motivation for building active documents; the document user interface

Paper documents and books are the most common user interface (UI) for information presentation. Most people are more comfortable using paper documents than using a computer. Just by looking at a book most people can quickly recognize titles, paragraphs, headings, sections, figures, lists, and so on. There are well understood (even if not always followed) UI and organization standards for paper documents and books. They usually follow the same organization, with a title page, copyright and publisher information, contents, chapters, and an index.

Because documents are so familiar, it follows that more document-based computer applications are being built. The currently most popular personal accounting software,

CCC 0894–3982/94/020075–13 *Received 5 April 1993*
Revised 12 January 1994

Quicken by Intuit, does not have a generalized graphical user interface (GUI)—it instead has a UI which simulates a paper check. Use of this natural UI makes this product easy to learn and hence extremely popular. Likewise, applications built with the document metaphor are easy to learn for most people.

1.3 Outline

An architectural overview of I6 is presented in Section 2, followed by a description of related work in Section 3. Sections 4 thru 7 contain descriptions of four applications of active documents constructed with I6, including interesting active document issues with each. Sections 8 thru 10 compare active document solutions with other solutions, discuss some implementation issues, and conclude with some of the main points presented.

2 ARCHITECTURAL OVERVIEW

2.1 Document structure

I6 is an object-oriented structured document editor. The main structure is called a *component*, which contains content as well as formatting information. For example, in the I6 representation of this paper, the current paragraph is a *para* component. The numbered section title is a *head* component, which contains an *autonumber token* from the *autonumber stream* named *list* as part of the *head* component's *prefix*. A component prefix is a special case of an *inline* component, which is used structurally to set off content that is formatted *in line* with its parent component. Inline components are used to represent nested structure, and often have different style attributes from the containing parent. I6 also has font and other style tokens that can be placed in the middle of a component text stream to change style without requiring an inline component. *Reference* objects exist to allow references that share content (usually an autonumber or page number) with other document objects.

Frames are containers used explicitly to control the placement of content including text and graphic objects such as *diagrams*, *images*, *charts*, *equations*, etc. The author creates a frame within a component. One of the formatting properties of a frame controls its location relative to its *token* (creation point) within its parent component. Expressing frame location relative to its anchor token allows greater layout flexibility than requiring the user to place the frame on a specific place on a specific page. For example, when content is inserted before the frame, the frame *knows* to move with its anchor to the next page. Each page has special *header* and *footer frames* set to have repeating content.

Tables are composed of *rows*, which are a special kind of component. Table cells are then special cases of frames, and thus can contain anything contained by a frame.

Microdocuments are component containers that can live inside frames.

2.2 Implementation architecture; extension language

Most of I6 is implemented in C. The heart of I6 is its object system, which has a dynamic class hierarchy, message set, and method associations per class. The object system includes features such as multiple inheritance, property inheritance, and *before* and *after* methods [1].

I6 also has a tightly integrated Lisp system, which is based on Common Lisp [2]. The I6 developer's environment includes an editor, listener, compiler, debugger, syntax checker, help system, a number of other utilities, and the entire source set for the Lisp portion of the system. In addition to the simple Lisp editor, GNU Emacs [3] has been customized thru its own Lisp environment to connect to running I6 sessions to edit and debug I6 Lisp, and to provide interactive I6 help.

Lisp scripts can be attached to any Interleaf object, stored within or external to a document. End-users need not know or care that the system contains Lisp, since generally they never see it.

The I6 UI is written entirely in Lisp. (There are Lisp bindings to a Windows-enriched Motif toolkit based on OSF Motif 2.0 and Microsoft Windows. This allows sharing over 95% of the UI code between Motif and Windows. The I6 Macintosh product has Lisp bindings to the Mac Toolbox, used to build the I6 Macintosh UI.) Callbacks are written in Lisp, although they usually simply send messages into *editor objects*, which are mostly written in C.

Lisp was chosen as the I6 extension language because it is interpreted (providing a fast prototyping environment) and has run-time binding. Run-time binding is required for easy delivery of active documents. This allows a user to simply cut and paste a document from an active system to a static one, then allowing the active document to introduce new classes and methods into the previously static environment. (See Section 9 for a discussion of activation and security issues.) Other features of Lisp that have proved useful include its exception handling, list processing, automatic memory management, packaging, and ability to be rebound to create problem-specific sub-languages.

3 RELATED WORK

The most intensive and well-documented research effort in active document technology has taken place at Xerox PARC. Spinrad [4] seems to have originated the term *active document.* Zellweger's Scripted Documents [5], Arnon's CaminoReal [6], Bier's and Goodisman's embedded buttons [7] and Active Tioga documents [8] were built using Tioga, Cedar's multimedia document editor [9]. The Tioga editor allows tagging of the nodes of a document tree and tagging of individual characters. In CaminoReal documents, editing of a mathematical object can send related mathematical objects to a "standard mathematical system," returning other objects that in turn are reformatted into the document. Zellweger's Scripted Documents contain sequences of activities in the form of external scripts to support such things as voice animation. Bier and Goodisman describe a prototype architecture where arbitrary document elements behave as buttons (control elements). Active Tioga [8] allows procedures written in the Cedar programming language and registered in a central database to be invoked as activities in response to document editing or redisplay.

Other active document research efforts have taken place at IBM, Apple, and the Weizman Institute of Science. Chamberlin et al. [10] describe the Quill document editor, which allows programmers to attach procedures written in the REXX programming language to individual objects of a document. While Quill used these procedures primarily for formatting purposes, it is clear that they could be used to add activity more generally to documents. Towner [11] describes a tool that imports database text fields into a document by adding special markup into the document. In this auto-updating tool, the activity—content updating—takes place only at a user's explicit request. Hansen [12] describes the Ness component of the Andrew ToolKit, which allows authors of documents to attach scripts that are written in the Ness programming language. These scripts are triggered by top level user events invoked via the mouse or keyboard. Goldberg et al. [13] add activity to ordinary email in order to facilitate the use of email for collaborative work.

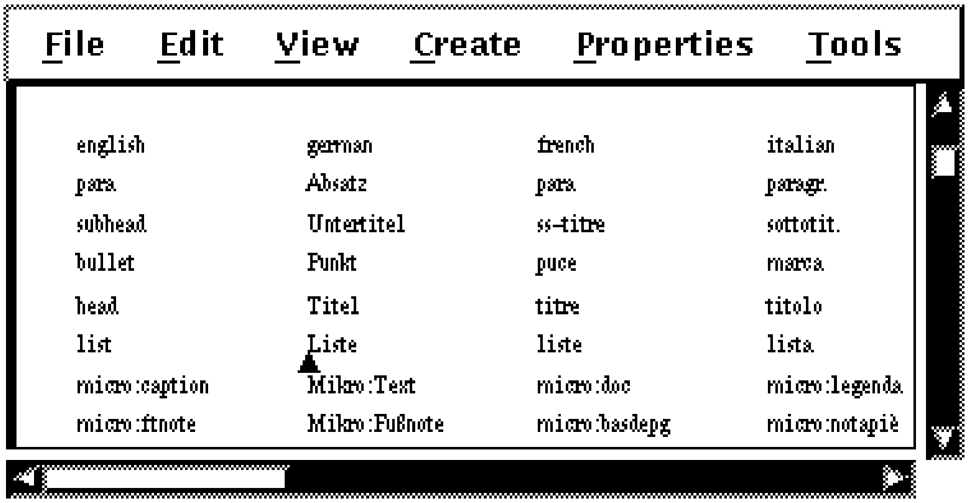
Interleaf’s active document architecture is described by English et al. [14], who define active documents as “structured documents and their processors in which the objects in the documents can be acted upon by, and can themselves act upon, other objects in the document or the outside world.”

4 EXAMPLE: AUTO-LOCALIZING TEMPLATES

I6 is distributed with a number of document templates—sample memos, letters, reports, etc. These templates usually contain some initial text and definitions of the master component types (sometimes called *style sheets*). I6 needs to be localized for fifteen languages and over forty countries, and this work includes the localization of the document templates. The memo template should initialize with page size 8.5″ by 11″ in the USA, but page size A4 in the UK. And the *para* component should be called *Absatz* in Germany.

In prior releases of the system, template maintenance was time-consuming and error-prone, as each change to a template required someone to edit the corresponding copy for each of the other languages and countries. I6 contains a system to allow the template designer (a typographer or graphic artist) to build *auto-localizing templates*.

The template documents contain tables of localization information, including the names of all components as well as values for localized variables such as page size. See Figure 1.



english	german	french	italian
para	Absatz	para	paragr.
subhead	Untertitel	ss-titre	sottotit.
bullet	Punkt	puce	marca
head	Titel	titre	titolo
list	Liste	liste	lista
micro:caption	Mikro:Text	micro:doc	micro:legenda
micro:ftnote	Mikro:Fußnote	micro:basdepg	micro:notapiè

Figure 1. A template localization table

A template document subclass is defined in one of the system Lisp libraries. This subclass has a custom *open* method that changes component names and sets other document properties according to the table-specified settings for the current language. This occurs while the document is opening, but before its content is shown. The open method then purges the localization table, and changes the class of the document back to the default document class, thus permanently deactivating this newly created document instance. This leaves its initial content in a state suitable for user completion.

The initial implementation of auto-localizing templates revealed general problems relating to confusion of building documents with building document *builders*. One problem is that active documents are often self-modifying, and thus one must be careful to save a copy of the original document. Another problem is that there is no consistent way to switch documents between static and active modes. This is exacerbated in this case since the last act of the subclassed open method is to turn the document back to the default (static) document class. Similar problems have been described in debugging Active Tioga documents [8].

Another problem with the template documents was a performance penalty users paid in order to have such run-time localization flexibility. On slower machines this added almost one second to the opening of a new document created from a template. At the cost of run-time localization flexibility, a site administrator can, at installation time, force the localization of the template documents in one instance for each language. This flushes the auto-localization templates and deactivates the documents, effectively freezing the template translations. This has proved acceptable because run-time localization is only important at a multilingual site; most sites are unilingual.

5 EXAMPLE: MULTIMEDIA EMPLOYEE DATABASE

We are presently building an active document employee photo album that is programmatically generated and has subclassed objects with custom selection methods. This builds in some ways on the active telephone directory described in [8] and the Biography Retrieval application described in [15]. The photo album currently has three types of data for each employee:

- Color image: a 35mm color photograph processed onto and extracted from a Kodak Photo CD.
- Voice: employee name as copied from our voice mail system.
- Text: information such as username, phone extension, and workstation name.

A Lisp program generates the photo album document from the above data. Each generated page contains a frame with a face image, a text description about that person, and a button component labelled with the user's workstation name. The face frames and the button component objects are subclassed, with new *select* methods provided for each. Clicking on the face plays the audio name, and clicking in the button component produces a list of processes being run on the remote workstation.

The code below defines the new frame subclass used to contain face images. A new select method is provided, which opens (plays) the audio object named by the current user.

```
(defvar face-class (obj-new-class doc-frame-class nil))
(mid:provide-method face-class mid:select #'face-select)
(defun face-select (frame &rest args)
  (audio-play-name (album-current-name)))
```

The code below defines the new component subclass used for the workstation machine name buttons, which on selection output a list of active processes. This `mach-select` method calls `tell-next` to forward the select message on to other select methods that might exist for the `mach-class` subclass or its parent classes.

```
(defvar mach-class (obj-new-class doc-cmpn-class nil))
(mid:provide-method mach-class mid:select #'mach-check)
(mid:provide-method mach-class mid:enter #'mach-check)
(defun mach-check (cmpn &rest args)
  (mach-ps-display (mid:get-substring cmpn t t))
  (tell-next args))
```

The object system allows interception of events on a specific object instance or on a class-specific set of objects. Thus there is no need to intercept mouse button events, nor the system-wide *select* method, nor even the default system *select* method for frame objects. The `face-class` frame subclass was created to have the special `face-select` method, allowing selection of only the face frames being handled by special code.

It is also possible to provide *instance methods* directly to the special objects instead of having to subclass them. But in cases where there are many special objects (such as face frames above), it is more efficient to create a subclass with the new method (putting all desired frames in this subclass) than it is by providing instance methods for all instances of interest.

6 EXAMPLE: INTELLIGENT MAINTENANCE AID

Intelligent Maintenance Aid (IMA) is an active document and expert system that was developed by United Technologies Corporation and Sikorsky Aircraft. IMA was initially developed to aid in helicopter maintenance operations, but is now being sold as a general equipment maintenance tool.

A field mechanic uses the active document interface to communicate with the expert system by selecting equipment condition choices presented on each screen. The choice made is sent by the document to the expert system, which uses failure history and test data to determine the next test to be performed by the mechanic. After sending a choice to the expert system, the document then asks it how to construct the next page. The expert system passes back an identifier for the next equipment diagram (which also includes instructions and a question) and a small set of choices that the document presents in a set of active buttons on the bottom of the page.

This application was developed using existing systems. Not only did the expert system independently exist, but so did all the equipment diagrams. All that was needed for this application was a document framework to link these pieces together. An IMA document uses I6 process communication Lisp to communicate to the expert system, Lisp code to activate the choice buttons, and Lisp code to construct following screens based on expert system responses.

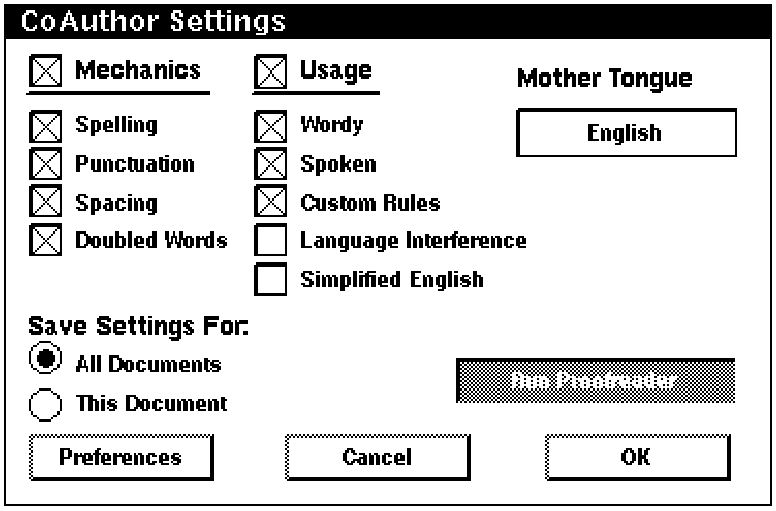
IMA is currently being adapted to operate on wearable computers, which include a head-mounted optical display and a voice activation interface. These will allow mechanics to work in cramped spaces with both hands free for equipment maintenance. Since the UI is a document, it is easily changed by adjusting fonts and pagination to accommodate smaller screens.

Document system facilities used by this application include printing, filtering to accept CALS [16] compliant documents, and automatic hypertext linking and indexing.

7 EXAMPLE: ORACLE COAUTHOR

CoAuthor is a self-learning proofreading tool used to check spelling, punctuation, and grammar. It can also check for simplified English [17], as well as look for common mistakes made by non-native English speakers.

The CoAuthor UI is implemented by a document beyond the lines described by Bier and Goodisman [7], as it has implemented a complete widget set using document objects. These include standard UI elements such as push buttons, toggle and radio buttons, text entry fields, and scrolling lists. See Figure 2.



CoAuthor Settings

☒ Mechanics
- ☒ Spelling
- ☒ Punctuation
- ☒ Spacing
- ☒ Doubled Words

☒ Usage
- ☒ Wordy
- ☒ Spoken
- ☒ Custom Rules
- ☐ Language Interference
- ☐ Simplified English

Mother Tongue: English

Save Settings For:
- ● All Documents
- ○ This Document

Figure 2. CoAuthor active document UI

The dialog windows are actually active documents that are tightly integrated with the user's source document being checked. A proposal for such an application was advanced by Terry and Baker [8].

To build this application, I6 Named Graphic Objects (NGOs) were constructed for each type of UI element. NGOs are objects created with standard I6 diagramming tools, then named by the user for easy reuse. In addition to NGOs giving end-users the ability to manage graphic objects by name, the document programmer can easily attach different methods on each type of NGO. Such a facility is essentially that provided by many stand-alone GUI builder systems.

When the user launches the CoAuthor dialog, it programmatically changes the appearance and content of its NGOs to show the current state of the proofreader. The user can then click, drag, or enter text into the NGOs to call the action specified for each NGO document object. These actions can modify the source document and/or the CoAuthor dialog. In addition, they can generate an interactive error report, which allows users to examine and deal with each error separately. Each error is linked to the source of the error in the document being checked.

In using document objects to represent UI elements, there are two common approaches used to change their appearances in I6 documents. First, one can programmatically apply different graphic properties or transformations to the object, such as changing its color or size or position. Second, one can use the conditional document assembly mechanism [18] to hide one document object quickly (such as an empty box) and instead show another object (such as a filled box).

CoAuthor made extensive use of document objects to represent UI elements that are standard in many GUIs. This had advantages because these elements could be edited and placed using the standard I6 document editor, and thus treated the same as other content created by the document developer. However, while document objects can be useful in emulating standard GUI toolkits, if it is a goal to be GUI compliant it is simpler to use actual GUI widgets within the document.

Users do not know that they are interacting with documents when they use the CoAuthor dialogs—indeed, one of the points of this example is that active documents are useful not only in presenting new UI paradigms, but also for easily constructing systems that use existing paradigms. The IMA application described above is an example of the former—its UI paradigm is that of a paper document. CoAuthor is an example of the latter—it uses Motif-like UI paradigms, although constructed with document objects.

8 ACTIVE DOCUMENTS SOLUTIONS VERSUS OTHER SOLUTIONS

When comparing active document solutions to other solutions, there are two things to ask. First, is the UI document-centric? And second, is the application built from a document object system (an active document engine), or from some other toolkit?

All of the examples described above could have been implemented using different toolkits than an active document engine. However, if they were implemented with another tool, not starting from a document class library, and resulted in the same UI and functionality, in fact they would be active documents—but active documents that were harder to build than building them on a document system.

The problems solved by the tools described above could have been solved using different UI paradigms as well—they did not need to be solved as active documents.

The key advantages for active document applications are the following:

- Ease of use. As mentioned earlier, building an application that uses the document metaphor can make an application easily understood by a very wide audience.
- Ease of implementation. Document object systems have rich capabilities for creating interfaces. The application designer can use existing objects from a document object catalog, or s/he can easily create new objects using the document editor.
- Core publishing functionality. If an application has a need for text and graphics presentation and editing, document system based solutions are much easier to use and have more powerful functionality than is available in general application toolkits. A simple example is the ability to have mixed fonts and/or text properties in part of your UI. A more advanced example is the ability to do automatic pagination of floating graphic frames in a document with tables and images. Scenarios such as these are easy to do with document-based solutions, but are not available in most parts of standard application toolkits.
- External publishing functionality. In addition to having the presentation and editing functionality within the document, document system based applications can also take advantage of all the other publishing facilities that exist within the document system. For example, the IMA application makes use of the system's printing and filtering capabilities.

- Performance. Document systems are usually carefully optimized for high performance text and graphics. Complex user interfaces built from document objects can be more efficient than ones built from a client/server widget based system. For example, the active document can be constructed ahead of time and saved in its proprietary performance-optimized format. This allows faster *open* of such interfaces than those that have to do a great deal of client/server communication to build each widget.
- GUI portability. Depending on the paradigm used to construct the UI, document-based applications can be constructed to be portable across many GUI systems. Some of this is due to GUIs not having different standards for document-like text and graphics presentation. Most GUIs specify the *trimmings* more than the document presentation. Thus, if you take advantage of the document metaphor, your application has a good chance of working well across different GUI standards. Document-based applications that use native GUI objects have the same portability issues as generic application development kits, namely the tradeoff between minimal but portable interfaces compared to rich but non-portable ones. I6 allows either approach. One example of the first was used in an Interleaf 5 (I5) set of visual tools for the customization of the user's document processing environment. We refer to its simplified UI as *Point And Pound,* as it mostly consists of GUI independent buttons that the user points at and selects to trigger activity. By contrast, I6 also allows the use of native widgets.

9 IMPLEMENTATION NOTES

9.1 What should an active document enabling system contain?

The most important element in an active document system is the underlying structured document model. It should contain a rich set of document objects (text, components, tables, autonumbers, page numbers, index tokens, reference objects, footnotes, frames, diagrams, beziers, images, files, documents, books, etc.), and a rich set of properties for each object.

There must be programmatic access to do everything that is possible from the UI, and more. There should be access for object creation and destruction; object navigation (ideally through different views, such as format and structure); getting and setting predefined and user-defined properties.

The system should be object-oriented to allow for easy active document building and reusability of active objects.

There should be a set of editor objects, which can act as user agents to intercede on behalf of the user between the UI and the underlying document objects. Editor objects are responsible for side effects such as posting choice dialogs and reporting error messages.

The system should have a full development environment with online hypertext documentation, active document development tools, integration with standard development tools, and support.

9.2 Active document seeding

An interesting aspect of active document development is how objects get *seeded*—that is, how otherwise static objects first get activity attached to them. In I6 the developer first evaluates some code to get a handle on an object. This can be done programmatically by navigating from the current document object, or by asking for the object at the cursor. Then the developer typically evaluates code to change the class of that object and to provide new methods for that class.

An object can be under edit by a normal document editing session, as well as under programmatic edit. Developers of active documents under I6 use both methods, the former often being easier. For example, a developer usually enters text or graphics into the active document by using the built-in editor for such objects. But the developer also often evaluates his or her document Lisp scripts to test them on the active document under edit. There is no race condition in permitting both types of editing, since there is only one event loop in the system. I6 processes end-user editing requests and programmatic requests in the order in which they arrive, whether from the current or an external process. The only difficulty of having documents open for both interactive and programmatic editing arises when some lower level programmatic interfaces mistakenly do not queue up redisplay requests, which leads to temporarily confused displays.

Once an object is made active, the developer has to provide a way for that object to remain active even after its document has been saved and closed. This can be done by providing an object *save* method. The document sends *save* messages to all objects at document save time, passing them the actual file stream object. A custom save method can write out code to be evaluated at open time, thereby reactivating such objects. The custom *save* method is usually a *before* or *after* method, which gets called *in addition to* the default save method for that object type. The default save method simply writes out the object itself.

An alternative to providing an object save method (which saves object-specific code and/or data next to the object in the main document file stream) is to save code on the document itself, either within the main document file stream or in an auxiliary file containing method definitions. The current version of I6 allows a document to be represented in several different *part files*, to allow high performance updating for some data such as autonumbers, and to allow some updates without needing to lock the main document file against editing. One such part file is the *methods* file, described below.

9.3 Activation; security issues

Active behavior occurs in these ways:

- If a document has an auxiliary methods file, it is loaded and evaluated when that document is first encountered by the software. Many I6 active documents only define their classes and autoload their methods in initialization contained in the methods file.
- If a document contains active objects within the document file stream (as opposed to the methods file), these become activated when the document is opened programmatically or by the user for viewing, editing, or printing.
- At software startup, all objects within the user's *profile* container are sent a *load* message, calling them to action. Users thus place initialization documents or scripts in their profile container.
- Document objects can have user-defined attributes for many purposes, such as to mark conditional content or to store data to be accessed by Lisp programs. There are some specially named attributes which, when they exist, are evaluated at document open

time. For example, such attributes are used by office memo templates to initialize the content of the *Date* and *From* fields.

- Users can *explicitly* activate an object by selecting it and issuing a Load menu command, or by explicitly evaluating some code that does this.

Users are not always cognizant of the first four scenarios above. Someone could simply copy an active document into a user directory, and when that user opened the directory, the activity would be triggered, possibly with undesirable results.

The system contains two facilities to limit activation. The first is to run the software with a *-static* option on the command line, which disables automatic activation described in the first four scenarios listed above. The second is to define an "active load hook" that gets called prior to each automatic activation. It is passed the current object and the code that is about to be evaluated. A sophisticated program can examine this code and decide whether or not to permit its activation. We note that examination of code by code is easily done in Lisp, which makes no distinction between code and data. A simple example of such an inspector is distributed with the system.

9.4 Active document delivery issues

I6 active documents can be completely self-contained and easily activated. Simply copying such a document from one filesystem to another *installs* that active document application. This suffices since the object methods are stored within the document, and the copy of the document will have its active objects activated exactly as the original.

The active document implementor can choose between building such self-contained documents (which contain all their own data and code) and building "layered applications." The latter use a code library scheme, in which the code is installed in a standard place, to be shared by all instances of that document class. Self-contained active documents are easier to move around, but in cases where there are many instances, a library scheme saves space and makes updates easier. Interleaf has delivered some self-contained active document applications that can automatically update themselves if a newer version has been encountered.

In our judgement, one important requirement for automatic activation of documents is run-time binding in the document extension language—this is one of the reasons we chose Lisp. With languages such as C++, for example, newly introduced active documents or active objects must first be compiled into existing systems.

10 CONCLUSIONS

Our experience with active documents leads us to the following positions:

- Lisp is a good language for active document development due to features such as the interpreter (which allows nice development tools), functions as data (it is easy to store and manipulate programs as data on document objects), and run-time binding (which is important for the delivery of active document objects).
- Use of the object system simplifies object activation, as discussed in the multimedia photo album example.
- Active documents are useful for both document-based UI (e.g., IMA) and emulation of standard UI paradigms (e.g., CoAuthor).

- Active documents need not appear active to all users (e.g., the auto-localizing templates).
- It is difficult to catalog complete requirements for an active document building system without having many examples. I6 extensibility benefited from experience with several dozens of active document applications built with I5.

We believe the following areas need more research:

- Non-programmer tools for building active documents.
- UI guidelines for active documents, such as how to switch between build and run modes.
- UI experimentation on how the document UI model fits into standard GUI environments —mixing document objects and standard GUI widgets.
- Tools for finding, editing, and debugging active objects within a document.

ACKNOWLEDGEMENTS

Thanks especially to Ed Blachman and Robert A. Morris for participation in a mini-seminar on active documents, and for careful review of this paper. Thanks also to several of our Interleaf colleagues for comments. Thanks to Francis H. Yu of Oracle Corporation and to Laryssa Kachorowsky and John Wronski of Sikorsky Aircraft for information about their active documents.

USAGE INFORMATION

Interleaf 6 (I6) is being released in October 1993. Interleaf 5 (I5) was released in 1989, and currently has over 200 000 users. The Developer's ToolKit (DTK) is a set of development tools and documentation used to customize I5, integrate it with other applications, and to build active documents and document-based applications. Over 1000 copies of the DTK have been sold, and Interleaf's customer support organization has worked closely with over 100 DTK customers. Internally more than 100 employees (in field offices, in engineering, documentation, QA, marketing, and customer support) have written some type of application based on I5. Universities can generally receive full I5 licenses (including DTK) for only the cost of media. We have not tracked university use of our DTK, although we have seen discussions and examples posted on Usenet.

REFERENCES

1. Bertrand Meyer, *Object-oriented Software Construction*, Prentice Hall, Hemel Hempstead, Herts., England, and Englewood Cliffs, N.J, U.S.A., 1988.
2. Guy L. Steele Jr., *Common LISP, The Language*, Digital Press, 1990.
3. Richard M. Stallman, 'Emacs, the extensible, customizable self-documenting display editor', *SIGPLAN Notices*, **16**(6), 147–56, (June 1981).
4. Robert Spinrad, 'Dynamic documents', *Harvard University Information Technology Quarterly*, **VII**(1), 15–18, (Spring 1988).
5. Polle T. Zellweger, 'Active paths through multimedia documents', in *Proceedings of the International Conference on Electronic Publishing, Document Manipulation and Typography* [19], pp. 19–34.

6. Dennis Arnon, Richard Beach, Kevin McIsaac, and Carl Waldspurger, 'Camino real: an interactive mathematical notebook', in *Proceedings of the International Conference on Electronic Publishing, Document Manipulation and Typography* [19], pp. 1–18.
7. Eric A. Bier and Aaron Goodisman, 'Documents as user interfaces', in *Proceedings of the International Conference on Electronic Publishing, Document Manipulation and Typography* [20], pp. 249–262.
8. Douglas B. Terry and Donald G. Baker, 'Active Tioga documents, an exploration of two paradigms', in *Proceedings of the International Conference on Electronic Publishing, Document Manipulation and Typography* [20], pp. 105–122.
9. Daniel Swinehart, Polle T. Zellweger, Richard Beach, and Robert Hagmann, 'A structural view of the Cedar programming environment', *ACM Transactions on Programming Languages and Systems*, **8**(4), 419–490, (1986).
10. Donald D. Chamberlin, Helmut F. Hasselmeier, and Dieter P. Paris, 'Defining document styles for WYSIWYG processing', in *Proceedings of the International Conference on Electronic Publishing, Document Manipulation and Typography* [19], pp. 121–138.
11. George Towner, 'Auto-updating as a technical documentation tool', in *Proceedings of the 1988 ACM Conference on Document Processing Systems* [21], pp. 31–36.
12. Wilfred J. Hansen, 'Enhancing documents with embedded programs: how Ness extends insets in the Andrew ToolKit', in *Proceedings of the 1990 IEEE International Conference on Computer Languages*, pp. 23–32, New Orleans, (12–15 March 1990).
13. Yaron Goldberg, Marilyn Safran, and Ehud Shapiro, 'Active mail—a framework for implementing groupware', in *Proceedings of the ACM 1992 Conference on Computer-Supported Cooperative Work*, pp. 75–83. ACM Press, (1992).
14. Paul M. English, Ethan S. Jacobson, Robert A. Morris, Kimbo B. Mundy, Stephen D. Pelletier, Thomas A. Polucci, and H. David Scarbro, 'An extensible, object-oriented system for active documents', in *Proceedings of the International Conference on Electronic Publishing, Document Manipulation and Typography* [20], pp. 263–276.
15. Ramana Rao, Stuart K. Card, Herbert D. Jellinek, Jock D. Mackinlay, and George G. Robertson, 'The information grid: a framework for information retrieval and retrieval-centered applications', in *Proceedings of the Fifth Annual ACM Symposium on User Interface Software and Technology*, pp. 23–32. ACM Press, (1992).
16. 'Markup requirements and generic style specification for electronic printed output and exchange of text', Technical Report Military Specification MIL-D-28000, CALS Policy Office, DASD(S) CALS, Pentagon, Room 2B322, Washington, D.C., (1988).
17. 'Specification for manufacturers' technical data', Technical Report A.T.A. Specification No. 100, Revision No. 30, AIR Transport Association of America, 1709 New York Avenue N.W., Washington, D.C. 20006, (1991).
18. Richard Ilson, 'Interactive effectivity control: Design and applications', in *Proceedings of the 1988 ACM Conference on Document Processing Systems* [21], pp. 85–92.
19. *Proceedings of the International Conference on Electronic Publishing, Document Manipulation and Typography*, Nice (France), April 20–22 1988. Cambridge University Press.
20. *Proceedings of the International Conference on Electronic Publishing, Document Manipulation and Typography*. Cambridge University Press, September 1990.
21. *Proceedings of the 1988 ACM Conference on Document Processing Systems*, Santa Fe, New Mexico, 5–9 December 1988. ACM Press.