The following paper was originally published in the
Proceedings of the Fourth USENIX Tcl/Tk Workshop
Monterey, CA, July 10-13, 1996

Tcl/Tk as an OpenDoc Scripting Part

Jim Ingham
AT&T Bell Laboratories

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org

Tcl/Tk as an OpenDoc Scripting Part

Jim Ingham
AT&T Bell Laboratories
jingham@mhcnet.att.com

Abstract

This paper describes the advantages that would accrue both to Tcl/Tk and to OpenDoc by importing the Tcl/Tk scripting system into the OpenDoc environment. First, it gives a brief outline of the OpenDoc compound document architecture. Then it describes the place Tcl/Tk could fill in the OpenDoc scheme. Finally, it gives a preliminary sketch of the areas in which Tcl/Tk would need to be modified in order to implement this merger.

Introduction

The OpenDoc component document architecture[1-6] promises to revolutionize how modern applications are deployed to users. Backed by the Component Integration Laboratories (including Apple, IBM and Novell), it promises to cure the bloat of modern applications, which feel the need to provide every feature possible within one shell. More important, OpenDoc provides unique opportunities for smaller developers, since they can concentrate on solutions in their particular area of expertise, without having to provide the whole environment within which their solutions can function.

The basic idea of OpenDoc is to cast the modules that comprise an ordinary application: text editors, spell checkers, picture editors, a differential equation solver, an FTP channel …, as separate "parts" that can be embedded in arbitrary combinations into a document shell. The document shell, provided by OpenDoc, controls the event loop and storage, arbitrates access to system resources like the menubar, mouse and keyboard, and provides a channel of communication among the parts for geometry management and data passing.

OpenDoc provides a rich language by which the parts communicate, but it does not come with many prebuilt features. While the geometry arbitration mechanism is well defined, the developer is free to choose the particular model of geometry management according to his or her needs. OpenDoc uses the Open Scripting Architecture (OSA)[1,7-8] to facilitate data passing among the parts, but does not provide any particular scripting language. Nor does it come with a toolkit for implementing ordinary GUI functions (buttons, listboxes,…).

This situation provides an interesting opportunity for deploying Tcl/Tk in a new context, which has many attractive features. If we could make Tk a "part" in the OpenDoc world, then the creators of OpenDoc content would get the ease of development and the rich widget set for their documents that has been such a boon in the X world. At the same time, Tk developers would gain the ability to embed whatever parts are available in the OpenDoc world into their application, without having to build extensions for those parts. This kind of embedding could fill the role that the complex VBX's fill in the Visual Basic world.

Also, since OpenDoc is intended to be a cross-platform architecture, the 7.5/4.1 versions of Tcl/Tk are a natural choice for the scripting architecture/widget set. In the rest of this paper, I will give an outline of how Tcl/Tk could be used as a tool in the OpenDoc environment. First I will provide a brief summary of OpenDoc. Then I will sketch the enhancements that need to be made to Tcl/Tk in order fulfill this role. There are several levels of OpenDoc incorporation, and while the tightest coupling would require some rather low-level changes to the Tk core, there is an intermediate level that could be done fairly surgically. I have started the work in some of these areas, mostly at this point as an assay of the level of difficulty.

Although there is much work to be done, there do not seem to be any show-stopping difficulties, and results are promising enough to make the effort worthwhile. This paper is really meant to be a call for an interest group. I intend to show why this is an interesting project and sketch its parts, in the hope that I can find some other developers who are interested in cooperating in its implementation. Note that the following discussion is in the context of the MacOS port of Tcl/Tk. There are also ports of OpenDoc to AIX and OS/2 currently available, with a Windows port due in the middle of the year[5], but I have no experience with these ports.

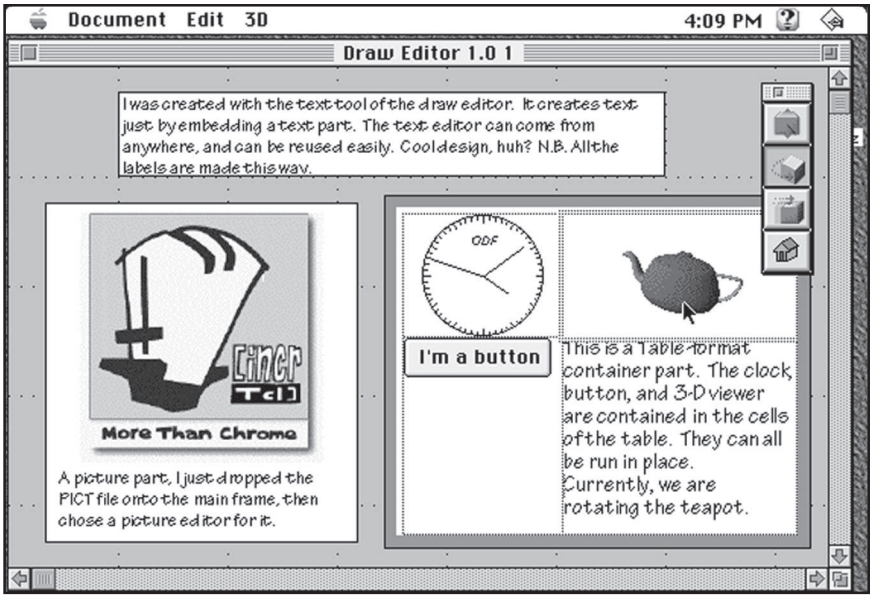

Figure 1. Here is an example of an OpenDoc document. The root part is a simple drawing editor. The Toaster picture is an embedded PICT, which is controlled by a Picture editor that comes with OpenDoc. The Clock, teapot and button are all embedded in a Table part. The teapot part, which is currently active, is a QuickDraw 3D document. This display was created with parts from Apple's OpenDoc Developer Release 5 CD.

Summary of OpenDoc

The fundamental idea of the OpenDoc model is that the document, rather than the application, should be the central focus of attention. The motivation is to give the content provider control over what elements go into each document: If I don't need pictures, then I don't need the picture editor most word processors now routinely include… And if I need a picture editor, or some other special service embedded in a text document, I should be able to choose one made by someone whose primary interest it was to provide that function, rather than the knock-off required to meet some feature spec sheet.

Each OpenDoc document is broken down into a number of "parts," each of which is controlled by its own "part editor." A part editor could be a text or pic-

ture editing facility, a spell checking facility, a TCP/IP socket interface… In a given document, there may be several parts that are controlled by the same part editor. By implementation, the part editors are actually class definitions, derived from a base class provided by OpenDoc, and the parts are instances of their part editor class. So although the parts owned by the same part editor share code, they are really independent entities, with their own data and state.

The user does not manipulate part editors directly. Rather, the developer of the OpenDoc part also makes a "stationery" document for the part editor. This can be as simple as a template for one part, or that part with some state information included, or can be a whole conglomeration of parts. The user opens the stationery document, which automatically makes a copy of itself for the user to edit. The stationery documents also support a drag and drop mechanism. So if you want to add a picture to a text document you are editing, you drag the stationery document for the picture part editor from the Finder onto your text document, and drop it wherever you want the picture to go, then interact with the picture part to load in the desired image. Thus you can make a complicated collection of parts, save it as a stationery document, and it becomes an atomic unit that you can later insert into another document.

Each part usually controls some screen area. For instance, a text part would have some area in its document's window to display its contents. Each contiguous bit of screen area in a window that a part occupies is called a "frame." Each part can have more than one frame, and the frames owned by a given part need not be in the same window. So, for instance, if I wanted to put a long bit of text into a newsletter document, I could break the text's part up into a number of frames, and spread the frames throughout the newsletter. There is no requirement that the frames be rectangular, though this has little implication for our purposes. More importantly, the frames of a given part can be constructed in such a way as to contain other parts.

This containing relationship can be used to solve long-standing problems, like inserting a picture into a text document. One frame of a text part could contain a picture part which is in charge of displaying the image. The containing frame (the text editor in our case) manages the geometry of the embedded part (the picture editor), but has no other responsibility for it. When redrawing is necessary, OpenDoc forwards update requests to all the embedded parts, and each part has the responsibility for drawing itself. Thus the coupling between an embedded part and its container is minimal. The embedding can go on to any depth, so a cell of a spreadsheet can contain a text part that contains an embedded picture, and so on.

To maintain consistency in this heterogeneous environment, all user interaction is routed through the OpenDoc shell. For each system service — the mouse, the keyboard, the menubar and the systems ports — there is an associated "focus." Parts negotiate, through the OpenDoc shell, for the focus for a given input stream. In general this is done by "activating" one part, which then requests the keyboard and mouse focus. There are user-interface conventions for how the active part is displayed, and how the activation is switched from one part to another (this is done by a single mouse click in the part to be activated, for instance). Once a part gets the focus for a given input, the OpenDoc shell will route all such events to that part.

Finally, the OpenDoc shell provides the facility for one part to send scripting events to another part in the document, using Apple's Open Scripting Architecture (OSA). The OSA provides a well defined way for parts to publish their interface to other parts, and to OSA aware editors. To achieve this communication, the parts build up "Apple Events," which describe a set of steps to be taken using this interface, and pass them back and forth. Any part that is OSA compliant, i.e. that has this interface, can take part in these scripting exchanges.

Tcl/Tk in OpenDoc

Let's start with a scenario of how Tcl/Tk could be used in OpenDoc. Suppose I have written a 2-D differential equation solver, which I have incorporated as a Tcl extension so that I can script it. Then I wrote a nice Tk front-end to control the simulation. Now I need to display the data. Since it is not entirely trivial to write a good 3-D data viewer, particularly one that can do convincing coloring and shading, rotations, etc., and perhaps produce a QuickTime movie of the results, I will look for a commercial package to do the task.

All this is possible in the context of ordinary applications, however, the presentation is fragmented. I start up my Wish, do the simulation, and then start up the 3-D viewer, play with it a bit, then go back to the Wish to redo the simulation. I have to have a data file somewhere to store the input cards, and the results somewhere else, and my movies in yet another file…

If both Tk and my 3-D viewer were OpenDoc parts, however, I could make up a single Document that contains as its parts, the Solver engine, the GUI and the viewer window, sort of a poor-man's Mathematica. Later, if I wanted to run a series of simulations, I could add a spreadsheet part to store a table of test cases, write a Tcl script to get the columns one by one from the spreadsheet, run the simulation for each column, and write back some results to the spreadsheet. I could even embed the QuickTime movies of the results as icons in the spreadsheet, if the spreadsheet was written to contain other parts in its cells. Finally, my Tcl script could drive the spreadsheet to do some analysis of the results.

All this would be within a single document, so I can hand it off to a colleague to use, without having to give complicated instructions on its use (for instance describing the naming convention that links the input and output data sets). Furthermore, I can make copies of the document, each containing a separate test sequence, and each copy is self-contained, with all its data and results in one place.

This picture relies on two separate facilities: the intercommunication of the parts which makes joint operations possible, and the common shell by which a single environment can be constructed around all these functions, some of which I control (the Tcl/Tk and solver engine), and some of which I don't even have libraries for (the spreadsheet and 3-D viewer). I will discuss in the next section what needs to be done to provide these two facilities.

The advantage of this integration goes in two directions. Tk brings to the OpenDoc world an extensible (I could incorporate my 2-D solver) scripting language, with a well-designed widget set. In return, Tcl/Tk applications will gain access, in a particularly seamless and attractive way, to custom facilities that are not available as libraries for incorporation into the Tcl shell.

The whole discussion is speculative at present, since quality applications available as OpenDoc parts are just coming on the scene. Cyberdog[9], Apple's OpenDoc based Internet Environment (with integrated e-mail, internet news service and WWW browser) shows great promise. It has some nice primitive widgets, like embeddable button parts that are linked to a URL, and on activation raise the appropriate browser window for that URL.

But this will likely change in the near future; OpenDoc won Best Technological Achievement of 1995 at Comdex[10], and has been consistently rated superior to the only real competitor, MicroSoft's OLE (see ref. 2 for an exhaustive discussion of the merits of OpenDoc over OLE). It was chosen by the Object Management Group as the standard desktop service for the CORBA environment[11]. The OMG is an industry group responsible for developing standards for object technologies.

There is an impressive list of commercial developers committed to writing OpenDoc parts (see http://www.cilabs.org for details). Claris announced that the next version of ClarisWorks will be an OpenDoc container part. Netscape Communications announced at this year's WWDC conference that it will provide some support for OpenDoc, but the details were sketchy. And at the same time, Tcl/Tk has an opportunity to make a contribution to this development precisely because there are no well established competitors.

Changes to Tcl/Tk

This section falls under two heads, OpenDoc compliance, and OSA compliance. It is worth pointing out that these two are independent tasks. In fact, the OSA compliance should be implemented regardless of whether the OpenDoc work is attempted at all. It is the logical way to implement exec and perhaps send on the Mac. Both the Tcl-based Mac Editor Alpha[12] and the Mac implementation of Perl offer Apple Event support. In the case of Alpha, this has made it a very powerful plug-in editor for the major Mac development environments. Without some kind of Apple Event support, Tcl/Tk will be a mute and isolated participant in the MacOS world. I have made a start at implementing the OSA interface to Tcl/Tk, and will have a beta release of this by the time of the conference.

OpenDoc:

There are two levels of OpenDoc incorporation offered by the Apple OpenDoc development group. One is to make Tcl/Tk a full fledged OpenDoc part, capable of being embedded in other parts, and of containing embedded parts. This is called a container part, in OpenDoc terminology. It is obviously the best way to proceed, since it would allow us to embed Tk controls in any other document, and to embed viewers, spreadsheets or whatever parts we wish within the Tk layout of our application. However, it also takes the most work, some of which is not entirely trivial.

The second is to make Tcl/Tk be an embeddable part, but not a containing part. This would allow us to put Tk controls and display widgets into a document alongside other parts, and use the Tcl interpreter to run our own extensions, and communicate with the other parts. However, we would not, for example, be able to embed the 3-D viewer into a Tk text widget, or use the Tk scrollbar widget to drive a non-Tk part.

NECESSARY ENHANCEMENTS: The full OpenDoc support requires changes or enhancements in four major areas, the event loop, windows, part activation, and part storage. The following is a bare sketch of each of these four areas.

The Event Loop: In any compound architecture like OpenDoc, the shell has to control the event loop. In order to switch control between the various parts, and also allow the parts to share idle time, there has to be a central router. In OpenDoc, each part has a HandleEvent method that does the actual business of responding to a single event. The document shell takes each event off the event queue, determines which part should respond to that event, and dispatches the event to that part's HandleEvent method.

The Tk event loop would have to be replaced by an event handler, which can do what it wishes with the events in Tk's internal event queue, but cannot poll for system events. This is not too difficult to implement. The current model in Mac Tk is that each time through the event loop, the top of Tcl's internal event queue is serviced. If there was an event in the queue, it is handled and the event loop is restarted. If the queue is empty, then the various event sources (file events, as well as UI events,…) are all polled for events. The new events are not processed at this time, but just added to Tcl's internal queue. Then the top event in the queue is serviced.

In OpenDoc, the handler would just put the event that woke it up onto the queue, then service the top event. The polling for file events, etc.. would be done when the handler was invoked for an idle event. It is yet to be seen whether this will cause performance problems, but it is the model for all the OpenDoc applications, and none of the currently available parts seem at all sluggish, so it should not be a problem.

Windows: One major change in Tk required by OpenDoc is that the main window of the application may not be a top level window. After all, we want to be able to embed a Tk layout into a document. This does not seem a very hard problem, however. We can just make an ODToplevel widget, which is just like a TopLevel, except that its window manager is the OpenDoc shell rather than the MacOS window manager. Tk would draw all its component widgets into this ODToplevel just as it would into an ordinary toplevel. Then it passes its final geometry request to its containing part.

There is some work to be done to implement the "wm" and "winfo" commands for these ODToplevels, but OpenDoc possesses a full set of calls to get geometry information, and arbitrate for new geometry, so the groundwork is well laid. One requirement for the

ODToplevel, and ordinary Toplevels, is that they must be registered with OpenDoc. This is so OpenDoc can do event routing, and dispatch update requests. This registration is accomplished by a fairly simple call, though we have to get access to our parts object to make it. Thus windows in OpenDoc Tk must have a part pointer added to their data storage. This can just be a ClientData pointer hung off the Tk_Window structure, so it is not too intrusive.

If we want to implement Tk as a containing part, we will also have to provide an ODFrame widget. This will be the containing frame for an embedded OpenDoc part. Each OpenDoc part is responsible for informing its containing part of its geometry requirements, so the ODFrame has all the information it needs to negotiate for position within the rest of the Tk layout. Further, each OpenDoc part has the responsibility to draw itself, so we don't need to do any work to present the contained part.

There are some User Interface guidelines which indicate which part is the active one in the document; the ODFrame widget would need some custom border drawing routines to conform to these requirements. Since OpenDoc does all the event passing, the ODFrame's responsibilities to its contained parts are limited to activating them when the frame is clicked in, and redrawing the borders in accordance with the UI Guidelines when the contained part becomes inactive.

Part Activation: When the Tk part is activated (for instance, when it is clicked in, or when it is first created) OpenDoc requires a number of setup routines be run. The part has to request focus for the events it is interested in (mouse events, keyboard events, as well as other system functions, like communications ports). This is done through set of fairly simple library calls, and will be easy to do. The UI requirements for displaying the active part (there is a little double line border in the current implementation) must be implemented. This frame would just be a special -relief option for the ODToplevel.

Again, when the part is deactivated, there is a protocol that must be followed to resign the foci which the newly activated part requests. The general UI standard for OpenDoc is that a part withdraw its floating palettes when it becomes inactive, and redraws them when it is reactivated. This would also have to be implemented.

Part Storage: Each part is responsible for reading itself in from storage, and writing itself out to storage. This is so the document can be saved and restored to this state later. In Tk, this is relatively easy, since the Tcl/Tk architecture allows you to query out the state of the interpreter. You know what all the global variables are, and you can just run the Tk tree to find out all the widgets, their contents and their positions.

You also need to track the proc definitions, what files are open, what fileevent handlers have been registered, etc. The state of the application can be "pickled" as this set of Tcl/Tk commands which restore this state. Doing this quickly and efficiently may take a little work, but all the facilities are already available in Tk. Of course, any extensions that you add to Tcl will have to have this ability to pickle themselves as well, if they are to coexist in OpenDoc.

Open Scripting Architecture:

This is the generic language by which parts in OpenDoc (and applications under MacOS) communicate with each other. The idea is that each part specifies the objects and operations that it wants to export to the world. For instance a generic part has objects like documents and menus. A word processor has, in addition, paragraphs, lines and words of text. These objects have various properties, like the entries in a menu, or the font of a character. There are also actions that can be performed on these objects, like setting the font of a word object, hiding a window, or telling an object to run one of its methods.

These verbs and objects are combined into "suites," many of which are standardized by Apple. There is a "Required suite" of operations that a generic application must perform: open, open document, print, quit and run. There is the "Core suite" that supports basic operations like get, set, delete and save. There is a "Text suite" that adds objects like characters, words, and paragraphs. You can support all of a suite, or any part of it that is appropriate. And of course, any part is free to add its own suites as well.

Each part (or program) has a Dictionary resource (the 'aete' resource). In this resource, each of the program objects and verbs is specified both by an internal four-letter code and by some "Human readable" information on the object. Another application can read this resource, and determine the scriptable objects in your application. An example of this is given

in Fig. 2. In this figure, Apple's Script Editor application has read in the dictionary, and presents the human readable component in the window at the lower left.

At startup the object and verb codes are linked to handlers within the part. When another part or application wants to send an instruction to the part, it constructs a data structure describing the operation and the objects it acts upon (called an "Apple Event"). This is a nested structure, so that objects

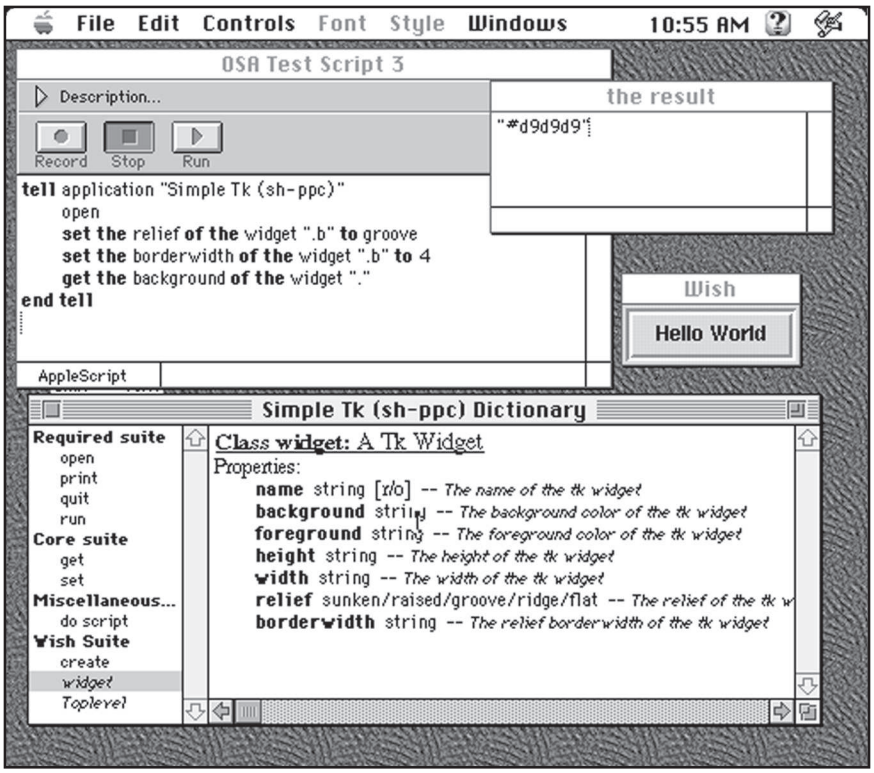

Figure 2. This is an example of an OSA model for Tk. There is a widget class that encompasses the basic widget behavior. Each of the widgets are derived from widget. The configuration options are accessed through the get and set verbs. The widget commands will be accessed by the "tell" verb. This is my current focus of interest, and I will release a working beta soon.

can be specified in quite complicated ways. For instance a complicated object specification like "the first paragraph in the window "My Article" whose font is "Courier"" can be quite easily built up.

Then it passes this Apple Event to the Apple Event Manager, which routes it to the target application or part as a "High Level" event. In OpenDoc, the shell passes the event to the parts HandleEvent method. A MacOS application receives the instruction as an entry in its event queue. In either case, the target application responds to the event by parsing it up into its parts, resolving all the object references, calling out the handlers associated with the verbs, and finally returning a reply Apple event, if the incoming instruction requested a result.

For our purposes, we do not need to provide an interface to all the facilities within Tcl/Tk, since it already has its own scripting language. If a part wants to build up a Tk GUI layout, the best way would be to create a Tk part instance and send it the necessary Tcl code. This is already possible in Tk4.1, using the standard "do script" event.

Note that the OSA provides another, even nicer, way to run Tcl scripts. You can register the Tcl interpreter as a "Scripting Component." A scripting component contains the interpreter as a shared library. Any part can ask the component manager for an instance of the interpreter, and then feed commands to it. For more details, see chapter 10 of ref. 7. This method is useful because you do not have to start up an external wish part to run your scripts: a part could contain the Tk component internally. It is not appropriate, however, if you want the Tk GUI to drive multiple parts.

Vince Darley has made a start on this approach for the MacOS by wrapping Tcl as a scripting component With this, you can use any of the standard script editors (for instance Apple's Script Editor) to write your Tcl code. However, before we can do the same for Tk within OpenDoc, we have to separate out the Tk event loop as I discussed in the previous section.

When designing an OSA model for Tk, we should concentrate on the things a client part will really want to be able to do. This consists mainly of allowing other parts to activate elements of the Tk GUI, to access and modify data in the text, entry, listbox and canvas widgets, and perhaps to modify configuration options of the widgets. I have made a start on this, as is

shown in Fig 2. I have implemented an OSA extension which can get and set configuration options, and, using the AppleScript standard "tell" construct, activate widget commands. The next job is to apply the objects in the OSA "text suite" to the text, listbox and entry widgets. I am currently working on this.

It would also be very useful for Tk to have some facility to build up an Apple Event and dispatch it to another application. The exec command is not currently implemented in the Mac port of Tk. This is because MacOS does not implement a Unix style exec. The Apple event mechanism fulfills this role in the MacOS. In fact, it is much more powerful than exec, since it also fulfills a role like the Tk "send" command. The dictionary mechanism is a very nice touch for documenting the interface.

The capability to build Apple Events already exists in the Tcl-based Mac Editor Alpha. It has a Tcl extension that allows you to build up arbitrary Apple Events from the object and verb codes, and send them. Peter Keleher has made the code available to Mark Roseman. It is based on a simple wrapping of a library (AEGizmos) provided by Apple, which is in the public domain, so we can very easily add it to Tk.

Another example of the use of this Apple Event builder is to implement "send" for Tk on the Mac. After all, "send" is no more than the doScript Apple event with the script as its data, sent between two Wish applications. This is not the best way to implement "send," of course, since it means that the command has very different implementations under X-Windows and the MacOS . But it would be a nice quick way to get this functionality. It could fill the need till a truly cross platform solution is devised.

Conclusions

First of all, I hope I have communicated some of the promise of the "document centered" User Interface philosophy. It allows the user to tailor "mini-applications," which contain the specific functionality they need. Moreover, each copy of the document contains both the description of the "application" that the user has assembled, and its state when he or she last left it. So, once constructed, the document can be replicated, reused, edited, and embedded as an atomic element in another document. OpenDoc transfers many of the "application designer" functions to the user of the application, and offers a flexibility and level of control that is quite promising.

However, at this stage the OpenDoc world is lacking a good solid GUI toolkit. This, by the way, is not some oversight on the part of the OpenDoc development group. The idea was to create an open architecture that developers could tailor to their own needs, rather than forcing a whole package down everyone's throats. This situation offers great possibilities, however, for someone who can deploy a solid GUI toolkit in OpenDoc. There are several reasons to think Tcl/Tk would fill this role elegantly.

Its extensibility would be a great asset, just as it is for stand-alone applications. The widget set has proved popular because it is easy to use, and yet very flexible. Currently OpenDoc only has a solid implementation under MacOS. However, as it is deployed to more platforms, we can repeat the steps we took to embed Tcl/Tk into MacOS OpenDoc, with the confidence that the engine is well tested on those platforms. This would not be true of any widget set that was developed primarily on the Mac.

I will conclude by describing the steps required for the incorporation of Tcl/Tk into OpenDoc. First of all, we need to refit the event loop part of Tcl, converting it into a handler based architecture. I made a quick test of this with one of the early 4.1 beta's, and there were no substantial impediments. I have been waiting for Tk to come out of beta before attempting the job in earnest.

The second crucial step is write an OSA interface to Tk. This can be done as a true extension, without modifying the Tcl/Tk core. The results will be of sufficient utility to MacTk in general, that the work should be done as soon as possible. I have made substanital progress on this task, producing a version that implements the basic widget objects. It can query and set the configuration options, and run widget commands. I will make a first draft of the inter-

face to the Text widget, and then release what I have done. Look to Mark Roseman's Mac resources page[13] for the results.

The next step is to embed Tk into OpenDoc. The simplest way to do this is to start by allowing only separate, Tk-controlled toplevels. These only need to be registered with OpenDoc, so that they receive events; other than that they do not differ from their Wish counterparts. This was easy in the early beta, and should suffice to get a test version running. Then we need to write an ODTopLevel widget, and finally the ODFrame widget.

Since I did my early experiments, the OpenDoc Development Framework[14] has also come out of beta. This is a framework from Apple Computer for writing OpenDoc parts. It takes care of many of the Human Interface details, and isolates the system specific parts of an OpenDoc part. It looks very promising, and can speed the writing of OpenDoc parts. I will use this framework to implement the TK embedding.

References

[1] Apple Computer Inc.
OpenDoc Programers Guide For The MacOS
Osborne McGraw Hill Press, December 1995

[2] R. Orfali, D. Harkey & J. Edwards
The Essential Distributed Objects Survival Guide
John Wiley & Sons, 1996

[3] A. MacBride & J. Susser
Byte Guide To OpenDoc
Osborne McGraw Hill Press, 1996

[4] Apple Computer OpenDoc Home Page:
http://www.opendoc.apple.com
You can get OpenDoc, the developer's kit, Apple's OpenDoc White Paper, and many other goodies from this site.

[5] IBM's Club OpenDoc Home Page:
http://www.software.ibm.com/clubopendoc/index.html
IBM is responsible for the OS/2, AIX and Windows Ports of OpenDoc. You can find information related to these ports , and other OpenDoc goodies here.

[6] CILabs Home Page:
http://www.cilabs.org
CILabs is the industry group in charge of coordinating the standards for OpenDoc. They help organize the interpart communications, and set up classifications for various part types (spreadsheets, text editors, database access, …)

[7] Apple Computer Inc.
Inside Macintosh: Interapplication Communication
Addison Wesley Publishing Company, 1993

[8] Dave Mark
The Ultimate Mac Programming Guide
IDG Books Worldwide, 1994
The first section of this book is about implementing OSA in a traditional Mac application. It also contains several useful reprints from Develop Magazine in the appendices.

[9] The Cyberdog Home Page
http://cyberdog.apple.com
Here you can download Cyberdog, read the Cyberdog white paper, and soon get the SDK for Cyberdog as well.

[10] The press release is at:
http://product.info.apple.com/pr/press.releases/1996/q2/960206.pr.rel.opendoc.html

[11] The press release can be found at:
http://www.cilabs.org/pr/pr.omg.html

[12] The Alpha Home Page is:
http://www.cs.umd.edu/~keleher/alpha.html

[13] Mark Roseman's MacTcl resources web page is:
http://www.cpsc.ucalgary.ca:80/~roseman/mactcl/

[14] The OpenDoc Development Framework Home Page:
http://www.odf.apple.com