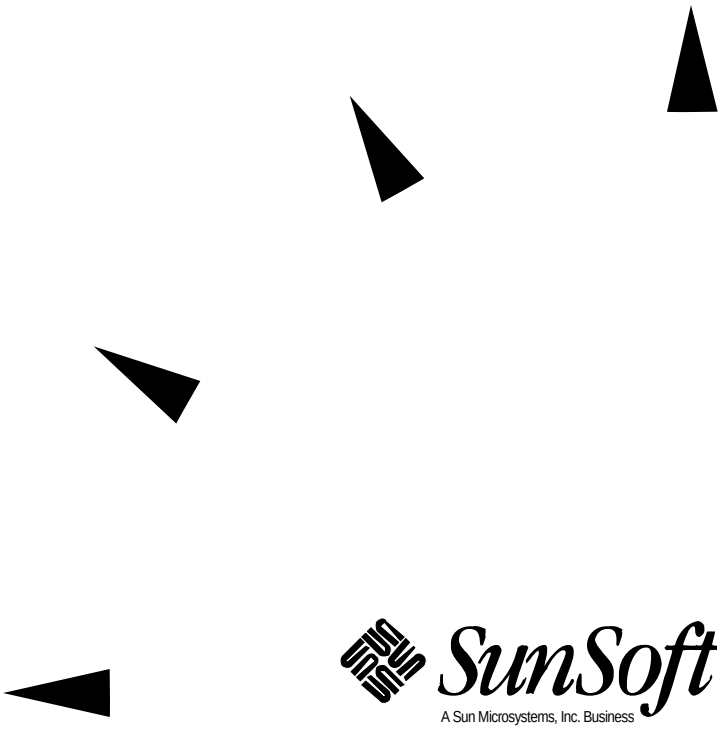
OpenStep User Interface Guidelines



2550 Garcia Avenue
Mountain View, CA 94043
U.S.A.

Part No: 802-2109-10
Revision A, September 1996

SunSoft
A Sun Microsystems, Inc. Business

Table of Contents

Tables

Preface

This manual contains guidelines that tell you what an OpenStep GUI should look like and how it should behave. It does not tell you how to use Interface Builder, Icon Builder, and the other tools to build an interface; the guidelines it contains are simply standards for the results you should achieve with the development tools.

Communicating with Your Users

Your OpenStep GUI should communicate with your users. There are different ways to measure this communication, but two points are particularly important.

- Users should be able to start up an application, look at its interface, and know what they can do (for example, “I can enter statistical data, chart it, and print the charts”). The guidelines explain how to achieve this kind of communication through the effective use of menus, windows, panels, and controls.
- The objects that make up your interface should behave in predictable ways. Users should be able to look at the menus, windows, panels, and controls on their screens and know how to operate them. The guidelines describe the characteristics your interface objects should exhibit to achieve this kind of communication.

Using the Application Kit Effectively

The OpenStep Application Kit (App Kit) supplies standard interface objects, such as panels, buttons, and menus, with their basic appearance and behaviors defined for you. You can include these objects in your interface and then complete them by adding some application-specific behavior. For example, a button's appearance and its basic reaction (it highlights when pushed, and so forth) are supplied, but you need to define what *your application* will do when the button is pushed.

Most features and options of an application can be represented in its interface by one of the standard objects supplied in the App Kit. Many of the objects in your own interfaces, then, will be App Kit objects that you complete with application-specific behavior. You can also develop custom objects of your own design that supplement those supplied in the App Kit.

The Types of Guidelines

To use the development tools effectively, you need to understand the relationship between behaviors supplied in the App Kit and the application-specific results you define—you need to know what has been supplied and what you will be adding. To help with this, *User Interface Guidelines* includes several types of standards or guidelines:

- The design principles that were applied to the overall OpenStep desktop, which define the appearance and behavior of the workspace, including application icons and the dock. Your application needs to conform to these principles and be consistent with the workspace behavior.
- Guidelines for the overall appearance and interaction of your application. These guidelines help you decide when to use standard windows, when to use panels, what kinds of commands should appear in menus, how many levels of submenus you can use, and so on.
- Descriptions of the objects supplied in the App Kit, which emphasize the characteristics built into them (how they respond to mouse and keyboard operations). These descriptions help you decide which objects will most effectively represent the features of your application.

- Specific conventions that should be followed when designing an OpenStep interface, such as the guidelines for keyboard shortcuts. These include "reserved" and "required" shortcuts, suggestions for when to use shortcuts, and guidelines for choosing the shortcut characters.
- Guidelines for using the standard objects supplied in the App Kit. These include objects such as the Open and Close panels and the File and Edit menus.

The point of view emphasized is how to choose the objects that will most effectively present your application's features to your users.

Who Should Use This Book

If you are developing an OpenStep application with a graphical interface you need to be familiar with the guidelines covered in this manual. The material is appropriate for those who have never designed or built a GUI and also for those who have experience with other GUI development environments.

Before You Read This Book

Before attempting to develop an OpenStep interface application, you should be familiar with the workspace, the dock, the File Viewer, the applications supplied with OpenStep software, and the general look and feel of the OpenStep interface. If you haven't used the OpenStep interface, the following end-user manuals will help you learn about it:

- *QuickStart to the OpenStep Desktop*
- *Using the OpenStep Desktop*

How This Book Is Organized

This manual presents interface guidelines in the following chapters:

Chapter 1, "A Visual Guide to the User Interface," is an illustrated introduction to the interface objects supplied in the App Kit.

Chapter 2, "Design Philosophy," explains the basic principles that apply to all OpenStep GUI's, including the OpenStep Workspace. Pay particular attention to the three paradigms for user actions.

Chapter 3, "User Actions: The Mouse and Keyboard," extends the concept of action paradigms, which was introduced in Chapter 2. It looks at the specific mouse and keyboard actions that operate interface objects. It covers both supplied and developer-defined characteristics.

Chapter 4, "Windows in the OpenStepInterface," covers the different types of windows supplied with the App Kit and the appropriate uses of each type. It also includes a detailed discussion of standard window interaction.

Chapter 5, "Panels," is a detailed discussion of panel characteristics. It also lists and describes the standard panels (such as the Save panel and the Open panel) supplied with the App Kit. These descriptions indicate what behavior has been implemented for the standard panels, and what remains for you to implement.

Chapter 6, "Menus," is a detailed discussion of menu behavior. It also lists and describes the standard menus supplied in the App Kit. Some of the necessary behavior has been implemented for the standard menus, and some, which is application specific, has been left for you to design and implement.

Chapter 7, "Controls," provides detailed discussions of the different control types and when they are most effectively used. It explains what behaviors have been implemented for each, and what remains for you to do. It also lists guidelines for any custom controls that you develop.

Chapter 8, "The Interface to the File System," describes the OpenStep conventions for using the Solaris file system.

Related Books

The following manuals cover the tools you use to actually build OpenStep GUI's:

- *OpenStep Development Tools Guide*, especially Chapter 3, which covers the Interface Builder and Chapters 15 through 18, which are tutorials that make extensive use of the Interface Builder.
- *OpenStep Programming Reference*, which covers the Application Kit and the GUI class libraries.

What Typographic Conventions Mean

The following table describes the typographic conventions used in this book.

Table P-1 Typographic Conventions

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% You have mail.`
`AaBbCc123`	What you type, contrasted with on-screen computer output	`machine_name% su` `Password:`
AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type `rm` *filename*.
AaBbCc123	Book titles, new words or terms, or words to be emphasized	Read Chapter 6 in *User's Guide*. These are called *class* options. You *must* be root to do this.

A Visual Guide to the User Interface 1

This chapter illustrates the different types of windows and controls supplied in the App Kit. Besides showing you what these interface objects look like, it includes some general remarks about the way each object is typically used. Extended discussions of object behavior and hints for creating custom objects are available in Chapters 3 through 7.

OpenStep Workspace

The following design principles were established before the OpenStep interface was designed:

- The “look and feel” should be consistent across applications, so that interface objects can provide cues that guide users in their interactions with OpenStep. Objects should have consistent and reasonable responses to user actions.
- The mouse should be the user’s primary input device.
- Colors should be used effectively—the OpenStep color scheme makes extensive use of black, white, and grays to outline the interface objects, but uses full color to accent and enhance them.
- The overall appearance should be simple and easy to understand, with shading that gives a three-dimensional effect.

The basic interface objects, such as windows and menus, were designed to conform to these principles. The appearance and behavior of each object was carefully planned, so that they would work together as a whole.

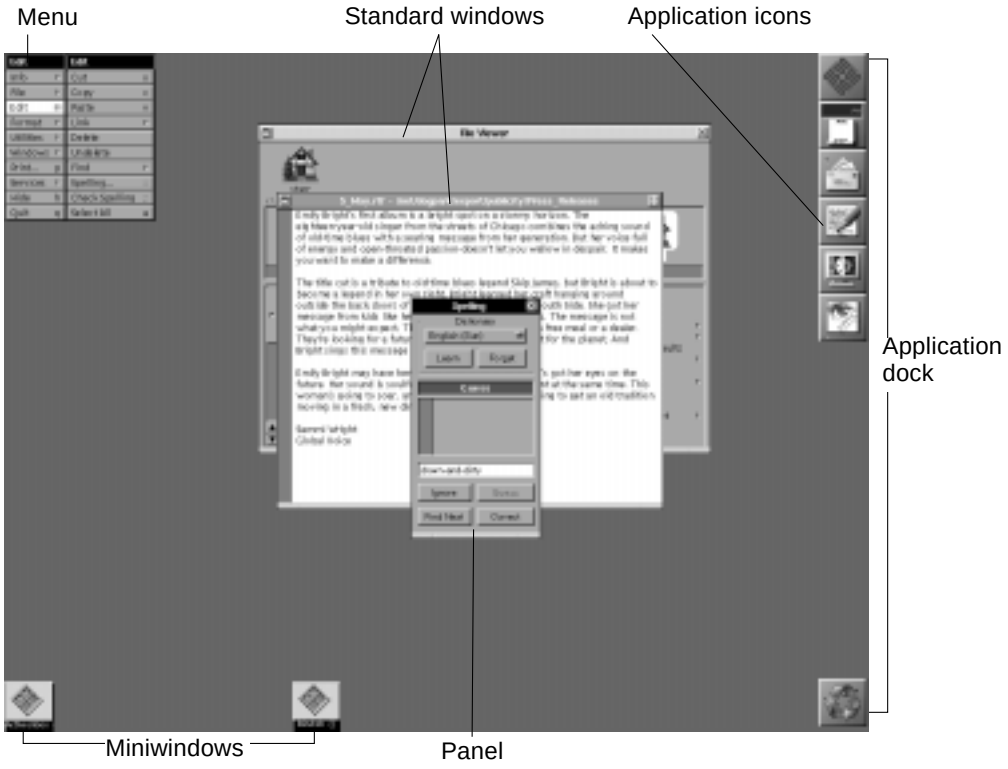


Figure 1-1 The OpenStep Workspace

Figure 1-1 is a typical view of the OpenStep interface in use, with a number of interface objects working together in the user's *workspace.* Two applications (File Viewer and Edit) are represented by *standard windows.* Edit also has a *menu* and a *panel* open. Other applications are running, but they are only represented by their *application icons* and, in some cases, miniaturized windows known as *miniwindows.*

Although many applications can run at once, only one can be *active.* The active application is the one that accepts user input, such as typing and mouse clicks. Users can tell which application is active by looking at the menu displayed in the upper left of the screen; it is always the menu for the active application. In Figure 1-1, the active application is Edit.

Types of Windows

As shown in Figure 1-1, three type of windows are used to make application features available to your users: standard windows, panels, and menus. Two other kinds of windows, application icons and miniwindows, appear in the workspace, but these do not make any application features available.

The active application always has at least one standard window or panel open. When it has more than one open standard window or a combination of standard windows and panels, one window will be the *key window,* which accepts user typing. The key window is identified by its black title bar. In Figure 1-1, the key window is the Find panel. (The key window is covered in greater detail in “Application and Window Status” in Chapter 4.)

Standard Windows

The parts of a standard window are shown in Figure 1-2. Not every standard window needs every part. The File Viewer window, for example, has no close button, which prevents beginning users from accidentally closing a window they need to use their computers. You decide which parts to include in the standard windows you create.

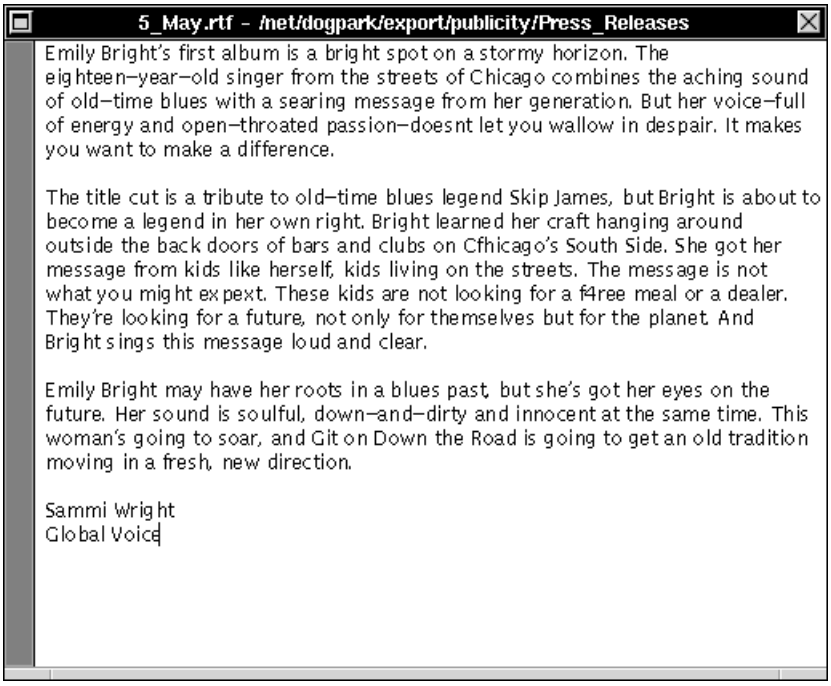


Figure 1-2 A Standard Window

More information about the appearance of windows in general, and standard windows in particular, can be found in Chapter 4, "Windows in the OpenStep Interface."

Panels

Panels can look just like standard windows, but they have different roles in the interface. They let users perform secondary functions that support the work they are doing in standard windows. In text editors, for example, standard windows display the documents users are editing, and panels are opened when the application needs to obtain information for print, save, search, and other commands.

Figure 1-3 An Ordinary Panel

The Font Panel in Figure 1-3 is an *ordinary panel*. When an ordinary panel is open, users can move from the panel to any other window (including panels) of the active application.

In some situations the application must get the user's undivided attention—when, for example, the user must confirm a possibly destructive command. *Attention panels* were designed for these situations. When an attention panel is open, the user cannot move to any other window in the application or issue any application commands. (It is possible to move to another application.) Because attention panels are different than ordinary panels, they look different, as shown in Figure 1-4.

Figure 1-4 An Attention Panel

The App Kit supplies a number of commonly used panels (including the two illustrated in this section) complete with text and controls. You can use them without building them yourself. Using standard panels also helps your users, because they have less to learn in a new application. Chapter 5, “Panels,” contains information about using standard panels and creating application-specific panels.

Menus

Menus give users access to all of an application’s features. Users should be able to look at the menu commands in an OpenStep application and get a good idea of what it does. As shown in Figure 1-5, menu commands are grouped into a *main menu* and its *submenus.*

Figure 1-5 Main Menu and Submenus

Users can always choose menu commands with the mouse. In some cases a *keyboard alternative* —combination of keys that can be used instead of a mouse click—may benefit them. To choose a command from the keyboard, users hold down the Command key and type the character shown in the menu command. For example, to quit an application, the user can either click on the Quit menu command or hold down the Command key and press “q.” “Implementing Keyboard Alternatives” in Chapter 3, covers keyboard alternatives in more detail.

The appearance of the main menu and a number of frequently used submenus (File, Edit, Info, for example) have been standardized. These menus should have generally the same commands, in the same order, in any application Chapter 6, “Menus,” describes the standard menus and commands, and gives guidelines on creating application-specific menus and commands.

Miniwindows

When a user clicks on a window's miniaturize button (the left button in its title bar), the window shrinks down to become a miniwindow, as shown in Figure 1-6. To get the full-size window again, the user double-clicks on the miniwindow.

Figure 1-6 Miniwindows

Application Icons

Running applications are represented on the screen by icons. Users can start applications by simply double-clicking on the right icon.

Application icons can be *freestanding* or *docked.* Docked icons line up along the right edge of the screen. They stay in the dock even when the applications they represent are not running, making it easy for users to locate the icons, double-click them, and start up the applications. Freestanding icons appear in the workspace after users start applications from the File Viewer, and they disappear when users quit those applications. Users can customize their environment by dragging application icons into and out of the application dock.

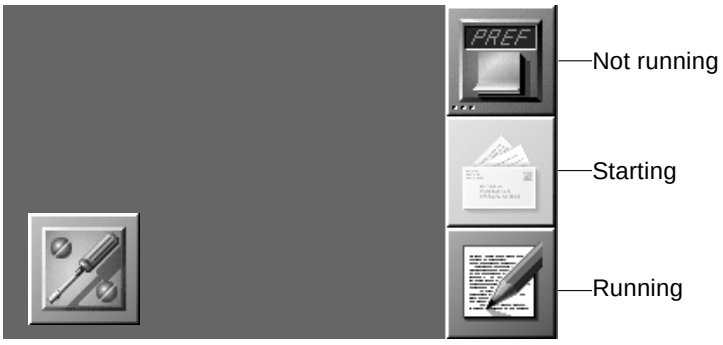


Figure 1-7 Docked and Freestanding Icons

The dock varies the appearance of docked icons to indicate application status. Three small dots (similar to an ellipsis) in the lower left corner of a docked icon indicate the application is not running. The dots disappear when the application is started up. During start-up, the icon is highlighted. These changes are illustrated in Figure 1-7.

Types of Controls

Controls are interface objects that appear on windows and let users give information to an application or start its next action. Controls usually appear on panels or menus, although they can appear on standard windows. Figure 1-8 shows some of the controls supplied in the App Kit. (Menu commands are actually controls, but they are covered separately, in “Menus” earlier in this chapter.)

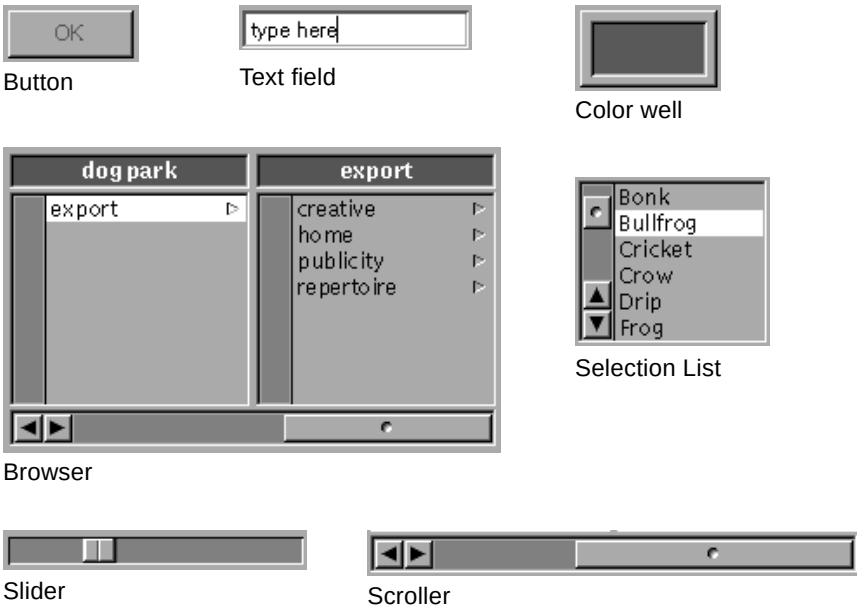


Figure 1-8 Controls

The following sections contain some general observations about using each type of control supplied in the App Kit and show some of the variations in appearance that are possible. Chapter 7, “Controls,” covers the behavior of App Kit controls in detail and discusses creating your own control types.

Buttons

Buttons are most often used to start an action or set a state. Buttons that initiate actions tend to be simple in appearance, such as those in Figure 1-9.

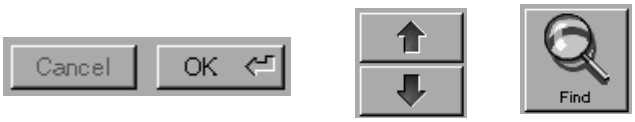


Figure 1-9 Simple Buttons

Among the buttons that initiate actions are those that open *pull-down lists*. See Figure 1-10. A pull-down list combines a button and a menu-like list. The button opens the list, and the user chooses an item to initiate an action. For example:

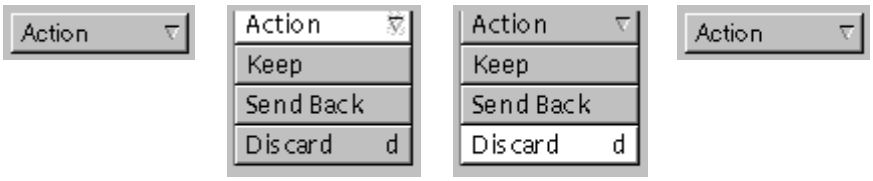


Figure 1-10 Pull-Down List

Buttons that set a state tend to be more complex in appearance. Some typical state buttons are shown in Figure 1-11.

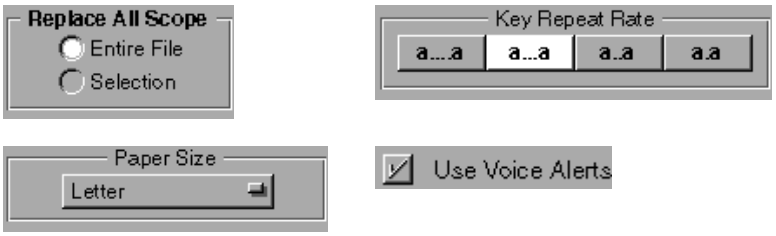


Figure 1-11 Typical State Buttons

Pop-up lists, like pull-down lists, combine a button and a list. The button opens the list and users can choose an item to set its state. Unlike a pull-down list button, the title on a pop-up list button will change to display the current state (such as “Centimeters” in Figure 1-12).

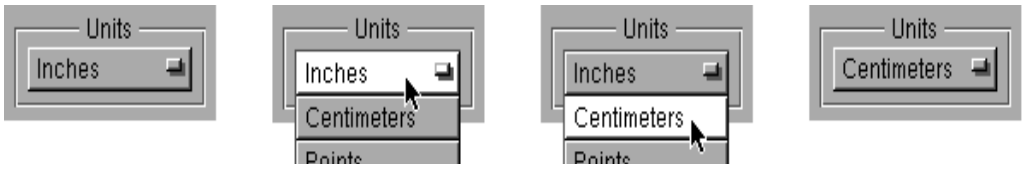


Figure 1-12 A Pop-Up List

Text Fields

Text fields let users enter textual data by typing on the keyboard. The application can work with the data as soon as the user presses Return or clicks on an action button associated with the text field. Whenever possible, the tab key should let users move to the next text field. Figure 1-13 shows three text fields.

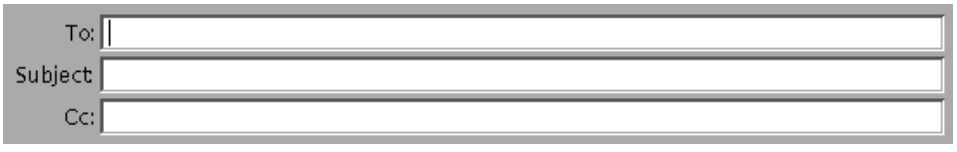


Figure 1-13 Text Fields

Sliders

Sliders set values somewhere between minimum and maximum values you determine when you create the slider. Users can change the value by dragging the slider's knob. Refer to Figure 1-14.

Figure 1-14 Slider

Color Wells

Color wells, shown in Figure 1-15, let users set the colors of interface objects or lines they are using in drawings. Applications often display a set of color wells on a dialog, with predefined colors, from which the user can choose. Frequently one color is chosen for the outline of a graphical object and another for its fill. Users can change the colors available from color wells by dragging in a color from another well, or by clicking on the well's border, which opens the color selection panel.

Figure 1-15 Color Wells

Scrollers

Scrollers let users move documents in the display area by dragging the knob or pressing the arrow buttons. (Alt-clicking on the arrow buttons scrolls by a screenfull.) Figure 1-16 shows a typical scrolling display area, with vertical and horizontal scrollers.

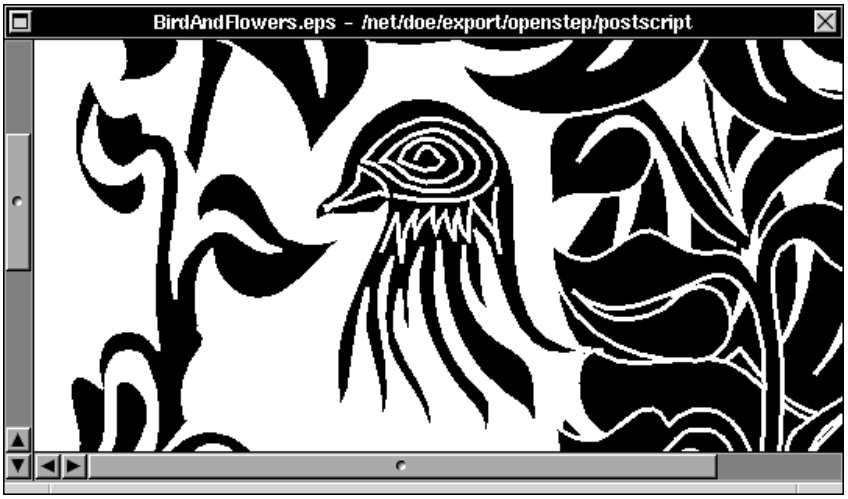


Figure 1-16 Scrollers

Browsers and Selection Lists

Browsers are used to show lists of items when there are hierarchical relationships between the items. The names of folders and files are typical of the information displayed in browsers. Users can move up and down a hierarchy represented in a browser by clicking on items that include the arrow symbol . *Selection lists* resemble browsers, but show only one level of items. In both browsers and selection lists, users can select items by clicking on them. Figure 1-17 shows a typical browser and a selection list.

Figure 1-17 Browser and Selection List

1

Design Philosophy 2

Any user interface you build for an OpenStep application should meet the needs of both novice and experienced users:

- For novice or infrequent users, it must be simple and easy both to learn and remember. It should not require any relearning after an absence from the computer.
- For experienced users, it must be fast and efficient. Nothing in the user interface should get in the way or divert the user's attention from the task at hand.

The challenge is to accommodate both these goals —to combine simplicity with efficiency.

The interface objects introduced in Chapter 1 help you reach both goals. They have recognizable properties of real objects, making it possible for your users to rely on analogies to everyday experience when they approach the computer. They will understand that windows in the interface behave much like separate sheets of paper, and buttons work like the real buttons they are familiar with. The same qualities of the user interface that provide simplicity for novice users can also result in efficiency for more expert users.

2

Basic Principles

Four basic principles were considered when the overall look and feel of the OpenStep user interface was established. You should consider the same principles when creating interfaces for your own applications:

- The interface should feel natural to the user.
- The user is in control of the workspace and its windows.
- The mouse (not the keyboard) is the primary instrument for user input.
- The interface is consistent across all applications.

Each of these principles is discussed in more detail below.

The Interface Is Natural

The great advantage of a graphical user interface is that it can feel natural to your users. In a carefully designed interface, the workspace becomes a visual metaphor for some aspect of the real world, and the objects displayed on the screen can be manipulated in ways that are analogous to the ways physical objects are manipulated. This is what makes a user interface intuitive—it behaves as your users, with experience of real objects in the real world, expect it to.

The similarity of graphical objects to real objects should be at a fundamental, rather than superficial level. Interface objects need not resemble physical objects in every detail, but they do need to behave in ways that your users' experience with real objects would lead them to expect.

For example, objects in the real world stay where your users put them; they do not disappear and reappear. Your users should find the same qualities in graphical objects. Similarly, although graphical dials or switches do not duplicate all the features of physical dials and switches, they should be recognized immediately by your users, and they should suggest the same actions that real dials and switches suggest.

The User Is in Control

The workspace and the tools for working in it (the keyboard and mouse) belong to the user, not to any one application. Users should be free to choose which application and window they will work in and to rearrange windows in the workspace to suit their own tastes and needs.

Users working with an application should be given the widest possible freedom of action. It is inappropriate for an application to arbitrarily restrict what its users can do. If an action makes sense, it should be allowed.

Avoiding Modes

Applications should avoid setting up arbitrary modes that restrict their users. Modes often make programming easier, but they also prevent your users from determining what happens next.

In OpenStep interfaces, modes are used in only three situations:

- In attention panels, which are covered in Chapter 5, “Panels.”
- In the modal tool paradigm, which is covered in “Three Action Paradigms” in Chapter 2.
- In *spring-loaded modes*, which last only while the user holds a key or mouse button down. See “Window Order” in Chapter 4 for more information.

When you use a mode in one of these situations, the mode should be freely chosen by the user, it should be visually apparent that the situation is modal, and you should provide an easy way out of the mode.

When You Should Act for the User

Even though the user is generally in control, it is sometimes appropriate for the application to act on the user’s behalf without waiting for instructions. Consider the panel that appears when users choose the Print command. For some items on this panel, like “page size,” a state must always be selected. Selecting a default page size of “letter” is an action that your application can reasonably take for its users.

The purpose of acting for the user is to simplify a task and possibly make a user action unnecessary. Therefore, when the application acts it must respond as though the user had acted. If the page size control, for example, changes in appearance when a user changes its state, it must change in the same way when the application sets its state.

Actions made on the user’s behalf should be simple and convenient. Otherwise, they can be annoying or confusing, and reduce the user’s sense of control over the workspace. If there is any doubt whether an application should act on the user’s behalf, then it probably should not. It is better for the application to do too little than too much.

The Mouse Is the Primary Input Device

Every aspect of an application, every possible operation by a user, is represented by a graphical object displayed on the screen. Graphical objects are operated mainly by the mouse, and the keyboard is principally used for entering text. The mouse is the more appropriate instrument for acting on graphical objects.

One of the goals of user interface design is to make mouse actions as natural for users as touch typing is for experienced typists. This is only possible if mouse actions follow established paradigms that users can rely on. To help achieve this, three paradigms have been developed for user actions on OpenStep interface objects. They are covered in “Three Action Paradigms” on the next page.

Although the mouse is the primary input device, the OpenStep interface also allows *keyboard alternatives* to mouse actions. They can be efficient shortcuts for experienced users. Keyboard alternatives are always optional; however, a keyboard operation without a corresponding mouse-oriented action on screen is not acceptable. (“Implementing Keyboard Alternatives” in Chapter 3 covers the details of implementing keyboard alternatives.)

The Interface Is Consistent

When every application has the same basic user interface, all applications benefit. Consistency makes each application easier to learn and increases the likelihood it will be used and accepted.

Just as drivers become accustomed to a set of conventions on public highways, users tend to learn and rely on a set of conventions for their interaction with a computer. Although different applications are designed to accomplish different tasks, they all require some common actions—selecting, editing, scrolling, setting options, making choices from a menu, managing windows, and so on. You application’s behavior is predictable only when you follow the conventions for these actions.

Interface conventions permit computer users, like drivers, to develop a set of habits and to act instinctively in familiar situations. Instead of being faced with special rules for each application, which would be like each town defining its own rules of the road, users can carry their knowledge about interface objects from one application to another.

Three Action Paradigms

Graphical user interfaces work best when they have well-defined paradigms for user actions with the mouse. These paradigms must be general enough to support the widest possible variety of applications, and also precise enough that users are always aware of what actions are possible and appropriate.

The OpenStep user interface supports three paradigms of mouse action:

- Direct manipulation
- Targeted action
- Modal tool

Direct Manipulation

Most interface objects respond directly to manipulations with the mouse—a button is highlighted when pressed, a window comes forward when clicked, the knob of a slider moves when dragged. Direct manipulation is the most intuitive of the action paradigms and the one best suited for moving and resizing graphical objects. For this reason, windows can only be reordered, resized, and moved by direct manipulation.

By directly manipulating the icons that represent documents, applications, mail messages, or other objects stored in the computer's memory, users can manipulate the objects the icons represent. For example, dragging an icon to a new location will change the position of a file in the file system's hierarchy.

The other paradigms—targeted action and modal tool—include some direct manipulation. When users select objects as targets for actions, the objects highlight themselves in some way, providing feedback to the users. And when users select modal tools from palettes, the appropriate tool icons are highlighted, reminding your users which tool is in use.

Targeted Action

Your applications provide controls, such as buttons and scrollers, which let users choose what the application will do next. Clicking on a close button, for example, doesn't just highlight the button; it also removes the window from the screen. Controls are devices, like light switches and steering wheels, that let your users carry out actions.

All controls act on targets. For some controls, such as the Quit menu command, the entire application is the target. For others, such as the close button in a window title bar, the target is a window. And for still others, such as the Cut menu command, the target is a subset of one window's contents, such as a range of text the user has selected.

In some of these cases (the Quit command and the close button) the target of the action is implied, but in others (the Cut command) the user must explicitly select the target. User-selected targets are frequently text strings or editable graphics. They can also be other types of objects, such as windows (the target of the Close Window menu command) or file icons (the target of the Workspace Manager Destroy command).

When the target of an action must be selected by the user, the user first selects the target and then chooses the control that initiates the action. With the Cut command, for example, users select a range of text in a document and then choose the command from the Edit menu.

This kind of targeted action, with an explicit selection, is the normal paradigm for controlling or operating on objects. It has the advantage that users can choose a target and then initiate a sequence of different actions. Users who select text, for example, can change the font, the point size, and then copy the selection to the pasteboard, without changing the target. Another advantage is that a single control can act on a series of different user-selected targets, making it extremely efficient and powerful. The Cut command, for example, can delete text, graphics, icons, and other objects.

For those situations in which direct manipulation is the most natural way to perform an operation, it is preferable to targeted action. However, since direct manipulation is not sufficient for many operations, targeted action is actually the most frequently used paradigm. Direct manipulation is an easy, natural way to resize a window by dragging its borders. It is neither easy nor natural to set the size of text by dragging the letters to a new height.

Modal Tool

With the modal tool paradigm, users determine what their mouse actions will do by selecting a tool. Depending on the tool that is chosen, mouse actions (clicking and dragging) produce different visual results. For example, a graphics editor might provide one tool for drawing circles and ovals, another for rectangles, and a third for simple lines.

Each tool sets up a *mode*—a period of time when the user's actions are interpreted in a specific way. The mode limits the user's freedom of action to a subset of all possible actions, which is normally undesirable. But with the modal tool paradigm, the limits imposed by the mode are justified in several ways:

- The mode is not hidden. The altered shape of the pointer and highlighted state of the tool make it apparent which mouse behaviors are in effect.
- The mode is not unexpected. It is the result of a user choice, not the by-product of some other action.
- The way out of the mode (usually clicking on another tool) is apparent and easy. It is also available to the user at any time.
- The mode mimics the way things are done in the real world. Artists and workers choose the appropriate tool (whether it is a brush, hammer, pen, or telephone) for the task at hand, finish the task, and choose a different tool for the next task.

These modal tools are often displayed in palettes with related tools, each tool enabling a specific set of mouse behaviors. The pointer assumes a different shape for each tool, so that it is always apparent which one has been selected, and the tool remains highlighted in the palette.

The modal tool paradigm is appropriate when a particular type of operation is likely to be continued for some length of time (for example, drawing lines). It is not appropriate if the user is put in the position of constantly choosing a new tool before each action.

Figure 2-1 shows a typical palette of modal tools; the freehand line tool is highlighted to indicate that it has been selected, and the pencil-image pointer shows that a mode is in effect.

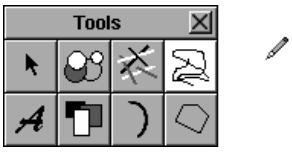


Figure 2-1 Palette of Modal Tools

Paradigm Extensions

Users come to expect familiar operations throughout the user interface. It is your responsibility to make sure the action paradigms your application uses are apparent to its users—controls should look like controls (like objects that fit into the targeted action paradigm), palettes of tools should look like palettes of tools, and so on.

You should also make sure that the paradigms used in your applications fit the actions users will actually be taking. It would not be appropriate if your users were required to select a modal “moving tool” just to move objects. Graphical objects should move, as real objects do, through direct manipulation.

Applied properly, the three paradigms described in this chapter can accommodate a wide variety of applications and user actions. Yet, over time, as programmers develop innovative software, new and unanticipated operations might require extending these paradigms.

A paradigm extension should be your last resort. All possible solutions within the standard user interface design covered in this chapter should be exhausted first. You must weigh the functionality added by extending the paradigms against the ill effects of eroding interapplication consistency for your users. Make sure that any extensions you develop are obviously different to your users from the existing paradigms.

If a paradigm extension is required, it should be designed so that it grows naturally out of the standard user interface, and adheres to the general principles discussed above.

Testing User Interfaces

The success of an application’s interface depends on real users. There is no substitute for having users try out the interface, even before there is any functionality behind it, to see whether it makes sense to them and lets them accomplish what they want.

User Actions: The Mouse and Keyboard

3

Chapter 2 introduced three basic paradigms—direct manipulation, targeted action, and modal tools—that guide OpenStep users when they operate windows, menus, controls and other interface objects. This chapter looks at the specific mouse and keyboard actions—clicking, double-clicking, dragging, and so on—that users perform on these objects.

3

Mouse Actions

The mouse moves the pointer in the workspace. Users typically move the pointer to an object and then use the mouse button to act on the object.There are two basic mouse operations:

- Moving the mouse to position the pointer. When the pointer is focused on an object, it is said to be positioned “on” or “over” the object. (With the standard arrow pointer, the point of the arrow indicates the user’s point of focus; other pointer shapes have other graphic devices to indicate their points of focus.)
- Pressing and releasing the mouse button.

From these basic operations, the standard mouse actions are derived:

- Clicking
- Multiple-clicking
- Dragging
- Pressing

Guided by the action paradigms, users can perform these mouse actions (and combinations of them) to rearrange windows, operate controls, and edit documents.

Note – The result of using App Kit objects includes recognition of appropriate mouse actions and automatic execution of the application-specific behavior you specify in *action methods*. The menu command object, for example, will execute its action method when clicked. You write the action method, which might open a panel or perform some other action. The text object responds to a different set of mouse actions, including clicks, double-clicks, triple-clicks, and dragging. If you create a custom control or a custom content area, you will probably need to write code to handles any mouse actions you want the control or area to respond to. This is covered later in this chapter in the sections “Actions for Custom Objects” and “Other Action Keys.”

Clicking

Users *click on* an object by positioning the pointer over it and operating the mouse button. The mouse button is pressed and released quickly, and the mouse is not usually moved during the click. The timing of the click is generally not significant.

A click indicates that the user's attention is focused at a particular screen location. The interface's response to the click depends on the object at that location. If the object is a window, clicking on it brings it to the front and usually *selects* it to receive characters from the keyboard. If the object is a menu command, button, or other control, clicking on it initiates whatever application-specific action has been defined for it. In a text area, a click selects a location for the *insertion point* (the place where typing will begin). In a graphics editor, it may select the location for a Paste command.

Note – The OpenStep interface provides two mouse functions, Select and Menu. Select is used for clicking, double-clicking, choosing menu commands, and other operations on interface objects that are described in this chapter. Select is normally assigned to the left button (left-handed users can reverse the functions of the buttons). In this manual, which covers operating the interface, "clicking the button" and "clicking on an object" mean clicking the button that is currently performing the Select function. For more information, see "Changing the Function of the Mouse Buttons in this Chapter.

Multiple-Clicking

Users *double-click* an object by positioning the pointer over it and pressing and releasing the mouse button twice. The second click must be within a short interval of the first to be recognized as a double-click. (Users can adjust the time interval within which two clicks are recognized as a double-click, to suit their individual needs, with the Preferences application.) The pointer cannot move significantly during the interval between the two clicks, which ensures that the double-click is focused on a single screen location.

Users *triple-click* an object by rapidly pressing and releasing the mouse button three times in succession. Triple-clicks are subject to the same pointer movement and time interval constraints as double-clicks.

Some objects respond differently to single, double- and triple-clicks.

Pressing

Users *press* an object by positioning the pointer over it, and pressing and holding the mouse button. Some types of object, such as the arrows in a scroller, respond to pressing and begin to act as soon as the mouse button is pressed. They do not wait for the button to come back up and complete a click.

Pressing an object resembles a click because the mouse button will be released eventually. The object is said to be pressed, rather than clicked, if the action begins when the mouse button is pressed and ends when the button is released.

Dragging

Users *drag* by pressing the mouse button, holding it and moving the pointer with the button down. This action is most often used in direct manipulation of objects.

In general, a dragging action includes a click (since the mouse button is pressed before dragging and released after it is complete). Dragging a window, for example, will also bring it to the front, as though the user simply clicked on it. There are, however, a few situations in which objects should respond differently to dragging and clicking. These are covered in "When Dragging Should Not Imply Clicking" later in this chapter.

Mouse Responsiveness

The pointer moves when the user moves the mouse, but the relationship between the two movements is not constant. Rapid mouse movements move the pointer farther than slow ones. This feature is called *mouse speed*, and users can adjust it with the Preferences application.

Changing the Functions of the Mouse Buttons

OpenStep provides two mouse functions, Select and Menu. Select is used to click, double-click, and choose menu commands, and is normally assigned to the left menu button. Menu opens a copy of the active application's main menu at the pointer location, and is normally assigned to the right menu button. On a three button mouse, such as a Sun mouse, the middle button is not active, while a two button mouse uses both of its buttons.

Left-handed users may be more comfortable if they reverse these functions. They can do so with the Mouse Preferences panel in the Preferences application. (See “Bringing the Main Menu to the Pointer” in Chapter 6 for details of the Menu function.)

Selecting Objects with the Mouse

The modal tool and targeted action paradigms require users to *select* objects. They do so by clicking on them or by clicking and dragging. A variety of objects can be selected, including:

- Tools in a tool palette
- Cells in a matrix or fields in a form
- Icons in a file viewer
- Items in a list (of files or mail messages, for example)
- Characters in editable text
- Graphical elements of editable artwork
- Position of the insertion point in text or graphics

Selecting an object distinguishes it from other objects of the same type. It does not change the selected object in any way. (It does not include such operations as clicking on a radio button or choosing a menu command.) In some cases, such as clicking on a window to bring it forward or clicking in a text area to position the insertion point, selecting is a complete operation in itself, followed by another operation such as moving to a control on the window or typing at the insertion point. But in most cases, users select an object to designate it as the target for a subsequent operation (this is the targeted-action paradigm).

In areas where users are allowed to insert new material (text, graphics, and so on), they can select objects already displayed or select a location for inserting. In a text field, for example, it is possible to select characters that have already been typed or an insertion point for more typing.

Note that windows are activated, rather than selected. Users can both activate a window and select a control or text location on it with a single mouse operation.

This rest of this discussion concentrates on selections in editable material, although the rules often apply to other kinds of selection.

Clicking to Select

If the anchor point (the pointer's location when the mouse button was pressed) and the endpoint (the pointer's location when the mouse button is released) are substantially the same, the user's action is a click. A click sometimes selects the item under the pointer and sometimes simply selects the clicked location as the insertion point for text or a modal tool. In a graphics editor, for example, a click can select an existing figure.

In text, a single click always selects a location as the insertion point, where characters can be entered from the keyboard. The insertion point is normally marked by a blinking vertical bar located between characters. If the user clicks on top of a character, the insertion point is positioned at the nearest character boundary. Clicking in a margin, or in an empty area away from any text, puts the insertion point next to the nearest character.

Multiple-Clicking to Select

A single click in a text area selects the location of the insertion point, but multiple clicks select a linguistically meaningful unit of the already existing text. Double-clicking in text normally selects a word, and triple-clicking normally selects a paragraph (defined as the text between two return characters).

If the user selects a character or word with a multiple-click and then drags, this selects additional text in the character blocks selected by the multiple-click. For example, after double-clicking to select a word, dragging after the double-click selects all other words that are even partially within the range defined by the anchor and end points of the drag.

Dragging to Select

Dragging selects everything between the anchor and end points of the drag. Exactly what is included in the selection depends on the type of material through which the pointer is dragged. See “Selection in Text and Selections in Graphics” later in this chapter for details.

Note – The App Kit supplies a text object and a browser with all of the selection mechanisms described in this section except one—the text object does not implement discontinuous selection. If you define your own selectable data, you will have to implement selection yourself, in accordance with these guidelines.

Extending a Selection

Pressing the mouse button to begin a click or drag operation normally cancels the current selection to begin a new one. But holding down either the Alt key or the Shift key modifies this action, allowing users to extend the current selection instead of canceling it.

Continuous Extension

Clicking and dragging when the Alt key is held down results in a new selection, which includes the initial selection and everything that lies between it and the pointer’s final position (where the mouse button is released). The Alt key is thus an alternative to dragging as a way of selecting a range—the user can click on one location to establish an anchor point, hold down the Alt key, and click again to determine the end point.

If the initial selection is itself a range, Alt-clicking and Alt-dragging will extend one edge of the selection (the edge closest to the pointer when the mouse button goes down) to the pointer’s final position (where the mouse button is released). The Alt key thus also provides a way of adjusting the boundaries of the initial selection. Alt-clicking outside a selected range extends the range to the point on which the user Alt-clicks. Alt-clicking inside a selected range repositions the closest edge of the selection to the point on which the user Alt-clicks. Refer to Figure 3-1 to see the effect of Alt-clicking.

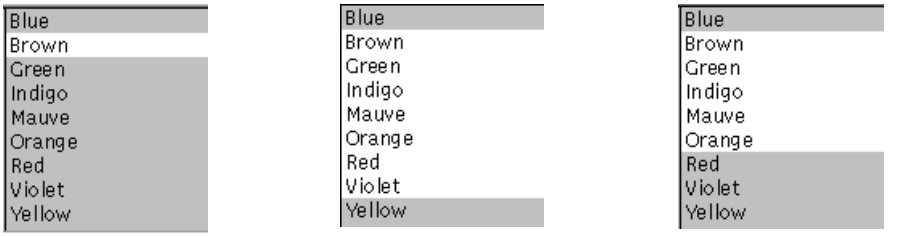


Figure 3-1 Alt-Clicking

If the initial selection is the result of a multiple-click the Alt key extends it, just as dragging would. Double-clicking a word, holding the Alt key down, and clicking on another word elsewhere in the text extends the selection to include both words and all those between the two.

Discontinuous Extension

The Shift key lets users add to, or subtract from, the current selection. Additions do not have to be continuations of the current selection, so discontinuous selections can result.

Discontinuous selection is common for editable graphics, icons, and items arranged in a list. It is not usually implemented for normal text.

To extend a selection, the user clicks and drags as usual while holding the Shift key down. New material is selected, but the previous selection also remains. This is illustrated in the middle column of Figure 3-2.

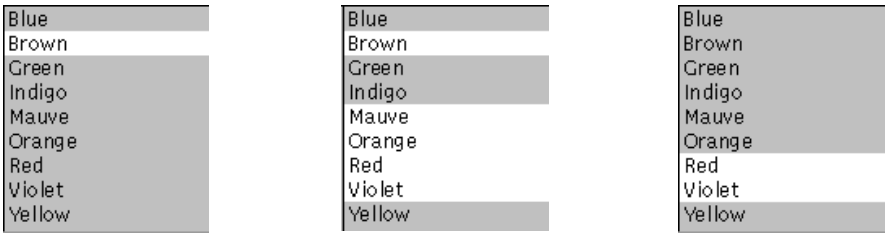


Figure 3-2 Shift-Dragging

To subtract from the selection, the user holds the Shift key down while clicking on or dragging over some part of the current selection. Shift-clicking and Shift-dragging deselect material that has already been selected. While keeping the Shift key down, the user can first select material, then deselect it, then select it again.

Shift-dragging either selects or deselects; it never does both. Which it does depends on the status of the item under the pointer when the mouse button goes down:

- If the item is not currently part of the selection, Shift-dragging will select it and everything the user drags over. It will not deselect material that happens already to be selected.
- If the item is currently selected, Shift-dragging deselects it and any other selected material that is dragged over. It will not add unselected material to the selection.

How the Arrow Keys Affect a Text Selection

In a text area, the keyboard arrow keys are used to position the insertion point or, if modified by the Alt key, alter the current selection. Unlike the mouse, which can select anywhere within a document, movement by the arrow keys is only relative to the current selection. The arrow keys have nothing to do with the pointer, which is controlled only by the user's mouse movements.

The following descriptions assume that the initial selection, before the user touches an arrow key, is a range of text. The simpler case, in which the current selection is not a range but an insertion point, is not directly addressed, but it can easily be derived from the descriptions given.

- When used alone (without a modifier key), the current selection is canceled and the insertion point moves. The left arrow key puts the insertion point one character before the beginning of the current selection. The right arrow key puts the insertion point one character beyond the end of the current selection. These keys will move the insertion point to the previous or next line if necessary.

- The up arrow key puts the insertion point one line above the beginning of the current selection, and the down Arrow key puts it one line below the end of the current selection. As the up and down Arrow keys move it from line to line, the insertion point maintains the same approximate distance from the left margin. It falls at the end of any line that is shorter than that distance, but comes back out to the original distance when a line that is long enough is encountered.

More information on handling the arrow keys is in "Other Action Keys" later in this chapter. Modified arrow keys—for example, Alt-arrow—are discussed in "Implementing Modifier Key—Arrow Key Combinations" later in this chapter."

Implementing Selection

"Selecting Objects With the Mouse" earlier in this chapter described how your users can select the objects supplied with the App Kit. There are two situations in which you need to implement selection yourself: when your application has areas of editable text and when you include custom objects and allow your users to make multiple selections of them. This section describes the proper interface behaviors for these situations.

Note – Even if your application uses the supplied Text object for its editable text areas, you still need to implement the modifier-arrow key combinations described in "Implementing Modifier Key—Arrow Key Combinations" later in this chapter.

When implementing selection, make sure that your application never moves the selection out of the user's view. When necessary, the display must scroll itself to make the new selection visible. Users, of course, can choose to move a selection out of view with a scroller. But as soon as the user makes a new selection or extends the initial selection (by such actions as pressing an Arrow key), the application selection should scroll the selection back into view.

Note – To hide the pointer temporarily, use the single-operator function `PSobscurecursor()`.

When Discontinuous Selection Is Not Implemented

Sometimes an application implements selection in an area without implementing discontinuous selection. In situations like this, the Shift key should act just like the Alt key during a selection, and both keys should result in continuous selections.

For example, discontinuous selection is not implemented in the App Kit text object, so both Shift-clicking and Alt-clicking extend the selection continuously. This saves the user from making errors due to pressing the wrong key when trying to extend the selection.

Selection in Text and Selection in Graphics

Dragging a rectangle to make a selection has slightly different results in text and graphics. In text (and other serially arranged material) the selection includes the entire series of characters between the anchor and end points. In graphics (and other material consisting of objects that can be independently arranged, such as icons or the graphic elements that make up a picture) the selection generally includes everything that is even partially within the rectangle defined by the anchor and endpoints. Figure 3-3 shows the difference between selections in text and graphics.

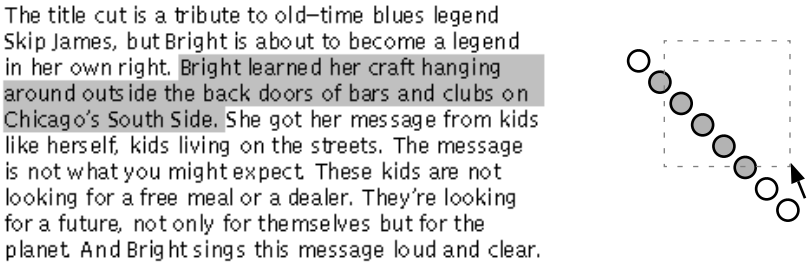


Figure 3-3 Selecting Objects by Dragging the Pointer

Managing the Pointer

The pointer is generally handled by the App Kit. There are, however, situations in which an application needs to explicitly hide the pointer or change its shape.

Changing the Pointer

Your applications can change the pointer from the standard arrow  to any other image of equal size (16 pixels by 16 pixels). When doing so, you must specify what part of the pointer acts like the tip of the arrow. That point, the pointer's *hot spot*, should be apparent to your users from the image itself. For example, if the pointer is an X, the hot spot should be where the two lines cross.

A shape other than an arrow is more useful in many situations. A typical example is the I-beam pointer used to select the insertion point in text areas. This pointer has its hot spot in the center of the beam.

It is often a good idea to change the shape of the pointer to indicate that the user has entered a mode. In applications that use the modal tool paradigm, the pointer should change to indicate which tool has been selected. For example, the pointer might look like a pencil while thin lines are being drawn in a graphics application, or like a wide brush when painting in broad strokes.

If mouse actions are valid only in a certain area, the pointer should revert to its normal shape when it leaves that area. It is best not to change the pointer too often, however. To avoid confusing your users, use the standard arrow wherever possible.

Hiding the Pointer

A visible pointer is essential for mouse actions, but it can get in the way when users are concentrating on using the keyboard. The text object automatically hides the pointer (makes it disappear from the screen) when users begin to type. The pointer returns to the screen as soon as the user moves the mouse, which signals a shift in attention away from the keyboard and back to the mouse.

The pointer should also be hidden whenever the user selects an insertion point or a range of text. The new selection is a good indication that the user is ready to begin typing, and hiding the pointer prevents confusion between the I-beam pointer and the vertical bar representing the insertion point. Unless it is hidden, the I-beam can obscure the vertical bar. In this case too, the pointer must reappear as soon as the user moves the mouse. (This is not currently implemented by the text object, so each application should do this for itself.)

Mouse Actions for Custom Objects

This section describes the behaviors you should implement for custom controls, application-specific document areas, and other custom objects. It describes how these objects should respond to clicking, dragging, and other mouse operations. You do not need the guidelines in this section if you are only using the standard Application Kit objects.

Reacting to Clicks

When one of your users clicks on an object on screen, the object should provide immediate, visible feedback as soon as the mouse button goes down. Depending on the action paradigm in effect, however, the correct behavior for the object may be to wait until the mouse button is released before it does anything more:

- If the purpose of the click is direct manipulation of the object, the object should react when the mouse button goes down. For example, when a window is clicked on, it comes to the front of the screen without waiting for the mouse button to go up. Similarly, when editing text, the user is committed to a new selection as soon as the mouse button is pressed.
- If the click is intended to initiate a targeted action or choose a modal tool, then, in general, the object should act when the mouse button goes up. This gives the user a chance to change his or her mind. If the user moves the pointer off the object before releasing the button, the action is canceled. Suppose, for example, that the user presses the mouse button while the pointer is over the Cut command in the Edit menu. The command is highlighted, but the current selection is not actually cut until the user releases the mouse button. If the user moves the pointer away from the menu before releasing the mouse button, the command will not be carried out.

Note – You can implement multiple-clicks so that they act when the mouse button is pressed the second (or third) time, instead of waiting for the mouse button to go back up (as is usual for a single click). This implementation can help improve the perceived speed of your application.

First Click on a Window

Sometimes the intention of a click is simply to bring a window forward. Your application needs to decide whether the user wanted the click to bring the window forward, or to bring it forward and do some work in the window. Use the following guidelines:

- If the user chooses a particular control—clicking on a button or a scroller, for example—the click should not only bring the window forward, but should also operate the control. Since controls are small, it is reasonable to assume that the user chose to position the pointer over the control and click on it.
- If the click is generally within the content area of the window, but not over a control, the click should just bring the window forward, without otherwise acting on it. In particular, it should not alter the current selection.

If the user chooses to double-click in the content area of the window, the normal double-click action should be performed. Double-clicking on a word should select the word whether the window is in front or not.

When Dragging Should Not Imply Clicking

In general, the main action performed by an object should be initiated by a single click, and dragging should be recognized as a click and initiate the same action. But there are situations in which you don't want dragging an object to initiate its action.

For example, the main action associated with a docked application icon is activating an application, but users can also move icons by dragging them. Users should be able to move icons without activating their applications, so dragging and activating must be separate actions.

There are three ways to handle a situation like this:

- Require users to hold down a modifier key when the object is dragged.
- Implement the object so that dragging does not perform a click.
- Implement the object so a click does nothing, and a double-click is required to perform the object's action.

In the case of the docked icon, the first solution (requiring a modifier key to drag the icon) is not appropriate. Moving an icon is a common operation, and unmodified dragging is the natural way to do it.

The second solution (a single-click activates the application and dragging does not perform a click) is not appropriate either, mainly because users who intended to drag the icon could easily click on it accidentally. An accidental click would begin the icon's main action (activating an application), which takes time, brings up windows, and cannot be reversed until the application is fully started. In addition to this, if a single-click started the application, it would be inconsistent with the way application icons behave in the File Viewer. In the File Viewer, a single-click selects the application's file. It is not started up until the user double-clicks it.

So the only acceptable implementation for docked application icons is the third possibility, in which a single-click does nothing and a double-click starts the application.

Another example, in which the second solution (dragging does not imply a single click) was implemented, is the File Viewer's shelf. When the user clicks on the icon of a folder on the shelf, the File Viewer changes to show the folder's path. However, when the user drags the icon off the shelf, the File Viewer does *not* perform the click's action—it does not change the current path. One of the reasons this solution works is that the shelf differentiates clicking from dragging by having a threshold for dragging. Until the user drags the icon a certain distance, it does not move. Once the user has committed to dragging the icon, it cannot be clicked. The icon also looks very different when it is clicked on (it is highlighted) then when it is dragged (it moves, and a dimmed copy is visible in its old location).

In general, for this kind of implementation, where dragging does not perform a click, the following conditions must be met to be successful:

- The user gets clear visual feedback that indicates whether the object is reacting to a click or to dragging, and it is difficult for the user to do the wrong thing. For example, if the user starts to drag the object but then decides to put it back, the action should not be treated as a click.
- The single-click's action matches similar uses of single-clicks elsewhere in the interface.
- The action initiated by the click does not have consequences that the user might want to avoid and that cannot easily be stopped or reversed. This condition is important because the user might accidentally click on an object when intending to drag it.

When to Use Multiple-Clicking

You should only implement double-clicking for a custom object when it logically extends the action of a single-click. Triple-clicking should only be implemented when there is an action that logically extends a double-click. There are two reasons for this rule, one philosophical and the other programmatic:

- Complex mouse actions are best remembered and understood when they appear to grow naturally out of simpler actions.
- Every double-click includes a single-click (the first click in the sequence), and every triple-click includes a double-click. At the time an application receives one click, it cannot know whether additional clicks will be received. So it must first act on the single click, then the double-click, then the triple-click.

For example, the first click of a double-click on a File Viewer icon selects the icon, just as a single-click would. The second click activates the application associated with the icon. A single click in text selects an insertion point, a double-click extends the selection to a word, and a triple-click extends it further to a full line, sentence, or paragraph.

Quadruple-clicks (and above) become increasingly difficult for users to produce or understand. They are neither used nor recommended in the OpenStep user interface. Even triple-clicks should be used sparingly.

Dragging From a Multiple-Click

The act of pressing a mouse button to initiate dragging can be part (the last part) of a double-click or triple-click. If the user does not immediately release the mouse button and begins dragging at the end of a multiple-click, the dragging action can be assigned a meaning that is related to the meaning of the multiple-click.

In text, for example, double-clicking selects a word and dragging from a double-click selects additional words within a range of text. If triple-clicking selects a line, dragging from a triple-click will select additional lines within the range.

How to Use Dragging

Dragging can be implemented in a variety of situations. The three most common are:

- Moving objects, such as windows and the knobs of scrollers
- Defining ranges, usually to select the objects within the range
- Sliding from one object to another in order to extend an action initiated in the first object to the second object

The following sections cover these situations in detail.

Moving Objects

The user can drag a movable object by positioning the pointer over it, pressing the mouse button, and moving the mouse while the button is down. The object should move in such a way that it remains aligned with the pointer on screen. When the object is constrained within an area or a track as a scroller knob is, it should remain as closely aligned with the pointer as possible.

The Application Kit contains support for moving objects between and within applications. It even changes the pointer to indicate whether the object is being moved, copied, or linked. See “Managing the Pointer” earlier in this chapter for more information on changing the pointer.

Defining Ranges

The user can also drag over an area or through a series of items such as text characters to define a range. The position of the pointer when the mouse button is pressed is the *anchor point*. Its position when the mouse button is released is the *endpoint*. The difference between the anchor point and endpoint determines the area or objects inside the range. Figure 3-4 shows anchor point and endpoint locations.

Dragging to define a range is mostly used to make a selection (such as a string of text characters or a group of icons) for the targeted-action paradigm.

When the user drags to define a rectangular range, as in a drawing program, applications often drag out, or “rubberband,” a rectangle that shows the area between the anchor point and endpoint. See “Selection in Text and Selection in Graphics” earlier in this chapter for more information on the dragging operations that define a rectangular range.

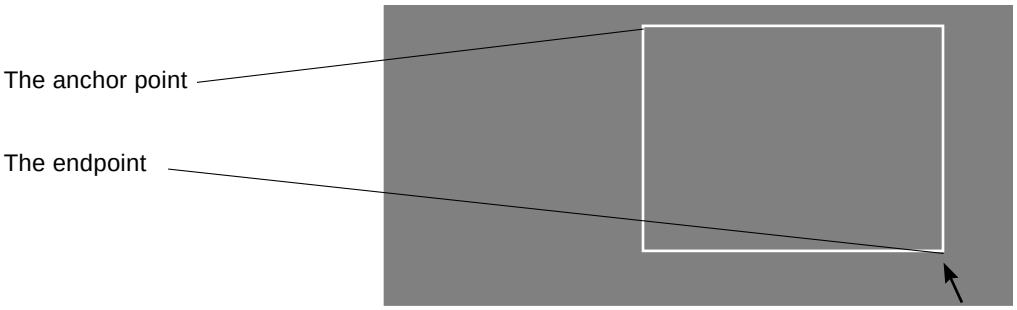


Figure 3-4 Dragging to Define a Range

Sliding From Object to Object

Sometimes a group of closely related objects reacts to dragging as though the user clicked on one of the objects. No matter which object in the group was under the pointer when the mouse button was pressed, the object chosen is the one that is under the pointer when the mouse button is released. (Notice that when an object is not in a group, it is chosen only when it is underneath the pointer both when the mouse button is pressed and when it is released.)

When working with menus users can press the mouse button when the pointer is over one command and release it when it is over another, which chooses the command that is under the pointer when the button is released. Similarly, when working with a tool palette, users can drag the pointer from one tool to another, or when working with a set of radio buttons they can drag the pointer from button to button.

The grouped objects do not have to be all of the same type. For example, a user can drag from a button that controls a pop-up list through the list to make a selection, or from a menu command that opens a submenu into the submenu.

If the user can drag from one object to another in a group of objects, then this fact should be apparent from the way the objects are displayed. Usually, such objects are displayed in a single row or column, as close to each other as

possible. For example, graphical radio buttons should be displayed next to each other, to distinguish them from ordinary buttons. (Graphical radio buttons are discussed in detail in Chapter 7, “Controls.”)

When to Use Pressing

For the most part, pressing is an alternative to repeated clicking. It should be used wherever a control’s action can be repeated with incremental effect. Clicking a scroll button is a common example. One click scrolls one line of text, a second click scrolls another line, and so on. Pressing the scroll button scrolls continuously, until the mouse button is released.

Pressing is also used to begin the mouse action of sliding from one object to another. If a button controls a pop-up list, the user presses the button and drags through the list to choose one of its options. After pressing a menu command to bring up a submenu, the user can drag into the submenu.

Using Modifier Keys With the Mouse

You can use modifier keys to change the results of mouse operations. Modified mouse actions should be implemented for optional or advanced features of your user interface, because they are harder to remember and require more coordination to produce. They typically extend or alter the effect of the standard mouse action. For example:

- Dragging a file icon from one directory window to another moves or copies the file to the new directory, depending on whether the directories are on the same disk or a different disk. Command-dragging always moves the file. Alt-dragging always copies the file. Control-dragging is similar to copying, but the new “copy” of the file is simply a link to the old copy. The user knows what will happen because of the visual feedback the application provides. It changes the pointer shape (to a  , for example, when Alt-dragging).
- Clicking on a scroll button scrolls a line of text. Alt-clicking scrolls a larger amount of text.
- Alt-clicking on a window moves it to the front without making it the key window or activating its application.

- Clicking selects a new insertion point in text. Alt-clicking extends the selection to include everything between the current insertion point and the point of the click. If an application implements discontinuous selection, Shift-clicking will select a new insertion point without dropping the old selection. If discontinuous selection is not implemented, Shift-clicking acts like Alt-clicking.
- Clicking selects an icon in a directory window. Shift-clicking adds new icons to the current selection. (Shift-clicking of a selected icon removes it from the selection.)

Note – The Caps Lock key does not work for Shift-clicking or Shift-dragging—the Shift key must be held down manually. This way, users will not find themselves Shift-clicking by mistake when they intend only to click.

The Control key (with no other modifier keys) is often used for mouse actions that create or act on links. If your application's documents can receive linked information, then you need to implement the Control–double-click accelerator, as described in the discussion of the Link Inspector panel in Chapter 5, "Panels."

Some modifier keys should be used only in limited circumstances:

- The Help key should be used only for Help-clicking, which brings up help on the clicked object.
- The combination of Alt and Control should be used only as a substitute for the Help key. (This is necessary because some keyboards do not have a Help key.)

Applications should avoid distinguishing between the left and right key of the Shift and Alt pairs. Users do not expect such a distinction except for certain computerwide, potentially destructive operations, such as resetting the computer's processor. Also, there is no hardware-independent way to differentiate the left and right modifier keys.

Keyboard Actions

In theory, the keyboard is used only to enter text, and the mouse is used for all other operations, such as making selections and using controls. In reality, a number of actions are initiated from the keyboard. A few, such as Return and Enter, are traditional keyboard actions. Others are keyboard alternatives that

provide convenient shortcuts for initiating mouse actions. There are some special key combinations that modify mouse actions. Figure 3-5 shows a typical Sun keyboard layout.

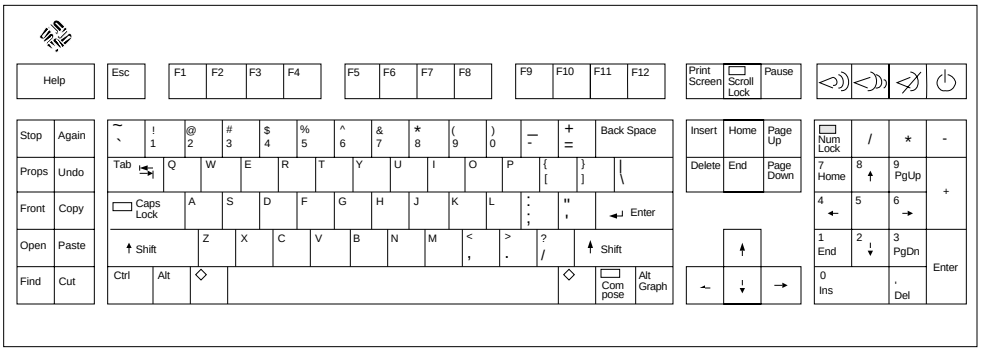


Figure 3-5 Typical Keyboard Layout for a Sun Computer

A computer keyboard includes standard typewriter keys, which are used to enter letters, numbers, and common symbols. This characteristic is covered in *Using the OpenStep Desktop.*

The computer keyboard also has keys that are not found on typewriters, including Alt, Control, Command, Help, Enter, and the Arrow keys. These keys, along with the Shift key, are used in keyboard actions and must be considered when developing an application.

Modifier Keys

Shift, Alt, Control, Command, and Help are known as *modifier keys* because they can modify the results of a mouse operation or another key on the keyboard. If a user presses Shift-3, for example, it yields the # character, and pressing Command-c issues a Copy command. Pressing a modifier key by itself does nothing.

To use a modifier key, the user must hold it down and press the key (or perform the mouse action) to be modified. More than one modifier key can be used at a time. (Command-Shift-c is a valid combination.)

Use of modifier keys is summarized the following paragraphs.

- The Shift key modifies keystrokes to produce the uppercase character on alphabetic keys and the upper character on numbers and other two-character keys. It is sometimes used with other modifier keys.
- The Alt key modifies keystrokes to produce an alternate character to the one that appears on the key. In general, the characters produced are special characters that are used relatively infrequently. To find out which alternate characters are generated by which keys, see *Using the OpenStep Desktop*.

Note – On SPARC keyboards, this function is mapped to the Alt key. On x86 keyboards, it is mapped to the right Alt key.

- The Control key modifies keystrokes to produce standard ASCII control characters. Some control characters are generated by single character keys—for example, Tab is Control-i, Return is Control-m, and backspace (Shift-Delete) is the same as Control-h.

Note – On SPARC keyboards, this function is mapped to the Control key. On x86 keyboards it is mapped to the Ctrl key.

- The Command key provides a way to choose menu commands with the keyboard rather than the mouse. As an alternative to clicking on a menu command with the mouse, the user can press the Command key in conjunction with the character displayed in the menu next to that command. For example, Command-c chooses the Copy command.

Note – On SPARC keyboards, this function is mapped to the Meta (◆) keys. On x86 keyboards, it is mapped to the left Alt key.

- The Help key does not modify keystrokes. It is used only to modify mouse actions, as described in “Using Modifier Keys with the Mouse” earlier in this chapter.

Note – On SPARC keyboards, this function is mapped to the Help and F1 keys. On x86 keyboards, it is mapped to the F1 key.

Special Command-Key Combinations

Several Command-key combinations produce special effects. Some play a particular role in the user interface. Others, in effect, give commands to the computer itself rather than to just one application. They cannot be used for functions other than those listed below.

- Command-. (period) should let users cancel the current operation in the active application. (The Esc key also performs does this.) Although the Application Kit has code to support Command-., it is not automatic. An application must ask for this functionality.
- Command-space should be used for file name completion. In contexts where it is appropriate for the user to type a file name (such as in an Open panel), Command-space bar displays as many characters as match all possible file names in the directory. If the user first types enough characters to identify a particular file and then presses the space bar with the Command key down, the remaining characters of the file name are filled in. (In many applications, the Esc key also performs file name completion.)

Other Action Keys

This section describes the kinds of results that you can appropriately implement for the non text keys that initiate actions. These keys generate non display characters that tell the application to initiate some action. The action will depend on how the application handles the special character. Some typical responses to these special character keys are listed here.

- The Return key moves the insertion point or prompt to the beginning of the next line, much like the carriage return key of a typewriter. When data is entered in a text field or form, Return moves the insertion point and informs the application that the data is ready for processing.
- The Enter key, like Return, signals that data is ready for processing. It need not move an insertion point or prompt to the beginning of the next line. Enter can also be generated with Command-Return.
- The Backspace key, usually located above the Return key, as shown in Figure 3-5, removes the preceding character in text or deletes the current selection. Shift-Backspace generates the backspace character, which moves the insertion point back one character. In most applications, the Del key has the same function as the backspace key.

- The Tab key moves forward to the next tab stop, or to the next text field in a sequence. Shift-Tab moves backward to the previous tab stop or text field.
- The Arrow keys move the symbol that is used in some contexts to track where the user is writing or entering data—for example, the insertion point in a document processor. The arrow keys' action is described in "How the Arrow Keys Affect a Text Selection" later in this chapter, and "Implementing Modifier—Arrow Key Combinations" earlier in this chapter.

Handling Arrow Characters

Because the Arrow keys generate the same character codes as the Symbol font's arrow characters, text objects should check to see which key generated the character. The Arrow keys never produce visible arrow characters. However, when a non arrow key (perhaps modified by the Alternate key) produces an arrow character code, it *should* produce visible arrow characters, and not result in Arrow key functionality.

For example, Alt-f should produce a visible left-arrow symbol, as shown in the *User's Guide*, instead of moving the insertion point left one character.

Implementing Keyboard Alternatives

Many users find it faster and easier to operate graphical objects with their keyboards than with the mouse. The standard menus include keyboard alternatives for some operations; you should consider providing keyboard alternatives for the operations that will be most frequently performed in your applications.

Note – For most applications, keyboard input is handled automatically. Text entry and display are handled by the App Kit's Text object, and keyboard alternatives are automatically converted into clicks on the controls they are associated with. All you have to do is choose keyboard alternatives (as discussed later in this chapter) and then specify them in Interface Builder™. If your application does not use the Text object for its text entry you will need to write code that handles keyboard input.

Keyboard alternatives consist of a single alphanumeric keystroke, which is modified by the Command key (and possibly another modifier key). They are most often used for menu commands, although they are permitted for a panel's buttons and pull-down lists, as well.

Although a keyboard alternative is associated with a graphic object, the object does not have to be on the screen for the alternative to function. Keyboard alternatives for menu commands and panel buttons will work when the menu or panel is hidden.

This section contains guidelines for assigning keyboard alternatives to application functions. There are three sets of keyboard alternatives—reserved, required, and recommended. Each set is listed in its own table.

Notice that users can use the Preferences application to change the keyboard alternatives (for every application), which means the keyboard alternatives you set up are only the initial settings.

Reserved Keyboard Alternatives

If your application provides a function listed in Table 3-1, you must use the command and reserved keyboard alternative assigned to it. For example, if your application opens files it must have the Open command, and the function Command-o must be the keyboard alternative.

You cannot use these reserved characters as alternatives for any other commands. If your application does not allow the user to open files, it will not have an Open command and must not use Command-o as a keyboard alternative. (See "Standard Menus and Commands" in Chapter 6 for more information on the listed commands and menus.)

Table 3-1 Reserved Keyboard Alternatives

Keyboard Alternative	Command	Menu
Command-?	Help	Info menu
Command-a	Select All	Edit menu
Command-c	Copy	Edit menu
Command-h	Hide	Main menu
Command-n	New	Document menu
Command-o	Open	Document menu

Table 3-1 Reserved Keyboard Alternatives *(Continued)*

Keyboard Alternative	Command	Menu
Command-p	Print	Main menu
Command-q	Quit	Main menu
Command-s	Save	Document menu
Command-v	Paste	Edit menu
Command-w	Close Window	Windows menu
Command-x	Cut	Edit menu
Command-z	Undo	Edit menu

Required Keyboard Alternatives

If your application provides a function listed in Table 3-2, you must use the command and required keyboard alternative assigned to it. For example, if the application has a Find panel, you must use Command-f as the keyboard alternative for bringing it up. See “Standard Menus and Commands” in Chapter 6 for more information on the listed commands and menus.

If the application does not implement a listed function (it does not have a Find panel), you can assign its required keyboard alternative (Command-f) to another menu command. However, to preserve interapplication consistency, it is strongly recommended that you first try to use characters other than those on this list.

Table 3-2 Required Keyboard Alternatives

Keyboard Alternative	Command	Menu
Command-;	Check Spelling	Edit menu
Command-b	Bold (Unbold)	Font menu
Command-d	Find Previous	Find menu
Command-e	Enter Selection	Find menu
Command-f	Find Panel	Find menu
Command-g	Find Next	Find menu

Table 3-2 Required Keyboard Alternatives *(Continued)*

Keyboard Alternative	Command	Menu
Command-i	Italic (Unitalic)	Font menu
Command-t	Font Panel	Font menu
Command-C	Colors	Varies

Recommended Keyboard Alternatives

If your application provides a function listed in Table 3-3 *and* you want to set up a keyboard alternative for it, you must use the recommended keyboard alternative. For example, if you provide a keyboard alternative for the Copy Ruler command you must use Command-1.

If you don't use the keyboard alternative for the recommended function, you can use it for another command. For example, if you don't set up an alternative for the Copy Ruler command, Command-1 can be used as a keyboard alternative for another command.

Table 3-3 Recommended Keyboard Alternatives

Keyboard Alternative	Command	Menu
Command-1	Copy Ruler	Text menu
Command-2	Paste Ruler	Text menu
Command-3	Copy Font	Font menu
Command-4	Paste Font	Font menu
Command-j	Jump to Selection	Find menu
Command-m	Miniaturize Window	Windows menu
Command-r	Show Ruler	Text menu
Command-P	Page Layout	Format menu
Command-S	Save As	Document menu
Command-V	Paste and Link	Link

Application-Specific Keyboard Alternatives

Because the OpenStep user interface is visual, every operation—every menu command, every scrolling operation, and so on—has a graphical representation that can be operated with the mouse. Keyboard alternatives are just that: alternatives. You should never set up an alternative for an operation that cannot be performed with the mouse.

Keyboard alternatives are allowed only for the commands in a menu, the buttons in a panel, or the items in a pull-down list. The characters used as keyboard alternatives must be visible to your users in the menu, panel, or list. Menus display them on the commands, and pull-down lists follow this example. A panel can identify the keyboard alternatives for its buttons in any way appropriate to the design of the panel.

When you decide which operations will have keyboard alternatives, your main consideration should be frequency of use. It is better to assign a keyboard alternative to a frequently used command than to one that is used less often. Infrequently used commands—such as the Info Panel command—should never have keyboard alternatives.

You should also try to assign keyboard alternatives to commands that are needed while users are working with the keyboard (for example, the commands in the Find menu). Keyboard alternatives for these commands let users keep working without moving from the keyboard to the mouse and back again.

You can also use keyboard alternatives that enable proficient users to work with one hand on the keyboard and the other on the mouse. For example, Command-x, Command-c, and Command-v allow users to select with the mouse while carrying out cut, copy, and paste operations from the keyboard. These keyboard alternatives free users from having to move the pointer out of the region where they are working just to click on a command.

If a keyboard alternative is assigned to one command from a set of parallel commands that control formatting or viewing data (a set of commands that lets users sort items in various ways, for example), the command that restores the default setting should have a keyboard alternative, too. Keyboard actions can then take the user to an alternate format and back to the default, rather than just half way.

Note – You do not need to assign a keyboard alternative to every command. Remember that users can create their own global keyboard alternatives with the Preferences application.

Choosing the Character

Any character except the period (.) and the space can be used for a keyboard alternative, but some characters have been set aside for the reserved, required, and recommended keyboard alternatives. If the character is a letter, it can be either uppercase or lowercase, although lowercase characters are preferred because they do not require the user to press two modifier keys (Command and Shift) at once.

When choosing the character for a keyboard alternative, try to make it mnemonic. If possible, it should be the first letter of the command it performs. If it is closely related to a command that already has a keyboard alternative, then you might want to choose a character physically near the existing one.

For example, the Find command's keyboard alternative is Command-f, taken from the first letter of the command. Two commands related to Find have keyboard alternatives that use the keys next to "f." The Find Next command uses Command-g, because it is just to the right of "f" and Find Previous command uses Command-d, because it is just to the left of "f."

Using the Alternate Key

If necessary, your application can have keyboard alternatives that use the Command and Alt key together to operate them. This is not desirable; you should exhaust all reasonable possibilities using the Command key or Command-Shift before you use Alt. See Figure 3-6.

If you do set up a keyboard alternative that requires the Alt key, the alternative character should be displayed in italics.

Figure 3-6 Using the Alternate Key in a Keyboard Alternative

Notice that the italicized character identifies the *key* that the user presses while holding down Command and Alt, rather than the *character* that would be typed if the user held down Alt.

Note – Keyboard alternatives operated with the Alt key are difficult to recognize, and the significance of italic characters in menus is not explained in the OpenStep user documentation. If you implement any keyboard alternatives operated by the Alt key, make sure that the documentation for your application explains the meaning of the italicized character.

When Mouse Operations and Keyboard Alternatives Differ

A keyboard alternative should generally do exactly what the corresponding mouse action does. There are, however, a few cases in which it is acceptable to have the keyboard alternative do just a bit more than the mouse operation. These cases are rare and they often go unnoticed by users because the differences are both subtle and intuitive.

For example, the Edit application has a Find panel which is brought up with the Find Panel menu command or its Command-f keyboard alternative. When opened from the menu, the panel usually remains open until the user explicitly closes it, since it may be used for several Find operations before it is closed. But a user whose attention is focused on typing often wants to find a word in a document and then resume typing without returning to the Find panel.

Edit handles this by implementing a slightly different behavior when the panel is opened with the keyboard alternative. If a user opens the panel with the keyboard *and* initiates the search by pressing Return, Edit assumes that this user wants to continue typing as soon as the word is found, and it closes the panel when the Find operation is complete. The user does not need to switch to the mouse and activate the document window.

A keyboard alternative that has been handled like this simply does what the user expects and accomplishes either or both of the following:

- Reduces the number of clicks or keystrokes the user needs to perform
- Eliminates the need to switch from the keyboard to the mouse and back again

In general, you should start with keyboard alternatives that result in the same behavior as their associated mouse actions. In those rare cases when the keyboard alternative should differ from the mouse action, the need becomes apparent in everyday use.

Implementing Modifier Key-Arrow Key Combinations

If your application allows text selection, you are encouraged to implement the characteristics described in this section for modifier key-Arrow key combinations. These combinations retain the basic directional orientation of each arrow key but alter the behavior of the key somewhat. Before reading this section, you should understand the standard arrow key behavior, which is described in "How the Arrow Keys Affect a Text Selection," earlier in this chapter.

Control-Arrow Combinations

An arrow key modified by the Control key should move the insertion point to the edge of the current display. The left arrow key moves the insertion point in front of the first visible character on the line where the current selection begins, and the right arrow key puts it after the last visible character on the line where the current selection ends.

The up arrow key moves the insertion point to the first visible line of the display, directly above the beginning of the current selection. The down arrow key moves the insertion point to the last visible line, directly below the end of the current selection.

When the insertion point is already at one edge of the currently visible display, the Control-Arrow combination that would normally move it to that edge will scroll the display (by the amount of a page scroll) and move the insertion point to the edge of the new display. When the insertion point reaches the beginning of a line, the left Arrow key does not move it further (for example, to another line). The right Arrow key does not move it beyond the end of a line.

Shift-Arrow Combinations

An Arrow key modified by the Shift key should move the insertion point from word to word. Shift-left arrow moves it to the left of the current selection to the beginning of a word. Shift-right Arrow moves it to the right of the current selection to the end of a word.

Shift-up Arrow moves the insertion point to the beginning of the word one line directly above the beginning of the current selection. Shift-down Arrow moves it to the end of the word one line directly below the end of the current selection. As the up and down Arrow keys move the insertion point from line to line, they choose words that lie directly above or below the original starting point. (In other words, the location of the insertion point can be calculated by the same rules that determine the edge of the selection when the user double-clicks and drags directly upward or directly downward.)

Alt-Arrow Combinations

An Arrow key modified by the Alt key should extend the current selection by one character or one line at a time. The edge moved first becomes the active edge; any subsequent motion with an Arrow key will move the active edge—left or right, up or down.

If, for example, the user presses Alt-left Arrow first, it moves the beginning of the current selection (its left edge) one character to the left (toward the beginning of the document). After this, the left and right Arrow keys will both move the *left edg*e, either one character to the left or one character to the right. If the user presses Alt-right Arrow first, the end of the current selection (the right edge) moves one character to the right (toward the end of the document) and becomes the active edge, after which both keys move the right edge.

Alt-up arrow and Alt-down Arrow are similar. If the user presses Alt-up Arrow first, the top edge of the selection moves up one line and becomes the active edge. If the user presses Alt-down Arrow first, it moves the edge of the current selection down one line, after which both keys move the edge of the selection.

Other Arrow Key Combinations

More than one modifier key can be used in combination with an arrow key, with additive results. Thus, Shift-Alt-right Arrow extends the selection to the end of a word, and Control-Shift-up Arrow places the insertion point at the beginning of a word in the first visible line of the display. Remember that Alt and Control cannot be used together because that combination is reserved by the system as an alternative for the Help key.

Windows in the OpenStepInterface 4

The OpenStep user interface communicates to its users by displaying text and graphics in windows. Many of the windows also receive input from the users. This chapter presents the guidelines for window appearance and behavior that will help you develop windows that communicate effectively with your users. It also covers standard window behavior in detail.

Detailed coverage of the other types of windows is in separate chapters: for panels, see Chapter 5, “Panels,” and for menus, see Chapter 6, “Menus;” for pop-up and pull-down lists, see Chapter 7, “Controls,” and for miniwindows and application icons, see Chapter 1, “A Visual Guide to the User Interface.”

Note – In documentation for your users, try to use the term *window* only for standard windows, although you can refer to panels and menus as special types of windows. You should refer to miniwindows, lists, and icons only by those names; don’t use the generic term *window* for them, as it would imply common implementation that is lacking.

4

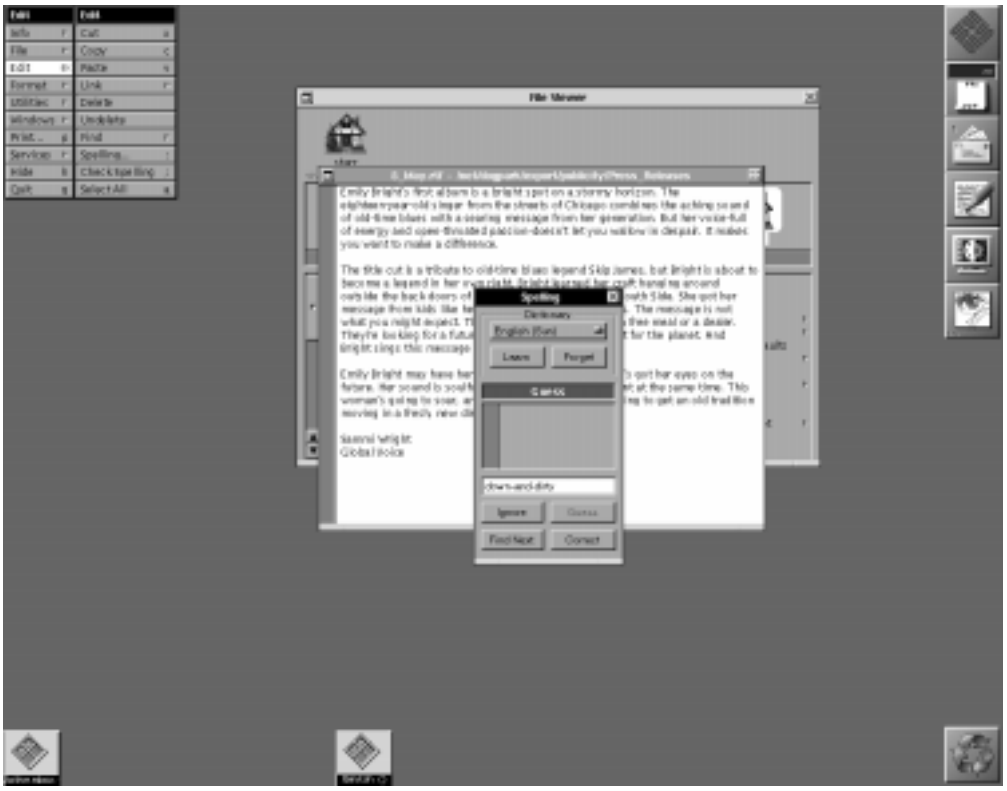


Figure 4-1 OpenStep Window Interface

How Windows Work

This section covers the basic appearance and behavior of all OpenStep windows—standard windows, panels, menus, pop-up and pull-down lists, miniwindows, and application icons.

Note – The window behavior described in this section is automatically handled by the Application Kit's NSWindow class and its subclasses. The application-specific behavior you design and implement is covered in "Implementing Windows" later in this chapter.

Parts of a Window

The parts of a window are illustrated in Figure 4-2.

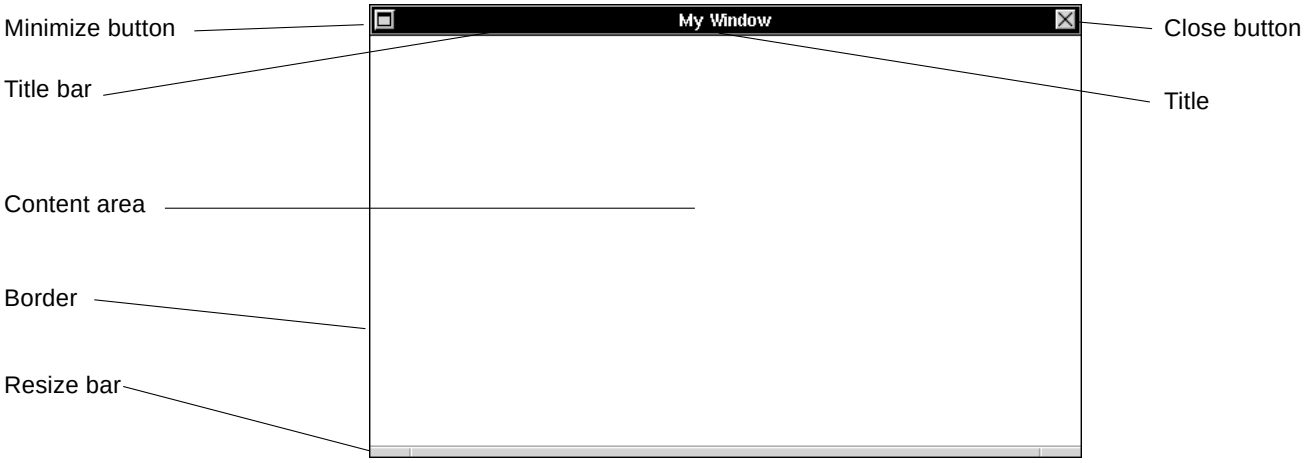


Figure 4-2 Parts of a Window

Every window has a content area where the application can be used to enter text and images.

Standard windows, panels, and menus also have title bars above their content areas and borders that surround both content area and title bar. The title bar is the user's control center for one of these windows. The title bar displays the window's title, if it has one, and often contains buttons the user can click on to dismiss it from the screen. In addition, the user moves the window by dragging its title bar.

Panels and standard windows can also have resize bars, which appear at the bottom of a window, below the content area but within the border. By dragging any of the regions of the resize bar, users can alter the sizes and shapes of the windows on their screens. The resize bar is the only window control outside the title bar.

Window Order

As users work, they move windows around and move from window to window. This freedom is essential, but it does allow them to put larger windows over smaller ones. If there were no restrictions on window ordering, users could cover their menus and docked icons with standard windows and even lose sight of attention panels and pop-up lists that required immediate attention.

To prevent problems of this sort, which would certainly confuse users, the different types of windows (menus, panels, standard windows, and so on) are assigned to different *levels*. A window is only allowed to cover windows in lower levels. There are seven levels:

- Windows that appear in a *spring-loaded mode*—pop-up lists, pull-down lists, and menus opened with the mouse's Menu button—are assigned to the frontmost, or first, level. These windows remain on screen only while the user holds the mouse button down, and so they only momentarily obscure other windows. Putting them in the first level guarantees that they will not come up behind other windows.
- Attention panels are assigned to the second level. Like spring-loaded windows, they are on screen temporarily. But unlike spring-loaded windows, which disappear when the user releases the mouse button, attention panels must be explicitly dismissed. They are assigned to the second level, so they cannot be covered by other windows. This keeps them in front of the user until they are dismissed, thus encouraging prompt user responses.
- The main menu is assigned to the third level. Unless some user action has opened a spring-loaded window or an attention panel, the main menu is the frontmost window on the screen.
- Other menus are assigned to the fourth level, just below the main menu. They can cover each other, but not the main menu.
- Docked application icons occupy the fifth level. They can be covered by spring-loaded windows, attention panels, and menus, but not by the ordinary windows of your application.

- Floating panels are in the sixth level. Floating panels are defined and discussed in Chapter 5, “Panels.”
- All other windows are grouped in the seventh level. This is the largest level, and it holds most of the windows on the screen. These windows can cover each other, but cannot come in front of spring-loaded windows, attention panels, menus, the dock, or floating panels.

This seven-level system keeps windows in the upper levels in view and readily available to the user; it prevents them from being inadvertently lost in a large pile of windows. Although attention panels, menus, and docked application icons can cover other windows, the user can get them out of the way when necessary: attention panels can be attended to and dismissed, menus can be moved to the side or closed, and the dock can be slid mostly off screen.

Some other points worth noting are listed below:

- When a window is first placed on screen, it comes up at the front of its level to get the user’s attention.
- When two windows belong to the same level, either one can be in front. When two windows belong to different levels, however, the one in the higher level will always be above the other.
- Every window has its own position in the order; even when two windows appear to by side-by-side, one is technically in front of the other.
- Even when a window is covered by other windows, it is considered to be on screen; it retains its ranking in the order and can be exposed by moving the windows above it to the side.
- Application and window status (which is the currently active application and which is the current key window) also play a part in determining the order in which the windows are displayed. See “Application and Window Status” later in this chapter for more information.

Window Characteristics

This section describes basic window characteristics. Notice that some of these behaviors are implemented by including optional parts (see Figure 4-2) in a particular window; you can modify the behavior of windows you create by adding or removing these parts:

- Reordering — Any window can be brought to the front of its level. Notice that it can only be brought to the front of its own level, which prevents users from moving a standard window in front of an attention panel or a menu.
- Moving — Any window with a title bar can be moved to a new location on the screen, as can any miniwindow or application icon.
- Resizing — Any window with a resize bar can be resized.
- Closing and miniaturizing — A window with the appropriate buttons in its title bar can be closed or miniaturized.
- Hiding and Retrieving — The Hide menu command lets users hide windows so they are not visible on the screen. This is not the same as closing them.

Reordering

Clicking on a window brings it to the front of its level, unless the click is on a title bar button. The window is reordered immediately when the mouse button is pressed. If the user is dragging the window to a new location, the window moves to the top of its level before it is moved.

Another way the user can reorder windows is to hold the Command key and press either the up-arrow or down-arrow. Command-up Arrow moves the backmost window (if it is in the lowest level) or standard window to the front of the level. Command-down Arrow moves the frontmost window to the back.

Moving

The user can drag any window by its title bar (if it has one). To drag a window, users must press and release the mouse button. This counts as a click and brings the window to the front of its level.

Resizing

Figure 4-3 Resizing a Terminal Window

If a window has a resize bar, users can change its size by dragging the resize bar. While the bar is being dragged, an outline of the window's border follows the pointer, as shown in Figure 4-3. When the user releases the mouse button, the window is resized to the outline.

Closing and Miniaturizing

A window's title bar can display two buttons, which are shown in Figure 4-4:

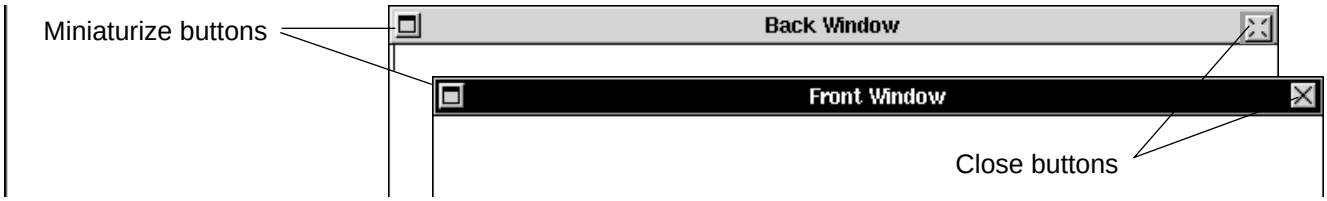


Figure 4-4 Title-Bar Buttons

The window in front has both buttons as they normally appear. The *miniaturize button* is on the left, and the *close button* is on the right. The window in back shows a *broken close button*, which indicates that the window is displaying a document that the user has edited but not saved. “Implementing Windows” later in this chapter has more information on the miniaturize and close buttons.

The user clicks the buttons either to miniaturize or close the window, as described in Table 4-1. The click does not bring the window to the front, make it the key window, or activate an application. (The key window and the active application are discussed in “Application and Window Status” later in this chapter.)

Table 4-1 Buttons in a Window’s Title Bar

Button	What It Does
Miniaturize button	Replaces the window with a miniwindow. The miniwindow represents the window on screen and gives the user access to it. Double-clicking the miniwindow causes it to disappear and the window that was miniaturized to reappear.
Close button	Closes the document displayed in the window, and removes the window from the screen.

Miniaturizing

Miniaturizing a window clears it from the screen without destroying it or changing its contents. From the user’s point of view, the window is transformed into a miniwindow. Double-clicking the miniwindow reverses the miniaturization.

Most standard windows and some panels have a miniaturize button. They can be miniaturized using either the button or the standard Miniaturize Window menu command. A group of windows representing a single document can be miniaturized into a single miniwindow, as described in “Document Menu” in Chapter 6.

Users cannot work in a miniaturized window, but programs can continue to update the window’s display. If you begin compiling a program in a Terminal window and then miniaturize the window, the compiler messages will be visible when you unminiaturize the window.

Miniaturizing differs from closing in a number of ways:

- Miniaturizing preserves the window as it was last seen on screen. A window that is closed cannot necessarily be brought up in the same state.
- Miniaturizing a window leaves a miniwindow behind so that it can be brought back up on the screen. Closing a window does not provide the user with a way to do this.
- Miniaturizing a window that displays a file will not the file or change the way it is displayed. Closing a window usually closes the file it displays.

Closing

The close button removes a window from the screen. What this means depends on the type of window:

Table 4-2 Closing a Window

Type of Window	Effect of Closing the Window
Menus and panels	A menu that is closed is removed from the screen, but the user retains a way to reopen it quickly with a command in another menu. Panels that are closed can be reopened in the same way. (See Chapter 6, “Menus,” for more information on menus.) When a panel that was closed is reopened, it assumes its former size and location and retains its former state. From the user’s point of view (and programmatically) it is the same panel that was closed.
Standard windows	Closing a standard window usually removes it from the application as well as from the screen. From the user’s point of view, the same window cannot necessarily be brought up again. The selection in effect when the window was closed might not be preserved, and the new window will not necessarily be located in the same place or have the same dimensions as the old one, especially when the user moved or resized the window before closing it.

Hiding and Retrieving Windows

The Hide menu command lets the user clear the screen of all the windows that belong to one application. This clears the workspace and makes it easier to work in another application.

When an application is hidden-only its application icon remains on the screen. When the user double-clicks the icon, the hidden windows reappear, arranged as they were before being hidden. Users can resume working in the application, picking up again at exactly the point where they left off.

Double-clicking the icon has one other effect: it activates the application (as discussed in the next section), which may cause the menus and panels of another application to disappear while those of the newly activated application reappear.

Double-clicking the icon for a running application activates it and brings its windows to the front, even if the application was not hidden. (The user can also bring covered windows forward using commands in the Windows menu, as described in Chapter 6, “Menus.”) The application’s menus also return to the screen.

If the user holds down the Command key while double-clicking an application icon, the application is activated as usual, but in addition all other applications are hidden.

Notice that when a window is completely obscured by other windows, it is covered by them but not hidden in the sense used here. Users can make a covered window visible by moving the windows above it to the side. This will not work for a hidden window, which is completely removed from the workspace.

Application and Window Status

Since more than one application can run at a time in OpenStep, the screen is likely to display windows for several applications. Users must be able to pick the application and window they want to work in. There are three OpenStep concepts that help them do this:

- The application that the user is currently working in is known as the *active application*.
- The windows that are the current focus of user attention in the active application are known as the *key window* and the *main window*. These are usually one and the same.

The terms key window and main window describe functional roles. Both roles can be assumed by the same window, or each can be assumed by a different window:

- The key window is the window that receives characters from the keyboard.
- The main window is the window containing the selected target for controls.

Keep in mind that the active application, the key window, and the main window, are not fixed properties of specific applications and windows, but concepts that describe an application's or a window's status at a particular point in time. The status will change as the user works with the computer, and OpenStep provides feedback to indicate where the active application, key window, and main window are. This is discussed more fully in the three sections that follow.

Active Application

From all of the applications running on a computer, the user can select only one to be the active application. The user must activate an application before being allowed to enter text in its windows or use its menus.

The appearance and behavior of the active application are distinguished in the following ways:

- It is the only application with visible menus. When an application is deactivated, its menus are hidden from view. When it is reactivated, they are restored to the screen.
- It is the application that owns most, if not all, of the panels that are visible on the screen. In general, panels behave like menus. They hide when the application is not active and return to the screen when the application is reactivated. (In some situations it is appropriate to implement a panel that remains on screen when the application is not active; see Chapter 5, "Panels," for guidelines.)
- It is the application that receives the user's keyboard actions. Typing and keyboard alternatives can affect only the active application. When there is no active application the user's keystrokes have no effect.
- It is the application that contains the key window and main window (if there is a current key window or main window), and its windows are likely to be in front of the windows of other applications.

Application Activation

In general, the task of selecting the active application is left to the user. The user's action can be direct, such as starting up the application or clicking on one of its windows, or indirect, such as having one application send a message to another application.

The only exception to this principle is when the user hides or terminates the active application. In this situation, OpenStep guesses which of the applications with windows on the screen should be activated next, which often saves the user from clicking to choose the new active application.

The actions that activate an application are:

- The user starts it up, unless the user activates another application while the first one is starting up.
- The application owns the panel or standard window that is frontmost on screen after the user hides or terminates the current application.
- The user double-clicks a miniwindow belonging to the application, or double-clicks the application's freestanding or docked icon. Double-clicking a docked icon starts up the application if it is not already running.
- The user clicks on one of the windows that belong to the application, provided the window is not a miniwindow or application icon.
- The application receives a message from another application that asks it to do some work requiring interaction with the user. Receiving a message from the Workspace Manager that asks the receiving application to open a file is one example. See "Activating an Application" later in this chapter for details.

Application Deactivation

There can be only one active application per workspace (that is, one per Window Server) at a time. Whenever the user chooses a new active application, the currently active application is automatically deactivated. The Application Kit and Workspace Manager take care of this task.

The active application is also deactivated when:

- The user hides its windows (by using the Hide command).
- The user terminates it (by choosing the Quit command).

In either case, if there are other applications with panels or standard windows on the screen, then the Workspace Manager activates the application that owns the frontmost panel or window. If no other application has panels or a standard window on screen, no application becomes active.

In addition, an application that sends a message to another application with intention of activating the other application should deactivate itself just before sending the message. See “Activating an Application” later in this chapter for details.

Note – A deactivated but running application can still do work. It is deactivated only in the sense that the user cannot interact with it without reactivating it.

Key Window

Users expect to see their actions on the keyboard and mouse applied to a specific window in a specific application. They need to know which window will be affected before they act—there should be no surprises.

Because the user is controlling the pointer location with the mouse, the user can place the pointer over a window and know that the next mouse action will affect that window. The window where typing will appear must also be obvious to users, so OpenStep visually identifies the key window (the window where typing will appear).

To identify the key window for users, the App Kit highlights its title bar by turning it black. This is illustrated in Figure 4-5.

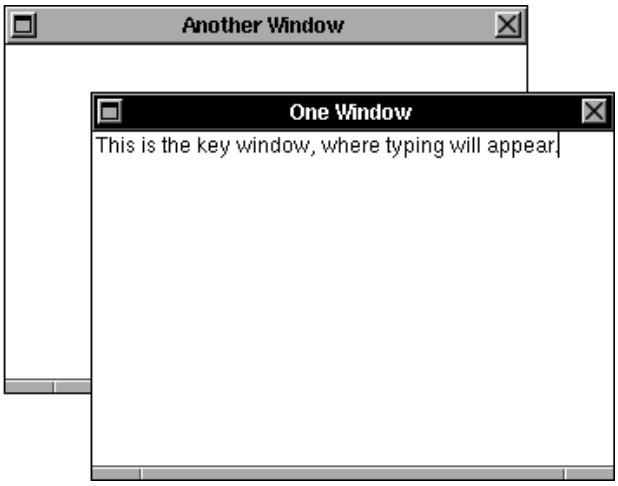


Figure 4-5 Key Window Highlighting

You can think of the title bar highlighting as a pointer for the keyboard. It moves from window to window as the key window changes. Key-window status also moves from application to application as the active application changes. Only one window on the screen has the highlighted title bar, and it must belong to the active application. There is just one key window per computer. Even a system that has two screens, but only one keyboard, has at most one key window.

A window does not have to become the key window to receive, and act on, keyboard alternatives. It must, however, belong to the active application.

Since the key window belongs to the active application, its black title bar has the secondary effect of helping to show which application is currently active. The key window is the most prominently marked window in the active application, making it key in a second sense: it is the main focus of the user's attention on the screen.

Main Window

The *main window* is the standard window where the user is currently working. It is the ultimate focus of any user action carried out with a menu or panel. When a user opens the Find panel, for example, it becomes the key window because the user must type in criteria for the Find operation. But the panel is

just an instrument with which the user is working on a document displayed in another window. That other window, the one displaying the document, is known as the main window.

Whenever a standard window becomes the key window, it also becomes the main window. When the key window status moves from a standard window to a panel, the standard window retains main window status.

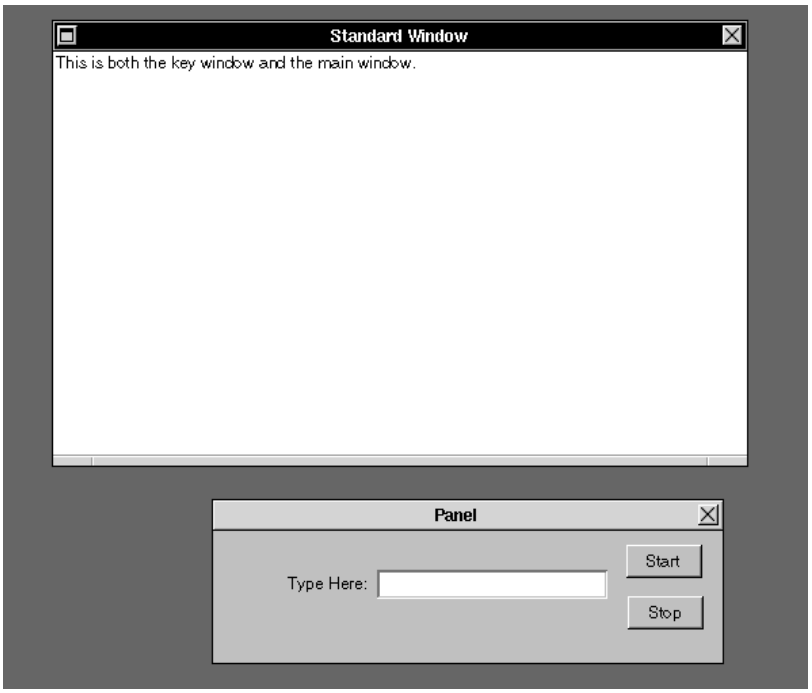


Figure 4-6 The Main Window Is the Key Window

To identify the main window in situations when the main and key windows are different, the Application Kit highlights the main window's title bar in dark gray. (When the main window is also the key window, it has only the black highlighting that indicates key window status.) Figure 4-6 illustrates highlighting the main window when it is also the key window. Figure 4-7 illustrates highlighting when the main window is *not* the key window.

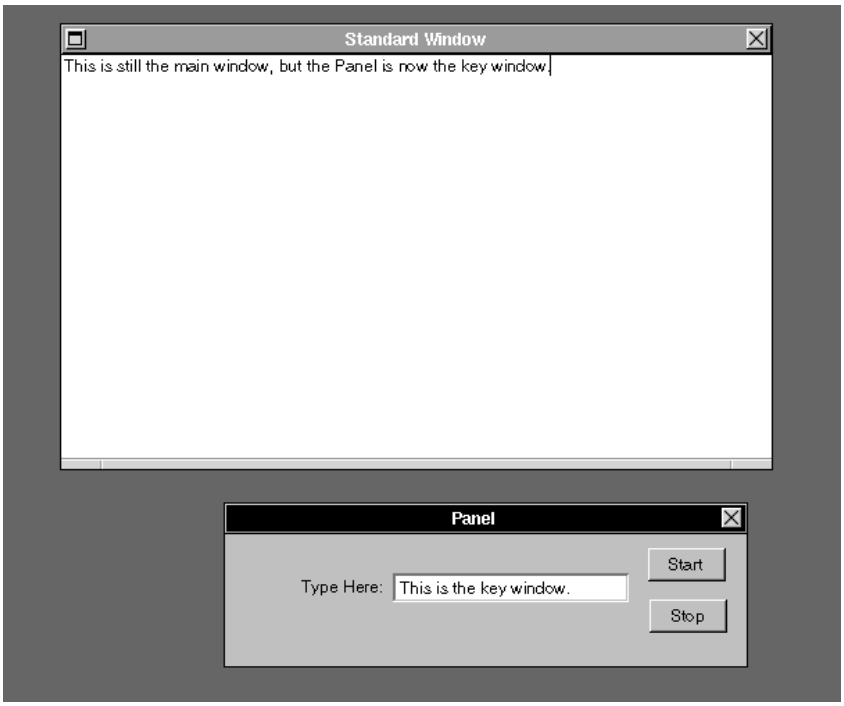


Figure 4-7 The Main Window Is Not the Key Window

A menu command can operate on either the key window or the main window, depending on the nature of the command. For example, the Paste command can be used to enter text in a Find panel when it is the key window. But the Save command will always save the document displayed in the main window, and the Bold command will make the current selection in the main window boldface. The rules for this behavior are:

- The action called for by a command is first directed to the key window. If the key window can handle the action, it does.
- If the key window is a panel and cannot handle the action, the action is next directed to the main window.

Note that this order of precedence is reflected in the way windows are highlighted: the key window is always highlighted, but the main window is highlighted only when it is not the key window.

The main window and key window are always in the active application. The main window follows the key window, as user actions shift the focus from window to window and from application to application.

How Windows Become the Key Window or Main Window

In most situations the user, rather than the application, selects the key window and main window. This section describes how different user actions select a main or key window and the part that the App Kit plays. Those few situations in which your application needs to choose its own key window are covered in "Choosing the Key Window" later in this chapter.

In the Currently Active Application

The user designates a new key window by clicking on any of the windows displayed by the active application. If the new key window is a standard window, it also gets main window status. If the new key window is a panel that accepts text entry, it gets key window status, but the original main window retains main window status and gets main window highlighting (a dark gray title bar). Notice that the user cannot select a main window without also making it the key window.

When the user closes or miniaturizes the window that has key window status, the App Kit chooses a new key window (or main window) for the active application. When the user closes the last open window, so that the application has no more windows on screen, no new key window can be chosen, although the application still remains active.

When an Application Is Activated

When an application is activated, one of its windows gets key window status and one (usually the same one) gets main window status. Whenever possible, status is assigned as a result of user action:

- If the user activates the application by clicking on a window that accepts keystrokes, that window gets key window status. If it is a standard window, it also gets main window status.
- If the user activates the application by double-clicking on a miniwindow, the miniaturized window is brought up on screen and gets both key window and main window status.

If the action that activates the application does not directly select a new key window, each window retains it previous status. If the user reactivates an application by double-clicking its icon, the windows that had key and main window status when the application was deactivated retain their status.

Note – When an application is activated, its key window may be highlighted before the newly deactivated application's key window loses its key window highlighting. This is an aspect of OpenStep's multitasking environment, in which users can begin working in one process (the new active application) before their instructions to another process (the previously active application) have been completed. Although the deactivated application's key window may retain its highlighting for a short time, it is not the key window—all keyboard actions are directed to the newly activated application.

Results of Clicking on a Window

Clicking on a window has two separate, but related, results:

- The window usually becomes the key window (and in most situations the main window), and the application that owns it is activated. Standard windows always get key window status when clicked, but panels might not. This is covered in detail in Chapter 5, "Panels."
- The window comes to the front of its level.

The first is a change in the window's status, the second in its position on screen. Both must happen before the user can work in the window. Moving it in front of other windows makes its contents available; making it the key window allows the user to type in it and give it menu commands.

In the OpenStep interface, these two results of a mouse click, while logically related, can take place separately. If the user Alt-clicks on the window's title bar, the actions will bring the window to the front without making it the key window or activating its application. Alt-clicking on title bars thus lets users rearrange and reorder windows on the screen without changing window or application status.

Implementing Windows

This section covers guidelines for designing and placing the different types of windows used in your application.

Designing Windows

The only windows that have a fixed size are miniwindows and icons. The initial sizes of all other windows are determined by the application developer. Standard windows are usually larger than panels, and panels are usually larger than menus, but there are no absolute rules.

When designing a panel or standard window, you should keep a substantial part of it free of objects that will respond to the user's first click. Make it easy for your users to find a place to click within the window and select it.

Try to limit the number of panels and standard windows your users need to bring up in order to use the application. Too many windows result in a cluttered screen that might confuse users. Even two windows will be excessive if users cannot tell which one they are supposed to work in. Finally, if your application clutters the screen, it will be difficult for your users to work in more than one application.

Placing Windows

One of the basic principles of the OpenStep user interface is that users control their own workspaces. One aspect of this control is the freedom to rearrange windows to suit their own tastes and needs. To maintain user window arrangements, window that are dismissed and brought up again must reappear in their previous locations.

To avoid making the user rearrange windows unnecessarily, each of your panels and nondocument standard windows should remember its own location when it is dismissed. The next time the window is brought up, it should reappear in the same location. Suppose, for example, the user brings up a Find panel, moves it to a new position, and then closes it. The next time the user brings up the panel, it should come up in its user-defined position, even when the user has quit and restarted the application.

Whether your document windows should also remember their position depends on the nature of the application. For example, Mail message windows do not remember their positions because users typically open many documents at once, and thus need the application's help in positioning the windows. However, an application such as a drawing program that is typically used for editing one file at a time should probably let the user determine each document window's default location.

The first time a window comes up, its position is determined by the application. To ensure a consistent user interface, all applications should follow these guidelines for initial locations of windows:

- When an application starts up, its main menu should appear in the upper left corner of the screen, unless the user has specified a different location for it.
- Standard windows should come up to the right of the main menu, allowing enough room for submenus that might later be attached to the main menu. Some applications also allow room for panels to come up to the left of the standard window and below the main menu.
- Attention panels should come up centered in the upper part of the screen, where they cannot be overlooked.
- No part of any window (other than miniwindows and icons) should be placed off screen unless the user has put it there.

Programmer's Note – There are three methods you can use to help panels and non-document standard windows remember their window position. Calling the `setFrameAutosaveName:` method once per window makes the window save its position in the defaults system whenever necessary. The next time the window comes up, it appears at the last-saved position. A less automated way of handling the window position is to call `saveFrameUsingName:` every time you wish to save the position, and call `setFrameUsingName:` to set the window's position when it is being brought up.

These methods are not appropriate for document windows, since there is no easy way to guarantee unique names for documents. If your application saves the positions of document windows, you should use the `saveFrameToString:` method to save a representation of the window's position into the document itself. When opening the document, you should position its window using `setFrameFromString:`.

Implementing Standard Windows

The standard window is the most commonly used type of window, and standard windows are used as the principal windows in every application. If an application lets the user edit files, each file should be displayed in a separate standard window. If the application is a game, the game board should be displayed in a standard window, and if the application is a simple accessory like a clock, the clock face should be displayed in a small standard window.

Every standard window has a title bar. Most also have the window controls—resize bars, close buttons, and miniaturize buttons. This section covers choosing titles for your windows and the user actions you need to implement when your windows have the window controls. It also describes the situations in which it is acceptable to omit the window controls.

Choosing a Title

If a window displays a document that can be saved, the title bar of the window should display the name of the document, an em dash (—) and the path for the folder that contains the document. Use two spaces on each side to set off the em dash. The following example shows the correct title for a window that displays a document named "jobRecords" which is in `/net/server/export/records`:

```
jobRecords  —  /net/server/export/records
```

The title bar is not a good place to show status (such as what the application is currently doing). Status is better displayed in the window's content area or in a panel that accompanies the window. When status is displayed in a window, small, dark gray text is often used (as it is in the Workspace Manager's File Viewer).

Programmer's Note – You should use the `setTitleWithRepresentedFilename:` method of the NSWindow class to set the title of a document window. To produce this window title:

```
jobRecords  —  /net/server/export/records
```

you should send the window a `setTitleWithRepresentedFilename:` message, and the argument to this method should be a pointer to an NSString containing the string `/net/server/export/records/jobRecords`.

Using the Resize Bar

Most standard windows—especially those with scrollable contents—should have a resize bar. The bar gives users control of their environments by letting them determine how much screen space is used for to the contents of the window.

Figure 4-8 Resize Bar

If a window has a resize bar, you should be careful that the window remains useful and legible, no matter how large or small the user makes it. OpenStep lets you constrain a window's dimensions so that it does not become too big or too small too be useful, and you can also restrict it so that it only grows and shrinks by fixed amounts. An example of this is the Workspace Manager File Viewer, which will only grow or shrink by the width of its browser columns, eliminating the possibility of showing only a partial column.

Using the Miniaturize Button

In general, each standard window should have a miniaturize button  . Exceptions to this rule occur when your application has an essential window—the application cannot be used without it. When a window is miniaturized, it should remain miniaturized until the user explicitly unminiaturizes it.

Because a miniaturized window is not likely to be the focus of user attention, applications should not alter the appearance or contents of a miniaturized window without the user's knowledge. They can, however, continue to do work that the user requested before miniaturizing the window. Terminal, for example, will complete any commands that the user entered in a Terminal window before miniaturizing it. Changes that were not initiated by the user before miniaturizing the window are unacceptable. Changing the font used in a miniaturized window, for example, is unacceptable unless the user specifies a font change for *all* windows.

The Windows menu has a command which is a counterpart to the miniaturize button and miniaturizes the key window. You can also add a command that miniaturizes several related windows into a single miniwindow to the Document menu. See “The Windows Menu” and “The Document Menu” in Chapter 6 for information on these commands.

Using the Close Button

Most standard windows have close buttons  . In some situations the close button is not needed. The Preferences application, for example, has only one standard window, and it cannot be used without that window; so the window has no close button. Compare this to the Edit application, which allows users to open and close multiple documents and also allows them to issue useful commands, such as File / New, when no windows are open.

Your application should break a window’s close button  whenever it contains unsaved work. From the user point of view, a broken close button means that the application will not let users lose work by accidentally closing the window without saving the work. If the user tries to close a window with a broken close button or quit the application that owns the window, the application should bring up a Close or Quit panel, which gives the user an opportunity to save the work. (See Chapter 5, “Panels,” for more information on these standard panels.)

Note – If an application uses multiple windows to display a single file and any one of them contains unsaved work, you should display a broken close button in all of them. The application should not bring up a Close panel, however, until the user closes the last window for the file.

The Mail application breaks the close button on a Send window as soon as a user types in the message area. If the user tries to close the window (either directly or by quitting the application), Mail puts up an attention panel that forces the user to either confirm the close or cancel it.

If an application does no work that can be saved, but merely shows data that might change, you can use the close button to indicate the status of the data. Break the close button to indicate that data in the window is not up to date. The application should also provide a way for the user to force the window to update. For an example of close buttons used in this way, look at the File Viewer.

Just as there is a command in the Windows menu (“miniaturize”) that corresponds to the miniaturize button, there is one that corresponds to the close button (“Close”). It has a keyboard alternative, Command-w (for “window”). See “The Windows Menu” in Chapter 6 for details.

Implementing Window and Application Status

Most aspects of window and application status are handled by the Application Kit. You must choose the initial key window and decide which windows are allowed to become the key window. (For information on when to make a panel the key window, see Chapter 5, “Panels.”) You need to make sure the application activates itself in the appropriate way, as discussed in the following paragraphs.

Choosing the Key Window

In general, every standard window in your application should be permitted to act as the key window, even if it does not respond to keyboard actions. Giving key window status to a window will focus attention on it and prevent the user from typing in any other window. When the key window does not accept typing, it should beep as it receives keystrokes, informing the user that typing is not appropriate.

When an application is activated on startup, it should designate one of its windows to be the initial key (and main) window. If the application opens a document file for the user, the window that displays the document should be the key window.

Activating an Application

Applications do not usually activate or deactivate themselves explicitly. When an application exchanges messages with the Workspace Manager, uses services, or provides services, activation and deactivation are handled by the system. For example, when the user chooses the Mail Selection service from Edit’s Services menu, the Edit application is deactivated. Mail is then activated, on the condition that no other application is currently active. Since Edit has been deactivated, this condition will be met unless the user activates another application in the meantime.

The only time an application needs to explicitly activate or deactivate itself is when it communicates with another application without using the services system or the Workspace Manager. This might happen when two applications work together closely by sending messages directly to each other. If the intent of a message is to activate the receiving application, then the sender of the message should deactivate itself just before sending the message, and the receiver should conditionally activate itself when it receives the message. If the intent of a message is not to activate the other application, then neither application should activate or deactivate itself. In general, a message should conditionally activate the receiving application whenever the user might work in it—even if it is only to operate a scroller.

Note – Applications should avoid activating themselves unconditionally. Unconditional activation violates the principle of user control, since it ignores the user's desire to turn to something else.

Programmer's Note – As described in this subsection, most applications do not need to explicitly activate or deactivate themselves. However, when necessary, an application can conditionally activate itself with the following code:

```
[NSApp activateIgnoringOtherApps:NO];
```

An application can deactivate itself as follows:

```
[NSApp deactivate];
```

Avoiding Activation When Dragging

When a user drags an object, such as a color or a file, between two applications, both the area that originally contained the object (the source) and the area to which the object is dragged (the destination) need to be visible. Sometimes the user needs to move the windows of one or both applications to make this possible. Once the user starts to drag the object, a change in window order may cover up the destination. This could happen when the source application is not active. When the user begins to drag, the source application is activated, bringing its windows forward and perhaps covering the destination.

To prevent this, there is an exception to the standard activation behavior in this situation. When a user drags an object from one application to another, the drag should not activate the source's application. It should activate, however, when the user clicks anywhere in the source's window or begins a drag anywhere in the window except the source area.

Programmer's Note – Avoiding activation when dragging objects is fairly simple to implement. First, each `NSView` that contains draggable objects should override `acceptsFirstMouse:` so that it returns YES. This enables the `NSView` to receive events whether or not its window is the key window. Next, the `NSView` should override the `shouldDelayWindowOrderingForEvent:` method so that it returns `YES` when the passed mouse-down event occurred over a draggable object. (The `shouldDelayWindowOrderingForEvent:` message is sent just before the `mouseDown:` message for the event.)

That is all you have to do if you use the dragging system. The dragging system automatically calls the `preventWindowOrdering` method (which prevents the window's application from being activated) if the object is dragged. Unless the `preventWindowOrdering` method is called, the window's application is activated, as usual.

Panels 5

Panels support the work your users do in the application's standard windows. There are two general classes of panels:

- Most panels let users give instructions to the application. Developer-defined controls on the panel allow users to construct logically complex commands or instructions for the application. The Font, Find, Page Layout, and Open panels are examples of this type.
- Other panels give information to the user. Help panels, the Info panel, and attention panels that display warnings are examples of this type.

For both types, the application's developer uses the panel to create a dialog between application and user that is highly structured in both form and content.

5

Note – The App Kit includes a number of standard panels and callable functions that create the standard attention panels. When standard panels don't have the features you need, you can enhance them or create custom panels. When you create custom panels, you are responsible for following the guidelines in this chapter, so that your users recognize them as ordinary panels or attention panels.

How Panels Work

The panel is a type of window, and its most basic characteristics are those shared by all types of window. These are covered in Chapter 4, "Windows in the OpenStep Interface." Only the unique characteristics of panels are covered in this section.

Ordinary Panels

Ordinary panels look and behave very much like standard windows. They are typically in the same level as standard windows and compete with them for screen space. They have title bars, but do not usually have miniaturize buttons. Ordinary panels differ from standard windows:

- A panel can never become its application's main window.
- A panel should not become the key window unless it accepts typing from the keyboard.
- Panels are generally removed from the screen when the applications that own them are deactivated.
- You can make a panel into a "floating panel" by assigning it to the sixth level, which puts it above standard windows. See "Floating Panels" in this chapter for more about this.)
- Attention Panels

When an application opens an attention panel, it creates a mode that prevents the user from working in any of the other windows of the active application. The mode limits the user to rearranging. Nothing else in the application—title bar buttons, text entry, miniwindows, or controls in other panels—will work. The only menu commands that will work are those that can operate on the

attention panel itself. Cut, Copy, and Paste, for example, will work when an attention panel includes an editable text field. The mode remains in effect until the user responds to the attention panel and explicitly closes it.

While an attention panel is open, the user can activate another application and work in it. When the original application is reactivated, however, the attention panel is still displayed and the mode it created is still in effect.

Attention panels differ from ordinary panels in the following ways:

- An attention panel has an empty title bar.
- An attention panel is closed by one or more buttons in the content area, and does not have close buttons in their title bars.
- An attention panel stays on screen—even when its application is not active—until closed by the user.
- An open attention panel has key window status while the application that owns it is active.
- An attention panel is in the second level, above everything on the screen except spring-loaded windows such as pop-up lists.
- An attention panel is opened just above the center of the screen, so that users cannot overlook it. (Users can move attention panels.)

Because an attention panel creates a mode that disables the other functions of its application, it must have a distinctive appearance that immediately tells the user a mode is in effect. Some of the graphical features that create this distinctive appearance are illustrated in Figure 5-1.

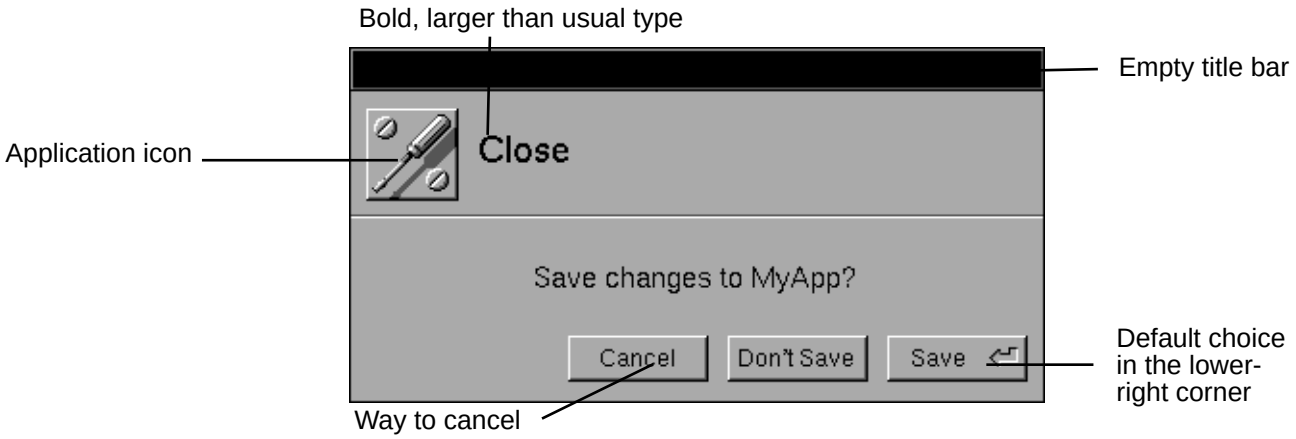


Figure 5-1 Attention Panel

Attention panels are closed when the user responds to the situation described by the panel. The buttons on the panel present the user with one or more possible responses. When the user chooses a button, the panel is closed and the mode ends.

Implementing Ordinary Panels

Because ordinary panels are used in many ways, you have considerable freedom when you design and implement them. This section covers guidelines for those aspects of ordinary panels that are different from standard windows.

Window Considerations

Title bar buttons and key-window status are handled differently than they are for standard windows. For more detailed information on the title bar buttons and on key windows, see “Implementing Windows” in Chapter 4.

Using the Resize Bar

If you decide that the ability to resize a panel might help your users, you should give it a resize bar. If necessary, you can set limits on the panel’s dimensions that prevent users from making it too large or too small.

Using the Miniaturize Button

You can put a miniaturize button on an ordinary panel, but it is rarely needed. Since your users can use menu commands to close a panel or reopen it, the miniaturize button is generally redundant. You can make an exception if shrinking a panel to a miniwindow would be more convenient for users, or if a panel persists on screen when its application is not active. See “Persisting Panels” later in this chapter for more information.

Using the Close Button

Every ordinary panel should have a close button so the user can dismiss it when it is not needed. You don’t need to break the close button, since panels do not display documents that users will be saving.

Panels That Become the Key Window

Panels that accept text entry and panels that are used independently of other windows should be allowed to get key window status. The Find panel, for example, becomes the key window because users can type in the word for which they are searching, and the standard Info panel becomes the key window because it displays information that is independent of the application’s other windows and becomes the focus of the user’s attention. On the other hand, the tool palette in a graphics application should never become the key window, because it is operated only by the mouse and is always used in conjunction with a document window.

Even if a panel accepts typing, you can delay giving it key window status until user action (such as clicking on a text field) indicates the user is ready to begin typing. Both of the following conditions must be true for you to implement this behavior:

- Text entry is not essential to using the panel.
- Users typically do not enter text when using the panel.

This is most often true for panels on which most of the control devices are not text fields, but are buttons, selection lists, and so on, and when there are alternatives to text entry. For example, the user can type an item or select it from a list. The Font panel is an example. This kind of panel should not show any selection until the user indicates a readiness to begin typing.

Note – Panels that need to avoid becoming the key window until the user indicates a readiness to begin typing can use the `setBecomeKeyOnlyIfNeeded:` method of the NSPanel class to do so. However, panels that should never become key—a tools palette, for example—must use a different way to avoid becoming the key window. Each panel must remove key-down and key-up events from its event mask.

Relinquishing Key-Window Status

After a panel becomes the key window, it should retain key window status only as long as necessary. If user actions on the panel affect the main window, key-window status should return to the main window when those actions are completed.

The Font panel, for example, gives up key-window status when the user clicks on the Set button on a Font panel and changes the font of the text currently selected in the main window. In all likelihood, the user is finished with the Font panel at this point and is ready to resume working in the main window. By managing key window status in this way, you allow the user to resume working in the main window immediately, without clicking on it just to make it the key window.

Exceptions to Ordinary Panel Characteristics

In general, ordinary panels are unobtrusive. They occupy the lowest level, and they disappear when their application is deactivated. In some situations that call for a more prominent panel, you can implement special features.

Persisting Panels

By default, an ordinary panel is removed from the screen when its application is deactivated. Users see only the panels that belong to the active application. This prevents confusion that could arise if similar panels from two different applications (two Find panels, for example) were on screen at the same time.

If one of your panels contains information that is relevant to the user's work in another application, you can override this default behavior and allow the panel to remain on screen after its application has been deactivated. This should be a rare occurrence.

An example is the Workspace Manager's Info panel, which contains system-level information, such as the amount of memory in the computer. Because the user might want to copy this information down—possibly into an e-mail message—this panel persists even when the user deactivates Workspace Manager and activates the mail application.

Floating Panels

By default, an ordinary panel is in the lowest level with standard windows. In some situations, however, it is useful for a panel to float above the standard windows and the other ordinary panels in the application. Typical of these situations is a small panel that contains a palette of drawing tools, which is most useful if it floats above the application's other windows. Figure 5-2 shows an example of a floating palette.

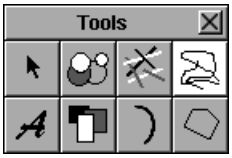


Figure 5-2 A Floating Panel

The following conditions should be true for you to implement a floating panel:

- The panel is oriented to the mouse rather than the keyboard. This means a panel you allow to become the key window should not be made a floating panel with one exception: It becomes the key window only when the user is ready to type (see "Becoming the Key Window" earlier in this chapter).
- It is important for the panel to remain visible while the user works in the application's standard windows. This is true when the user must frequently move between the standard window and the panel (as for a tool palette) or when the panel displays information that is relevant to the user's actions in the standard window (as in some inspector panels).
- It is small enough that it will not obscure much of what is behind it.
- It does not remain on screen when the application is deactivated.

Notice that panels float for some of the same reasons that menus do.

Panels With Variable Contents

Two types of ordinary panels—*multiform panels* and *inspector panels*—were designed for displaying specialized information in a limited amount of space. Both multiform and inspector panels can be used for many different purposes, even within the same application.

Multiform Panels

A multiform panel is a panel that can take a number of different forms. It has a pop-up list or a set of radio buttons at the top that which let the user choose which form it takes. Figure 5-3 shows a panel which lets the user choose among seven different forms.

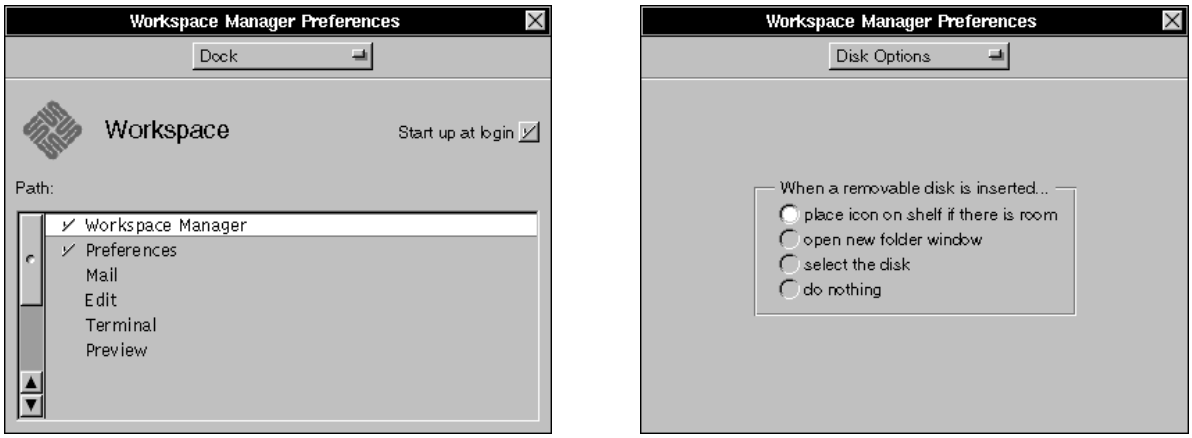


Figure 5-3 Two Forms of a Multiform Panel

Multiform panels conserve screen space by combining many related panels into a single panel. Since not all of a multiform panel's contents are visible at the same time, you should not use a multiform panel for an application's most basic functionality.

Inspector Panels

An inspector panel is a panel that displays information about the currently selected object. An inspector panel usually lets users set properties of the object, as well as view them. The Font panel (described in “Standard Panels” later in this chapter) is an inspector panel that displays font information for the current selection. Figure 5-4 shows the Workspace Manager Inspector panel, which lets users view and set information about the currently selected file or folder. Inspectors are often multiform panels, with each form displaying a different kind of information about the selected object.

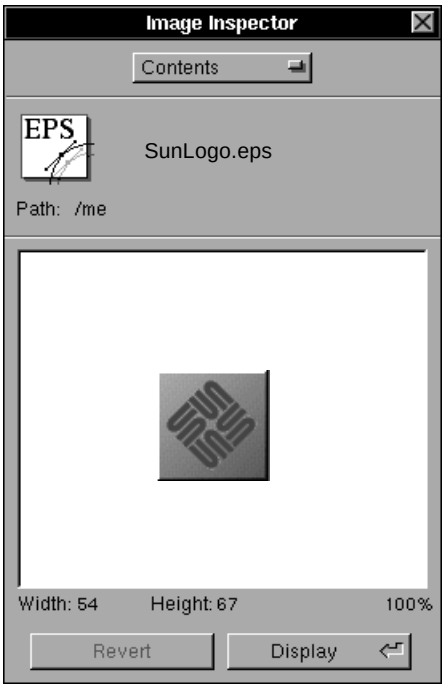


Figure 5-4 Inspector Panels

5

Implementing Attention Panels

Because attention panels create modes that severely limit your users' freedom of action, you should use them only when they are really necessary. The guidelines for deciding when a panel should be implemented as an attention panel are:

- The panel gives the user information about the current context. The panel usually identifies an error, warns of potentially dangerous or unexpected results from the current course of action, or informs the user of a condition that makes it impossible to carry out the requested action. They may also supply information the user needs to proceed intelligently.
- It interrupts an action and gives the user an opportunity to take corrective steps—as does, for example, the panel that interrupts the Quit command to let the user save altered files before the application terminates.
- It clarifies or completes a user action—as does, for example, the panel that completes the Save As and Save To menu commands. In this case, the menu command must have three dots after its name—for example, "Save As...". This is discussed under "Commands That Open Panels" in Chapter 6.

Attention panels that interrupt or complete an action must have a Cancel button, which gives the user the option of cancelling the action. If Cancel is chosen, the application should return to its previous condition, as though the user had never initiated the action. When a panel informs or warns, it should, if possible, let the user choose what to do in response to the information it displays.

Naming an Attention Panel

Attention panels that open as the result of a command or a command-like action should be named for the action that brings them to the screen. The panel that appears after the user chooses Save, Save As, Save To, or Save All should be named Save. The panel that comes up when the user wants to close an edited but unsaved document should be named Close, whether it is invoked from the close button or through the Close or Close Window commands. The panel's name appears just to the right of the application icon, as shown in Figure 5-5.

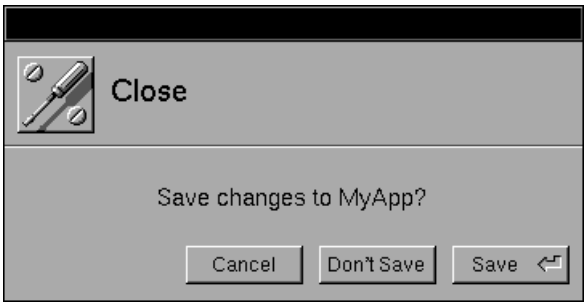


Figure 5-5 Name of an Attention Panel

Default Option in an Attention Panel

When a user action opens an attention panel that has more than one button, the button that allows the user to continue with the action should be the default button.

The default button should not perform any action that was not implied by the user request that brought up the panel. It should be the “safest” of the actions that are available to the user (for example, Open Copy rather than Open Anyway when the user tries to open a document that someone has already opened). It should not contradict what the user set out to do, which means you should not use Cancel as the default button.

The default button in an attention panel should be located in the lower right corner of the panel. In most cases, users should be able to operate it by pressing the Return key (when the panel is the key window), and you should mark it with the return symbol to indicate this. When a default button has dangerous side effects, it may be better to omit the return symbol and require that users click on the button.

Closing an Attention Panel

Each action that can close an attention panel is represented by a button. These buttons should be located along the right and lower edges of the panel, and the default button should be in the lower right corner. Only the default button should be operable by the Return key.

Naming the Buttons in an Attention Panel

Label each button clearly with a verb or verb phrase that describes its action. Users should be able to read the names of the buttons and choose the right one. They should not need to read other text on the panel. Avoid “generic” labels like Yes and No, because they are not clear and lead to user errors. Avoid using OK unless it is the only button in the attention panel.

Good names for attention panel buttons include:

Cancel
Close Anyway
Don’t Close
Don’t Save
Explain
Open
Open Copy
Open Anyway
Quit Anyway
Replace
Revert
Review Unsaved
Save
Save All
Set

Optional Explanations in an Attention Panel

An Explain button can offer users a way of getting more information before dismissing the panel. A typical example is shown in Figure 5-6.

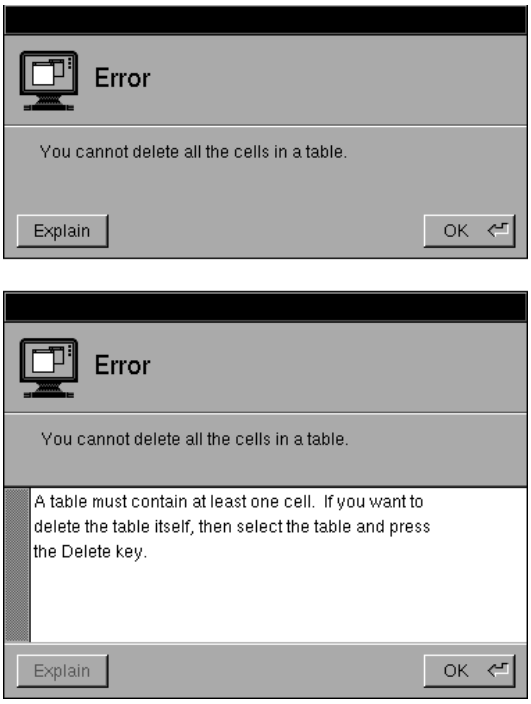


Figure 5-6 Explain Button on an Attention Panel

Standard Panels

Some panels are used in many applications. The Info panel, which gives certain kinds of general information about the application, appears in every application. The Font panel, which lets the user set the font of the text selection, appears in every text processing application.

This section describes all the OpenStep standard panels. Some of them are provided by the App Kit; the others you build yourself, following the guidelines in this section. If there is a standard panel for a function you are using in an application, you should use it instead of designing a new panel.

If an Application Kit panel does not provide all of the features you want for your application, you can customize it by adding controls and information to what the App Kit already supplies. Figure 5-7 shows the standard Open panel on the right and a customized Open panel on the left. A check box (named Use Simple Mode) has been added to the customized panel.

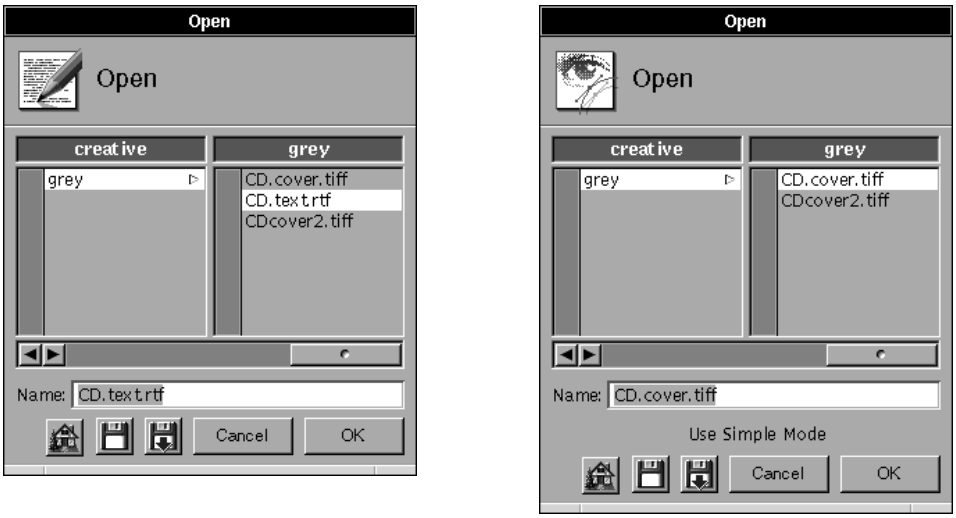


Figure 5-7 Standard and Customized Open Panels

Note – To customize a panel that is implemented by the Application Kit, construct an `NSView` that contains the information and controls you want to add. Add your `NSView` to the panel by sending it in a `setAccessoryView:` message.

Table 5-1 lists and describes the standard panels. The sections that follow the tables explain how to implement those panels that are not fully implemented by the App Kit.

Table 5-1 Standard Panels

Panel	Description
Close	An attention panel that opens when a user tries to close a document with unsaved changes. For more information on this panel, see “Implementing the Close Panel” in this chapter.
Colors	An ordinary panel provided by the App Kit. It lets a user preview and specify colors in any of the following modes: color wheel, grayscale, red-green-blue (RGB), cyan-magenta-yellow-black (CMYK), hue-saturation-brightness (HSB), custom palette (which loads an image from which the user can choose colors), and custom color lists. A Colors panel sometimes works with color wells. See “Color Wells” in Chapter 7 for information on color wells. For information on the command that opens the Colors panel, see “Font Menu” in Chapter 6.
Find	An ordinary panel that lets the user enter a string for an application to search for. For more information on this panel, see “Implementing the Find Panel” later in this chapter.
Font	An ordinary panel provided by the Application Kit. It lets the user preview fonts and change the font of the currently selected text. For more information on the interface to fonts see “Font Menu” in Chapter 6.
Help	An ordinary panel provided by the Application Kit. Use this panel to display on-line help text. It can display text, graphics, and link buttons (which lead to other information). For more information on creating help, see “Using the Help Panel” later in this chapter.
Info	An ordinary panel that displays information about the active application. For more information on this panel, see “Implementing the Info Panel” later in this chapter.

Table 5-1 Standard Panels *(Continued)*

Panel	Description
Link Inspector	An ordinary panel provided by the App Kit. It lets a user get and set attributes of the currently selected linked information. If your application's documents can receive linked information, it should have this panel. For more information, see "Using the Link Inspector Panel" later in this chapter and "The Link Menu" in Chapter 6.
Open	An attention panel provided by the App Kit. It lets the user designate a file to open. For more information, see "Using the Open Panel" later in this chapter.
Page Layout	An Application Kit panel that prompts the user for information needed to display documents on screen, or print them. If your application has extensive page layout features, you can replace this panel with one that has information and controls for those features. For more information, see "The Format Menu" in Chapter 6.
Preferences	An ordinary panel that allows the user to determine some details of the application appearance and behavior. For more information, see "Implementing the Preferences Panel" later in this chapter.
Print	An attention panel provided by the Application Kit. This panel opens whenever the user prints a document or other data. It prompts the user for information needed to print, and displays the following options: send the output to a printer, save the output to a PostScript file, send the output to a fax modem, preview on screen, and cancel any of the preceding actions, even after they have started. For more information, see "The Main Menu" in Chapter 6.
Quit	An attention panel that should open when the user tries to quit an application that has some unsaved or incomplete work. For more information, see "Implementing the Quit Panel" later in this chapter.
Save	An attention panel provided by the Application Kit. It prompts the user for the name under which the file should be saved. For more information, see "Using the Save Panel" later in this chapter and "The Document Menu" in Chapter 6.
Spelling	An ordinary panel provided by the Application Kit to help the user check the spelling of text. See "Checking Spelling" in Chapter 6 for more information.

Implementing the Close Panel

When the user closes a document that has been edited but not saved, your application must open a Close panel and let the user choose among canceling the close operation, saving the document before closing, or closing without saving. This panel should have at least three buttons:

Cancel Don't Save Save

Save is the default option, because many users do not think of closing a document and saving as separate operations—for many, closing implies saving.

If closing the document or window has consequences other than losing unsaved work, you must still use a Close panel to inform the user. When the user closes a terminal emulation window, for example, the emulator application should open a Close panel that tells the user closing will terminate the currently running command. In a situation like this, when there is no alternative to the "side effect" of the operation initiated by the user, the Close panel should have these buttons:

Cancel Close Anyway

Note – Do *not* open a Close panel unless there is work that will be lost.

Implementing the Find Panel

Figure 5-8 Find Panel

This panel, shown in Figure 5-8, should have at least two controls: a text field in which to enter a text string and a button that initiates a search for the next occurrence of the string. In most cases you should also have a button that will find the previous occurrence of the string.

Note – Although we recommended that you use verbs as button titles, it is not appropriate in this panel for two reasons. First, changing the titles to Find Previous and Find Next obscures the difference between the buttons—it is easy for users to miss the second word on a button. Second, the word "Find" is already used twice on the panel (and no other verb is), making it redundant in the button titles. For more information on naming buttons, see "Choosing the Button's Image or Label" in Chapter 7.

Find panels often have more controls than the panel in Figure 5-8. Typical additions are:

- Check box that lets users indicate whether capitalization matters in the search (by default, it should not)
- Check box that lets users indicate whether partial words should be considered matches (by default, they should)
- Second text field, named "Replace With" that allows the user to specify a second text string, which will replace the first when it is found
- Radio buttons that allow the user to indicate the range over which the search should be conducted (by default, the range should be global)
- Options that allow the user to specify objects other than text strings (such as paragraph characteristics) for the search

Although the Find panel usually stays open until the user explicitly closes it, there is one situation in which the application closes it for the user; see "When Mouse Operations and Keyboard Alternatives Differ" in Chapter 3.

This panel is opened by the Find Panel menu command, and the Find Next and Find Previous commands correspond to the buttons on the panel. For more information, see "The Find Menu" in Chapter 6.

Using the Help Panel

The App Kit's Help panel is part of the OpenStep help system. As supplied to you, the help system includes help text for basic interface tasks, such as using menus and scrollers, and skeletal help for menus and panels that are implemented in the App Kit. You write help text for application-specific objects and tasks, and link it to the basic text.

When users Help-click on a menu or panel implemented in the Application Kit, the supplied help text is displayed. For example, Help-clicking on the Print panel opens a Help panel with basic information on the panel's features.

You can add text for the objects in your application and the tasks associated with them. To do this, override the help files for Application Kit panels and commands with new files containing links to more task-oriented help. When you have added your own help text, you should modify the supplied index and table of contents so that they refer to your help text.

Programmer's Note – The Help panel is part of OpenStep's support for on-line help, which also includes:

- A question mark pointer which appears when the user presses the Help or F1 key (or on keyboards without a Help key, Alt-Control)
- Help text for basic user tasks (such as using menus and operating scrollers)
- An easy way to connect your interface objects to help text, so that Help-clicking on your objects will open Help panels with your help text

To add help text to your application, you first add a help folder using Project Builder. This help folder includes template files for the Help panel's index and table of contents. You can create and modify these and other help files, along with the help links in them, with the Edit application in developer mode. Both Interface Builder and the Application Kit have support for associating help with the user interface objects in your application.

For more information on implementing on-line help, see the discussion of `NSHelpPanel` in the *OpenStep Programming Reference*.

If your application uses the Help panel, you need to implement Help-clicking for each of its objects. (Otherwise, when the user Help-clicks on an object, the Help panel will open but will probably show inappropriate help.) Help-clicking on an object results in the panel automatically displaying the most specific help available. For example, when the user Help-clicks on a button, the help system first looks to see whether the button has help associated with it; if so, the Help panel displays it. If not, the help system looks for more general help, such as that associated with the window or even the application that the button is in.

Writing Note – You are encouraged to reuse the help text from OpenStep applications such as Edit, Mail, and Workspace Manager. After copying text from one of these applications' Help panels, you can paste it into a file in your application's help folder. You should then modify the text and update the help links it contains so that it suits your application. For example, Edit has help on working with documents, text, graphics, and color, as well as on printing. With a few changes, much of this help will be appropriate for other document-based applications.

You are also encouraged to use the design of OpenStep's help. Help in OpenStep applications is task-oriented, with Help-clicking serving as a way of discovering the tasks an object is used for. At the end of every help file is a list of related tasks, with links to the help for each task. The objects that have associated help are generally menu commands, standard windows, panels, and a few important buttons such as those in Mail's Compose window. If supplying this level of help is not practical, then you should at least ensure that Help-clicking on any object results in help on a relevant subject, even if the help is very general.

Implementing the Info Panel

Each application must have an Info panel (an example is shown in Figure 5-9), which displays a small amount of basic information about the application:

- Name of the application
- Application icon
- Copyright information
- Current version of the application
- Names of the authors (optional)

Figure 5-9 Info Panel

The Info panel should not offer help for using the application or give more information about it (such as its history or purpose).

Interface Builder includes a sample Info panel that you can use in your application. For information on the command that opens the Info panel, see "The Info Panel" in Chapter 6.

Using the Link Inspector Panel

When you use this App Kit panel, you should provide the following keyboard alternative for its Open Source button: hold down the Control key and double-click on a linked item. This should perform the same action as selecting an item and choosing the Open Source button.

Using the Open Panel

Whenever this panel is opened, it should list the contents of the folder with which the user was most recently working. To determine which folder this is, examine the window that is currently (or was most recently) the application's main window. The document displayed in this window will be the document most recently edited; the Open panel should list the contents of the folder that contains this document. If the application has not had a main window since it was started, the panel should show the user's home folder.

If your application can open only certain types of files, the Open panel should list only files of those types when it opens.

Implementing the Preferences Panel

Most applications have a Preferences panel, which is an ordinary panel that allows the user to determine some aspects of how the application looks and behaves. Figure 5-10 shows an Edit Preferences panel. Users typically use this panel infrequently. They should not need to open it during normal work with the application.

Figure 5-10 Edit Preferences Panel

Preferences panels typically include controls for such things as:

- Default font size
- Format for displaying data
- Whether the application should make backup files
- Default size of the application's windows
- Options that make expert features available
- Application's keyboard alternatives

Do *not* use the Preferences panel for appearance or features the user might want to reset from time to time during a session.

The contents of the Preferences panel should be valid throughout the application; the appearance of the panel should not change depending on the window or data currently selected. The preferences should carry over from session to session; most will also affect the way the application works during the current session.

All of an application's options must be settable from within the application. It is not acceptable to have options that are settable only by using the `dwrite` command.

Preferences panels are often implemented as multiform panels, which reduces their dimensions and lets you organize the options. The Preferences panel is opened by the Preferences command in the Info menu. For more information, see Chapter 6, "Menus."

Implementing the Quit Panel

Whenever the user tries to quit an application that has unsaved or uncompleted work, the application should open a Quit panel. This should happen, for example, when the user has edited a document and not saved the changes, or when the application is still computing (a command is executing in a UNIX shell). Notice that in these situations, one or more of the application's windows should have a broken close button. For more information on breaking the close button, see "Using the Close Button" in chapter 4.

The Quit panel is an attention panel. In most cases it has three buttons, arranged as follows:

Cancel Quit Anyway Review Unsaved

You could also add a Save All button that saves every unsaved document, exactly as the Save All command does. The buttons function as follows:

- *Cancel* cancels the Quit command and allows the user to take some alternative action.

Note – When your application opens a Quit panel because of a logout or a poweroff, the panel should not have a Cancel button, because the application cannot cancel the logout or poweroff.

- *Quit Anyway* continues the Quit operation, regardless of the consequences.
- *Review Unsaved* is the default button. If the user chooses this button, the application should cycle through its unsaved documents, opening a Review Unsaved panel for each one. The Review Unsaved panel functions like the Close panel, allowing the user to decide whether the document open for review should be saved, but its name reflects the different manner in which it is invoked.

  The user can cycle through all the unsaved documents, using the Review Unsaved panels to indicate which should be save. The user can also click on the Review Unsaved panel's Cancel button, which terminates the review process and returns the user to the Quit panel.

  You should only include a Review Unsaved button when the application is document-oriented.

Note – Do *not* open a Quit panel unless work is about to be lost.

Using the Save Panel

This Application Kit panel should come up showing the folder of the document being saved (the document in the main window). A document's folder is reflected in its window's title bar. For example, if the title bar shows

```
UNTITLED-1  —  /net/server/export/documents
```

the Save panel should come up showing `/net/server/export/documents`.

Menus 6

Menus reveal your application's features to its users. Users often browse the menus to see what features are offered and explore those that seem useful.

Because of this, you should use the menu system's hierarchical nature and group your application's commands into a logical and meaningful set of menus. A well-designed menu hierarchy will help users understand the structure of an application and find the commands they need.

How Menus Work

This section describes the basic appearance and behavior of OpenStep menus, which are built into the Application Kit. It covers the main menu, submenus, menu commands and keyboard alternatives. It also describes the user actions that operate menus.

The Application Kit supplies the basic menu functions covered in this section. In particular it supplies:

- The ability to respond to menu commands. (The chosen command is highlighted, and the action you define for the command, such as opening a submenu or a panel is performed.)

- Abilities of menus to open, close, hide, and return; of the main menu to change when the user changes the active application; of submenus to be torn off, and so on.
- Ability of a menu command to open a submenu.
- Ability of commands to handle keyboard alternatives.

Menus you create when you work in Interface Builder have these abilities. To add an application-specific command to a menu, for example, you drag a menu command from the Palettes window to the menu. You can then name the command, give it a keyboard alternative if necessary, and associate it with an action. Figure 6-1 shows typical menus.

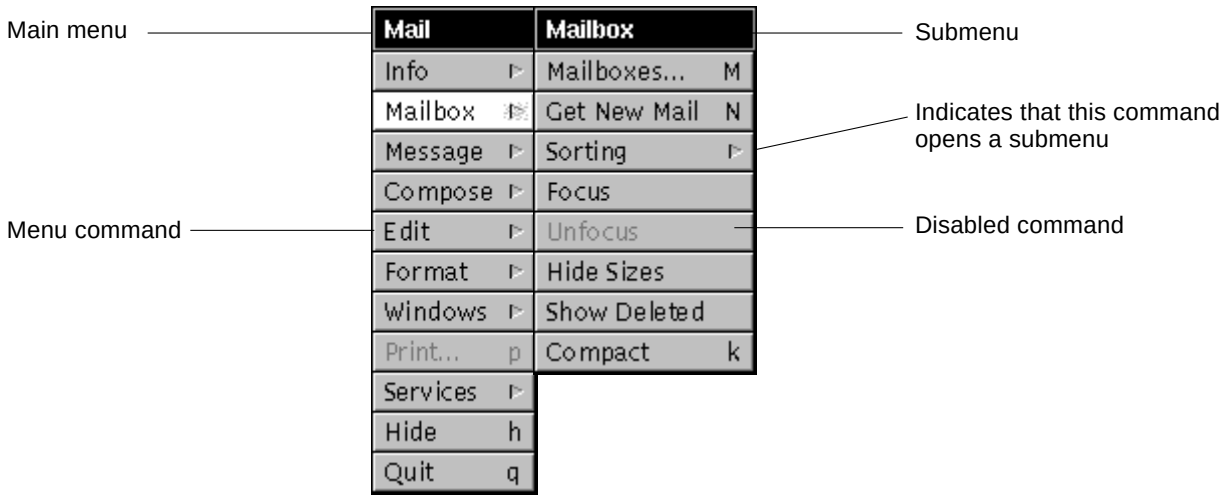


Figure 6-1 How Menus Work

Basic Menu Functions

The main purpose of a menu system is to display commands from which users can choose. To make this a relatively simple process for the developer and ensure easy use, the OpenStep interface implemented the following guidelines for overall menu functions in the App Kit:

- Menus are arranged hierarchically. A command can open a submenu that has its own commands.
- Menus are assigned to the third and fourth levels, which keeps them floating above everything else on the screen except spring-loaded windows (such as pop-up lists) and attention panels.
- Only menus for the active application are displayed. When the user moves to another application and activates it, current menus disappear and menus for the newly activated application take their places. The App Kit remembers which menus the user has opened and displays them whenever the application is reactivated. For more information, see “Window Order” and “Miniaturizing” and “The Active Application” in Chapter 4.
- Menus cannot be miniaturized. They do not need to be, since they are small (do not cover much of the screen) and are easily reopened after they have been closed.

The basic user operation is choosing a menu command with the mouse. Simply clicking on a command will choose it; users can also drag the pointer through the content area of a menu and release the mouse button when the pointer is over the desired command. (Dragging will highlight each command that the pointer passes over; a command is chosen when the button is released.)

The rest of this section covers these points in more detail.

Main Menu

Every OpenStep application has a main menu, which is displayed on the screen when the application is active. It contains standard commands used in every application and application-specific commands that you supply.

Programmer's Note – Project Builder creates a main menu with some standard commands for your application. You can add commands and submenus in Interface Builder. Note that some of the standard commands available in Interface Builder have actions or submenus already associated with them. If you add the Windows command to your main menu, you get the Windows submenu.

In the smallest applications, the main menu may be the only menu. Most applications need more than one menu to hold all of their commands. In OpenStep, this is handled by submenus—commands on the main menu identify general functional areas, and when chosen they open submenus with more specific commands. The hierarchical relationships between the main menu and its submenus give users access to all of the application's menus through the main menu.

The default location for main menu is the upper-left corner of the screen. Users can change the default location by dragging it to a new location. They can also change the main menu's location for all of their applications (except those whose main menus have already been moved) with the Preferences application.

The main menu does not have a close button; it is displayed whenever the application is active. When the user activates another application its main menu replaces the original main menu. When the user reactivates the original application, its main menu reappears.

Bringing the Main Menu to the Pointer

When the user presses the mouse button that is mappped to the MENU function, OpenStep opens a copy of the main menu at the pointer location. See Figure 6-2. Notice that the pointer must be over a window that belongs to the active application, rather than the screen background or an inactive window.

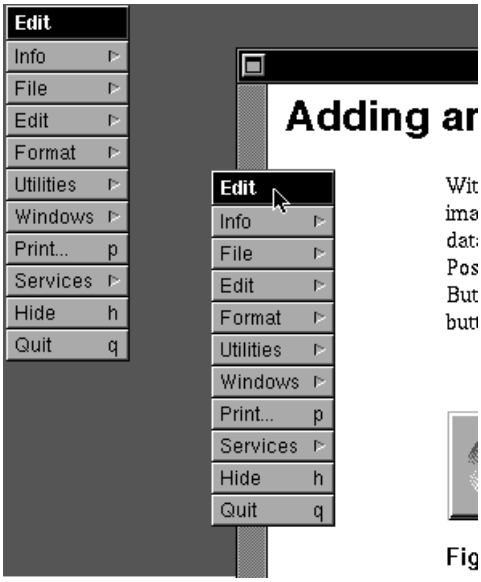


Figure 6-2 Bringing the Main Menu to the Pointer

As long as the mouse button is down, the user can drag into this copy of the main menu, open its submenus, and choose one of the available commands. When the mouse button is released, this copy of the main menu and any submenus the user opened will disappear. Submenus that may have been opened from the permanent copy of the main menu are not affected.

Users can use the Preferences application to assign the MENU function to either the left or right mouse button. See “Changing the Functions of the Mouse Button” in Chapter 3 for more information on the MENU function.

Submenus

The main menu is created by Project Builder. All of the menus you create are submenus. They may open directly from the main menu, or they may open from one of its submenus (making them “sub-submenus”). Users open a submenu by choosing a *controlling command* on a menu that is already open.

Users can drag from a controlling command into a submenu and move the pointer to one of the submenu's commands. As long as the mouse button is held down, the submenu remains open and the controlling command remains highlighted. When the user releases the mouse button over a command, it is executed and the submenu disappears.

The OpenStep menu system gives users two options for working with submenus. They can be *attached* to their parent menu or *torn off*.

Keeping a Submenu Attached

The easiest way for users to open a submenu is to click on the parent menu command that opens it. They can also drag to the parent menu command, and release the mouse button. Either action will open the submenu and keep it attached to its super menu. The parent menu command will remain highlighted, indicating that its submenu is attached.

When attached, a parent menu and an attached submenu behave like a single window. User actions that move or close the parent menu will also move or close the submenu. An attached submenu has no close button of its own. A submenu attached to the main menu is assigned to the same window level as the main menu.

An attached submenu can also have its own attached submenu. This is illustrated in Figure 6-3. The Mail menu is attached to the Services menu, which is attached to the main menu. Moving the main menu moves all three.

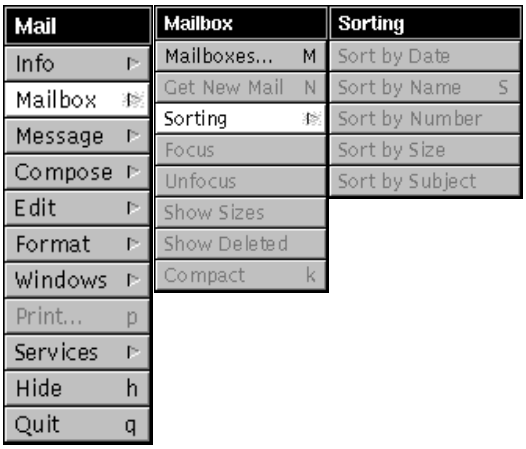


Figure 6-3 Attached Submenus

Tearing Off an Attached Submenu

After opening a submenu, the user can tear it off and drag it to another location. See Figure 6-4. The torn-off menu can be moved close to the location where the user is working. The user can also close the parent menu and work only with the submenu. When a menu is torn off, it gets its own close button. Any submenus that were attached to the submenu when it was torn off remain attached and move with it.

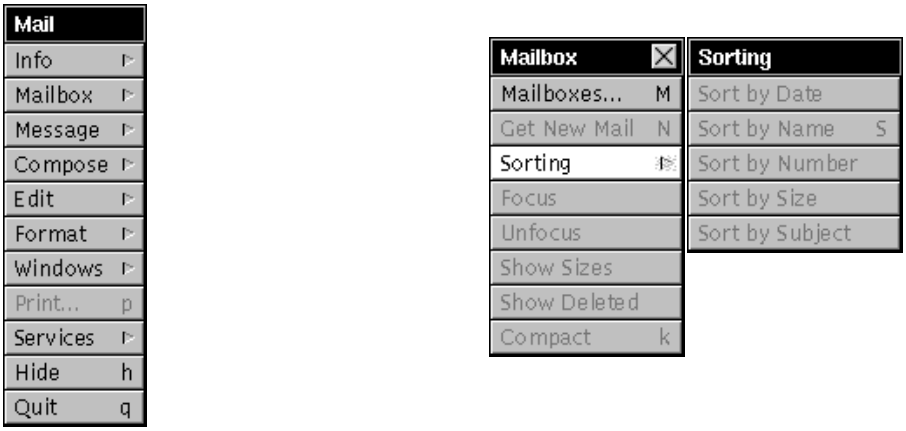


Figure 6-4 Detached Submenu

Once a submenu has been torn away from its parent menu, it stays where the user puts it. To reattach it, the user must close it and then reopen it from the parent menu.

If the user presses the mouse button while the pointer is over the controlling command that opened the torn-off submenu, a second copy of the submenu temporarily appears in the attached position (next to its parent menu). Figure 6-5 shows this condition.

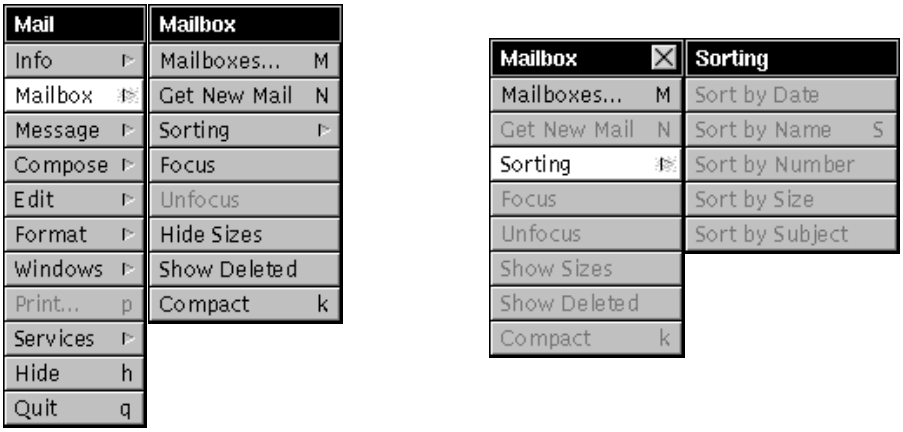


Figure 6-5 Temporarily Attached Submenu

Closing a Submenu

An attached submenu can be removed from the screen in three ways:

- By choosing its controlling command again. In Figure 6-4, clicking on the Mail command (on the detached Services menu) will make the Mail submenu disappear.
- By choosing any other command in the parent menu.
- By removing its parent menu from the screen. For example, when a torn-off parent menu is closed, its attached submenu disappears from the screen.

A torn-off submenu is removed from the screen by clicking on its close button.

Commands

Menu commands use the targeted-action paradigm. Some commands—Cut, Copy, Paste, and Miniaturize Window, for example—require the user to select the target. Others—such as Hide, Quit, and Info—do not require selection, because the target is assumed by the action.

When a command is chosen (by either a mouse action or a keyboard alternative), it is highlighted. When the user uses a keyboard alternative to choose a command in an off-screen menu, some visual feedback is necessary.

OpenStep provides it by highlighting the menu's controlling command or, if the parent menu is not visible, the parent menu's controlling command. This indicates that the keyboard alternative has in fact invoked the command.

Many commands control submenus. The actions of these commands are simply to attach the submenu to the menu. These commands are identified by the submenu symbol  .

Other commands cause panels or standard windows to appear on screen:

- Some open a standard window—the New command in the Document menu, for example, or the Console command in the Workspace Manager Tools menu.
- Some put an attention panel on screen to help clarify or complete the command. For example, the Save As command produces a panel that asks the user to type in the file name the user wants to use for the document.
- Others open a panel that can stand on its own, independent of the command that produced it. Sometimes the panel simply imparts information to the user—a Help panel, for example. But usually it acts as a control panel where the user can give instructions to the application—the Font and Find panels, for example. Such panels are similar to submenus in that they open a range of options to the user.

The highlighting behavior of a command depends on its action. If it controls a submenu, it remains highlighted while the submenu is attached. If it controls an attention panel, it remains highlighted until the panel is closed. If it opens a panel that is not an attention panel, it does not remain highlighted.

You can disable a command, as described in "Disabling Commands" later in this chapter. Disabled commands have dark gray text (instead of the usual black) on the usual light gray background. They are completely inoperative and are not highlighted in response to user actions.

Keyboard Alternatives

Users can often use keyboard alternatives, instead of the mouse, to choose commands. A keyboard alternative is a combination of a character and the Command key. For example, holding down the Command key and pressing the "p" key is the standard keyboard alternative for the Print command. Keyboard alternatives are discussed in detail in "Other Action Commands" in Chapter 3.

6

Implementing Menus

The App Kit supplies the format for a basic appearance and functions for OpenStep menus. This section covers what you do to create a menu system for your application:

- Designing the menu hierarchy
- Choosing command names
- Disabling commands

Designing the Menu Hierarchy

When designing your application's menu hierarchy you begin with a main menu that was created by Project Builder. As you add features to your application, and design menus that will make those features available to your users, keep the following guidelines in mind:

- Use as many of the standard menus (described later in this chapter) as possible. Using the standard menus is one of the easiest ways to ensure consistency between applications, and it will speed up the development process
- Keep your application's menu hierarchy as shallow as possible. In general, a menu should be no more than two steps away from the main menu. It is even better when you can keep it one step away, but don't let it become too long (too many items) or confusing just to avoid the second level. Figure 6-6 shows a hierarchy that has too many levels.

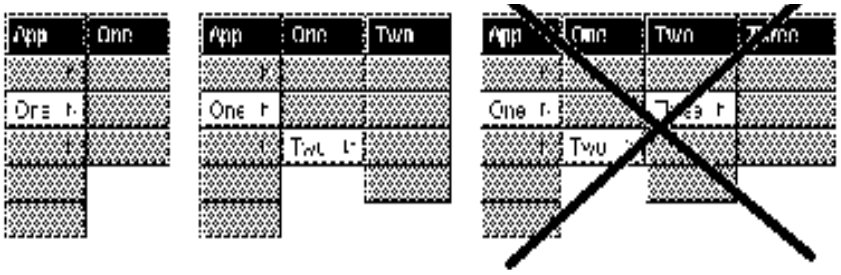


Figure 6-6 Levels of Submenus

- A menu should never have fewer than two commands, unless you add and remove commands dynamically and the menu has fewer than two commands in some situations. If an application has a menu with only one command in it, move the command up one level and replace the controlling command that opens that menu. A specific example of this is discussed minify Menu" later in this chapter.
- A menu can have as many submenus as it has commands, although only one at a time can be attached to the menu. A menu should appear only once in the menu hierarchy—it should not be the submenu of two menus.

Choosing Command Names

Names for commands should be short. Limit them to a single word whenever you can, and use very short phrases when you cannot. Avoid abbreviations in command names, especially abbreviations that are not standardized or widely used. Apply the same capitalization rules to all applications in the same language. (For English, commands are capitalized as they would be in a title—the first and last words begin with uppercase letters, as well as major words in between.)

Command names should be unique. No two commands, even when they are in different menus, should have the same name.

Commands That Perform Actions

If a command performs an action, its name should begin with a verb, giving a command name that reads like a short imperative sentence (addressed to the application). Examples of this include Hide, Open, Save As, and Revert to Saved.

When a menu command is used to toggle a state or condition, you should change its name rather than using two menu commands. The clearest way to inform the user of situation like this is to change the verb in the command's name. Do not, for example, create a menu with both Show Ruler and Hide Ruler commands. (Note that one of the commands would always be disabled.) Some examples of good names are listed in Table 6-1:

Table 6-1 Menu Command Names

First State	Second State	Notes
Show Ruler	Hide Ruler	
Show Grid	Hide Grid	
Use Grid	Ignore Grid	*Do not* use Grid On and Grid Off.
Bold	Unbold	*Bold* is treated like a verb in this command.

Commands That Open Panels

If the main purpose of a command is to perform an action, and it opens a panel to help the user through the action, name the command for the action rather than the panel. Follow the naming guidelines for action panels (see "Commands That Perform Actions" in the previous subsection). The standard Save, Save As, and Save To commands, for example, are action commands that happen to open a panel (named the Save panel). Notice that the panel is given a name that reflects the command's name.

If the main purpose of the command is to open the panel, then name the command for the panel. This usually results in a name that is a noun phrase, instead of the verb phrase used for action commands. The Preferences command, for example, opens the Preferences panel, and the Spelling command opens the Spelling panel.

With one exception, a command that opens a panel must have three dots after its name (for example, "Preferences..."). The exception is for panels that are warning panels, such as the panel that comes up when the user tries to revert to a saved version of a document. Because the user would complete the action if the warning panel did not open, you don't want to imply that the command opens a panel where the user does work. In addition to this, new users often examine an application's panels by choosing the menu commands with "...". You do not want these users to choose commands that open warning panels. For example, the Workspace Manager Log Out command does not have three dots, but it always opens a warning panel.

Do *not* put three dots after commands that open standard windows (like the New Viewer command in the Workspace Manager, or the New command in the standard Document menu).

Note – Use three periods (not the ellipsis character) to produce the three dots.

A command that opens a panel should not usually have the word “panel” in its name, since the three dots indicate that it opens a panel. It is acceptable to use “Panel” in a command name that would otherwise be identical to the name of an existing command name. For example, when the command that opens the Info panel is in the Info menu, the command is named Info Panel. When an application has no Info menu, the command is named just Info.

Commands That Open Submenus

Commands that open submenus usually begin with nouns, but verbs or adjectives are acceptable when they are clearer. It is important that these commands have names that clearly identify what the submenu contains. For example, it is a bad idea to have both a Tools menu and a Utilities menu under the main menu, since most users will not be able to remember the differences between the two.

If a command opens a submenu of actions, it might be appropriate to have the command name include the target of the actions. This scheme is used in the Document menu: the Open command can be read as Open Document, the New command can be read as New Document, and so on.

Commands That Open Standard Windows

Commands that open standard windows should begin with “New,” as in the standard New command that appears in the Document menu, or match the title of the window they open. The Workspace Manager’s View menu, for example, has a New Viewer command that opens a Viewer.

Sample Command Names

Some representative names for commands are listed in Table 6-2:

Table 6-2 Sample Menu Command Names

Command Name	What the Command Does
Cut	Performs an action
Font	Attaches the Font menu
Font Panel...	Opens the Font Panel
Hide	Performs an action
Info Panel...	Opens the Info panel (used only when Info is already used)
New	Opens a new document in a standard window
Preferences...	Opens the Preferences menu
Save As...	An action command that happens to open a panel
Select All	Performs an action
Show Graphics	Switches to Hide Graphics when graphics are already visible

Disabling Commands

When an application is in use, it is not uncommon for commands to have no valid function. For example, when a text editor does not have any documents open, its Save and Close commands have no function. In situations like this, a menu command that has no current function should either be disabled or it should open a panel explaining the function is not currently available. The text editor mentioned previously, for example, should have its Save and Close commands disabled, as shown in Figure 6-7.

Figure 6-7 Document Editor Menu

When users chooses a disabled command with a keyboard alternative, the computer should beep. This lets the user know that the command is not valid, even if the command is not visible.

If an invalid command opens an explanatory panel, it should explain why the command is inappropriate and offer assistance. The panel must provide more information than just that the command will not work, since that information can more directly be conveyed by disabling the command.

Graphical Devices in Menu Commands

The area to the right of a command can be used for two things: the keyboard alternative character or the submenu symbol. (Commands that control submenus cannot have keyboard alternatives.) Nothing else is permitted in this area.

In addition, menu commands should not use arbitrary graphical devices, such as check marks, to show state. There is almost always a more appropriate way to display current state in the OpenStep interface—for example, by using buttons or check boxes in a panel or by designing objects that can be directly manipulated (such as those in the Edit application's ruler).

Assigning Keyboard Alternatives

"Other Action Keys" in Chapter 3 lists standard and recommended keyboard alternatives.

Standard Menus and Commands

This section describes the standard menus supplied by the Application Kit, Using as many of them as possible will speed application development and help you achieve consistency between applications. It also covers the standard arrangements for commands in each menu.

Menu and command names are shown in U.S. English. If you are working in another language, there will be a different set of standard names (for example, *Quit* is *Quitter* in French applications) but you should use them.

Programmer's Note – The standard menu commands and much of their functions are supplied by the App Kit, Project Builder, and Interface Builder. The App Kit even manages the names of some supplied commands (such as changing Bold to Unbold) and disables some commands, such as Heavier when they are invalid. However, you should double-check that each menu command works properly.

Main Menu

Every application should have its main menu laid out as described in this section. Figure 6-8 shows a typical main menu layout.

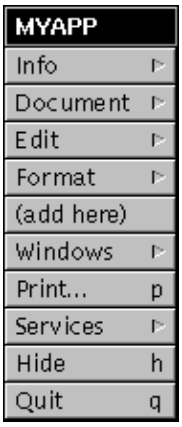


Figure 6-8 Main Menu

The main menu's title should be the application's name, shortened when necessary. For example, the main menu of Interface Builder is titled IB because "Interface Builder" would waste screen space.

Table 6-3 describes Main Menu commands.

Table 6-3 Main Menu Commands

Command	Action
Info	Attaches the Info menu, which contains commands that give general information about the application, as well as let the user set general preferences about how the application works. Info is the first command in the main menu in part because it can be read in conjunction with the application name in the title bar (for example, Info about Edit, Info about Draw, and so on). See “Info Menu” later in this chapter.
Document	Attaches the Document menu, which has commands that affect a document as a whole—opening, saving, and closing, for example. This menu is named differently in different applications, so it is important that the command be in a prominent, well-defined location (second). See “Document Menu” later in this chapter.
Edit	Attaches the Edit menu, which contains commands affecting the current selection. Every application that can have editable documents or selectable text must have this menu. See “Edit Menu” in this chapter.
Format	Attaches the Format menu, which contains commands affecting the layout of documents, including the font and paragraph format of text and the arrangement of graphic images. See “Format Menu” later in this chapter.
Windows	Attaches the Windows menu, which contains commands affecting the windows that belong to the application. See “Windows Menu” in this chapter.
Print...	Opens the Print panel, which permits the user to print a document. You can omit the Print command if your application does not print. In general, the Print command is assumed to print the document in the main window. If a panel can be printed (for example, one that contains a registration form), then to avoid confusion the panel might contain its own Print button.

Table 6-3 Main Menu Commands *(Continued)*

Command	Action
Services	Attaches the Services menu. This menu lets the user choose services provided by the system or by other applications. See "Services Menu" later in this chapter.
Hide	Hides all the windows of the application. See "Hiding and Retrieving Windows" in Chapter 4.
Quit	Terminates the application. If quitting the application might cause the user to lose work, then the application should open a Quit panel. Otherwise, the application should not require confirmation of a Quit command.

The Info, Services, Hide, and Quit commands should be in the main menu of every application. The other commands described above should be included when appropriate.

Adding Commands to the Main Menu

The main menu works best when it is short (so that commands are easy to find) and narrow (so that it does not take up much screen space). Try to limit your main menu to no more than 11 or 12 commands.

The main menu is also, for the most part, a menu of menus. The commands you add should generally be commands that open submenus.

When designing your application's user interface, you can move a command that the guidelines place in a submenu up one level to the main menu, provided that:

- The main menu is short enough to accommodate another command.
- The command provides functionality that is considered central, even crucial, to the application. For example, a text editor might bring the Font command up to the main menu from the Format menu, but a spreadsheet would not.

Like any other command that is added to the main menu, a command that is raised from a submenu should generally control another submenu.

When a command is promoted to the main menu, you should, for continuity, put it immediately after the command that controls the submenu it was in. For example, if the Font command is promoted from the Format menu, it follows the Format command. If the Find command is promoted from the Edit menu, it follows the Edit command, as shown in Figure 6-9.

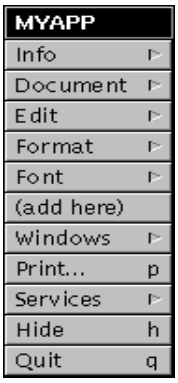


Figure 6-9 Adding the Font Command to the Main Menu

Info Menu

The Info menu contains commands that let the user view and set information about the application as a whole. See Figure 6-10. A License command is an example of the kind of command you could add to this menu. Table 6-4 describes Info Menu commands.

Figure 6-10 Info Menu

Table 6-4 Info Menu Commands

Command	Action
Info Panel...	Opens a panel that displays a small amount of basic information about the application. This standard panel is described in “Implementing the Info Panel” in Chapter 5.
Show Menus	Displays and tiles all the application’s menus. There is currently no support for this command in the App Kit. You can implement it or not as you see fit.
Preferences...	Opens the application’s Preferences panel, which permits the user to customize the application. This standard panel is described in “Implementing the Preferences Panel” in Chapter 5.
Help...	Opens a panel with helpful information on how to use the application. If you implement this command, you should use the standard Help panel, which is described in “Using the Help Panel in Chapter 5. If you do not implement this command, then when the user Help-clicks on an object in your application, the system opens a panel informing the user that the application does not use the OpenStep help system.

If your application does not support any of the commands in the Info menu except Info Panel, you can omit the menu and make the Info command open the panel instead of opening the menu. If you do this, add the three dots to the command.

Document Menu

This menu contains commands that affect a document as a whole. (Commands that affect selected parts of a document are generally in the Edit menu.) Applications that do not open or save documents of some kind and do not have a New command may omit this menu. Figure 6-11 shows a typical Document Menu, and Table 6-5 describes menu commands.

Figure 6-11 Document Menu

The command for of this menu (the second command in the main menu) should indicate the type of thing that the Open command opens and the Save command saves. It might be Document, Project, File, Model (for spreadsheets), or Shell (for a terminal emulator). Never call this menu Window, since most applications include the standard Windows menu.

Table 6-5 Document Menu Commands

Command	Action
Open...	Opens the Open panel so the user can open a file. Opening a file also opens a window (or windows) that displays it.
New	Opens a new, unnamed file and a window to display it in. This new document should be in the same folder as would be displayed in the Open panel. (See the section “Using the Open Panel” in Chapter 5.)
Save	Saves any changes in the document displayed in the current main window to a file. If the document has never been saved to disk, this command should have the same effect as the Save As command.
Save As...	Saves the document displayed in the main window, as changed, by writing it to a new file with a name supplied by the user. The document displayed in the main window corresponds to the new file, and the window’s title is changed accordingly. This command places a Save panel on screen that asks the user to type in a file name or cancel the command.
Save To...	Saves the current version of the document displayed in the main window, by writing it to a new file with a name supplied by the user. Save To is similar to the Save As command, but Save To does not replace the window’s current file with the new one. You can choose whether to implement Save As or Save To, or both in your application.
Save All	Saves every document that is open in the application. This is a shortcut for performing the Save command on every open document.
Revert to Saved	Replaces the current version of the document displayed in the main window with the version saved on disk. This undoes any changes made to the document since it was last saved.
Close	Closes the document in the main window, and all the windows used to display that document. It is completely parallel to the Open command. See “Windows Menu” later in this chapter for information on a related command, Close Window.

If your application uses more than one window to display a document, you can add a Miniaturize command that miniaturizes all of the windows associated with the currently selected document (the document of the key window) into a single miniwindow. There is a standard command in the Windows menu (Miniaturize Window) that miniaturizes a single window.

Performing an Implicit New Command

If the user starts up an application by double-clicking an application icon rather than a document icon, the application should, if appropriate, provide the user with a new document to work in (performing an implicit New command). This is much friendlier to a new user than simply putting a menu on screen. Users should be permitted to disable this action through a preference.

It is almost always appropriate for general-purpose applications to perform an implicit New command. However, it is not appropriate if the application cannot produce a new document without user input. It is also not appropriate if producing a new document has side effects, such as modifying the file system by creating a new folder or adding a file that might persist even if the user decided not to save the new document.

When an application is started up automatically at login or from another application, it should not perform an implicit New command.

If the user opens another document without touching the new one that was provided at startup, the application could automatically close the new one. But this is not a requirement of the user interface.

Uneditable Documents

If a document is opened that the application will not allow the user to save (even with a Save As command), it should not permit the user to edit the document on screen. Waiting until the user is ready to save changes is too late.

Edit Menu

The Edit menu contains commands that alter the selection in the current key window (or in the main window if the key window does not respond to the command). Commands should be dimmed when they cannot operate on the current selection. See Figure 6-12.

Figure 6-12 Edit Menu

Table 6-6 describes commands contained in the Edit Menu shown above.

Table 6-6 Edit Menu Commands

Command	Action
Cut	Deletes the current selection and copies it to the pasteboard.
Copy	Copies the current selection to the pasteboard without deleting it.
Paste	Replaces the current selection with the contents of the pasteboard.
Paste As	Attaches a submenu that permits the user to paste the current contents of the pasteboard into the document in a specified data type. The submenu lists the possible data types, as discussed in “Paste As” in the next subsection.

Table 6-6 Edit Menu Commands *(Continued)*

Command	Action
Link	Attaches the Link menu, which contains commands for manipulating linked information. See “Link Menu” later in this chapter.
Delete	Deletes the current selection without copying it to the pasteboard, thus leaving the contents of the pasteboard intact. The Delete key has the same effect.
Undo	Undoes the last editing change. This usually includes all changes since the user last made a selection, including the selection of an insertion point.
Find	Attaches the Find menu, which contains commands related to the Find panel. See “Find Menu” later in this Chapter.
Spelling...	Opens the Spelling panel.
Check Spelling	Finds the next misspelled word without bringing up the Spelling panel.
Select All	Makes the entire contents of the file the current selection.

Applications that permit the user to edit text or graphics should support at least the Cut, Copy, Paste, and Select All commands. It is strongly recommended that you also implement the Undo command.

Programmer’s Note – To includes this menu, drag it from the Palettes window of Interface Builder. It is already associated with the text object, which provides all the menu’s functions except Find, Paste As, Link, and Undo.

Paste As Menu

The Paste As menu, shown in Figure 6-13, is rarely needed because applications can take care of pasteboard data types without user intervention. However, sometimes it is useful for the user to be able to specify the format in which data is pasted. For example, the user of a page layout program might want to choose whether text is pasted as ASCII or Rich Text Format (RTF), or whether graphics are pasted as EPS or TIFF.

Figure 6-13 Paste As Menu

This menu should include only the types appropriate for its application. As usual, you should disable any invalid menu commands. For example, when the pasteboard contains only text data, any graphics formats should be disabled.

Checking Spelling

The Spelling and Check Spelling commands in the Edit menu are intended to provide a uniform user interface for the spell check function that parallels the interface for Find. (Find is discussed in “Find Menu” later in this chapter.) These commands for checking spelling are supported by the App Kit’s NSText object; custom objects will need custom code to check spelling.

The Spelling command opens the Spelling panel, which is described in Chapter 5, “Panels.” The Check Spelling command is equivalent to the button on the panel that searches for and selects the next misspelled word in the main window; it permits the user to find the next misspelled word without bringing up the panel. If the application cannot find the next misspelled word until the user takes some action within the Spelling panel (for example, loads a dictionary), Check Spelling opens the panel. (This function is parallel to Save, which opens a panel when the user must first supply a file name, or Find Next opening the Find panel if the user needs to enter a text string to search for.)

If an application has spelling options that cannot be accommodated in a panel, Spelling and Check Spelling should be replaced by a Spelling command that opens a submenu. That menu could have Spelling Panel and Check Spelling commands.

Link Menu

The Link menu provides a standard interface for the functions of receiving and supplying *linked information*. See Figure 6-14. Linked information is information, such as a graphic image, that has been copied from another file or application, but will be automatically updated when the original is modified. See the *Using the OpenStep Desktop* for more information on working with links. Table 6-7 lists Link Menu commands and their actions.

Figure 6-14 Link Menu

Programmer's Note – The App Kit supports links with its `NSDataLinkManager` and `NSDataLink` classes. See the *OpenStep Programming Reference* for detailed information on how to make your application receive and supply links.

Table 6-7 Link Menu Commands

Command	Action
Paste and Link	If the last Copy operation was performed in an application that can supply linked information, this command pastes the contents of the pasteboard and links it to the original information.
Paste Link Button	If the last Copy operation was performed in an application that can supply links, this command pastes a button that, when clicked, opens the document that contains the original information. Link buttons are discussed in "Implementing Buttons" in Chapter 7.
Publish Selection	Creates a link file. When this link file is dragged into documents that can receive linked information, the end result is as if a Paste and Link command had been done. This command places a panel on screen that asks the user to type in a file name or cancel the command.
Show Links	Highlights or unhighlights all linked information in the current document. The name of this command must alternate between Show Links and Hide Links, depending on the state of the document.
Link Inspector	Opens the Link Inspector panel, which is discussed in "Using the Link Inspector Panel" later in this chapter.

An application that can receive linked information (one that implements one or both of the Paste and Link and Paste Link Button commands) should also implement Show Links and Link Inspector.

When users choose the Show Links command, the application should highlight all the linked information in the main window, as shown in Figure 6-15. In the window shown in the illustration, the picture at the lower left is the only visible linked information, so it is the only picture that is highlighted with a chain pattern.

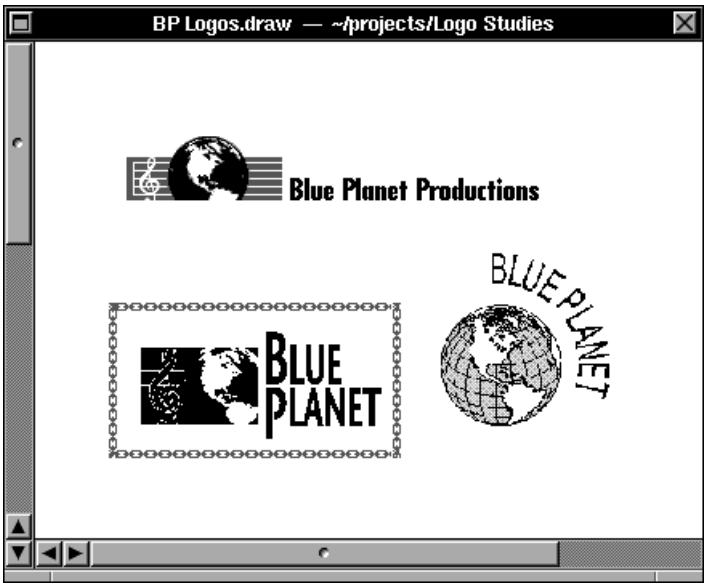


Figure 6-15 Linked Information

Find Menu

Applications that display large amounts of text are encouraged to include a Find menu like the one illustrated in Figure 6-16. Other applications might also find this menu useful, but because it is designed most specifically for text, a variation of it might better meet their needs. Table 6-8 describes Find Menu commands.

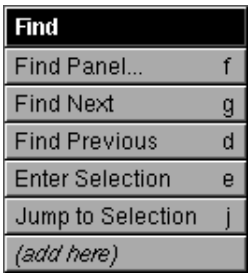


Figure 6-16 Find Menu

Table 6-8 Find Menu Commands

Command	Action
Find Panel...	Opens the Find panel, makes it the key window, and selects everything in the text field labeled Find so that the user can easily enter new text. If the panel is already on screen, the command brings it to the front, makes it the key window, and selects the Find field.
Find Next	Searches forward for the next occurrence of the string in the panel's Find field.
Find Previous	Searches backward for the previous occurrence of the string in the panel's Find field.
Enter Selection	Enters the current selection into the panel's Find field so that Find Next and Find Previous can search for it.
Jump to Selection	Scrolls to display the beginning of the current selection.

Find Next and Find Previous begin searching at the current selection. If the search is successful, the text that is found is selected and becomes the starting point for the subsequent search. Neither command requires the Find panel to be on screen. However, if the panel's Find field is empty, Find Next and Find Previous both open the Find panel, make it the key window, and select its Find field. This is exactly what the Find Panel command does. These other commands do it as a convenience to the user, who has indicated an intention to do a search.

The Find panel is further described in "Implementing the Find Panel" in Chapter 5.

Format Menu

The Format menu, shown in Figure 6-17, should hold the principal formatting commands available to users of your application. For applications that deal mainly in numbers, they may be commands that format the text display of floating-point numbers or the graphical display of numeric data. For text processors, they may include the commands that would otherwise go into the Text menu, plus others. Table 6-9 describes commands in the Format Menu.

Figure 6-17 Format Menu

Table 6-9 Format Menu Commands

Command	Action
Font	Opens the Font menu, which has commands that alter the font of the current selection. (See “Font Menu” on page 6-32)
Text	Attaches the Text menu, which lets the user choose the format of the selected blocks of text. (See “Text Menu” on page 6-35.)
Page Layout...	Opens the Page Layout panel, which lets users determine how documents are to be printed and displayed on the screen.

If you promote the Font command to the main menu and have no other commands to add to the Format menu, the Format menu would become little more than a container for the Text menu. In this circumstance, the commands that would otherwise go in the Text menu should be placed directly in the Format menu. A separate Text menu is needed only when there is reason to isolate these commands from other formatting commands, or to shorten what would otherwise be an excessively long Format menu.

When commands from the Text menu are placed in the Format menu, they should follow the Page Layout command, so that the Copy Ruler and Paste Ruler commands end the menu.

Font Menu

Applications that support text entry and editing should provide a Font menu (shown in Figure 6-18) and Font panel. The Font panel is described in "The Standard Panels" in Chapter 5. It contains controls that let users preview and set font attributes. The Font menu has a command to open the panel, and commands to make common adjustments to a font.

Figure 6-18 Font Menu

Each command alters one aspect of the font, such as its size or style, while leaving other aspects intact. The Font menu and Font panel target currently selected text. The Preferences panel should be used to alter the default font. Table 6-10 describes Font Menu commands.

Note – When you drag this menu in from Interface Builder's Palettes window, you get the menu and its functions from the NSText object with little additional work. Make sure that every command works, dims, and changes it name as it should.

Table 6-10 Font Menu Commands

Command	Action
Font Panel...	Opens the Font panel.
Bold	Makes the current selection bold, if it is not bold already, and makes it unbold if it is. The name of the command must alternate between Bold and Unbold depending on the selection.
Italic	Makes the current selection italic or oblique, if it is not already, and makes it unitalic if it is. The name of the command must alternate between Italic and Unitalic depending on the selection.
Underline	Underlines the current selection, if it is not already underlined, and removes the underlining if it is. When the current selection is already underlined, the command name must change to Ununderline.
Larger	Makes the current selection one point larger.
Smaller	Makes the current selection one point smaller.
Heavier	Uses a heavier typeface to display the current selection.
Lighter	Uses a lighter typeface to display the current selection.
Superscript	Moves the currently selected text up an appropriate amount for a superscript. Choosing the command again moves the text that much higher.
Subscript	Moves the currently selected text down an appropriate amount for a subscript. Choosing the command again moves the text lower by the same amount.
Unscript	Returns the selected superscripted or subscripted text to the normal baseline of the text.
Copy Font	Copies from the current selection all the text attributes listed in this menu, including font family, font size, bold, italic, underlining, superscript, and subscript.
Paste Font	Alters the current selection so that it has all the font attributes previously copied with the Copy Font command.

Note – When the current selection is an insertion point, commands in the Font menu affect the characters that are typed next, rather than any existing text (unless the text area can contain only one font). If an area of text can have only one font, then selections on the Font menu and Font panel change the font of all text in the area.

The only commands that must be included in the Font menu are Font Panel, Copy Font, and Paste Font, although frequently used commands like Bold and Italic should be present in most cases. Applications that are not text intensive may decide to omit less frequently used commands, such as Heavier and Lighter.

Each command leaves other font attributes intact. For example, Bold will change 11-point Times Roman to 11-point Times Bold and 24-point Courier Oblique to 24-point Courier Bold Oblique.

If there is more than one font in the selection, the Larger and Smaller commands make each one point larger or smaller than its current size. The other commands make only the change that is appropriate for the first character in the selection. For example, if the first character in a multifont selection is italic, the Unitalic command will remove the italic trait from all the text in the selection, but will not change any text that is not italic. If the first character is not italic, the same command (now called Italic) will italicize the entire selection, but it will not alter any text that is already italic.

The Colors command, which opens the Colors panel, often appears in the Font menu. You can put it somewhere else; each application should place the Colors command in a menu that indicates the kind of objects the Colors panel can affect. In Mail and Edit, colors can be applied only to characters (and not to their background), so the Colors command is in the Font menu.

Text Menu

The Text menu, shown in Figure 6-19, contains a set of formatting commands that affect text. These commands are supported by the Application Kit's NSText object.They can be isolated in a Text submenu, as shown here or included directly in the Format menu. If you have other formatting commands that are more important to your application, those commands, rather than those listed in Table 6-11, should go in the Format menu.

Figure 6-19 Text Menu

Table 6-11 Text Menu Commands

Command	Action
Align Left	Aligns the text at the left margin, leaving a ragged right margin.
Center	Centers the text between the left and right margins.
Align Right	Aligns the text at the right margin, leaving a ragged left margin.
Justify	Aligns the text at both the left and right margins.

Table 6-11 Text Menu Commands *(Continued)*

Command	Action
Show Ruler	Displays a ruler in the text area, if the ruler is not currently visible. Otherwise, this command hides the ruler. The name must alternate between Show Ruler and Hide Ruler, depending on the state of the text area. The ruler is a scale containing controls that affect the format of a paragraph (such as margins and tabs).
Copy Ruler	Copies the ruler settings in the first paragraph of the selected text.
Paste Ruler	Alters the paragraphs containing the text selection to have the settings most recently copied with the Copy Ruler command.

An application, such as a word processor, which includes many other text-related commands, can put all of them in the Format menu, in the order that is most useful. All applications should, however, use the command names shown above in the Text menu, no matter which menu the commands are in. (See "Format Menu" in this chapter.)

Windows Menu

The Windows menu (see Figure 6-20) contains commands that affect the windows belonging to the application. You may replace this menu with one more suitable for your application. If the application that has multiple windows per document, for example, you might add commands that organize those windows. Table 6-12 describes Windows Menu commands.

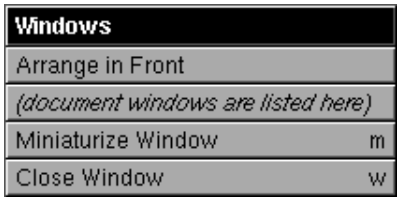


Figure 6-20 Windows Menu

Programmer's Note – This menu, with all of its functionality, is provided for you. Drag the Windows command into your main menu from the Palettes window of Interface Builder.

Table 6-12 Window Menu Commands

Command	Action
Arrange in Front	Stacks and offsets all the application's document windows (those that can become the main window and are created using Open and New commands) at the front of the screen. While this command is recommended, it is not mandatory.
Miniaturize Window	Miniaturizes the key window if it has a miniaturize button. The affected window need not be a document window.
Close Window	Closes the key window if it has a close button. If the window is the last one (or only one) open displaying a document, it also closes the document, just as the Close command would. (See "Document Window" earlier in this chapter, for a description of the Close command.)

The commands in this menu bring windows to the front of the screen or remove them. Other kinds of commands, even if they affect windows in some way, should be located elsewhere in the menu hierarchy.

You can replace Arrange in Front with an Arrange command that opens a panel or menu giving the user more choices concerning which windows to arrange and how they should be tiled or stacked.

The commands inserted below Arrange in Front list current document windows. Each command brings one window to the front and, if possible, makes it the key window. The Application Kit creates this list and dynamically adds a command for each new document window that it is opened. Because the Application Kit creates the command names from the title bars of the document window, improperly titled windows can lead to the Windows menu becoming too wide. Window titles are discussed in "Choosing a Title in Chapter 4. Figure 6-21 is an example of dynamically created commands for document windows.

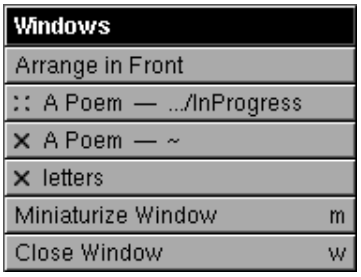


Figure 6-21 Windows Menu Listing Document Windows

Services Menu

This menu is required; it contains commands that invoke services provided by the system or by other applications. Refer to Figure 6-22. Commands are placed in the menu dynamically by the Application Kit when it becomes aware that other applications are service providers. This means there is no need for you to specify the order in which commands are added.

Figure 6-22 Services Menu

Programmer's Note – To take advantage of services provided by other applications, you should first drag the Services menu from the Palettes window of Interface Builder. Then, if you use the NSText object in your application, your application automatically gets many services.
Your application can get more kinds of services if you write a few lines of code.

Providing Services

Programmer's Note – To provide services to other applications, your application advertises its services in a section of its executable file or in its file package. (File packages are special folders that look and behave like files.) These services are automatically included in the menus of any applications that can accept them.

If you application provides services, you can specify how menu commands that request services should be worded. The following guidelines apply:

- Each command should begin with a verb and should name the application that will respond to the request. If the name of the application can be interpreted as a verb, it can be the first word of a command phrase (for example, Chart Selection for an application named Chart). Otherwise, the verb that begins the command should be followed by a preposition and the name of the responding application (Open from Workspace).
- If an application responds to more than one service request, it can arrange the commands in a submenu under the application's name. Commands in the submenu do not have to name the application, but should, like all other commands, begin with a verb.

However, if the application name is commonly interpreted as a verb, the submenu commands can consist of words that would meaningfully follow the application name in a phrase, much as the commands in the Paste As menu. For example, Figure 6-23 shows the Sort command:

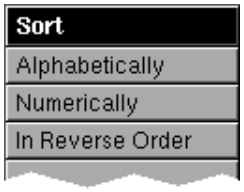


Figure 6-23 Names of Services

Service requests conditionally activate the other application, but only if user input might be required. For example, Digital Webster is likely to require the user to scroll the display, but the Workspace Manager does not need the user's help to get a file opened. The application that opens the file will become active, but the Workspace Manager will not.

Adding a Tools Menu

Since it is easiest for users to find a command when it is in a submenu with related commands, the commands that open panels and special non-document windows should be located throughout the menu hierarchy in the appropriate submenu. For example, the Font Panel command is in the Font menu, the Open command is in the Document menu, and Page Layout is in the Format menu.

In some cases, a window or panel is an independent tool with a functional domain all its own, and you may find it difficult to group the command that controls this window with your other commands. Some common examples of this are the palettes in a graphics program, the Inspector in Interface Builder, and the Console in the Workspace Manager. If your application has two or more such commands, consider collecting them together in a Tools menu. Figure 6-24 shows a Tools menu from the Workspace Manager.

Figure 6-24 Tools Menu

The Tools menu should not be considered a default location for commands that open windows or panels. If a window or panel is not truly a tool, its command should go elsewhere. If you can put a command into a submenu by function, you should do so.

Controls

7

Controls represent actions your users can initiate or states they can set. Each type of control has a slightly different “feel,” the result of the specific appearance and behavior built into it. Each action or settable state you add to your application should be represented by the control that will feel most natural to the users.

How Controls Work

This section explains some basic characteristics exhibited by all types of controls, lists the standard controls, and introduces the concept of custom controls. It is followed by sections that cover in detail the appearance and actions of standard controls.

Basic Control Functions

When a user acts on a control with the mouse or keyboard, two things happen:

- The control's appearance changes, to let the user know it is responding. Each type of control responds to specific mouse and keyboard actions and changes its appearance in specific ways. These reactions are part of its definition.
- The control sends some instructions to the application. The instructions associated with a control are designed and implemented by the application developer. They are part of an *application's* definition.

You can compare these two characteristics of graphical controls with control devices in the real world. A mass-produced switch, for example, can be installed on a variety of different machines. The manufacturer of the switch provides it with a user interface, which invites specific user actions, but product engineers give those user actions different meanings in different machines.

Standard Controls

As you develop an application, you design the actions that you want it to perform and identify the states that users will be able to set. Then you need to decide which control will best represent these actions to your users.

The App Kit supplies a number of standard controls, and each has an appearance and behavior option. For most actions or states you implement, you should be able to find a standard control that works well. The standard controls are covered in the following order:

- Buttons
- Text fields
- Sliders
- Color wells
- Scrollers
- Browsers and selection lists
- Menu commands (See Chapter 6, "Menus.")

If you cannot find the appearance and action you need in one of the standard controls, you can create a custom control.

Custom Controls

The App Kit makes it relatively easy to design your own controls, but you should follow these guidelines:

- A control must provide immediate feedback to let the user know that it is responding and initiating an action. Just as users can look at a dial on a stove to see whether it has been turned, a graphical control must change its appearance in response to user actions. Don't assume that users will notice a reaction elsewhere in the application as user feedback.
- A custom control should have a distinctive appearance and behavior. Do not design controls that look so much like standard controls that users will confuse them.
- The behavior of a control should be apparent from its appearance. Users who are familiar with OpenStep should be able to recognize new types of control objects and know, almost instinctively, how to operate them.

Buttons

Buttons are the most common controls. They are used in title bars (the miniaturize and close buttons), on attention panels (for Cancel and the other options that close the panel), and in most other situations that call for a control.

You can create buttons in a variety of different shapes and sizes. Figure 7-1 shows the standard types.

How Buttons Work

There are some differences in function among the buttons; this makes different buttons suitable for different situations. The most significant difference is between buttons that act and buttons that set a state:

- *Action* (or *one-state*) *buttons* tell the application to perform a task, such as scrolling a document forward or beginning a search.

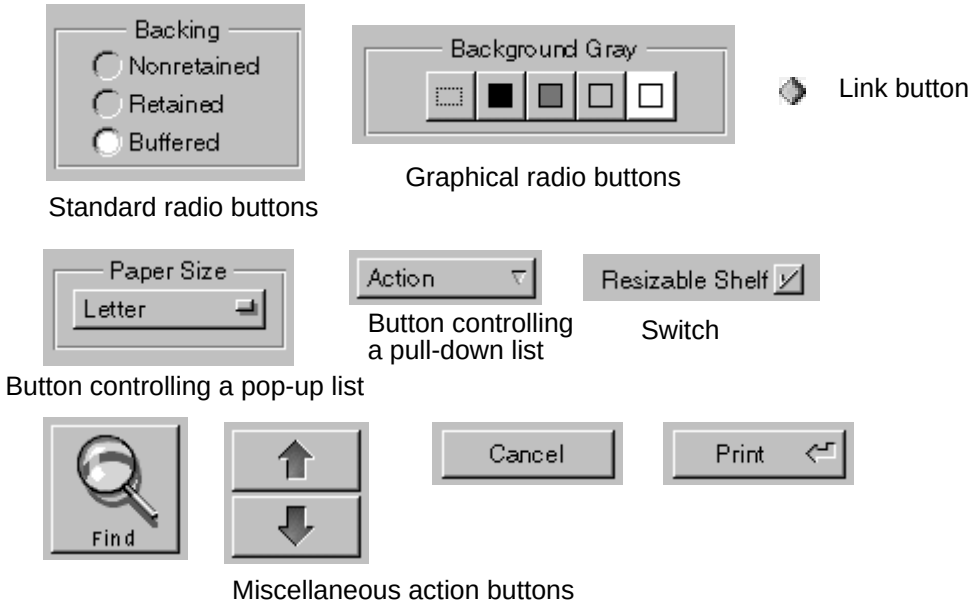


Figure 7-1 Types of Buttons

- *Two-state buttons* set a state on or off. A typical example is the button that lets users indicate whether the text search should find only whole words. Standard two-state buttons include switches and radio buttons. This category also includes pop-up lists.

Another difference is the specific user action that will sends a button's instruction to the application:

- Most buttons respond to a complete click and send their instructions to the application when the user releases the mouse button (assuming the pointer is still over the button when the mouse button is released). If the user presses over the button but moves the pointer away before releasing the mouse button, the instruction will not be sent.
- Other buttons respond when pressed. They send their instructions to the application as soon as the user pushes the mouse button down. This type of button typically repeats its instruction at regular intervals if the user holds the mouse button down and keeps the pointer over the button. This user action (holding down the button on a scroller, for example) gives the

appearance of continuous action. Users who keep the mouse button down can drag away from the control button and then back again to stop and restart the action. They can also press, release, and press again to restart the action.

All types of buttons respond visibly as soon as the mouse button goes down, and they maintain this “responding” appearance until the mouse button is released or the pointer is moved away. Even when the mouse button is released, control buttons maintain a responding appearance until any action the user initiated is complete. (In most cases this effect is momentary, and the user generally sees the button changing as soon as the mouse button is released.)

Buttons That Open Lists

The Application Kit supports two kinds of button/list combinations: pop-up lists and pull-down lists. Pressing the button for either of these controls opens a list, which stays open as long as the mouse button stays down. Users can drag through the list and choose an item.

Pop-up lists and pull-down lists look similar, and users choose items with the same action. One is an action button, however, and the other is a multistate button. This means that you use them in different situations.

Pop-Up Lists

A pop-up list functions like a set of radio buttons—it presents a set of values from which users can choose. (Compared to a set of radio buttons, the pop-up list saves screen space and prevents overcrowding in panels.) Each list is controlled by a button with a special symbol: . When users press the button, the list pops up. It stays open while the mouse button is down. See Figure 7-2.

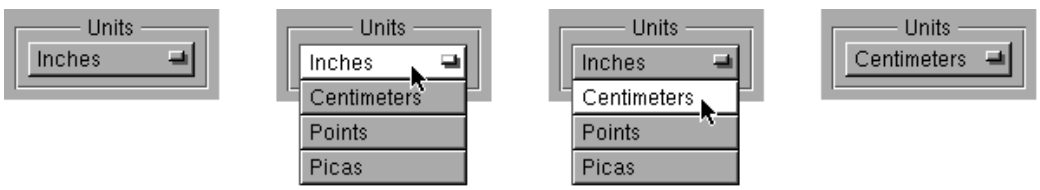


Figure 7-2 Pop-Up Lists

When users open a pop-up list, the item that matches the button's label appears on top of the list. Users can drag through the list and release the mouse button on another item, which selects it. When a new item is selected, the button's label changes. The button displays the new selection, but no action is taken.

Pull-Down Lists

A pull-down list looks like a pop-up list, but it initiates an action rather than setting a state. See Figure 7-3. The button that controls a pull-down list has a different symbol  .

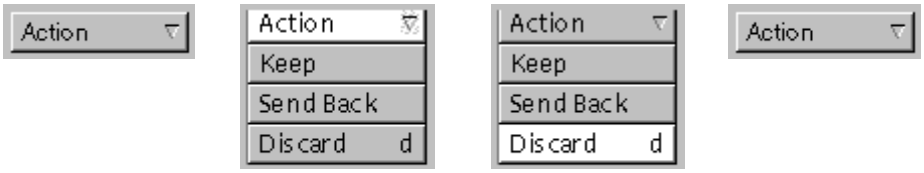


Figure 7-3 Pull-Down Lists

Users can drag through a pull-down list and release the mouse button to select an item. When they do, the control initiates the selected action, and the list closes without changing the label. Notice that this behavior of pull-down lists is like the behavior of a menu.

Implementing Buttons

Before you implement any type of button, you must do three things:

- Design the button's action.
- Choose an image, a label, or both for the button.
- Decide how to manage the button's appearance when it is clicked.

Of course, before using a button you should be certain that it is the best control for the job. Reading "Choosing the Appropriate Control" later in this chapter, should help.

Designing the Button's Action

A one-state (action) button should always perform the same action. It may seem efficient to vary a button's action according to the application's state—to switch between Erase and Restore, for example—however, it is better to provide separate buttons for each action and disable those that become irrelevant. This lets your users click safely on the same location whenever they want to perform an action without considering which state the application is in. It is acceptable, however, to change buttons that initiate time-consuming actions into Stop buttons while the action is in progress. See "Implementing Stop Buttons" later in this chapter.

One-state buttons are generally labeled with a verb or verb phrase that describes the action (such as Find), but some have only a graphic image (such as the arrowhead in a scroll bar button). Guidelines for assigning labels are covered in "Choosing the Button's Image or Label" in the next subsection.

Two-state buttons should never perform actions. Keep in mind, however, that changing a state can change the display. An inspector panel for a graph document might, for example, have a set of radio buttons that represent options for the graph type (line, bar, and so on). When the user clicks on one of these, it resets the graph's type, and the graph is redrawn. However, it would not be acceptable for the radio button to create a second graph of the new type.

Ideally, the display should be updated whenever a user clicks on a two-state button. This is not practical, however, when updating would take a long time or would be difficult to reverse. Consider giving users control over the display in these situations. If redrawing a graph takes a long time, you could provide a Redraw Graph button. (Remember that the button's own appearance must change immediately.)

Buttons with more than two states are not recommended; it is very difficult to convey their significance to the user.

Choosing the Button's Image or Label

A button's label should indicate what will happen when it is pressed or clicked. Keep in mind that a label identifying the *current* state (such as AM or PM), will probably be misunderstood. Users are more likely to assume the label represents the state they will set by clicking. A button labeled On, for example, is more likely to be interpreted as "Press here to turn something on" than as "This is now on."

To avoid this, use images and highlighting to show the current state, and use the button's label to indicate what it *does*. Labels that display the current state, such as one that switches between AM and PM, should only be used when the information they refer to is clearly visible. AM/PM buttons will probably be understood if you put them next to a digital representation of the time, but not if they stand alone. These and other two-state buttons are shown in Figure 7-6.

- Buttons should clearly look like action buttons or two-state buttons. Two-state buttons that do not have two distinct states and action buttons that do not look like they performs actions will confuse your users.
- Button labels should be dimmed (using gray text) whenever their buttons have no effect. Dimmed buttons are completely disabled—pushing them should not highlight, push in, or transform their appearance in any way.
- Button labels should be capitalized like menu commands. The first and last words should begin with capital letters, and the words between should be capitalized as they are in titles.
- Buttons that always open panels (except warning panels) should have three dots (...) at the end of the label.
- Action buttons that can be chosen with the Return key should display the Return symbol in their labels (see the Create button in the left panel in Figure 7-4). Some user actions, such as activating another application and moving key window status away from the panel, will prevent Return from choosing the button. In these situations, remove the Return symbol to prevent user confusion (see the right panel in Figure 7-4).

Figure 7-4 Using the Return Symbol on a Button

Programmer's Note – App Kit panels automatically remove the Return symbol when the panel loses key window status. So will attention panels created with `NSRunAlertPanel()` and its related functions.
For other panels, you need to explicitly manage the Return symbol. Do it in your implementations of the `windowDidResignKey:` and `windowDidBecomeKey:` delegate methods of the NSWindow class.

Changing the Button's Appearance During a Click

A control button's appearance should be transformed when the mouse button goes down, to indicate that it is responding. (This transformation should also reflect what the button will do; buttons that set a state should reflect the new state as soon as the mouse button goes down. See Figure 7-6, in which the Bold button uses a check mark to represent its state, for an example of this.) There are three ways of doing this:

- You can highlight it.
- You can highlight it and change the shading so it appears to be pushed in.
- You can change the image it displays.

In most situations, you should respond to user actions by highlighting the button and pushing it in. When buttons are right next to each other (such as scroll buttons and graphical radio buttons) you should not push them in, because they have no space around their bezeled edges and look less three-dimensional than normal. (It is possible to have a button pushed in without highlighting it, but this is not recommended because it is difficult for users to see.)

Note – Highlighting can be done automatically by the Application Kit or by changing the image to a custom "highlighted" image.

Figure 7-5 shows the recommended transformations for action (one-state) buttons.

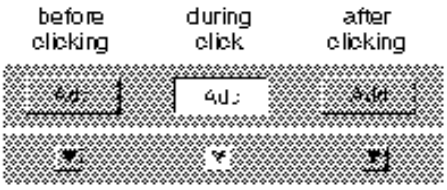


Figure 7-5 Changes in Appearance for a One-State Button

Figure 7-6 shows the recommended transformations for two-state buttons.

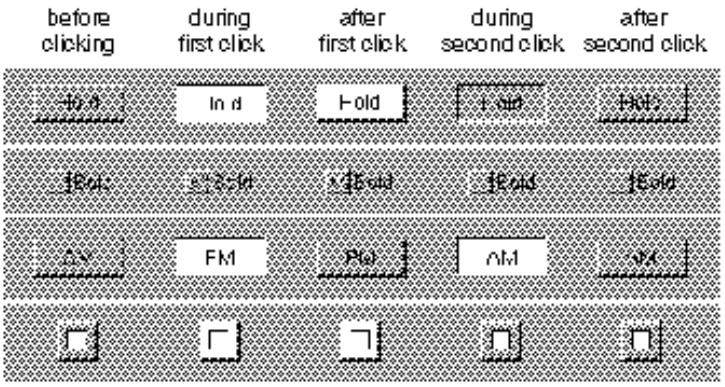


Figure 7-6 Changes in Appearance for a Two-State Button

Implementing Pop-Up and Pull-Down Lists

You must provide the titles for pop-up lists. The normal way to do this is with a titled box around the pop-up list's button. Figure 7-7 shows a box titled "Units" providing the title for a pop-up list.

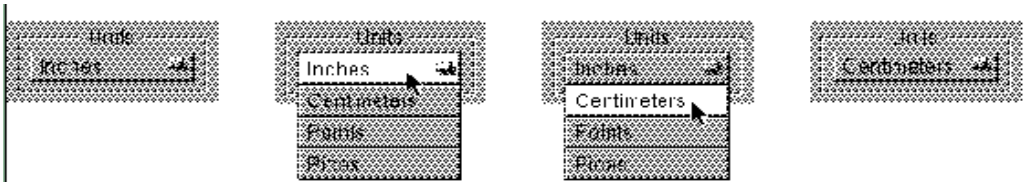


Figure 7-7 Titling a Pop-Up List

Pull-down lists do not need titled boxes because their labels do not change.

When you use a pop-up or pull-down list, make sure the list does not blend with nearby objects when it opens. An open list may also place items next to another object's label, leading users to interpret them together. Finally, an open list can obscure other objects that would help users decide what list item to select.

Programmer's Note – When an item in a pop-up or pull-down list opens an attention panel, default behavior for the list is staying open until the panel is closed. But the list is in a higher tier than the attention panel, and it will cover the panel. To avoid this, dismiss the list before opening the attention panel. Have the list item call the `performSelector:object:afterDelay:` method to schedule the execution of a method 1 millisecond in the future. This method should then open the attention panel.

Implementing Link Buttons

Link buttons are usually created by users rather than developers. Since they often appear in custom content areas which are usually unique to an application, link buttons need a little more support from the application than the other types of buttons. When you implement link buttons, make sure they have the following behavior:

- Clicking a link button highlights it (briefly) and opens the document that contains the information referred to in the link. An unmodified click should never select the link button.

  If the user presses the mouse button with the pointer over a link button and drags away without releasing the mouse button, the link button should lose its highlight. If the user keeps the mouse button down and drags back over the link button, the link button regain its highlight.

- Shift-clicking a link button selects it.

Implementing Stop Buttons

Users who initiate actions that might continue for some time can cancel those actions by holding down the Command key and pressing the period (.) key. (This is described in “Implementing Keyboard Alternatives” in Chapter 3).

You can make it more obvious to your users that they can interrupt the action if you give the button that controls the action a stop state. If you implement a stop state, use the transformations shown in Figure 7-8.

Figure 7-8 Stop Button

The button’s action should be started when the user completes the first click (releases the mouse button). Similarly, the button’s action should be terminated when the user finishes clicking the button in its stop state.

Text Fields

Text fields are controls that allow users to type a single line of data, such as a file name, a part number, or an address. See Figure 7-9. Users can select text and edit it. If the user types more text than will fit in the field, the entry automatically scrolls so that the insertion point stays visible. The data is processed only when the user presses Return or clicks on a button that is associated with the field.

Figure 7-9 Text Field

A text field should have a white background and be surrounded by a bezeled border that makes it appear inset from the surface of the screen. When the text field is temporarily disabled, the text becomes gray (just like the label of a button), but the background color does not change.

A text field that is not usually edited or selected but can be (such as a name associated with a file icon in the File Viewer) should display text on a gray background and have a border with no bezel. When the user selects the text, the text field's background should turn white, and the selected text's background should turn light gray.

Text fields can be titled and arranged in groups to produce an on-screen form, such as the one illustrated in Figure 7-10.

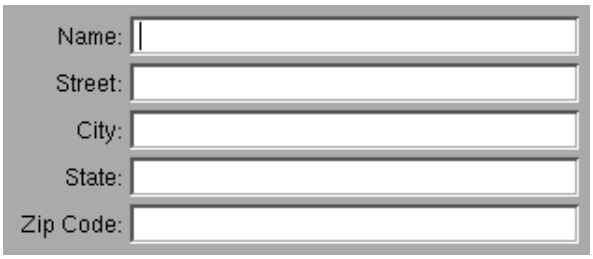


Figure 7-10 Titled Text Fields

When there is more than one text field in a window, the Tab key moves the selection—the point where typing will appear—from one field to another. Refer to Table 7-1.

Table 7-1 Moving from One Text Field to Another

Character	Action
Tab	Moves from one text field to the next one in the series. In Figure 7-10, Tab moves the current selection from the Name field to the Street field to the City field, and so on.
Shift-Tab	Moves from one text field to the previous one in the series.

When the user presses Return after typing in a text field, the field usually initiates an action. Data might be entered and processed, a search might begin for text that matches the string in the field, or a document might be saved to a file name the user typed. Exactly what happens is in the application's definition.

You should include a button that has the same action as Return (and is marked with the Return symbol) on the window or panel. This button's label is an explicit statement of what Return will do. From the user's point of view, Return is simply a shortcut for the action of the button.

Figure 7-11 shows a button with the Return key's action. The user can start printing a document by clicking on the Print button or pressing Return while a text field is selected. When the user presses Return, the Print button is pushed in and highlighted, as if the user acted on it directly.

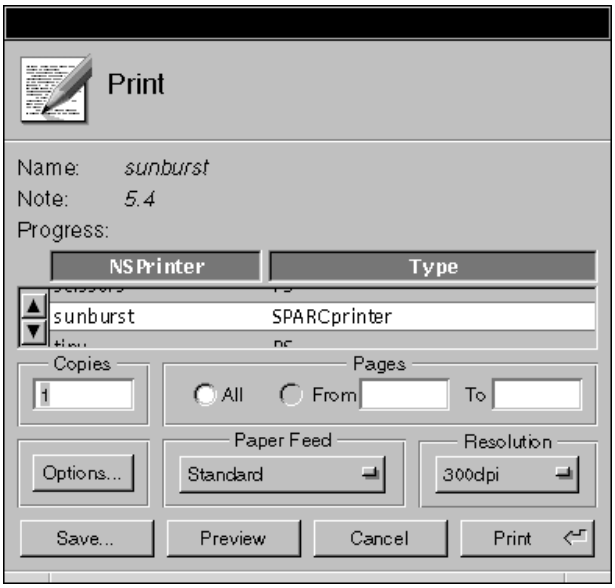


Figure 7-11 This Print Button Is the Equivalent of Pressing Return

- If the action associated with Return is one that the user is likely to repeat with different values in the text field, you can have the Return action select the text in the current field, allowing the user to replace it and press Return again.
- When you combine text fields in a form, you might decide that Return should not perform an action that processes user input. Instead, it will simply do just what Tab does—move the selection to the next field. Users must act on a button or other control to enter the data they type into the form.

Text fields generally accept data without restrictions on type, but in some situations you need to control the data type of the input. Typical examples of restrictions you can place on input data include:

- Unsigned or signed integers
- Unsigned or signed floating-point numbers
- Dates

If a user types data that is not accepted, the contents of the text field should be selected and highlighted. The user can make any necessary corrections and try again.

Sliders

Sliders are controls that let users set a value on a scale (between some maximum and minimum). The scale can be continuous or incremental (divided into intervals). It consists of a vertical (Figure 7-12) or horizontal (Figure 7-13) *bar* and a *knob* that moves on the bar.

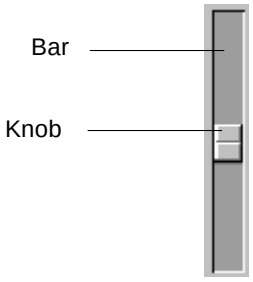


Figure 7-12 Slider

The position of the knob indicates the current value. Users can change the value by moving the knob, or by positioning the pointer somewhere over the bar and pressing the mouse button. The knob will jump to the pointer's location; releasing the mouse button will fix the knob in its new location, dragging will move it along the bar.

When the slider has an incremental scale, the knob jumps to the nearest incremental position when the user releases the mouse button. Users can press the Alt key and drag the knob to values between the normal scale.

Users should always be able to see some feedback when they manipulate the slider's knob—this is usually done in a text field or label next to the slider (as shown in Figure 7-13).

Figure 7-13 Slider with Label

Programmer's Note – Your application should specify the increment amount to be used for Alt-dragging. Otherwise, Alt-dragging has the same effect as unmodified dragging.

Color Wells

Color wells are controls that let users manage the colors in their documents. See Figure 7-14. They give users access to all of OpenStep's color definition tools, but the sequence of user actions required to use them is more complex than the actions required for other controls. You typically implement color wells on Inspector panels, where they allow users to set the colors of document objects.

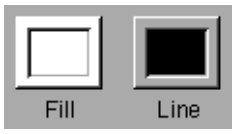


Figure 7-14 Color Wells

To work with a document object's colors, the user selects the object and opens its Inspector panel. The panel should have color wells, one for each part of the object with a color. In Figure 7-14, the Inspector has two color wells because the selected object has two colors: fill and line.

When the Inspector is first opened, the color wells should display the object's current colors. Users can change the colors in the wells and apply the new colors to the objects. These are separate actions, so users can perform some complex operations that copy colors from one object to another or create a new color and apply it to several objects. The basic user operations are:

- Setting the color in a color well.

  To do this, the user drags a color into it, from either the Colors panel or another color well. Users who want to define a color on the Colors panel can open it by clicking on the color well's border.

  If the user selects the color well's border, the color in the well will change when a new color is selected in the Colors panel (every time the user clicks on a color). This is a good way for users to preview colors, but users who don't understand the selection mechanism might find it confusing. The application should deselect the color well's border whenever the user is not actively selecting colors. For example, you should deselect all color wells when the panel thy appear on is miniaturized.

- Applying the color to a document object.

  When a object is selected, the color wells should change color to reflect the object's colors. For example, if a white box with a red border is selected, then the Fill well shown in the previous figure should contain white, and the Line well should contain red. When the user changes the color in a well, that change should be reflected in the selected object. For example, dragging a swatch of green from the Colors panel into the Line well should immediately make the outline of the selected box green.

  Whether or not a color well's border is selected should have no effect on whether the well affects the object that is currently selected

When deciding whether or not to user color wells, consider the following:

- They are powerful because they give access to whole color section mechanism, but indirect, so you should use them only when users need the full range of color choices.
- An alternative to using a color well, when the group of acceptable colors is small, is to use graphical radio buttons (as pictured in "Buttons" earlier in this chapter).
- Another alternative, when a wide range of colors is needed, is to use the Colors panel alone. The user can change an object's color by selecting it and then choosing the new color in the Colors panel.
- You could also use a custom control that is more appropriate in appearance and functionality than a color well.

The Colors panel is an Application Kit panel, covered in Chapter 5, "Panels." It opens when users selects a color well's border; they can also open it directly with the Colors command.

Scrollers

Scrollers are controls that allow users to work with documents larger than the display area, by moving the display area to different parts of the document. You will probably add scrollers to most viewing areas in your application.

Figure 7-15 Scrollers

Figure 7-15 shows the area on a window which is available for viewing documents, along with the document itself, which is much larger than the viewing area. Scrollers move the viewing area over the surface of the document.

How Scrollers Work

Figure 7-16 shows the three parts of a scroller: the *bar*, the *knob*, and the *scroll buttons* (which are optional). This scroller is vertical and scrolls the viewing area up and down. A horizontal scroller scrolls it from side to side.

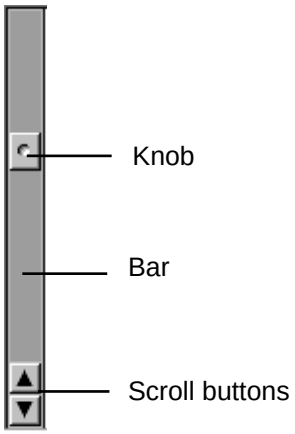


Figure 7-16 A Vertical Scroller

If the viewing area has scrollers but the document fits in the viewing area, the knob and the scroll buttons are removed, leaving a scroller in the form of a plain gray strip. This residual scroller indicates that functional scrollers will appear whenever the document becomes larger than the viewing area or the viewing area is reduced in size.

The Bar and Knob

The relative dimensions of the bar and knob indicate how much of the document is visible in the viewing area. The length of the bar represents the entire document, and the knob represents the part of the document that is currently visible. When the user deletes material from the document, the knob grows because the visible area is a larger percentage of the whole document; when the user adds material, the knob shrinks because it represents a smaller percentage of the entire document. The knob never becomes smaller than a square.

The location of the knob indicates which piece of the document is currently displayed. If it is at the top of the bar, the top of the document is being displayed; if it is at the bottom, the bottom of the document is displayed. The knob in Figure 7-16 indicates that about one third of the document area, near its top, is displayed in the viewing area.

Users scroll the viewing area by moving the knob in the bar. The knob responds to four kinds of user action:

- Dragging it to a new location. The viewing area moves as the knob moves.
- Pressing the mouse button in the bar but not on the knob. The knob jumps to the pointer, and the viewing area moves accordingly. If the user keeps the mouse button down, the knob can be dragged to a new location. With this technique, users can jump to the general part of a document they want to view (by pressing in the bar) and then fine-tune the viewing area (by dragging the knob).
- Pressing or clicking on a scroll button. The knob (and the viewing area) will move in the direction indicated by the arrow on the scroll button.
- Selecting something in the document and then extending the selection past the edge of the viewing area. This automatically scrolls unseen portions of the selection into view.

Fine-Tuning Mode

If a document is large, even small movements of the knob will make large changes in the display area. This makes it difficult for users to adjust the display area accurately by dragging the knob.

To help with this, scrollers have a fine-tuning mode that is activated by holding down Alt and dragging the knob. This changes the responsiveness of the knob, so that it (and the display area) move only slightly when dragged by the mouse. The knob moves in the direction it is dragged, but does not move as far as the pointer does. It moves slowly, representing the motion of the display area.

Once the Alt key is released, any subsequent dragging action will cause the knob to jump to the position of the pointer.

Scroll Buttons

The scroll buttons permit more precise scrolling than direct manipulation of the knob. In a text document, clicking on the vertical scroll button scrolls the viewing area by one line of text. When pressed, it scrolls continuously in one-line increments. Horizontal scroll buttons work in a similar way, scrolling a small, fixed amount either left or right.

- If the user drags from one button on a scroller to the other, without releasing the mouse button, each button acts as it is pressed. (Users cannot drag a scroll button on one scroller to a scroll button on another scroller.)
- If the user holds down Alt and clicks on a scroll button, the viewing area moves by an amount based on its own dimensions. Alt-clicking on a vertical scroller moves the bottom line (or two) to the top of the display area or the top line (or two) to the bottom. This provides users with a bit of overlap, so they know that nothing was skipped.
- Sometimes scroll buttons are displayed without the rest of the scroller. Since there are no knob and bar to indicate when the viewing area has reached a vertical or horizontal limit, you should dim the arrow on the corresponding scroll button.

Note – Interface Builder makes it easy to put scrollers around a document display area. The App Kit handles all scrolling characteristics, including Alt-clicking and Alt-dragging. You may want to adjust the amount the viewing area moves for a single click (even for graphics, it should be the distance comparable to a single line of text) and optimize drawing performance so that scrolling is as fast as possible.

Automatic Scrolling

When the user begins a selection (usually of text) in the display area and then drags to extend the selection beyond it, the display area will scroll continuously to bring the moving edge of the selection into view. This continues until the user releases the mouse button and sets the final edge of the selection. The amount scrolled increases as the edge of the selection moves farther from the display area. The application brings the location under the mouse pointer into view.

As the document scrolls, the scroller knob is adjusted to reflect the current position of the display.

Implementing Scrollers

If a document is taller than the display area, it should have a vertical scroller. If it is wider than the display area, it should have a horizontal scroller.

Note – When users scroll, they move the display area over the surface of a document. They often perceive, however, that the document moves (not the display area). This means that the knob and the document appear to move in opposite directions. To avoid confusion, documentation for your application should explain scrolling in terms of adjusting the visible portion of the document, rather than adjusting the document to make it visible.

The scroll buttons for both vertical and horizontal scrollers should occupy the lower left corner of the display area, where the two scrollers meet. This makes it easy for users to move from one set to the other.

If you use controls that operate on the document view, such as the page-scroll buttons as shown in Figure 7-17, you can put them in the area normally occupied by the scrollers (beneath and to the left of the document). Other types of control should not be put in this area.

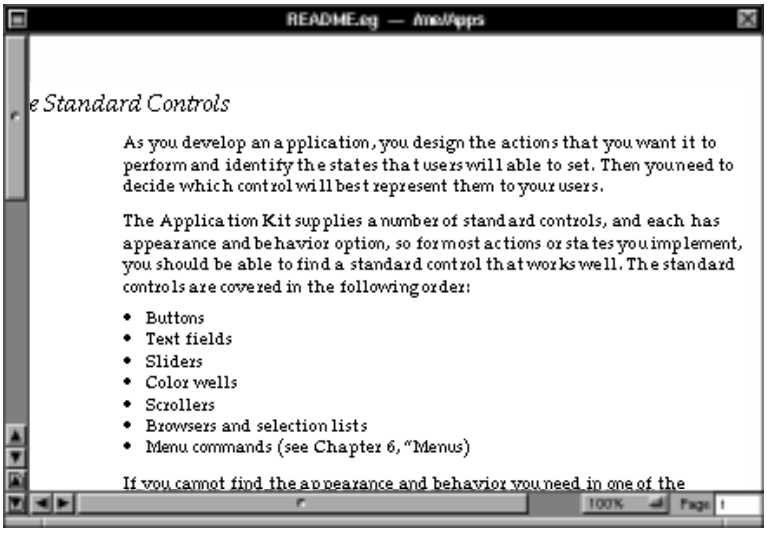


Figure 7-17 Page-Scroll Buttons and Controls

Controls that you can put in the scroller area include:

- An editable text field that displays the current page number can be located at the far right of the horizontal scroller (see Figure 7-17).
- A pop-up list that lets the user scale the display can be located in the horizontal scroller (see Figure 7-17).
- A pop-up list that controls the viewing mode (for example, preview versus drawing mode in a graphics application) can be located in the vertical scroller.
- Page-scroll buttons, which scroll from page to page or in increments matching the dimensions of the display area, can be grouped next to the scroll buttons. There is no current Application Kit support for page scroll buttons, and a precise arrangement is not specified. (See Figure 7-17 for typical page-scroll buttons.)

Browsers and Selection Lists

Browsers and selection lists are controls that display lists and let users select one or more items in the list. Browsers show several lists side-by-side, allowing users to work with data organized hierarchically, such as:

- Files and folders
- Cities, counties, and states
- Management structure of a company

Selection lists display data that exists on a single level. They usually have scrollers, but this is not required. See Figure 7-18.

Figure 7-18 A Browser and a Selection List

"Selecting Objects With the Mouse" in Chapter 3 summarizes the mouse actions that select items in a browser or selection list. Browsers and selection lists can also include text fields, in which users can select by using the keyboard and typing in the name of an item. The browser on the Save panel includes a text field that allows users to select from files appearing in the browser by typing.

If double-clicking on items appearing in a browser or selection list does anything, it should perform the same action as pressing Return (that is, the same action as the button marked with the Return symbol).

Choosing the Appropriate Control

In some situations, it is clear which control will best represent an action or state to your users:

- Scrollers are used when documents are displayed in display areas that might be too small for a complete document.
- Sliders are used when your users need to set a value within a bounded range (of colors, numbers, or sound levels, for example).
- Text fields are used whenever it is impossible or impractical to provide a list of all possible values.
- Color wells are used only with the Colors panel, and only where complex color characteristics are needed.
- Browsers are used only for data that is organized hierarchically.

In other situations, the choice is less clear. You can use a button or a selection for most of them, but these controls have many variants, and you want to pick the one that "feels right." The rest of this section analyzes different kinds of actions and states that must be represented by controls and identifies the buttons or selection lists that will work best.

Controls That Represent Actions

Actions your users can start should usually be represented by menu commands or buttons.

- In some cases, an action can be represented by both a menu command and a button.

- Although text fields can initiate actions when users press Return, they do not identify the action nor provide feedback to the users. You should always supply a nearby button that represents the action, labels it and responds when Return is pressed.
- Some actions are unimportant or rarely used and you don't want to use screen space to represent them as individual buttons. In these situations, you can make them into items in pull-down lists. Another possibility is to implement them as menu commands.

The Figure 7-19 shows a single action, printing, implemented by a menu command, a button, and a pull-down list item. Any of these controls opens the print panel. Print is normally implemented as a menu command.

Figure 7-19 Different Controls That Start the Same Action

Controls That Represent Settable States

Settable states can be represented by two-state buttons, pop-up lists, and selection lists. Menu commands should never be used to show state. A settable state can be more or less independent of other settable states, and the degree of independence influences your choice of a control to represent it:

- In some situations a settable state is completely independent of any other state represented on the panel. (The display size of text in a document is an example.) These states can be represented by a single control.
- Some settable states (options) are related to other options on the panel, but there is no fixed relationship among them. Users can choose any combination of options.
- In other cases, there is a mutually exclusive relationship among a group of options. Users must choose one, and only one, of the related options.

Note – Selection lists allow a relationship in which either no choice or one choice can be selected.

Controls that represent a state should only be used to show and set state and not to initiate actions. There are two apparent exceptions to this rule:

- Double-clicking an item in a selection list can initiate an action, but the double-click is actually a shortcut for selecting the item and then clicking on a button.
- Setting a state can have visible consequences, such as changing the displayed format of a document to change, but this is actually feedback that the state has changed, and not an action.

Displaying a Single Option

When a single option is truly independent of the other options on a panel, a switch is the preferred control. You could also represent it as a graphical two-state button, as long as it is very clear that the button has two states. Refer to Figure 7-20.

Figure 7-20 Control for an Independent Option (a Switch)

Displaying a Group With an Unrestricted Relationship

When you want to represent a number of related options that work together to determine something but do not have a mutually exclusive relationship, use one of the controls illustrated in Figure 7-21:

- Group of switches
- Group of graphical two-state buttons (they should not look like graphical radio buttons)
- Selection list

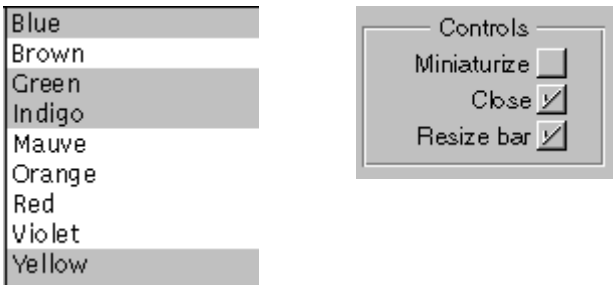


Figure 7-21 Controls for Options With an Unrestricted Relationship

Switches and graphical two-state buttons are preferred. Selection lists are less attractive and do not indicate how many selections can be made. (If you use two-state buttons, be careful that they do not look like radio buttons.) A selection list might be more appropriate when the list of choices can increase or decrease dynamically, or when space on the panel is restricted.

Displaying a Group With a Mutually Exclusive Relationship

Several kinds of controls can be used to represent options with a mutually exclusive relationship. These controls let the user choose one and only one option:

- Group of radio buttons (standard or graphical)
- Pop-up list
- Selection list

Figure 7-22 shows these controls as they would be used to set the background color of a text field. Because this use is inherently graphical and there are only a few valid choices, graphical radio buttons are the best choice, followed by standard radio buttons. A pop-up list is marginally acceptable for this use, and a selection list is the least appropriate choice.

Figure 7-22 A Group of Choices With a One-of-Many Relationship

The following considerations might help you decide which control to use:

- If the control will be used frequently, consider using radio buttons (standard or graphical), since they are easier to operate and more accessible to the user.
- If text does not adequately describe the choices, consider using a group of graphical radio buttons.
- If space is limited or the window or panel looks too complex, consider using a pop-up list.
- If the list of choices can increase or decrease, consider using a pop-up list or a selection list.
- If the list of choices can grow larger than the screen, use a selection list with a scroller.
- If the user needs to see more than one of the choices on screen to understand them, avoid using a pop-up list.
- If the control will usually appear at the edge of the screen, you might want to avoid a pop-up list.

Note – : A pop-up list usually pops up so that the current selection is under the pointer. If, however, the list is long and near the edge of the screen, it shifts so that the entire list can appear on screen, which may change the selection under the pointer. Users might unwittingly make a new selection while intending only to see what is in the list. When considering a pop-up list, think about whether it is important to avoid this result.

- If many ordinary text-based buttons are in the panel, a pop-up list might fit in better graphically.

The Interface to the File System 8

The OpenStep Workspace Manager provides an effective graphical interface to the Sun Solaris file system. Directories are represented by folder icons, and files are represented by a variety of icons which indicate file type and owning application. These objects are displayed on a specialized panel (the File Viewer) featuring browser controls that allow users to navigate the file system with mouse actions.

The file icons are fully-featured graphical objects, and users can act on the icons they see in the File Viewer (in accordance with the direct manipulation, modal tool, and targeted action paradigms) to manipulate files. The Workspace Manager, which includes the File Viewer, the Application Dock, and all of their characteristics, has been implemented for you. (These characteristics are documented in *Using the OpenStep Desktop*.)

This chapter explains some things you will need to know in order to fully integrate your application into the OpenStep environment—conventions for folders in which applications are installed, for creating file packages, new file extensions, and so on.

Note – Users who want to use a command line to interact with the operating system can use the Terminal application to open a VT-100 terminal emulation window and run a standard command shell.

OpenStep Folder Conventions

Although the OpenStep system runs on a full Solaris file system, the file system interface has adopted some conventions that make it easier for users to work with the file system:

- The File Viewer's default view shows all of the files available on the system, but the user can change the view and hide many of the system folders.
- OpenStep software creates some standard folders for OpenStep system software. It also establishes conventions for other software (including your own) that is installed at user locations.

The user's view of the file system begins with the root (/) folder. Inside the root folder, the OpenStep convention is to distinguish between software installed as part of the product and other software installed at a customer site, by creating and using the following folders:

- Home folder, for files the user creates and works with
- OpenStep folders, for software installed as part of an OpenStep installation, which include the developer tools, for users who install them.
- "Local" folders, for software installed at a customer site and available to all users
- "Personal" folders, for software installed by (or for) individual users
- A `/net` folder that represents network folders available to the computer

Home Folder

Traditionally, each user has a home folder that is identified with his or her login name, `/home/`*username*. Home folders of other users may also be visible in the File Viewer.

When typing in file names, users can specify their home folder as ~. For example, `~/openstep` refers to the `openstep` folder in the user's home folder.

OpenStep Folders

The OpenStep folders contain documents, resources, and the applications that are bundled with `OpenStep`.

- `/usr/openstep/Apps` contains supported applications that are likely to be used by all OpenStep users. These are general-interest applications such as Edit, Preferences, Mail, Terminal and Preview.
- `/usr/openstep/Developer` holds applications and files used to develop OpenStep applications. Some of the folders that can appear in `/usr/openstep/Developer` are listed in Table 8-1.

Table 8-1 Folders in `/usr/openstep/Developer`

Folder Name	Contents of Folder
`Apps`	Holds applications used by developers. This folder might not exist on all systems. Project management, application construction, and other window-based programming tools belong here. Like all other applications in the four OpenStep folders, these applications can be run from the Workspace Manager. Compilers, debuggers, and profiling tools are typically located in `/opt/SUNWspro/bin`.
`Demos`	Contains programs that demonstrate the capabilities of OpenStep. These are not full applications and are not supported by Sun. However, they include games and utilities that users might find interesting.
`Examples`	Contains source code for example programs. This folder might not exist on all systems. Its folders contain source code that you can study and compile.

- `/usr/openstep/Library` contains resource files, which are organized into several folders. Not all files or folders are on every system.

Local Folders

Software not supplied by Sun and installed with OpenStep should not be installed in `/usr/openstep`. This software belongs in folders known as local folders. The following list describes local folders, their creation, and their use.

- Local folders are not included on the release disk. They are created when they are needed.

- The local folders local folders you can create and use are `/usr/local/openstep/apps` and `/usr/local/openstep/Library`.
- The local folders should not contain any files that are part of the OpenStep product. They should contain applications and other information available to all users at a local site. A new font available to all the users of your network, for example, should be installed in `/usr/local/openstep/Library/Fonts`.
- Any user who logs in to a computer or boots from it over a network has access to the contents of the computer's local folders.

Personal Folders

Users can use `~/openstep/Apps` and `~/openstep/Library` folders (not `LocalLibrary` or LocalApps) in their home folders for applications and information that they alone have access to. Users who develop utilities for their own use or purchase private copies of word processors or spreadsheets should install them in `~/openstep/Apps`.

/ net *Folder*

The `/net` folder gives the user access to file systems that are physically located on remote machines. Immediately below `/net` are folders that represent the servers visible to the current computer. The next level of folders represents the file systems on those machines that have been exported to the network. For example, `/net/willow/export/misc` is the `misc` file system on a server named `willow`.

Note – `/net` will only contain other folders if automounting is in effect on the current computer and the file systems it attempts to mount have been exported.

All the folders in the root (`/`) folder, including `/net`, are physically located on the disk that the user's machine was booted from.

Search Paths in OpenStep File System

This section covers the uses of search paths in the OpenStep file system. There is a standard Workspace Manager search path, and you can create a search path for your own application to use.

The Workspace Manager Search Path

When the Workspace Manager searches for executable files it looks in the folders specified in the *search path*, The default search path includes six folders:

```
~/openstep/Apps
/usr/local/openstep/Apps
/usr/openstep/Apps
/usr/openstep/Developer/Apps
/usr/openstep/Developer/Demos
```

The search path is used in three situations:

- Finding the icons to display for the *files* associated with a particular application
- Finding the application to start when a user double-clicks on a file
- Finding services offered by applications

Before using the search path to find the application to start up, Workspace Manager first looks in the dock. Each application icon in the dock represents a specific executable in a specific folder, and Workspace Manager assumes that the user prefers the version installed in the dock to any other versions that may be available.

If an application is not represented in the dock, the Workspace Manager next looks in the current working folder (the folder containing the file the user wants to open). If it fails to find the application there it uses the search path.

The search path begins with `~/openstep/Apps`, since any user-installed applications would be there. `/usr/local/openstep/Apps` is next because it contains sitewide software. If nothing is found in the user and site folders, `/usr/openstep/Apps`, `/usr/openstep/Developer/Apps`, and `/usr/openstep/Developer/Demos`, which contain Sun-supplied software, are searched. This search order ensures that any user-installed software is found before (and overrides) the sitewide software, and the sitewide software is found before (and overrides) Sun-supplied OpenStep software.

Users can alter the default search path by setting a value for the Application Paths parameter in their defaults database.

Application Search Paths

If your application needs to look up data that could be in more than one place on the system it should use an ordered search path similar to the Workspace Manager search path. Three functions, `NSStandardApplicationPaths()`, `NSStandardLibraryPaths`, and `NSStandardFontPaths`, provide a programmatic interface to ordered search paths. Applications that use these functions can avoid hard-coded search paths.

File Name Extensions in OpenStep File System

This section covers the uses of extensions in the OpenStep file system, including extensions reserved for the Workspace Manager and custom extensions for your own applications.

Workspace Manager Extensions

OpenStep applications use file name extensions to identify different types of files. The extensions consist of the last period (.) in the file name and the characters that follow it. Mail, for example, uses the extension `.mbox` to identify its mailboxes. Typical names for mailboxes are `Active.mbox` and `Outgoing.mbox`.

The Workspace Manager also uses file name extensions to associate document files with applications. Every application that defines its own data file format appends an identifying extension to the names of its document files.

Custom Application Extensions

Your application should use a unique file name extension to identify its documents. The characters that follow the period can include only alphabetic and numeric characters. Your extensions must include at least three characters; extensions of fewer than three characters are reserved for OpenStep files. The extension `.example` is acceptable, but `.eg` is not, because it has only two characters after the period.

File Packages in OpenStep File System

A *file package* is a folder which looks like a single file in the File Viewer. Because users do not normally look inside file packages (unless they explicitly open it as a folder), they are not likely to alter or reorganize their contents.

Applications should create a file package when they work with groups of files that must be kept together. If, for example, your application displays information that is stored in independent text files, or if it makes use of a private utility program, or if it just loads archived objects from Interface Builder `.nib` files, you may want to keep these auxiliary files in close proximity to the main application executable file. A file package (with the `.app` file name extension) is the way to do this.

Similarly, if your application creates documents that are split into more than one file—for example, if text is in one file and artwork in another—these files can also be grouped in file packages.

File packages for documents should have the file name extension used for the application's document files. When a word processor uses a file package to store a document and its artwork, the file package should have the same extension as a text-only document stored in a single file.

Opening and naming the individual files within a document file package is entirely the application's responsibility.

Displaying File Names

When file names are displayed in browsers, users can see the complete path leading to the file. In other situations, as document window title bars, text strings are used to represent the file's name and path. In these situations, the correct format for the filename is as follows:

`jobRecords — /net/`*`servername`*`/export/records`

The file name should be followed by two spaces, an em dash, two more spaces, and then the path.

When there is not enough space to display the full file name—as in a menu command—you can shorten the path to the minimum that differentiates the file. The part of the path you don't show should be replaced with three dots.

For example, if a file called `jobRecords` is listed in a limited space, you can display it as:

```
jobRecords
```

If two files called `jobRecords` are listed in limited space, you might display them as:

```
jobRecords  —  .../records
jobRecords  —  .../backup
```

Files and Folders That Your Application Creates

Applications frequently create, manage and save files and folders that are not explicitly created by their users. When an application creates several files to store a single document, they should be grouped in a file package (see the preceding section for more information). In other situations, when the files hold information about the current state of an application, templates, or similar information, the file names and locations the application uses for them depend on whether users need direct access.

The guiding principle here is that the user's home folder belongs to the user. Applications should not put anything in the home folder that belongs to an application rather than the user. When the application must create unrequested files and folders, it should put them where the user can find them but where they are not likely to get in the user's way.

- If the user never needs direct access to the files or folders, they should be placed in a folder named `~/.openstep/.AppInfo`. If an application needs to create many such files, it should create a folder in `~/.openstep/.AppInfo` and store them there. If MyApp creates many unrequested files, it should put them in a folder named `~/.openstep/.AppInfo/MyApp`. (If the `~/.openstep/.AppInfo` folder does not already exist, the application can create it.)
- If the user might work with the unrequested files or folders independently of the application that creates them (for example, template files), the files and folders should be placed under `~/openstep/Library`. The names assigned to these files and folders should let users know which applications they belong to.

Note – You should generally use the defaults system instead of files to store small amounts of data. The functions supporting the defaults system are described in the *OpenStep General Reference* manual.

8

Index

A

B

C

D

E

F

H

I

J

K

L

M

T

U

W