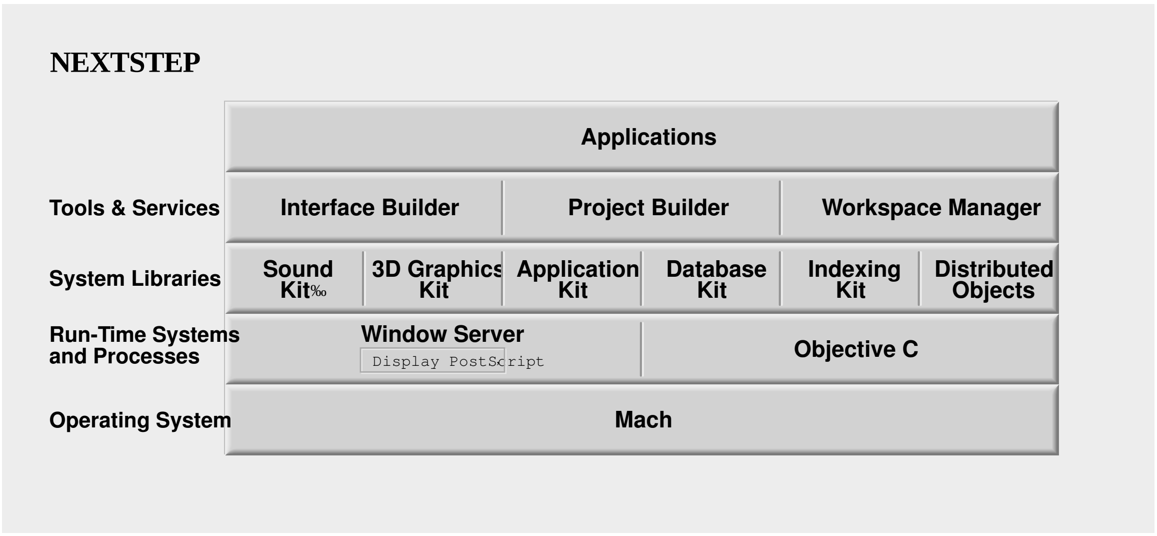
The real key to programming productivity with NeXTSTEP is the complete application framework provided for you. This framework, implemented by the Window Server and the Application Kit, provides the core functionality for any application and offers useful enhancements that can help make all applications more powerful.

NeXTSTEP is a multilayered object-oriented environment designed from the ground up to help improve programming productivity. Built on the Mach operating system, NeXTSTEP is made up of run-time systems and processes, libraries, and development tools. Together, these resources free you from much of the complexity of creating sophisticated and user-friendly applications.

Project Builder and Interface Builder are applications that allow you to create, test, and build graphical, object-oriented programs quickly and easily. They're discussed in "Step-By-Step Through a NeXTSTEP Application," earlier in this guide.



The Objective C language is used to implement many features of the NeXTSTEP environment. The Objective C run-time system enables many of the benefits of object-oriented programming described in the previous section.

This section focuses on the Window Server and the Application Kit, two NeXTSTEP components that cooperate to define the framework of every NeXTSTEP application.

THE WINDOW SERVER

Because NeXTSTEP's UNIX-compatible Mach operating system is fully multitasking, many applications can be running simultaneously. Each NeXTSTEP application has one or more windows associated with it, all capable of drawing their contents and receiving user events. If each application were required to manage window display and event-handling entirely on its own—as with some systems—the code required would be very hard to write and maintain.

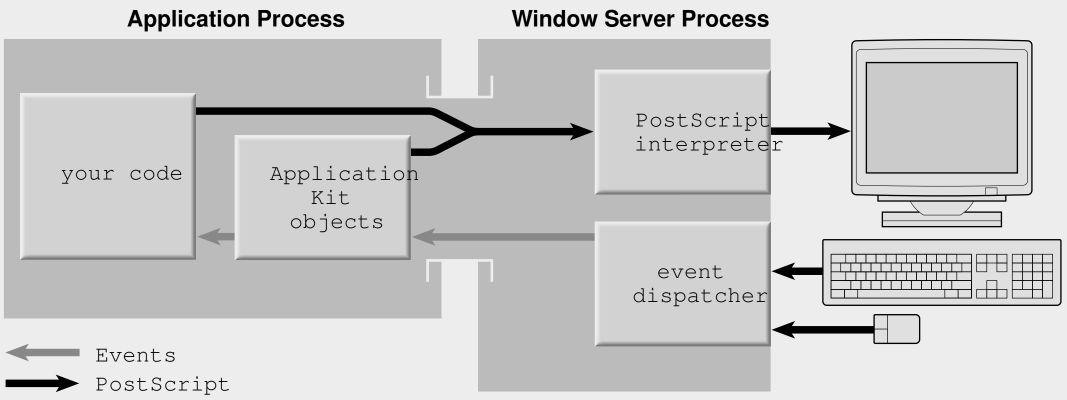
The Window Server is a process that unifies these tasks, providing window management, display, and event-handling for all running applications. Every application is a client of the Window Server, getting its information about user actions from the Window Server, and sending its drawing instructions to the Window Server.

A window can be thought of as a rectangular region used to display the graphic output of an application. Windows are also the focus of user input. They're where typing appears, and where user-interface items the user can manipulate with the mouse are located.

Window Server windows are low-level entities with some very rudimentary behavior. They can be resized, ordered in a certain front-to-back order, and displayed on the screen, but they don't have any user-interface controls, nor do they respond directly to user actions. The interface to Window Server windows—both the user interface and the programming interface—is provided by the Application Kit through Window objects.

Drawing instructions are sent to the Window Server in the form of PostScript, code. The Window Server interprets this PostScript code to render the output of each application to its windows, or passes it to a printer or other device

APPLICATION DATA FLOW



This figure shows the overall data flow for a typical application that accepts input from the keyboard and mouse and displays output on the screen.

for hard-copy output. Because it uses PostScript for all imaging, NeXTSTEP provides complete device-independence. The Window Server ensures that an application's drawing is always performed at the highest resolution and using the broadest spectrum of colors available for a specific output device.

In addition to rendering images, the Window Server sends each user keystroke and mouse click to the appropriate application in the form of an *event*. The event includes information about the type of action the user took, the location of the cursor, the window where the action occurred, and various other data. Because NeXTSTEP applications act primarily in response to events from the Window Server, they are said to be "event-driven."

Several Application Kit classes are designed specifically to handle the events distributed by the Window Server. This greatly reduces your responsibility in creating an event-driven application. Application Kit objects may take total responsibility for handling an event from the Window Server, or they may do some of the event handling and then send an Objective C message to one of your own objects. To write an event-driven application, you simply implement methods that respond to messages sent from Application Kit objects.

THE APPLICATION KIT

The Application Kit provides a complete collection of powerful, flexible classes that you can use to construct your own applications. A few of these classes are discussed in detail in this chapter; a summary of Application Kit classes can be found in the "Class Summaries" section.

The Application Kit defines the essential structure of a NeXTSTEP application. This structure is characterized by a single Application object, a Window object for every window the application needs, and various View objects that subdivide the territory within windows, that draw the contents of their territory, and that handle events within that territory.

Every application based on the Application Kit consists of a network of Objective C objects—some provided by the Application Kit, some that you define as subclasses of Application Kit classes, and some that you define as subclasses of other NeXTSTEP classes.

For example, the Application Kit directly provides most of the objects used in YourCall, including its Application, Menu, Window, Text, and Button objects. YourCall has one subclass of the Object class for its CallController object, used to transfer data between the user interface and the database, and another, CallRecord, to represent call data stored in the database.

Since much of the structure provided for every NeXTSTEP application by the Application Kit is embodied in three objects—Application, Window, and View—it's useful to understand examine their behavior in more detail.

The Application Object

Central to every application is a single Application object, which acts as a coordinator between the many objects within the application, between these objects and the Window Server, and between the application and other NeXTSTEP applications. The Application object is created for you automatically at run time if you construct your application using Interface Builder.

The Application object retrieves events from the Window Server and forwards them as messages to the appropriate objects. By acting in concert with other Application Kit objects, it greatly simplifies your responsibilities in writing an event-driven application.

The Application object can notify another object, its delegate, of significant happenings pertaining to the application—such as when the application is fully initialized and ready to receive events, when it becomes active (becomes the application that the user interacts with), when another application is activated in its place, or when the user quits the application. These notifications allow you to execute application-specific code at the appropriate times.

Window Objects

A Window object provides the basic user interface to a Window Server window. It enables a window to move, resize, come to the front of the screen, and hide automatically in response to the user's actions—you don't have to write the code to implement this behavior.

A Window object also provides a high-level programming interface to a Window Server window. All your application has to do is create a Window object; the Window Server window is created automatically through the object. To programmatically control a window, an application sends Objective C messages to the Window object, which interacts with the Window Server on the application's behalf.

Like the Application object, a Window object can have a delegate that is notified of significant happenings pertaining to the window and can respond appropriately. For example, the delegate can be notified when the user moves or attempts to close the window, and the delegate can dictate acceptable window sizes when the user resizes the window. Because the delegate can exercise a degree of control over one or more windows, you rarely need to create a subclass of the Window class in order to get the specific window behavior your application requires.

The YourCall application is a good demonstration of the utility of the Window class. It has a full-featured window that allows normal user interaction with the application, yet it has virtually no code devoted to window management.

Views

Views are Application Kit objects that know how to respond to keyboard and mouse events and that can output PostScript code in order to draw within a portion of a window. The window area that a View can draw within is identified by its *frame rectangle*, which specifies the size of the View and its location.

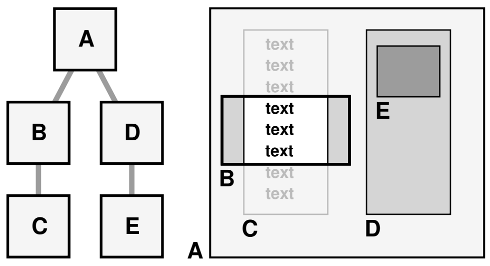
Unlike a window, which stores and displays rendered images for an application, a View does not store an image, but is an entity used to construct an image. In fact, every Window object must have one or more View objects that draw the contents of the window.

The View Hierarchy

Every View maintains a list of *subviews*, which are View objects that the View manages. In order to draw within its area of the window, a View can output PostScript code itself, or it can simply use its subviews to draw. Every Window object has one main View, known as its *content view*, which is used, either directly or indirectly, to draw the entire contents of the window.

The grouping of a Window's content view, its subviews, their subviews, and so forth, is known as the Window's view hierarchy, and is important for understanding drawing and event-handling in NeXTSTEP.

Here's an example that shows two ways of looking at a single view hierarchy:



On the left is the structural representation of the hierarchy. View A has two subviews, B and D, so View A is said to be the *superview* of View B and View D. View B and View D in turn each have a single subview, View C and View E.

On the right, the same view hierarchy is shown as it might appear on-screen. View A contains its subviews, which in turn contain their subviews. Note that a View "clips" its subviews; in this illustration only the portion of View C that is within its superview (View B) gets displayed.

One purpose of the view hierarchy is to determine drawing order and thus which Views appear in front of others. (Remember, however, that a View doesn't have to draw; it could simply leave its drawing to its subviews.) To draw a Window at the appropriate times, the system sends a display message to its content view. All Views have a display method that draws the View and then sends a display message to each of its subviews. This insures that each View in a Window's view hierarchy is drawn, and in the proper order. In the example above, the drawing order would be A, then B, C, D, and E.

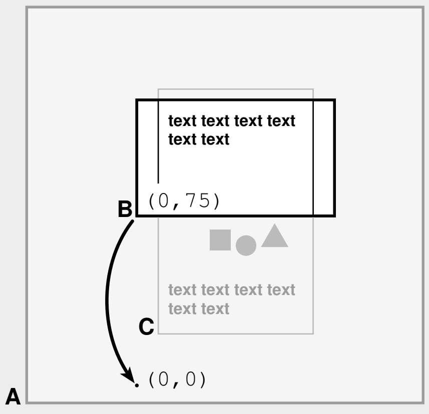
A View's Coordinate System

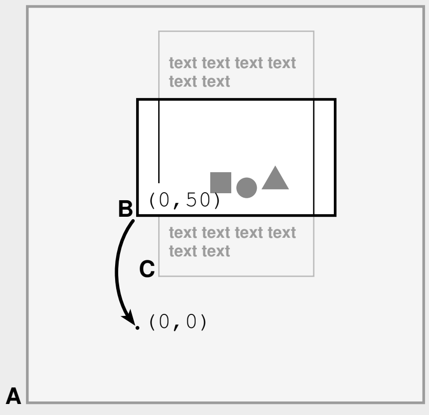
Each View maintains its own coordinate system for all drawing done within the View. The coordinate system is also used to locate the View's subviews. Each View's frame rectangle is specified in its superview's coordinate system. Thus the size and location of a View is relative to its superview.

Because each View draws in its own coordinate system, you can reposition a View within its superview or window without affecting its appearance. When you reposition a View, all its subviews are repositioned with it, since they are located within the View's coordinate system—regardless of the View's position in its superview, window, or on the screen.

By default, a View's coordinate origin is coincident with the lower left corner of its frame rectangle. The View class provides methods that allow you to reposition, scale, or rotate the coordinate system of any View. Such transformations don't affect the position of the View, but they apply to any drawing done within

SCROLLING WITH A CLIPVIEW





In this example, View A has one subview (View B), which is a ClipView object. View B in turn has one subview (View C), which is the View to be scrolled. Each subview is displayed within its superview's coordinate system. Since A's coordinate system and B's position within that coordinate system don't change, B doesn't move. However, B's coordinate system does change, so View C will move.

In the first picture, View B's coordinate origin is 75 units below B's lower left corner. In the second picture, View B's y coordinate has been translated by 25 units, so the origin of its coordinate system is 50 units below View B's lower left corner. Since View C is displayed within its superview's coordinate system, this has the effect of scrolling View C up. View C's own coordinate system has not changed, so View C doesn't have to make any drawing adjustments.

it, and they affect the position of the View's subviews.

The Application Kit's ClipView class provides an example of how manipulating a View's coordinate system can be useful. ClipView objects are used for scrolling other Views. If you have a View that knows how to display a large document, you can make it a subview of a ClipView object in order to display only a portion of the document on the screen. You can then send messages to the ClipView instructing it to translate its own coordinate system, thus moving its subview and scrolling the document the subview displays.

Since the document View maintains its own coordinate system, it doesn't need to make any adjustments for drawing based on its scrolled position on the screen. The document View doesn't even need to know that it's the subview of a ClipView, nor that it gets moved on-screen. (See "Scrolling with a ClipView," previous page)

To further simplify scrolling behavior, the Application Kit provides the ScrollView class. A ScrollView object coordinates the display between a ClipView and Scrollers, interface objects that allow the user to control scrolling of a document. The ClipView takes care of scrolling the document, and the ScrollView takes care of coordinating the user-interface for scrolling, so you can focus on the custom code for the document itself.

EVENT HANDLING

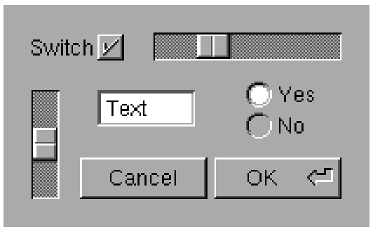
With most other graphical environments, the programmer must write an event loop that requests events from the system, and determines what happened, where it happened, and what to do about it.

With NeXTSTEP, events are retrieved and delivered to the correct objects automatically. The Application object gets an application's events from the Window Server and forwards them to the appropriate Window object. The Window object, in turn, handles the event if it pertains strictly to the window, or delivers it to the appropriate View object.

Events are sent to Views as Objective C messages. Each *event message* is named after an event type—such as keyDown: or mouseDragged:. For example, Buttons and other controls implement a mouseDown: method that sends the action message to the target object when the user clicks the button in the user interface.

To enable a View to respond to all events pertaining to it, you simply implement the methods that respond to messages sent automatically from the View's Window object. For example, if the View must handle mouse-down events, you implement a mouseDown: method, and that method is invoked when the user clicks in the View. You don't need to write the code that determines which View to send an event to, because the Application and Window objects already do that for you. If the View doesn't need to respond to mouse events, you simply don't implement the mouseDown: method.

Controls



The Application Kit provides two classes—Control and Cell—to simplify common event-handling even further. With a custom View subclass you must implement the custom response to events pertaining to your View. Controls and Cells, however, allow you to specify a target object and a specific action method within the target object that's to be invoked in response to user manipulation.

Examples of Application Kit Controls include Button (which sends a message when clicked) and TextField (which sends a message when the user types the return key). Another subclass of Control is Matrix, a Control that manages multiple Cells (which are in essence "lightweight" Controls). For example, the buttons on the form in the YourCall user interface are actually multiple ButtonCells contained in a single Matrix. Each ButtonCell in the Matrix has its own target and action.

From within Interface Builder, you can set the target for any Control or Cell object. Interface Builder knows what methods are implemented by the target object, and it lets you select one of the appropriate methods as the action. For example, when you hook up a ButtonCell object to its target and select an action, the target's action method becomes the ButtonCell's response to a mouse click.

Controls and Cells are useful because they already implement complete event-handling behavior; all you have to do is hook them up. For an example of hooking up objects, see "Step 5: Connecting Objects in an Application" in the "Step-by-Step Through a NeXTSTEP Application" section earlier in this guide.

The Responder Chain

As users work with an application, their focus shifts from one part of the user interface to another. For example, as users enters data in YourCall's form, they tab from one text field to the next. When the user performs certain actions—such as clicking "Spell Checking" in the menu—a message needs to be sent to the object that the user has currently selected.

The *responder chain* is a structure of objects that keep track of the currently selected object, enabling one object to respond to an event or other message if it can, or to forward that message to another object if it can't. Several key objects in the Application Kit (the Application object, Window objects, and View objects) inherit from the Responder class, which defines the responder chain.

Responder objects are linked in a chain that defines the sequence of objects a message should be offered to if it can't be handled by a particular Responder. The view hierarchy is used by default to build this chain; if a View doesn't respond to a message, it can pass the message to its next responder, which is its superview.

Each Window object keeps track of its *first responder,* the object that shows the current user selection. In a window with multiple text fields, it's typically the text field with the blinking cursor, the field that the user is currently editing. You don't set the first responder, users do; Views make themselves the first responder in response to a mouseDown: event.

Many messages are sent to the responder chain. An example of this is the Cut menu item. When the user clicks the Cut command, the menu doesn't know what object will ultimately respond, so it sends a cut: message to the first responder of the current Window.

Each responder in the responder chain is, in turn, given a chance to act on the message until one actually does. If the View that's the first responder doesn't want the message, the View above it in the view hierarchy is given the opportunity to respond. If that View doesn't want the message, its superview is given the opportunity to respond, and so on. If no View responds, the current Window, then the Window's delegate, the Application object, and the Application object's delegate will each in turn get the opportunity to respond to the message.

The benefit of the responder chain is you don't have to explicitly keep track of the receiver for a message; the Application Kit will keep track for you and deliver the message to the object the user has selected.

DRAWING

Most drawing in NeXTSTEP is done with the PostScript language. PostScript is a rich imaging language capable of describing any two-dimensional image. It incorporates operators for scalable outline fonts, lines, boxes, arcs, Bézier curves, full color support, and many other operations. Any PostScript figure can be rotated, scaled, and clipped to a specified portion of the output device.

Traditionally, PostScript has been used to describe the look of the printed page. NeXTSTEP does all its drawing with PostScript, regardless of where that drawing is being performed: on-screen, on a printer, or on a fax modem.

PostScript can describe any arbitrarily complex drawing. Here's an example of a figure that's easy to create with PostScript, yet would be difficult to do with most other graphics models. It demonstrates various rotated figures, an arbitrary shape filled with a color, the use of iteration to provide depth and fading to a font, and the use of a font as a clipping path for a radiant pattern. These effects were all created entirely with PostScript code:

Drawing is the true test of art.
Drawing
J.-A.-D. Ingres 1780-1867

The PostScript language is device independent, so your application has to understand only one imaging model. Output to a window on the screen is no different than output to a laser printer, a fax modem, or a phototypesetter. And because PostScript is a language, the output of your application can be saved as an ASCII file and interpreted by any application or printing device capable of handling PostScript.

PostScript and C are different languages, so you can't put pure PostScript code directly into C source code files. However, NeXTSTEP includes a complete library of C functions that correspond to single PostScript operators. In addition, a utility program called pswrap converts ("wraps") pure PostScript code into C functions that your program can call—enabling you to implement arbitrarily complex drawing code within your applications.

Adding to the two-dimensional drawing capabilities of the PostScript language, the 3D Graphics Kit provides NeXTSTEP applications with the ability to create photorealistic three-dimensional images using the RenderMan, language. The 3D Kit's N3DCamera class is a View subclass that integrates RenderMan and PostScript drawing. N3DCamera uses the Quick RenderMan‰ renderer for fast interactive drawing on-screen and the Photorealistic RenderMan‰ renderer for output to higher resolution devices. The 3D Graphics Kit thus maintains the unified imaging model provided in two-dimensional drawing by PostScript.

When Do Views Draw?

The Application Kit provides two mechanisms that Views use to draw. Views present their standard appearance to the user through *proactive drawing*. They indicate their response to user events through *reactive drawing*.

Reactive drawing is done in direct response to the user's actions; for example, a Button uses reactive drawing to highlight itself when the user clicks. It's usually performed by an event message such as mouseDown:. (It can also be done at regular time intervals for an animation effect.)

Proactive drawing is used to draw the View when it's first presented to the user, and to redisplay it when it's scrolled, or when its window is resized. It's also used to construct an image of the View in order to print it. When a View's contents change—for example, when the user edits text—the image presented through proactive drawing will change. Every subclass of View must implement the drawSelf:: method to draw itself proactively.

It's important to note how drawSelf:: is used. It's invoked automatically by the Application Kit; a View doesn't need to know when or why it's being asked to draw itself. The View simply produces a PostScript description of its current contents. The Application Kit takes responsibility for directing that output to the correct place; the Window Server takes responsibility for making sure that output is produced at the correct resolution and color values for the device producing the image.

MESSAGES BETWEEN APPLICATIONS

As you've seen, an object-oriented program consists of a network of objects that interact by sending messages to one another. NeXTSTEP extends this model so that messaging between objects in separate applications works exactly like messaging between objects within an application. Using the messaging facilities built into the Mach operating system, NeXTSTEP implements several types of object-oriented interapplication communication, each tailored to a specific area of application interaction:

- The Pasteboard class implements NeXTSTEP's extensible cut/copy/paste mechanism.
- ObjectLinks let users assemble documents from multiple sources that automatically update to reflect changes in their source components.
- Drag and drop protocols define a simple, graphical user interface for data sharing between applications.
- Interapplication Services enable applications to offer their facilities to other applications through a standard, context-sensitive menu.
- The Distributed Objects system provides a general mechanism for implementing peer-to-peer and client-server communications between applications.

With these facilities, NeXTSTEP extends object-oriented benefits to a higher level. Just as a network of objects interacts to create powerful applications, a network of NeXTSTEP applications can interact to give users even greater power and control.

The Pasteboard

The Application Kit's Pasteboard class allows easy transfer of data both within an application and between applications. Pasteboard objects give applications access to the pasteboard server, a centralized data server shared by all applications.

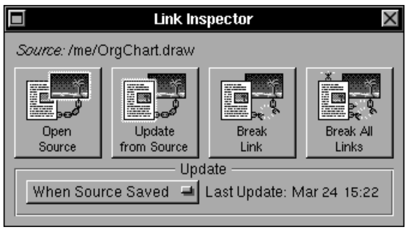
An application can use different pasteboards in the server to cut and paste different types of unrelated data: for example, one pasteboard can be used to cut and paste text, another to cut and paste font types, and a third for paragraph formatting information. Each pasteboard is represented by its own Pasteboard object, and each one can store multiple representations of related data. For example, the selection pasteboard could store both a PostScript description and a bitmap representation of an image. When an application puts several different representations of data on a pasteboard, other applications are able to select the richest data type they support for pasting.

Object Links

Object Links extend the Pasteboard's cut/copy/paste mechanism to let applications share data dynamically. Object Links are designed to support documents assembled from a variety of sources—reports from a database application, graphics from a spreadsheet, text from a word processor.

Object Links provide a simple programmatic interface to inter-document communication. LinkManager objects provide the mechanism for interapplication communication between documents. When the user cuts or copies data from a document, the LinkManager puts a reference to the origin document and application on the Pasteboard. When the user performs a Paste and Link operation in another document, the receiving document both accepts the data and registers the linked selection and its source. You simply create a LinkManager for each document, and the manager handles this for you.

Through the Link panel, Object Links also provide a simple user interface for link management.



Users can choose to update linked data manually or dynamically. They can open the source document and application from the destination document, or they can break the link so that the destination is no longer updated when the source is modified.

Drag and Drop

The dragging protocols let your application accept data and documents created in other applications and send data and documents to other applications, through a simple “Drag And Drop” user interface. The dragging protocols provide ways to define the types of data your documents accept and receive, and the image to represent document-specific data types your application can share.

Providing Services to Other Applications

As demonstrated in “Extending the Advantage,” any application can provide services to others by means of a dynamically-updated, context-sensitive menu. Any application can take advantage of services provided by other application through this same menu. No application needs to know in advance what services will be available in order to use them. Instead, applications simply identify the data type they’re currently working with—ASCII text, PostScript, TIFF images, and so on—and the Services menu offers the appropriate services.

To provide services to other applications, an application registers the pasteboard data types it’s willing to receive and return in order to provide its services. Other applications, service clients, are then periodically queried as to the types of data they are willing to place on, and receive from, the pasteboard.

For example, an optical character recognition (OCR) application could make its services available to change bitmap images into editable ASCII text. A menu item for the OCR application would then be created in every application that has a Services menu and can place bitmap images on the pasteboard and receive ASCII text in return. When the user selects an image in a client application, the OCR menu item is enabled.

If the user clicks the OCR command, the image is copied to the pasteboard and pasted into the OCR application. The OCR application converts the image to ASCII text and places the text back on the pasteboard. The text is then pasted into the client application, replacing the bitmap.

Once a service-providing application registers itself with the services system, its services are available to all NeXTSTEP applications. This extends the usefulness of applications you create, and enables your application to take advantage of the features of other NeXTSTEP applications automatically. To the user, it appears that the service is simply built into every application.

The Distributed Objects System

The Distributed Objects system provides a messaging model between objects in different applications that’s exactly the same as that between objects within a single application. Using Distributed Objects, applications share Objective

C objects, even among applications running on different machines across a network.

In the Distributed Objects system, an application initiates a communication process by registering an object offering to communicate. Other applications respond to that offer by requesting access to the registered object. When access is granted, the requestor simply sends messages to that object. On both sides of this dialogue, the syntax for the interaction is exactly like standard Objective C messaging.

The Distributed Objects system is particularly useful in developing *cluster applications*—closely related applications designed to work together, but coded and deployed individually. Cluster applications help you balance short-term demand with long term goals; they let you develop applications that implement the most needed functionality first, then extend that functionality by adding new applications to the cluster. The Distributed Objects system is also useful in implementing applications that let users on a network work cooperatively on a single document.

SUMMARY

The purpose of the NeXTSTEP development environment is to allow you to write rich, functional applications as easily as possible. You don't spend your time coding an interface, because Interface Builder does it for you. You don't need to write low-level event-handling code; you simply define responses to events your application receives as Objective C messages. You don't write a text engine, because the Application Kit supplies a powerful object for text entry and editing. You don't rewrite your application to do printer output, because the device-independent PostScript drawing that goes to the screen is equally suitable for producing printed output. You don't create special mechanisms for making applications work together, because NeXTSTEP provides those mechanisms for you.

In short, your efforts in writing a NeXTSTEP application are limited to those parts of the application that are actually unique. You don't spend a lot of time recreating work that has already been done or fighting the limitations of a confining software architecture.

Finally, because NeXTSTEP supplies you with an object-oriented development environment, your own code will tend to be both robust and reusable, thereby maximizing your long-term productivity.