A Close-Up of OpenDoc

By Kurt Piersol

OpenDoc is a new vendor-neutral standard for compound documents that offers cross-platform support.

In recent years, there's been an industry shift toward document-based computing and away from application-based computing. Starting in the 1970s at Xerox, and continuing with architectures like OLE from Microsoft®, this movement toward more natural ways of assembling documents is now gathering significant industry momentum.

OpenDoc is a new entrant into this field—one intended to be a vendor-neutral, open standard for compound documents. *Compound documents*, the key to this type of computing, are documents composed of many different kinds of content, all of which share a single file. These documents can contain almost any type of data, such as tables, charts, and text, as well as video, sound, note cards, or 3-D graphics.

A Quick Tour of OpenDoc

Typically you can edit any or all of these types of content in place in a compound document. This means that several editors can work on a document at the same time, unlike in today's computing environment where one editor owns the entire document.

Given a set of editors working together on the same document, there must be boundaries to sort out where one kind of content ends and another kind begins. Otherwise, it would be impossible to discern which editor should work on particular sections of the document. In OpenDoc parlance, these bounded sets of content are called *parts*.

Of course, the whole point of a compound document is to be able to mix the types of content, so a mechanism is needed that can put one part inside another without the parts losing either their identity or their boundedness. This process is called *embedding* because it's rather like the typical real-world embedding process. You can embed raisins into bread, for instance, but doing so doesn't change the fact that the bread is still bread and the raisins are still raisins. Still, the resulting raisin bread is tastier than either ingredient alone.

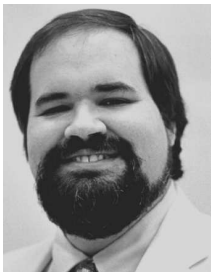
Thus, a compound document has various types of content, as well as parts embedded inside other parts. Each type of part has its own editor. All these parts share the same windows, as well as the same file on the disk or storage server. When you open an OpenDoc document, you're really looking at a collection of parts. When you edit the document, you're using a collection of editors.

These editors must work together smoothly. Editors have only a few basic tasks: storing the contents to disk (if needed), drawing the contents out to the screen or a printer, and letting the user act on the contents by accepting events such as mouse-clicks and keystrokes.

A compound document system needs to sort out the boundaries between the editors so that efficient editing can occur. OpenDoc does this by using a series of libraries that sort out the boundaries for the part editors. While this is an object-oriented system, it's important to note that OpenDoc isn't an object-oriented framework, because it wasn't designed to be extended through inheritance. Instead, it's an object-oriented interface between part editors that can be written in different programming languages and by different organizations. These libraries can be used via procedural code if required.

OpenDoc Components

OpenDoc has several important components that are used to organize a document's content and implement the sorting mechanism. The first of these is OpenDoc's layout system. OpenDoc



Kurt Piersol

helps the parts negotiate about layout so that they can avoid blasting bits over one another. The same layout system helps determine what editors are invoked and when they get mouse events. The layout system works for on-screen windows, off-screen bit maps, and printer contexts. It handles 2-D and 3-D graphics, overlapping parts, and multiple active parts that are updating asynchronously.

OpenDoc has an event-dispatching system that routes events to the correct part. It uses the layout system to let you activate parts directly, without allowing double-clicking or menus to get in the way.

OpenDoc's storage system helps parts store complex information in a shared file. It even helps with storing multiple document drafts and embedding information. This includes a data transfer system that helps the parts ship information—including embedded information—through the Clipboard, by linking, or by use of the drag-and-drop-mechanism.

Lastly, OpenDoc has a scripting system that lets users coordinate the actions of various part editors, either within a document or across a network and across platforms.

Rather than describing all these capabilities, I will instead focus on several of these areas in more detail. Some of the most interesting parts of OpenDoc revolve around how it dispatches events, handles storage, and does scripting.

OpenDoc Event Handling

OpenDoc is a cross-platform architecture, and event handling varies considerably in different GUI environments. For example, in the X Window System®, keystrokes go to the window that the cursor is over at that moment. On a Macintosh they go to the frontmost window at the insertion point. Other systems have yet other ways to determine where keystrokes should go. Similar interface discrepancies arise when you consider how to handle menus, windows, clipboards, and dialog boxes.

We members of the OpenDoc team made an early decision during the design process: Don't demand alterations of the human-interface environment. A good decision, but it left us with a problem. How could we make a mechanism that worked generically if the interface changed among platforms?

There were two options. One was to treat human-interface elements as a collection of special cases on each platform. The other, which we chose to implement, was to come up with an abstraction of the fundamentals of our problem and then fit it to each platform. Our solution was to build two major structures, the dispatcher and the arbitrator. The *dispatcher* is an object that helps the underlying platform's dispatcher to find the correct part editor. The *arbitrator* is a way for parts to tell the dispatcher which editor owns the stream of keystrokes, the menu bar, or any other shared resource. Together, they let OpenDoc work with the different human-interface models of different platforms.

The arbitrator is actually a table that shows the resource that can be owned and what part editor owns them. Each of these resources is called a *focus* of arbitration. To get resources, a part editor asks the arbitrator for a set of foci by name. The arbitrator then uses a two-phase commit mechanism to ask the present owners to give up ownership. The arbitrator asks the various owners to give up the resource in the first phase and then reassigns ownership in the second phase.

Networking and multiprocessing experts will recognize this technique as a standard way of preventing deadlocks involving resources. For example, imagine that one part editor owns the menu bar, and another has ownership of the keystroke stream. If each editor wanted the other's resource but refused to give up the one it owned, there would be a deadlock. By asking for resources as a set and assigning ownership in two phases, OpenDoc prevents this sort of "deadly embrace" between part editors.

OpenDoc's arbitrator is extensible, by both platform implementers and application developers. At any point, a new arbitrator focus can be added by creating an object called a *focus module* and adding it to the arbitrator. This extensibility means that new hardware resources can be managed through the arbitrator, as well as new software resources, such as server connections.

Most systems deliver events to windows, but because compound documents have parts that divide up those windows, a second stage is needed to get the events from the windows to the parts inside them. The OpenDoc dispatcher does this by taking human-interface events from the operating system and handing them to the correct part editor. This dispatcher is similarly open-ended.

Figure 1 shows how the OpenDoc arbitrator and dispatcher work. A stream of mouse and

OpenDOC has several important components to organize a document's content and implement the sorting mechanism.

The OpenDoc Arbitrator and Dispatcher

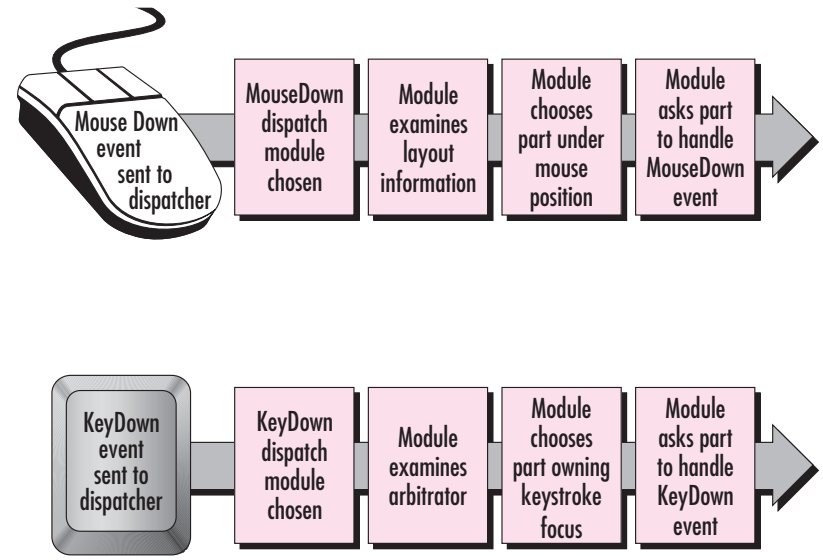


Figure 1. The OpenDoc arbitrator and dispatcher

keyboard events are directed to the dispatcher, which directs them to the appropriate editor handling that portion of the document. Where necessary, as in the case of the keyboard event shown in Figure 1, an arbitrator module may be required to decide which editor receives the keystroke.

Although some systems have nested windows that could theoretically be used for embedding, they usually don't have enough information about layout to make a very effective embedding tool. So OpenDoc provides a more sophisticated mechanism called a *frame*. Frames are part of OpenDoc's layout system, and they help the parts divide up the drawing area of a window into separate regions for each part. The use of frames allows OpenDoc to support overlapping parts that are simultaneously active, a requirement for many multimedia documents. The OpenDoc dispatcher reads the arbitrator and frame information to decide which part editor should get an event (see Figure 1).

Some systems, such as Windows, have a tremendous array of events that can be passed to applications. Others, like the Macintosh, have a much smaller set of such events. The OpenDoc dispatcher must be able to handle both extremes. It can have a new behavior added at runtime to handle new kinds of events. In addition, any part editor can monitor the dispatcher and watch the stream of events that passes through the dispatcher. Both monitoring and extension are accomplished through objects called *dispatch modules*, which are added at runtime to the dispatcher.

Storage in OpenDoc

Probably the single most interesting part of OpenDoc is its approach to storage. There are many different data formats on many different systems. Superficially, it appears that programmers have a perverse desire to define new storage formats that are different and incompatible. Even the same applications often have different formats on different machines.

But there are some really good reasons for the differences, however annoying they may be. There are some trade-offs that make choosing a file format difficult. The two most common trade-offs come when you must choose between standardization and innovation, and between publication and efficient editing.

The first trade-off problem arises when you want to add new features, but you have an existing file format of your own or there is a market need to support a particular format. Unfortunately, file formats are seldom extensible in a convenient way.

Then there's the second problem, which tends to arise when a standard interchange format has been proposed. The design of such formats is almost always centered around making them easy to read and extend. Unfortunately, this often means that they aren't designed for efficient editing or high performance on a particular machine. The compromise is often to support both formats and allow one to be rewritten as the other.

The worst problem from a compound-document point of view is that these formats don't mix very well. Efficient editing formats are often based on replaceable pieces (that is, strings of bytes) that can be randomly accessed through some sort of table of contents. This makes them very easy to edit efficiently, but they are very specific to a particular editor.

Efficient publication formats are often stream oriented, making them easy to understand for a program, without regard to its internals. Problems arise when you try to mix piece-based formats and sequential formats. If you insert piece-based information into sequential information, there are often unacceptable constraints about when and how new pieces can be added. When you put sequential information into piece-based formats, there are often problems with the size of the sequential information, and there's no way to extend the sequential information without

understanding the piece-table structure of the container.

OpenDoc addresses these problems by creating an open meta-format, a way to store both sequential and piece-based information in a file without conflict. Although OpenDoc can support many different storage systems, it specifies one implementation on each platform based on Bento, a compound-document storage format developed at Apple that was designed to address many of these problems, support multimedia, and work on many platforms (see Figure 2). OpenDoc storage makes no assumptions about whether formats are sequential or piece-based. It allows random or sequential access.

As shown in Figure 2, document parts can be stored anywhere within the file. These parts can then be reconstructed into a single stream of data by referencing the file's table of contents.

To explain OpenDoc's storage model, I'll work up from the simplest elements of storage to the full document structure. At the base of OpenDoc documents are stream-like entities called *values*. Every one of these values looks to an editor like a complete file. Each value has read, write, and seek operations, just like any typical file.

In addition, OpenDoc supports two extra operations, insert and delete, that insert and delete data into the middle of a stream without copying massive amounts of information. It does this by maintaining a table of contents that assembles random chunks of the file into the appearance of a stream. Inserts and deletes generally alter the table of contents instead of moving the contents around. Sequential formats can use stream operations, and piece-based formats can use insert and delete operations to alter data in place.

Values are collected inside objects called *storage units*. Every OpenDoc part has its own storage unit, which consists of a list of named properties, each of which has a list of typed values. Thus, a storage unit is a lot like a directory in a typical file system. Properties are a lot like named files, and values form the contents of each file.

Unlike a directory in a typical file system, however, a storage unit has the added advantage of being able to store multiple formats for every property. A file system has only one set of contents per name, whereas a storage unit can have several, each with its own type. This makes OpenDoc uniquely suited to storing multiple representations of a part. An OpenDoc part editor could, for instance, store both a standardized format and an efficiently editable format with a single name.

The Bento Storage Format

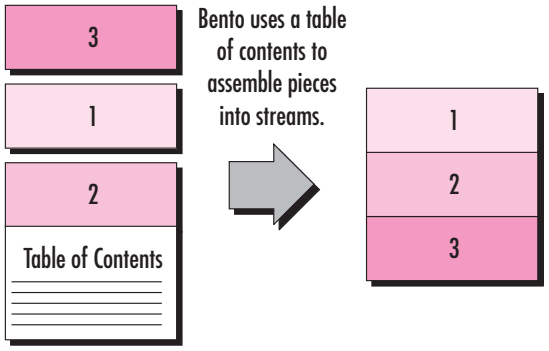


Figure 2. The Bento storage format

Storage units are collected into structures called *drafts*. A draft is really a list of storage units—a snapshot of the state of a document. When you make a draft in OpenDoc, you're really saving the state of the document for later retrieval, even if changes are made. OpenDoc is very efficient about how it stores drafts, using the information from read/write/insert/delete operations to store only the changes from the last draft.

One exciting thing about drafts is that values can use them to refer to storage units in a very robust and flexible fashion. Any value can include a reference to another storage unit in a draft. This allows pieces of data in various formats to have a standard way of referring to one another. As a result, a draft can support many different organizational structures, from simple hierarchies such as a file system to full hypertext webs.

Every draft has a pointer to the topmost part, or *root part*, of a document, which is used to open the draft into windows or to print the document. The rest of the document embedding is done using OpenDoc's reference mechanism. Thus, documents can have many different structures. The root part is in charge of it all, and it sets up the document structure based on its own content rules. A spreadsheet, for example, might allow embedding only on the main sheet, while a Rolodex editor might allow it on any card. The same part can be embedded in many locations.

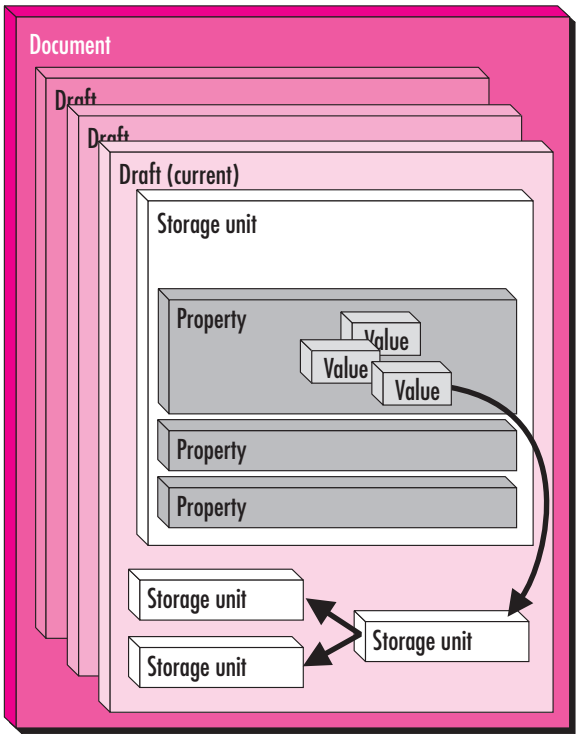


Figure 3. The OpenDoc storage model

Finally, there are documents that are just lists of drafts. One special draft, the *current draft*, represents the latest state of the document (see Figure 3). Users see the current draft upon opening a document. This structure is one of the key design points in OpenDoc. It supports more kinds of documents than any previous compound-document system. Even better, for part editors that don't need all its power, they can simply do stream I/O operations and ignore the rest.

Referring to Figure 3, values contain the basic atoms of data and look like files to OpenDoc editors. Values are collected inside storage units, which contain a list of typed values. A storage unit functions as a directory in a file system and can refer to multiple copies of the data in different formats. Storage units are collected into drafts. Each draft is a snapshot of the information. A document is composed of different drafts, the latest snapshot of which is contained in the current draft.

OpenDoc Scripting

OpenDoc scripting is designed with two major points in mind. First, scripting is a human interface, intended for users—not C programmers. Second, scripting is most valuable in the context of work-flow applications, where several kinds of data are used in some sequence to accomplish a task.

It's tempting for a designer to imagine that scripting, like automating user tasks, is just another programmatic interface: Just expose the innards of your editor and let users program it any way they like.

This doesn't really work, though. Imagine that you're creating a scripting system for a word processor. Internally, run-length encoding is an efficient way to store character-formatting information. However, at the scripting interface, you'd hardly want your users to understand the arena of run-length arrays so they can set the font of a word. Instead, you'd create an abstraction of the data that lets them select words, lines, or paragraphs, and let them set the font of any of those things.

The second major point of OpenDoc, work-flow, is just as important as the first point. A user usually has a task to complete that involves many distinct actions on many distinct kinds of data. To have a useful automation system for OpenDoc, you need to be able to script the actions of many parts in a single script. These parts could be in many documents, on many machines, across networks on many different kinds of hardware and operating systems. Even worse, some of the instructions need to be delivered through store-and-forward systems, such as E-mail. Clearly, this is a tough—but important—problem to solve.

The OpenDoc team chose to extend an existing solution to this problem. Over the past few years, Apple has created a scripting architecture that it calls OSA (Open Scripting Architecture), which solves much of this problem for the Macintosh.

OSA is a series of libraries that each address parts of the problem. There's a standard calling convention, called Apple Events, that allows applications to call one another over a network or on a single machine. OSA scripting systems can coordinate activities among many machines and many applications from a single script. Another library, the Apple Events Manager, simplifies the process of making and receiving Apple Event calls.

Another OSA library allows different scripting languages to call one another and can even be used to plug other scripting architectures into OSA. A good example of such an architecture is the OLE 2.0 automation interface. OSA provides a standard way to adapt other scripting architectures so that they can be called from OSA-compatible languages.

There's also a standard record of calls, called the Registry, that defines a common set of operations that most editors can support. Operations such as copy, paste, create, delete, and move are all defined in a standard way in the Registry.

Central to the design of the Registry commands is a standard way of naming selections in individual applications. This is one of the keys to verbal intuition and consistency in OSA. OSA's naming scheme, called *object specifiers*, lets users name individual objects or groups in a standard way. It lets users treat their documents as if they were a giant object database that they can query in a natural way. For instance, it's quite easy in OSA to set the color of "every cell whose value is less than zero" in a spreadsheet to simply "red" or set the color of "cell R1C2" to "green."

OSA also includes a standard way for applications to publish the kinds of objects they make available for scripting, as well as what can be done to those objects. This user terminology is described in a resource and allows OSA scripting languages to support many different language syntax forms. Thus, they can be internationalized to resemble natural spoken languages as well as existing computer languages. Since OSA was intended to be a cross-platform architecture, it fits in nicely with OpenDoc's goals.

However, the OpenDoc team needed to extend this naming architecture to handle embedding. OpenDoc includes a service, called the *name resolver*, that allows part editors to easily handle object specifiers. This service parses the object specifier and then calls back into the part editor to resolve object specifiers into pointers to the part editor's data structure. If the specification crosses a part boundary, the name resolver can switch contexts and begin resolving in the new part. It handles query-type specifiers and even does query optimization in some cases.

The OpenDoc team also needed to extend the OpenDoc dispatcher to deliver the Apple Event (which is called a *semantic event* in OpenDoc) to the correct part. This involved using the name resolver to determine which part should handle the semantic event and then dispatching the event to that part.

The result is quite striking. Users can refer to objects using the visible embedding structure of the document, saying things like "delete seconds 1 through 30 of the first movie of paragraph 1 of my document." These instructions can be delivered either in real time or through store-and-forward systems, thanks to OpenDoc's sophisticated naming scheme.

Getting Your Hands on OpenDoc

OpenDoc is being implemented right now. By the time you read this, it will probably be in its alpha-testing phase. Once that is finished, we plan to release the source code generally. A vendor-neutral organization, CI Labs (Component Integration Labs), will be the owner of OpenDoc (including a cross-platform version of OSA) and will be the point of contact for obtaining the source code. You can contact CI Labs at 688 Fourth Avenue, San Francisco, CA 94118, (408) 974-6549, or on the Internet at `cil@cil.org`.

There are white papers, specifications, and other documentation available from CI Labs right now, with more on the way. In the meantime, look for the various CI Labs companies to begin releasing seedings of the technology on their respective platforms.

Kurt Piersol *is a software architect at Apple Computer (Cupertino, California), where he leads design teams and coordinates design elements on a number of projects. Currently working on OpenDoc, he previously led the Apple Events project, coordinated the design of a Mac® Interapplication Communication system, and was the early technical leader for AppleScript™. You can contact him on BIM c/o "editors."*

Reprinted with permission, from the March 1994 issue of *BYTE* magazine. © 1994 by McGraw-Hill, Inc., New York, NY. All rights reserved.