OpenDoc—Building Online Help for a Component-Oriented Architecture

Melissa E. Sleeter

Human Interface Design Center
Apple Computer, Inc.
10500 N. DeAnza Boulevard, MS: 302-1H1
Cupertino, CA 95014 USA
Tel: 408 862-2911

E-mail: sleeter.m@applelink.apple.com

ABSTRACT

Component-oriented software allows end-users to extend or replace monolithic applications using components—software plug-ins that handle specific kinds of data and can be used to add functionality to documents. Building online help for component-oriented architectures raises issues that are exemplified in the specific case this paper examines—providing help for OpenDoc™ component software using the Apple Guide help system. Component-oriented architectures have characteristics that challenge a static, application-oriented help model, such as the original Apple Guide model. The solution requires extending a static help model in the following ways: generating a help view that is both component-oriented and dynamic, identifying a context for "context-sensitive" access, and defining how content will be integrated within the help view.

Keywords

Online help, component software, instructional design, modular design, semantic matching

INTRODUCTION

OpenDoc is a component-oriented software architecture available on the Mac OS, Windows, UNIX, and OS/2 platforms. Help for OpenDoc on the Mac OS platform was developed using the Apple Guide help system.

Component-oriented software architectures

Component-oriented software allows end-users to extend or replace monolithic applications using *components*—software plug-ins that handle specific kinds of data (text, spreadsheets, graphics, sound, network connections, etc.) and can be used to add functionality to documents. With components, users can easily create multi-media documents by adding to the document components that handle the different datatypes.

For example, the document in Figure 1 below includes separate components that handle text, graphic, and chart information. Users can edit each part of an OpenDoc document in place, without opening separate applications, by clicking a part to make it active and put its menus in the menu bar.

Figure 1. A component-based document

Component-oriented software represents a paradigm shift for both end-users and help developers.

Traditional software processes consist of a single application or operating system. In a component-oriented architecture, processes that behave like a single document, dialog box, control panel, or application actually consist of separate components. For example, all OpenDoc processes run within an environment called the OpenDoc *shell*, which provides certain basic behavior—opening and closing documents, adding components to a document, and basic editing behavior. Within any OpenDoc process, the user experiences the shell and the active component (with input focus) as a unified object and will want help for both. However, the shell is installed and updated independently of components.

This means that help content for the shell and components must reside in separate files.

"Context" becomes more complicated in component-oriented architectures. In traditional software architectures, the context with input focus (i.e., available menus) is the frontmost process, usually an application or operating system. In component-oriented architectures, the context with input focus is smaller than the process and is made up of several parts. In OpenDoc, for example, the context with input focus includes

- the active component (such as a word-processing component)—Only one component at a time has input focus.
- the OpenDoc shell—The menus provided by the shell (Document and Edit) are always available, along with the active component's menus.
- plug-in services (for example, a spellchecker)—Services may be available along with the active component.

The process-level context includes

- the active document (in a multi-document process)
- the active process (corresponds to a active application or operating system)

In traditional software, "context-sensitive" help usually means help for the current process. Component-oriented software requires that help developers consider and define a context for "context sensitive help" because a process consists of separate parts.

The Apple Guide help system

The Apple Guide help system supports the design and delivery of interactive on-screen instructions. These instructions are grouped in files, known as *guides*, which are written for fixed entities, such as applications. The example in Figure 2, SimpleText Guide, is a single help file that is available when the SimpleText application is active.

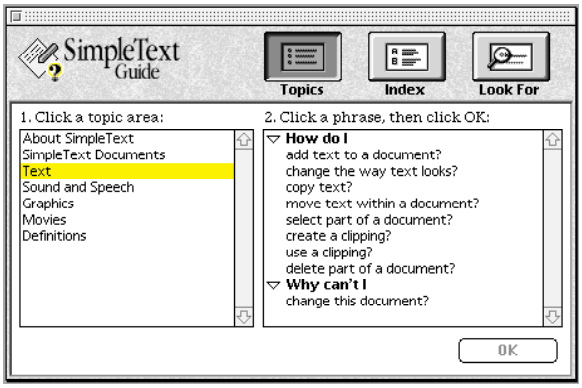

Figure 2. Apple Guide help for an application

When a guide is open, users can click buttons—Topics, Index, or Look For—to use alternate access strategies for help information. The left side of the access window shows a list of topics, a list of index terms, or a field into which the user can type a word or phrase. When the user selects a term on the left, a list of applicable help sequences appears on the right.

Apple Guide help is tightly integrated into the product its supports. As the user steps through instructions, the help system continuously interacts with the product software, using context checks to determine the state of the software.

A static help model

The original Apple Guide help model is static. It provides help for the active process—the operating system or an application—by opening a guide written specifically for the process. In the example below, a single file entitled "SimpleText Guide" provides help for the SimpleText application. When the application is active and the user invokes help, Apple Guide locates and opens the SimpleText Guide file.

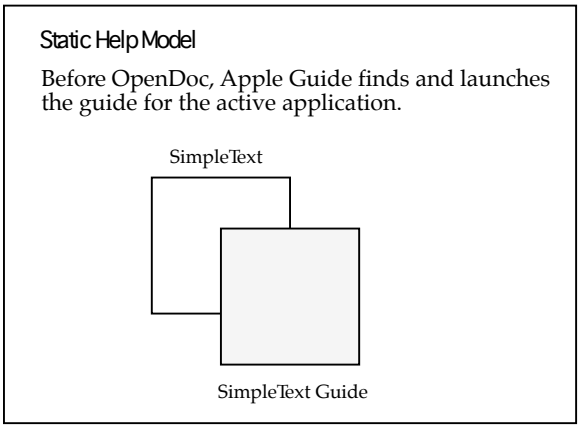

Figure 3. A static help model

Component-oriented architectures, such as OpenDoc, have characteristics that challenge this static, application-oriented help model:

- The functionality of a document (i.e., which components the user will include) is not known until the user creates the document. For example, a user can create an online sales report by starting with a text component, then adding components to handle spreadsheet and graph information. Another user editing the report could need help for any component in the document.

- Once created, a document remains dynamic because the user can add or delete components at any time. In the sales report example above, the user may decide to add a button component pre-loaded with a URL to connect readers directly to the company web site. Another user editing the report could now need help with the button component.

- Help developers can not know the context in which a component will be used. For example, an Internet-browsing application made of OpenDoc components may contain its own web browser component. However, the user can replace the original web browser with a third-party browser. A user asking for help will need help for the new browser, not the original one, and definitely not for both!

- The context for which help can be written (i.e., a static set of tasks) is usually only part of the context for which the user needs help (e.g., a document or application made up of many components). For example, help can be written for a single component, such as a button component. However, in a control panel made up of many components (including the button), the user probably wants help for the control panel not just the button.

- In some components, help must document separate "programming" and "use" modes. For example, help for a button part must tell users how to program the button (e.g., give it content, such as a sound to play, a script to execute, or a URL to connect to) and how to use the button (e.g., move the button or change its size).

METHOD

In developing an online help model that works for component-oriented architectures, we started by putting together a interdisciplinary team to cover the following functions: instructional design, human interface design, Apple Guide help system engineering, and Apple Guide scripting.

Based on a set of generated and observed user scenarios, the team extrapolated user needs. We found that our list included needs for both end-users and help developers:

- transfer existing online help skills—Apple Guide users are accustomed to choosing one help menu item and getting context-sensitive help for the frontmost process.

- minimize up-front choices when asking for help—Users shouldn't have to open separate help files for each component in the process.

- access to help for tasks that are provided by the OpenDoc operating environment, or shell, at all times—The shell is running whenever an OpenDoc process is active.

- recognize when several components do the same task

- recognize help for a particular component (i. e, reinforcement that the user is working in the right component)

- part help available wherever the part is used

- basic OpenDoc information that is available whenever the shell is active, is consistently presented, and doesn't add to developers' distribution overhead

- a stable set of tasks for which to author help

The resulting help model is based on the general principle that help works best when designed for both user needs and for the medium that conveys it [1]. The help model is designed to meet OpenDoc user needs within the existing Apple Guide help system. We also wanted to use as much as possible of Apple Guide's current help model and human interface, since they are based on extensive user studies [2].

A Component-Oriented Help Model

When the environment is both dynamic and component-oriented, it is reasonably obvious that a help view should be generated by combining modular help files and should be as dynamic as the process it supports. For OpenDoc help, the key design goal was to structure a model based on content that could be authored in modules, which could then be dynamically assembled and updated to provide context-sensitive access to help.

OpenDoc's help model has several characteristics that extend Apple Guide's original, static help model. We

believe that any component-oriented help model needs to have most of the same characteristics:

- For purposes of providing access, "context-sensitive" is defined as access to the entire process, rather than an individual component.
- Instead of opening a single file for the process, a help view is dynamically assembled. (Actually, it is not possible to write a single, static file for a dynamic entity.)
- The help view changes if components are added to or deleted from the process.
- Content modules correspond to the smallest set of stable tasks (i.e., a component or a set of tasks common to several components).
- Content is integrated in the help view. (The level to which integration is possible is one of the more interesting issues in providing help for component-oriented software).

Figure 4 below illustrates the new help model. In the example, Apple Guide provides help for a process that consists of three components: the OpenDoc shell, a button, and a web browser. When the user invokes help, Apple Guide locates help for all three components, then dynamically constructs a help view for the process.

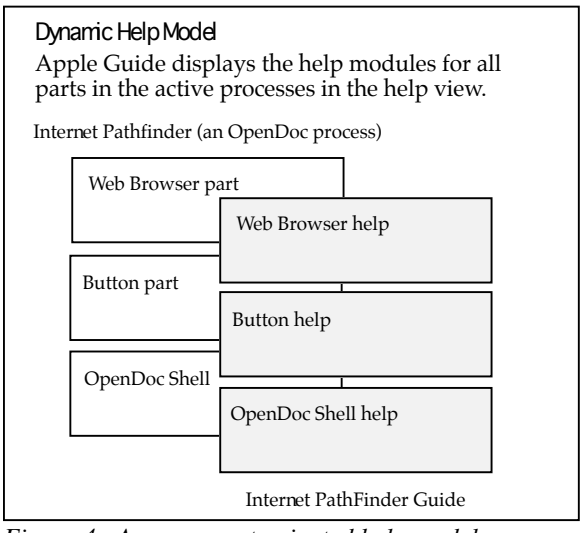

Figure 4. A component-oriented help model

The features described in the rest of this paper include both human interface elements and engineering features that were added to the Apple Guide help system to implement component-oriented help for OpenDoc.

Process-wide access

In order to provide context-sensitive access for component-oriented software, it is necessary to identify a context for which the user needs help.

In an active OpenDoc process, one component at a time has input focus. We discussed limiting access to help for the active component, but based on user testing it is often wrong to assume that users want help for the current tool when they choose help [3]. Besides, OpenDoc components often have narrowly focused functionality, similar to that of a tool on a tools palette. A single component often doesn't provide a large enough context to address the user's goals. Another problem with limiting access to the active component is that the OpenDoc shell and some components don't become "active" in a way that allows the user to invoke help. For example, a button component would execute its programmed action, rather than become active, when the user clicks it. Last, if we provide help at the component level, the user must choose up front where to look for help, as in Figure 5 below.

Figure 5. Too-many up-front choices

Ultimately, processes seem to provide the context most closely linked to user goals. When OpenDoc users invoke help, they will get help for the currently active OpenDoc process, including help for the shell and all components and services in the process. For example, help for the process shown in Figure 6 below would include help for the OpenDoc shell, parts 1 through 7, and any available plug-in services.

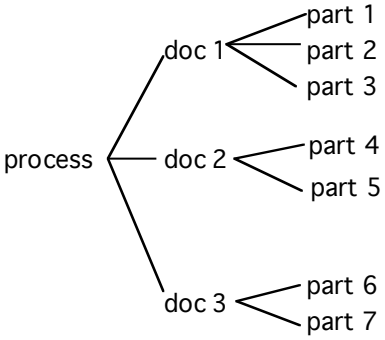

Figure 6. Process-wide help access

With process-wide help, we can minimizes the user's up-front choices. When an OpenDoc process is active, the help menu displays an item that is dynamically named for the current process, as in Figure 7 below.

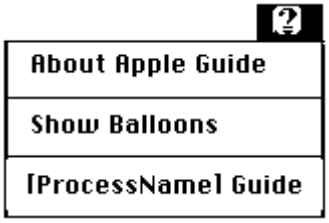

Figure 7. Single menu item for process-wide help

Dynamically assembled help view

In developing help for an open, component-oriented system, we found that we needed the greatest possible flexibility in structuring help content into modules and in getting content into the help view.

To create the help view, Apple Guide uses a qualifying resource attached to each help file. The resource specifies the conditions under which the help's content appears in the access window—based on the component or components with which the help file is identified. Using a qualifying resource gives help developers the ability to identify help content with one or more software components, providing great flexibility in structuring help content. Since the qualifying resource is defined by the help developer, it is not necessary to ask component developers for additional programming.

For example, help content that is common to several software components can be pulled out into its own file, with a qualifying resource that specifies several components. Whenever any of the specified components is in the active process, the help appears in the access window.

The help view is also dynamically named, based on the process name. (See Figure 7 above.) Apple Guide users are accustomed opening the help menu and choosing help with the same name as the active process. Since component-oriented processes are named by the user, help for such a process must be named dynamically. In OpenDoc, the process name appears in several locations in the interface (such as in the process menus). For Apple Guide users, seeing the process name preserves an expected one-to-one correspondence between the process and its guide. If the user changes the process name, the guide name also changes.

Modular structure

In component-oriented environments, help developers work with two separate help contexts—a context for which help can be written and a context for which the user needs help access. For example, users experience OpenDoc's operating environment—called the *shell*—as an integral part of every component because the shell's functionality and menus are available whenever OpenDoc is active. For purposes of access, the shell's help must be integrated into component help. However, the shell and components are independently installed and revised, so their help content must be authored and maintained in separate modules.

For OpenDoc, we found four stable entities for which help modules can be written: the OpenDoc shell (operating environment), components, services, and in some cases, "chunks" of content common to groups of related components.

Integrated content

Once the help context is identified, the help view is constructed by defining how content will be integrated. Integration of content serves two purposes:

- It allows users to search for help without having to know what help file the information belongs to—minimizing up-front choices.
- It shortens the search list because terms that occur in multiple components are displayed only once.

The second reason above provides a powerful incentive for finding ways to integrate content. Lists are easier to scan when terms are not repeated. We found at least one case in which a process could contain more than five components, each of which would repeat the same topic and its associated help sequences. Faced with a long list in which some topics are duplicated, users would have had great difficulty finding cases where several parts would do the same task.

In Figure 8 below, topics (on the left) are not integrated but appear under their component names. This view forces the user attention to components and away from content. Also, the topics in the example are difficult to scan. It isn't clear how items on the left relate to items on the right because both are two-level lists.

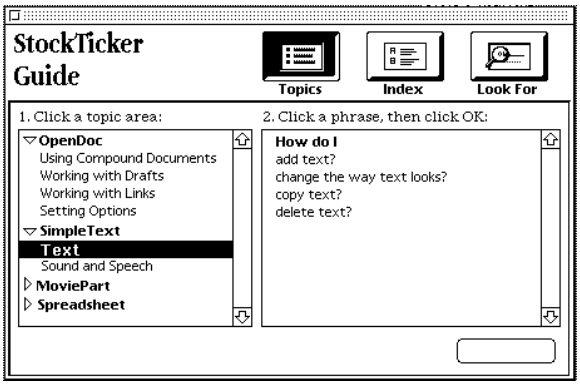

Figure 8. No content integration

For OpenDoc, we considered several levels of integration, based on the goal of allowing users to focus on tasks, rather than components. Essentially, OpenDoc's help provides integrated search capabilities so that any search based on a topic, an index term, or a "look for" keyword is process-wide.

When the user invokes help for an OpenDoc process, Apple Guide extracts the index terms and topics from all help files that qualify for inclusion in the help view. It resolves exact matches, then merges the items into an alphabetical list, as shown in Figure 9 below.

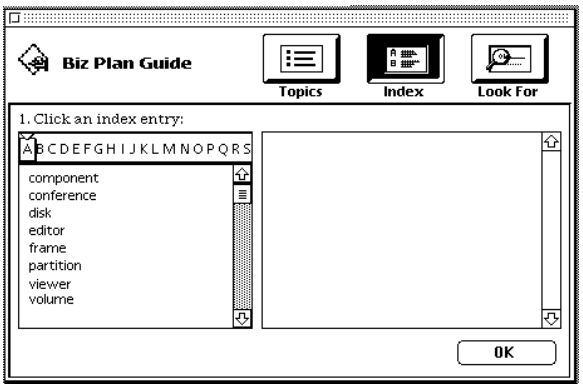

Figure 9. Integrated index terms

Once past the up-front choices, users need to be able to identify the component to which help information applies. This is necessary when more than one component in a process can do a task. It is also necessary when there is a semantic, but not a logical, match between terms. In Figure 10 below, two help files contain the index term "volume." Apple Guide recognizes a semantic match and displays "volume" only once in the access window. However, when the term is selected, users can distinguish between one component that uses "volume" to mean "loudness" and another that uses it to mean "a disk partition."

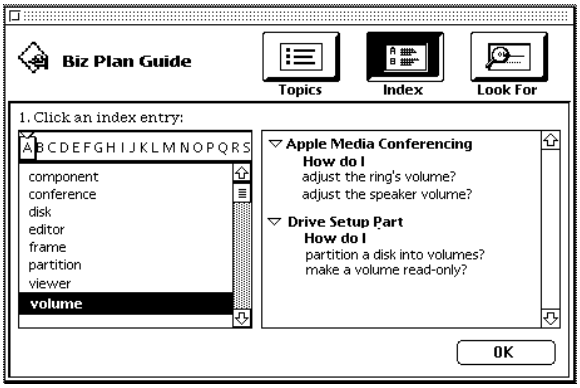

Figure 10. Recognizable component help

By associating help content with a component in the access window, the help system provides access to help for an individual part within a larger context.

In keeping with the goal of minimizing up-front choices, users should be able to search for keywords without being aware of which component the help is associated with. Apple Guide's Look For access window allows the user to type in a search term. The "hits" on the right group information with software components, as in Figure 11 below.

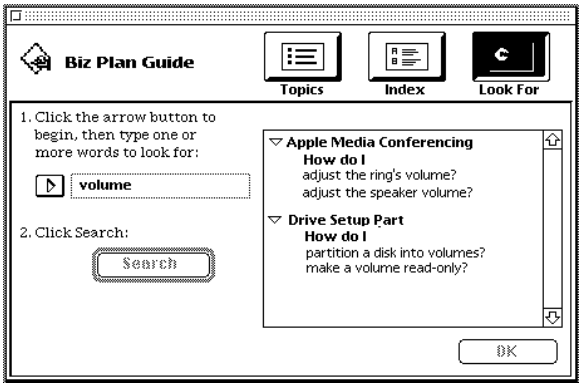

Figure 11. Integrated search across the process

IMPLICATIONS

While developing the OpenDoc help, we identified help issues that have implications for other component-oriented architectures.

Instructional Issues

As a result of integrating help content in the access window, we discovered several instructional issues. First, in order for alphabetized topics to work well in an open system, we found that verb phrases don't work well in an open system that aims at integrating content. For OpenDoc, topics are expressed as nouns, rather than verb phrases (i.e., "Movies" rather than "Playing Movies" or "Working with Movies"). The reason is that, when the user searches an alphabetized list, the verb can be misleading.

In Figure 12 below, the topic is “Text” rather than “Working with Text” so that users scan quickly find the activity they want to do. The help sequences (on the right) are still expressed as tasks.

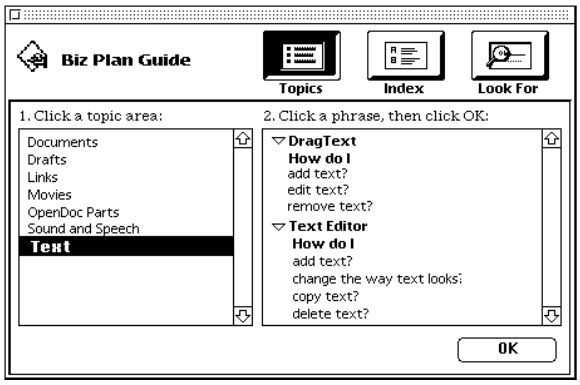

Figure 12. Standardized topics in the help view

Because exactly matching terms are resolved, we also looked for ways to standardize topics that deal with the same tasks. The OpenDoc component category names worked well for this purpose. (See the appendix for a list of category names.) Examples include “Text,” “Spreadsheet,” and “Chart.” We also found that topics and index terms can serve to draw together similar tasks done with different components. Figure 13 below illustrates an example where two components in a process will let the user connect to the Internet.

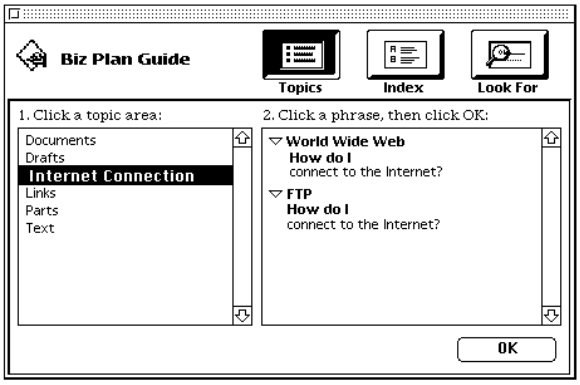

Figure 13. Topic used as branching point

Better content integration

Component-oriented help systems push us toward more highly integrated help content that lets users focus on tasks and treats components as incidental tools. The need to integrate help content makes more urgent the need for advances in language parsing and semantic matching.

For example, content could be more tightly integrated if we could resolve matches at the sequence level (e.g., match “How do I edit text?” with “How do I change text?”) Figure 14 below illustrates some of the difficulties of resolving overlapping content. When dealing with stated tasks, rather than simple terms, current help systems are unable to deal with differences in syntax and levels of abstraction.

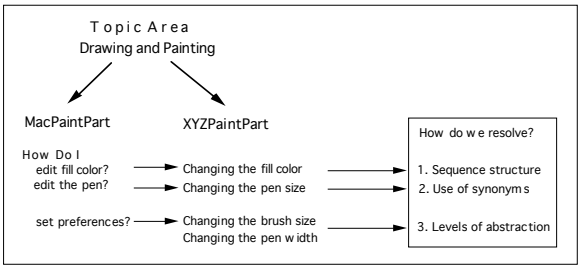

Figure 14. Limitations of semantic matching

With advances in semantic matching, we might be able to provide a help system in which users state a task for which they need help and the help system could constructs a path of steps from a knowledge base encompassing everything that can be done on the user’s system.

In the meantime, help developers for component-oriented environments need to develop ways to integrate content more fully. Extracting and alphabetizing index terms across the components in a process is a good start, but we need to do better. For example, basic OpenDoc terms are defined in help for the OpenDoc shell. It would be useful to directly access and display these definitions (i.e., using “hot text”) from component help. To do that, we need to be able to extract and merge at the content level.

Unexpected benefits of modular structure

Modularity can provide unexpected benefits. The help module for the OpenDoc shell provided a unique opportunity because it is included in the help view whenever OpenDoc is active (i.e., no matter what component is active). The shell’s help covers generic OpenDoc terminology and tasks, making the information consistent (because it exists only once) and essentially providing it free to component developers (because they don’t have to write or distribute the information).

We also found that the component-oriented help model can be extended to help for non-component environments. The benefits include greater flexibility in access to help content, smaller modules to revise and test, and the ability for third-party add-ons to bring in their own help.

CONCLUSION

Building help for component-oriented software forces help developers to re-think online help right down to the design model. The revision of a static help model to support OpenDoc for the Mac OS exemplifies the

issues facing help developers, provides a set of features that can be used as a starting point for implementation, and provides implications for the future of online help.

With the advent of component-oriented technology, help systems that are tightly integrated into the software product must become as modular as the technology they support. If the general trend of software is toward modularity, then perhaps the need to restructure help is a confirmation that help is moving in the right direction.

ACKNOWLEDGMENTS

The author would like to acknowledge the people who contributed to design and engineering for Apple Guide help for OpenDoc: Jose Arcellana, Dave Curbow, Elizabeth Dykstra-Erickson, Shemin Gau, Michael Gough, Devon Hubbard, Kevin Knabe, Brian McGhie, James Miyake, and James Palmer. Thanks also to Rebecca Reese for her editing support.

REFERENCES

1. Sellen, Abigail and Nicole, Anne. Building User-Centered Online Help, in *The Art of Human Computer Interface Design,* Brenda Laural, ed.. Addison-Wesley, Menlo Park, CA, 1990, pp. 143-153.

2. Knabe, Kevin. Apple Guide: A Case Study in User-Aided Design of Online Help, in *Proc. CHI '95 Human Factors in Computing Systems* (Denver, CO USA, May 7 - 11, 1995), ACM Press, pp. 286-287.

3. Duffy, Thomas M., Mehlenbacher, Brad, and Palmer, James E. . *Online Help: Design and Evaluation.* Ablex Publishing Corporation: Norwood, NJ, 1992.

APPENDIX — OPENDOC PART CATEGORY NAMES

Part category is a name that describes the kind of information a part handles. The names below represent the latest list of category names for OpenDoc parts. Where the names are simple, specific, and activity-based, they may be useful as standardized topics for open, component-based help.

Plain text (plain ASCII text)
Styled text
Drawing (object-based graphics)
3D graphic (3D object-based graphics)
Painting (pixel-based graphics)
Movie (movies or animation)
Sampled sound (simple sampled sounds)
Structured Sound (sampled sounds with additional information)
Chart (chart data)
Formula (formula or equation data)
Spreadsheet
Table (tabular data)
Database
Query (storied database queries)
Connection (network-connection information)
Script (user scripts)
Outline (outlines created by an outliner program)
Page Layout (page layouts)
Presentation (presentations or slide shows)
Calendar (calendar data)
Form (forms created by a forms generator)
Executable (stored executable code)
Compressed (compressed data)
Control Panel (data stored by a control panel)
Control (data stored by a control, such as a button)
Space (stored server, disk, or subdirectory data)
Project (project-management data)
Signature (digital signatures)
Key (passwords or keys)
Utility (data stored by a utility function)
Mailing Label (mailing labels)
Locator (locators or addresses, such as URLs)
Printer (stored printer data)
Time (stored clock data)