NeXT versus Sun: a Comparison of Development Tools

Executive Summary

The tools used for developing applications on NeXT™ and Sun® systems appear on the surface to be similar. Sun has many tools that serve roles similar to their NeXTstep™ counterparts. On closer inspection, however, the Sun tools are quite different.

Developers using both platforms have found that Sun tools lack essential, timesaving features. NeXT provides many features that can be used by applications with no additional work. Examples of these include standard dialogs, imaging, color and printer support, and a host of others. On the Sun these features are difficult (or, in some cases, impossible) to implement.

Finally, and perhaps most importantly, Sun's tools are not object oriented. None of the toolkits are designed to work with an object oriented version of C. Customization of Sun's tools is not done using any known Object-Oriented language. In some cases customization is not possible.

In addition, interfaces on the Sun are inconsistent and rudimentary, and perpetuate the notion that UNIX-based applications are hard to use. NeXTstep Applications are:

- Easy to use
- Richer in features
- More modular and easier to maintain
- Delivered in less time

Booz•Allen & Hamilton, a nationally recognized consulting firm, recently finished a study comparing NeXTstep to other platforms.[1] They found:

Over 82% of the developers and programmers surveyed ranked NeXTstep higher than other environments they had used (Sun, Macintosh®, MS-DOS®) in all major areas—development environment completeness, application quality, maintainability, and development time. . . .

100% of the respondents with Sun and NeXT workstation development experience stated that the ease and speed of development using NeXTstep was better than their experience with Sun workstations.

1. "Comparative Study, NeXTstep vs. Other Development Environments," Booz • Allen & Hamilton, Inc, 1992

I. Introduction: Development Architectures

Several common elements exist in all modern programming environments that are used to develop applications with graphical user interfaces (GUI): a window system, a toolkit, and a layout tool.

Window System — core functionality required to display graphics on the screen and receive events from the mouse and keyboard.

Toolkit — precompiled user interface elements, including windows, buttons and sliders.

Layout Tool — a program that allows the developer to prototype the user interface graphically. The prototype is then written in a form that the real application can be built without writing the code that places the windows and buttons on the screen.

II. Sun OpenWindows vs. NeXTstep: Two Different Visions

Sun's OpenWindows™ is a combination of two different windowing systems, one based on MIT's X11 Window System with drawing and event routines which conform to the Xlib protocol; and a second system, NeWS™, based on the PostScript® language. NeWS provides all of the services provided by X11, but uses a different protocol. Both the X11 and NeWS protocols coexist within the OpenWindows server.

Sun provides a number of user interface and windowing toolkits. These include two X toolkits [XView™ and the Open Look® Intrinsics Toolkit (OLIT)] and the NeWS toolkit (TNT). Sun also sells a prototyping tool, DevGuide™, which generates code for any of these three toolkits.

Sun's approach has been to provide a whole collection of discrete tools having very different origins and supporting a variety of user interfaces, toolkits, and application development environments. They have placed greater emphasis on providing variety than on providing an integrated development architecture that is designed from conception to aid programmer productivity. This difference is the crux of

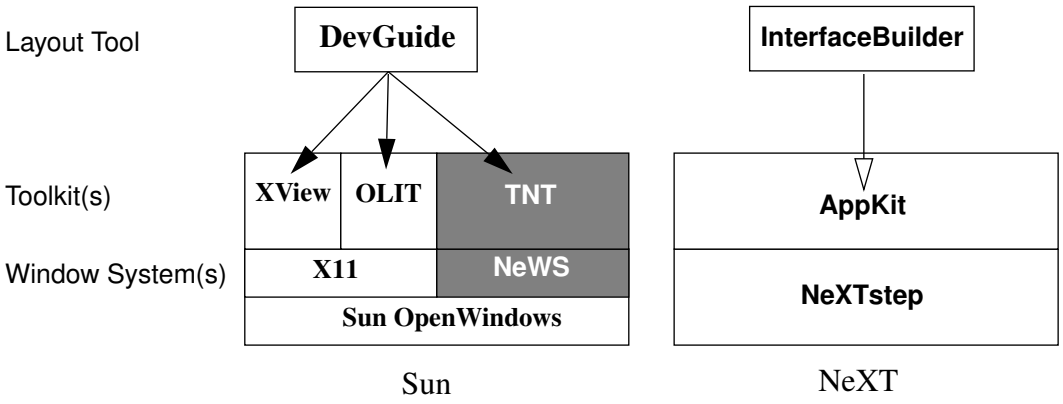

Figure 1 *Sun and NeXT Window Development Systems*

the difference between Sun's philosophy and NeXT's. The implications resulting from this philosophical divergence are far reaching.

At NeXT we believe today's application requirements are too sophisticated (and user expectations too advanced) to rely on such a mix-and-match approach to basic development tools. A shift beyond the limited toolkits and discrete tools of the '80s and the older procedural models of the '70s is needed if we are to achieve any significant advances in programmer productivity, end-user functionality and ease of use.

III. NeXTstep

NeXTstep is an integrated, consistent and complete object-oriented applications development environment. Beyond its mechanisms for creating single stand-alone applications, NeXTstep provides facilities to ensure that all applications developed under NeXTstep can cooperate with each other. NeXTstep is not merely a tool for producing applications—it is an environment in which applications work consistently and work together.

The tools in NeXTstep are tightly integrated. These tools include a windowing system, an advanced object-oriented programming language, an object editor, and an application toolkit. During their design, NeXT kept these primary goals in mind:

- Reduce substantially the time required to develop applications. NeXT believes that programmers should only have to write code that is unique to their applications.
- Encourage programmers to develop better applications by providing a rich suite of extensible objects that easily integrate advanced features such as video, sound, interapplications communications, music and multi-font text.
- Provide a consistent user interface across all applications, so all applications easier to learn and use. And to make developing user interfaces as easy as *using* them.
- Provide a single imaging model for drawing to screen, printer, fax machine, imagesetter, or other output device.

Reducing Development Time. NeXT built standard components for all common application functions such as event handling, printing, and file handling. The routines and User Interface dialogs in NeXTstep supply all of this.

Enabling Consistent Functionality Through Extensible Objects. A toolkit must be extensible in order to be truly useful. The toolkit designer must allow for situations that were not anticipated when the toolkit was originally conceived. NeXTstep achieves this by using a fully object-oriented (thus extensible) language—Objective C.

NeXTstep's set of Objective C objects implement standard kinds of functionality, and allows the application writer to customize these objects by subclassing them. For instance, the standard NeXT Font Panel allows the user to choose a single font; but if the developer is writing a program that will compare two or more font faces, then it is necessary to pick multiple faces at the same time. In its prebuilt state,

the standard Font Panel cannot accomplish this. But a subclass of the Font Panel can provide the multiple selection capability.

Creative application developers will always need to provide functionality beyond that anticipated by the toolkit designer. By allowing subclasses of existing objects (rather than requiring the creation of new code), the behavior of the customized objects is guaranteed to be consistent—even as those functions become more advanced. For instance, adding scaling or rotation, for example, to a standard print panel extends the capabilities already there within a familiar user framework.

In addition, when the system object is extended, every application relying on it will inherit its new capabilities. When NeXT extended the Print panel to include faxing, every application gained faxing capability at runtime, without recompiling, via NeXTstep Objective C's run-time binding.

NeXTstep combines extensibility with consistency—a winning combination for any developer.

Consistent User Interface. Standard dialogs make user interfaces inherently consistent, since each application is actually running the same code that provides these services. Every application can call the same Print Panel. It's easy for application developers to program to a common user interface standard, and less likely that the programmer will create an inconsistent interface.

Single Imaging Model. In contrast to Sun's variety of imaging libraries (X, NeWS, Pixrect, XGL), NeXTstep is built on a single imaging model. NeXTstep uses Display PostScript® (co-developed by NeXT and Adobe) to draw on the screen, rather than X or NeWS protocol, thus insuring that what is drawn on the screen is exactly compatible with the widest range of output devices, from desktop printers to imagesetters.

IV. Overview of NeXT and Sun Toolkit Functionality

The NeXT Application Kit

NeXT's toolkit is the Application Kit™ (AppKit). The AppKit is an object-oriented toolkit that is integrated with NeXT's object-oriented application builder: InterfaceBuilder™. The AppKit gives developers several kinds of extensible object-oriented building blocks:

Windows	All user interface objects that other systems have, plus many objects having significantly more functionality. Thus, NeXT's standard text object provides a multi-font text editing and spell-checking as part of standard functionality.
Event Handling	Objects managing responses to all keyboard and mouse events.
Media Integration	Objects for PostScript on the screen (**View**), manipulating sound (**NXSound**), video (**NXLiveVideoView**), and printing/faxing.
Data Exchange	Built-in support for industry standard data formats including TIFF, Encapsulated PostScript (EPS), JPEG compressed images, and Microsoft's Rich Text Format (RTF).
Common Dialogs	All of the dialogs (and the code implementing this functionality) common to all applications: panels to open, select, and close files, choose fonts, choose printers, and select colors.
Resources	Interfaces to resources available to the application: the font objects, NXColor, NXPrinter and many more.
Interapplication	Objects such as Speaker/Listener and pasteboards that allow applications to communicate with each other, as well as methods (Services) that let applications take advantage of a new application's functionality transparently, without re-compiling.
Printing	A complete architecture for application printing (described below).

Figure 2 *Sun and NeXT save dialogs*

The Sun Approach

The most obvious difference between NeXT's AppKit and the many Sun toolkits is the lack of a common, object-oriented foundation. As we will show in the detailed discussion of toolkits in a later section, none of the Sun toolkits uses the same object-oriented language that a programmer might use to develop the rest of the application.

Sun toolkits simply provide rudimentary user-interface elements, an event system, and access to window system resources. Comparing Sun's toolkits to the AppKit features listed above shows that the Sun toolkits provide some elements of windows (user interface objects) and event handling, while many of the features that are common to modern applications but are the most difficult to implement—media integration, data exchange, common dialogs, resources, and printing—must be designed and implemented completely by each programmer.

Data Exchange. Sun's toolkits do not specify which data types ought to be supported. As a result, Sun applications typically only allow ASCII text to be pasted from one application to another, or only allow files to be dragged in from the FileManager. Multifont text, graphics, and sound cannot be cut and pasted between applications. Unlike NeXTstep, a user working on a graphic in a drawing program cannot paste the image into a word processing document by merely clicking Copy and clicking on the word processing document, and clicking Paste.

Common Dialogs. Sun's toolkits do not supply dialog objects for operations that are common to all applications. Lacking such standard dialogs forces each developer to create this code, much of which may be difficult to implement, for each operation. As a result, most Sun applications have inconsistent panels for such functionality; and because of the difficulty in creating such functionality, what is provided is very rudimentary.

Typically the user must remember the pathname, then type it in completely and correctly when saving a file—thus reinforcing the perception that all UNIX-based applications are hard to use. For example, Figure 2 shows the Save dialog from Sun's TextEdit editor on the left. Contrast this with the Save panel supplied in NeXTstep, shown at the right. In the NeXT dialog, a hierarchical browser allows the user to scroll through the filesystem. A type-in field allows the user to type the filename in, if that is easier (in this field he only has to type in part of the filename—the rest will be completed by the system).

A Sun programmer trying to provide this same level of consideration for the user faces vastly more work as he must design, develop, and debug this code all for himself. In NeXTstep, adding such advanced ease-of-use functionality is itself easy; in fact, it is the easiest part of creating applications.

Interapplication Communication. Sun recently announced it is providing communications between applications with a separate C library (ToolTalk™). As one would expect in an environment in which tools are thought of as many separate entities, Sun's ToolTalk is not integrated with the Sun toolkits. Now that it's available, some Sun software vendors promise that their applications may be able to send and receive ToolTalk calls sometime in the future.

In contrast, NeXTstep provides a uniform message architecture, resting on top of its UNIX-compatible Mach operating system. The communications architecture of Mach was designed to support application and object messaging. And on top of Mach, NeXT has constructed a common messaging protocol based on a message sender object (Speaker) and a message receiver object (Listener). Speaker and Listener objects, part of every application, have been part of NeXTstep since its first release.

Figure 3 *Sun and NeXT print dialogs*

As a result, all NeXT applications allow other applications to send them a message to open a file and display it on the screen (openFile:ok:).

Capability such as this allows programs such as Digital Librarian™, and Lighthouse Design's Diagram! ™to ask external applications to open files and display data. Since this openFile is the same message sent by the Workspace to applications to open files, all applications that handle files respond to this message today.

Figure 4 *The NeXTstep Font Panel*

Resources. Although Sun's font support allows programmers to access font metrics, no built-in support shows the programmer exactly which user fonts are installed and available in the system. To discern which fonts are on the system, programmers must check the font directory very carefully, since this directory also includes several internal system fonts (e.g., to implement Open Look) that shouldn't appear in a list of available user fonts. Once the font is found, Sun fails to provide prebuilt tools for its selection or display—the developer must still design, develop, and debug those panels.

In contrast, NeXTstep provides the programmer with Font, FontManager, and Font Panel objects which give complete information about all fonts available on the system—as well as prebuilt objects to select and display them. Figure 4 shows the built-in Font Panel, that allows the user to pick the family, typeface, and size from all fonts available on the system. The developer does not have to write any code.

Printing. NeXT's AppKit provides an integrated architecture for printing, based on Display PostScript. Because PostScript is a standard imaging model for printers, all drawing done by application programs is immediately printable to a wide range of hardcopy devices. All of the application's rendering is based on the same PostScript code—whether to the screen, printer, or even fax machine.

On top of this elemental Display PostScript imaging model, the AppKit provides:

- Complete built-in dialogs for selecting a printer from the printers available on the network and laying out a page. Sun applications that choose a printer usually display a panel like that shown in Figure 3, requiring the user to remember which printers are available. In networked offices with many cross-functional teams where workers frequently visit different sites, this interface proves especially problematic, since users coming from another site will have no idea which printers are available locally.
  In contrast, NeXT systems show all of the available printers (as shown in Figure 3), along with an indication of their type. And using NeXT's PrintManager application, the user can make his printer

available to all users in a local network domain. Whenever a printer is added to the network, it automatically appears on the Print panel of all applications running on all machines in that domain.

- Standard NeXTstep Print panel features also include n-up printing (multiple miniature pages printed on a single page), print previewing (and the saving of previewed page images as PostScript files), and faxing.
  To send a fax, the user simply presses the Fax button included in every application's Print panel. After a fax request is initiated, the panel shown in Figure 5 appears, allowing the user to select from a list of commonly used fax numbers. Additional panels allow the user to choose a fax machine and fill in a cover sheet.
  Such functionality is provided via the AppKit for every NeXTstep application.
- Complete, built-in dialogs for changing fonts, plus multifont text objects based on industry-standard Adobe fonts. Sun's font technology not only lacks the font management and selection services provided by NeXT on all applications, but is based on their own proprietary, nonstandard F3 format. Thus, Sun documents are not necessarily "what you see is what you get" when printing to Adobe PostScript devices.
- Printing is built into all applications. When the user prints, he simply asks the application to display various parts of the current document on the printer, just as it would appear on the screen. The rendering is sent to a printing context which is spooled off to the printer chosen by the user (through the Print panel). Since even complex operations, such as pagination, are managed by the AppKit's printing routines, printing can often be added to an application in minutes.

In Sun's environment, in contrast, much of this is lacking:

- Most Sun development is done in X, which offers no printer support. Sun does offer the option of rendering in NeWS, a PostScript-like language, but it is printable only on its own Sun NeWS-Print™ printers.
- No standard dialogs are provided for choosing printers or fonts, or for publishing the availability of printers on a network. Sun uses the /etc/printcap file, which is different on every system.

Figure 5 *The NeXTstep FaxPanel*

- Sun offers no application-based printing. The application must create a file in the file system, then issue the appropriate UNIX command.

Table 1: NeXTstep versus OpenWindows features

	NeXT	Sun
Window (user interface)	√	√
Event handling	√	√
Media integration (rich text, graphics, sound, video)	√	
Standard data exchange formats	√	
Resources	√	
Interapplication communication	since 1.0	available in future
Application printing	√	
Standard panels for choosing printer	√	
Preview support in every application	√	
Fax support in every application	√	
Easily "publish" printers to network	√	
Font Panel	√	
Fonts	Adobe	Sun F3
Imaging model	Adobe PostScript	X, NeWS PostScript

Figure 6 *Interface Builder Palettes. Programmers can choose from an array of different kinds of palettes like the ones shown. Choosing one palette reveals its own collection of objects. The programmer then merely drags relevant objects off the palette and designs the user interface.*

V. Layout Tools

Tools to lay out user interfaces graphically are available from both NeXT and Sun. These increase programmer productivity in the following ways:

- Layout tools allow programmers to preview the user interface without wasting time on the run-edit-compile loop.
- Such tools simplify the process of constructing an interface so more team members, including non-programmers, can be involved in the user interface design.
- These tools improve the overall quality of applications. As layout becomes easier to change, programmers can more easily refine applications to meet the requests of users. In some organizations, graphic artists lay out the user interfaces—something that was impossible before the advent of graphical layout tools.

NeXT's interface development tool, Interface Builder, was developed together with the AppKit. Interface Builder's design goal is to describe the interfaces between components (objects) of an application. In this sense, the word interface is confusing, since it does not refer, simply, to user interface components. A more accurate name for Interface Builder might be Object Editor, since Interface Builder allows the programmer to:

- Create objects, including user interface objects, as well as other kinds of programming objects that provide underlying functionality to the application.
- Make connections between any two objects, either to inform one object of another object's existence, or to send messages from one object to another.
- Examine the object hierarchy in the AppKit and graphically subclass existing objects, producing appropriate code.

Using Interface Builder

Interface Builder presents the programmer with a complete palette of all of the AppKit user interface objects. Figure 6 shows some of the palettes available in Interface Builder. The palette on the left is a collection of user interface objects. In addition to these familiar objects, Interface Builder provides a standard set of prebuilt menus (shown in the center, above), a multilevel browser object, and a multifont scrolling text object (shown on the right).

Within Interface Builder, the programmer can view the AppKit hierarchy in a window that displays all of the classes in the application—including all of the AppKit's classes. A typical class viewer is shown in Figure 7. After selecting a class, the programmer can subclass an object, add instance variables and methods, and eject the Objective C code for the new subclass. By double-clicking on the class name, Interface Builder also allows the programmer to view the header file for the class.

Figure 7 *Interface Builder's class viewer*

Figure 8 *Connections in Interface Builder. On the left, setting the action for a button. On the right, setting the "custNumber" outlet*

Custom Objects

With the object subclassed, the programmer may wish to package the object in a form that other developer drag off the palette and use, just like any other AppKit object. Interface Builder is unique in providing this extensible functionality. The programmer simply adds a few methods, and packages the object(s) into a palette. This palette, and the compiled code for the underlying objects, can then be distributed.

Figure 9 *Custom Palettes. This is a palette sold by Objective Technologies providing a set of graph-making objects, which are used exactly like AppKit objects, and are dragged off the palette to be used.*

This extensibility, the ability to create new palettes of objects, is a very powerful facility, because it allows programmers to create many kinds of user interfaces or other types of objects that can be loaded into Interface Builder and then accessed out like any of the NeXT AppKit objects. Using this, a department could create database query objects containing prebuilt query forms for a corporate database.

Since these palettes are distributed as compiled code, software companies such as Objective Technologies and RDR can distribute and sell custom palettes rather than a complete shrink-wrapped product having fixed functionality. Figure 9 shows a palette sold by Objective Technologies.

Connections in Interface Builder

Interface Builder supports two types of connections. The most obvious type is an action which corresponds to a callback in a more traditional C programming environment. A programmer may, for instance, create a button on a panel that logs the user onto a database when the button is pressed.

In the example shown in Figure 8, a database object handles communication between the client user interface and the host database. Once a connection is made in Interface Builder between the button and the database object, all of the appropriate messages to the database become visible so the programmer can select which message will be sent to the database object when the button is pressed. This goes beyond the capabilities of other tools, since here the programmer's objects are accessed within Interface Builder, and the messages and methods created by the programmer can be accessed with Interface Builder and connected to the appropriate user interface objects.

Figure 10 *The DevGuide main window*

Interface Builder also lets the programmer construct new objects and create instance variables. Instance variables describe the other objects that a given object can access. An example of this is shown on the right in Figure 8. Here, the database needs access to a type-in field that receives input and displays results. For this the database uses an instance variable called custNumber which is hooked to the type-in field. The connection between the instance variable and the type-in field parallels the two components of an object in an object-oriented language: data and code (or instance variables and methods).

VI. Sun's DevGuide

Sun's DevGuide is a user interface layout tool for all Open Look toolkits. DevGuide can eject code for all of the three Sun toolkit described earlier.

Using DevGuide

Like Interface Builder, DevGuide presents a palette of user interface elements to the programmer—the set of Open Look elements. Immediately one difference is obvious: all of the Open Look elements are all delivered on one panel, rather than in separate functional views. A detailed comparison of Interface Builder and DevGuide reveals that the similarities between these tools are superficial. In contrast to Interface Builder, DevGuide lacks the following capabilities:

Menus DevGuide allows menus to be created, but provides no standard, prebuilt menus like Interface Builder does, such as Info, Document, Edit, Format, Windows, Services, Font, Text, Find, and Colors. In NeXTstep, functional menus contain items, such as font and color selection, which function as soon as you pull them off the Interface Builder palette. These types of functional menus do not even exist in DevGuide.

Matrix Interface Builder simplifies layout design by letting you drag any user interface item (such as a button or a field) into evenly spaced matrices using the mouse. But in DevGuide, each item must be laid out and aligned separately.

Browser While Interface Builder includes a multicolumn browser for navigating through hierarchical data, DevGuide (and the Open Look toolkits) only provide a single-level scrolling list, which is inadequate for navigating through hierarchies of information, including the UNIX file system.

Text Both tools provide an editable text view, but DevGuide only permits monofont text. NeXTstep provides a text object that is a complete word processor, having rulers, spell-checking, the ability to read and write Microsoft Rich Text Format (RTF), and dialogs and menus that can control all font and tabs/ruler manipulations.

Palettes DevGuide only allows you to access the prebuilt interface elements supplied by Sun, shown in Figure 10. In contrast, Interface Builder lets you access custom palettes that you have either purchased or created. DevGuide does

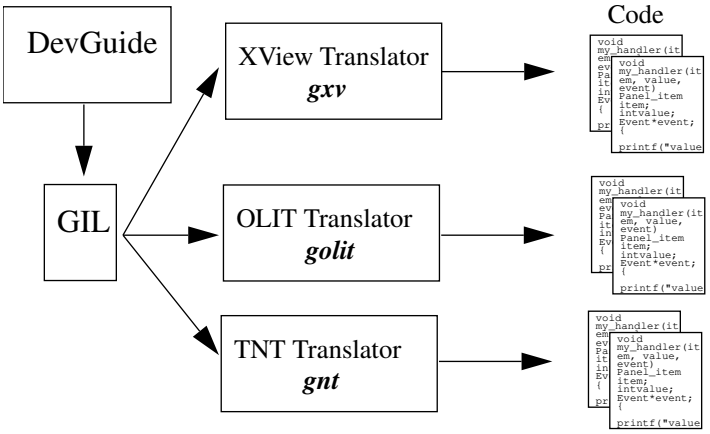

Figure 11 *The translation of DevGuide into source code*

not allow you to access elements that have been customized or extended.

Code Generation

To facilitate language and toolkit independence, DevGuide produces code written in an intermediate language: the Guide Interface Language (GIL). GIL describes the positions of user interface elements, and stores the name of an associated callback. This GIL language can then be converted by a number of translators into a language (C or PostScript) that is appropriate for a given toolkit (XView, OLIT, or NeWS), as shown in Figure 11.

In DevGuide, the programmer lays out the user interface and assigns callbacks, then the code is ejected. Generally two separate modules of code are ejected by the DevGuide translator:

xxx_ui.c C routines that create the user interface designed in DevGuide. This routine has a single entry point that creates the elements. Usually the programmer never touches this file, which is the equivalent of the Interface Builder nib file except it is program code rather than a binary archive.

xxx_stubs.c C routine templates for the callbacks. This file must be edited by the programmer.

DevGuide is not integrated with any object-oriented language, such as Objective C or C++, although Sun has provided a *gxv+* procedure which ejects C++ code for the user interface module.

This merely allows the user interface portion—the code that the programmer never modifies—to be in C++. The stubs files—which the programmer does modify—are ejected by gxv+ in plain ANSI-C.

Since a single stubs module is created for each DevGuide file, the modularity of a DevGuide application is tightly coupled with the DevGuide files. To break the program into smaller modules, the programmer must create separate DevGuide files for each different part of the user interface; for instance, he would create a DevGuide file for each window.

DevGuide can force the programmer to create different files for each window in order to keep these stub files from growing too large. When you are trying to design an entire user interface, this process can be quite confusing.

By forcing the developer to modularize the program around the components in the user interface, DevGuide does not permit functionality to be grouped into larger logical units. Instead, DevGuide forces the developer to break up modules based on user interface boundaries.

In Interface Builder, all of the code that generates the user interface is placed in the nib file. Callback routines consist of methods that can be related to any number of Objective C objects. A single interface file can contain a number of small objects that

handle various aspects of the interface. Or, if it makes more sense to have the callbacks for a number of nib files handled by a single object, this too is possible.

Summary: Comparing Interface Builder and DevGuide

ESL, a subsidiary of TRW recently completed a study in which they developed three applications on both NeXT and Sun platforms.[1] Using DevGuide on Sun and Interface Builder on NeXT, ESL concluded:

It quickly became clear that Interface Builder and NeXTstep offered a rich and flexible paradigm. Design sessions on the NeXT focused on the analytic task and requirements. Suggestions from users could be quickly implemented and modified on the NeXT. DevGuide provided more limited user interface options, and the DevGuide interface limited the interactivity of discussions with users.

In summary, the NeXT prototyping tools provided clear insight into the final appearance of the application early on and allowed for rapid modifications at this early stage. The Sun tools, however, enabled only some aspects of the user interface features to be prototyped. The behaviour of other parts of the interface would not be seen until much later in the process, when changes were more difficult.

Using DevGuide, the programmer can:

- Design the user interface and preview it.
- Assign the names of callback routines.
- Eject C code

Using Interface Builder, the programmer is able to:

- Design a user interface, and preview it.
- Assign callback methods, and create the objects which contain these methods.
- Subclass objects in the AppKit/Objective C hierarchy (including non-interface objects).
- View class definitions.
- Define the interfaces between all objects in the application.
- Eject Objective C code and a separate binary file containing the user interface.
- Perform project management tasks, including adding non-user interface modules, TIFF images, sounds, additional libraries, and allow Interface Builder to create the Makefile.
- Run Make.
- Using loadable palettes, load custom objects into Interface Builder and lay them out exactly like prebuilt AppKit objects.
- Enable the creation of a modular, maintainable application by allowing the programmer to break the task into objects that are logical units of functionality, not predetermined user interface boundaries. DevGuide makes it impossible to design programs in logical units since modularity is dictated by the user interface.

These differences are summarized in Table 2.

1. "Developing Custom Applications in a Heterogeneous Environment: A Comparison of NeXT and Sun" presented at the NeXT Federal Expo Technical Conference, Reston VA 5 December 1991.

Table 2: NeXT Interface Builder vs. Sun's DevGuide

	NeXT IB	Sun DevGuide
User interface layout	√	√
Connections to non-user interface code	Object methods	C function callback
Creates code	Objective C	C
Creates Makefile	√	√
Built-in menus that work straight off the palette	√	
Ability to make Matrices for user interface elements	√	
Project management for all of the program's files: code, images, sounds	√	
Integrated with object-oriented language	√	
Subclassing	√	
Access to toolkit class definitions	√	
Customizable palettes	√	
Integrated with Make	√	
Allows reasonable modularity	√	
Multiple modules in one interface	√	
Single module in multiple interfaces	√	

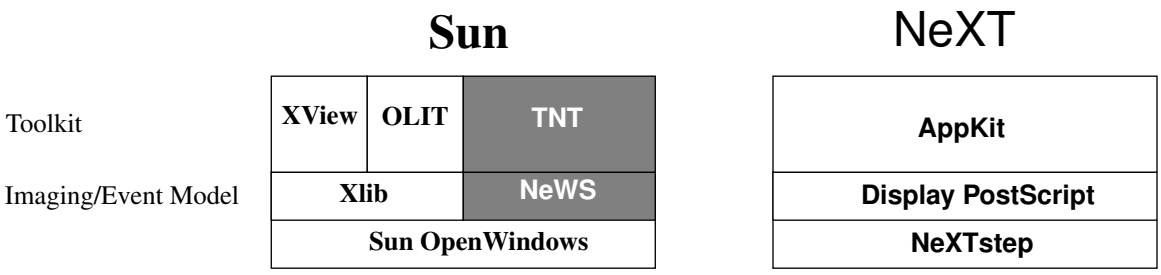

Figure 12 *Sun and NeXT Window Development Toolkits.*

VII.Detailed Discussion of Toolkits

Introduction

This section compares the toolkits available for Sun's Open Look development with NeXT's AppKit. Differences are noted in the following areas:

- Programming interface
- Maintainability
- Object-oriented approach
- Acceptance level with the developer community and commitment of the computer manufacturer

Sun's Toolkits

On Sun's OpenWindows, the primary toolkits used for Open Look development are:

XView Sun's primary toolkit. Started as SunView® (the window toolkit used under Sun's proprietary window system), it was later ported to X.

OLIT The Open Look Intrinsics Toolkit, written by AT&T. This toolkit was built on the X Intrinsics, and as such, it resembles the Motif® programming interface. OLIT is gaining acceptance, primarily among third-party software developers who want to create applications that are portable between Sun's Open Look and other machines running Motif.

TNT The NeWS Toolkit, Sun's newest toolkit. TNT combines an object-oriented window interface with the NeWS window system, which uses an extended PostScript language and an imaging model that can be printed on Sun NeWSPrint printers. This toolkit has been around in early versions for sometime, but has not gained much acceptance for a number of reasons.

While other toolkits for X Windows™ are available from third parties, XView and OLIT represent most of the development work being done today on Sun products. In addition, the toolkits described above are the toolkits supported by Sun's DevGuide, a tool that can be used to lay out user interfaces.

Toolkits available for Motif development are beyond the scope of this paper. Although Sun is the number one supplier of workstations using Motif, Sun does not support any of these toolkits, nor does it support the use of the Motif Window Manager mwm.

Table 3 describes the support and acceptance level of these toolkits.

Table 3: Toolkit Support and Acceptance

Toolkit	Users
NeXT AppKit	Used by all developers and NeXT
XView	Primary toolkit Sun uses for product development
OLIT	Almost all third party development is done in OLIT, due to similarity to Motif
TNT	Not widely used by either Sun or developers.

Object-Oriented Approach

At one time or another, Sun has claimed that all of these toolkits are object oriented—and in one sense they are. Sun argues that each toolkit contains user interface elements (objects), and a restricted calling library that allows the programmer to change

NeXT

Figure 13 *A typical NeXTstep application is composed entirely of objects.*

aspects of these objects. These calling conventions are referred to as methods (borrowing a term from object-oriented programming). In addition, the internal construction of these user interface elements represent an inheritance tree of objects. The new XView documentation makes a special point of showing how these objects are related: a panel is a window that is a drawable object, and so on. Since these toolkits are simply C libraries, however, it's difficult to believe they are truly object oriented. If Sun's claims were true, all user interface toolkits would be object oriented.

All user interface toolkits must provide user interface objects, and they always have a restricted language to manipulate the attributes of these objects. Since an inheritance tree is typically used in the design of such a toolkit, object-oriented programming is ideally suited for the implementation of the internals of a user interface toolkit.

Of course, the technology used in the internals of the toolkit is less important than the API presented to the programmer. Here are a few key questions to ask about the object-oriented nature of any toolkit:

- Does the toolkit use the same object-oriented language that the programmer might use for the rest of the code, or does it force the programmer to learn a different one?
- Is the toolkit aware of the paradigm being used for nonuser interface objects? In other words, can user interface objects send messages to nonuser interface objects, and vice versa?

The issue is how well the toolkit is integrated with traditional object-oriented programming languages. What language/syntax do the various toolkits use?

XView uses a C Library instead of a programming language to communicate with user interface elements. Subclassing is difficult, involving the creation of a custom XView Library—so subclassing is rarely used.

OLIT also uses a C Library. Subclassing is done with the Widget syntax used in Motif. Although the Widget syntax was designed to be more extensible than XView, it is still not a programming language.

TNT uses an Object-Oriented language, NeWS PostScript.

Why is this important? Because it's easier for the programmer to learn only one set of techniques for extending objects. Functionality can be migrated from user interface objects into non-user interface objects when it's more logical to do so. This all creates an environment where program modules are broken up into logical, reasonable, designed pieces. The resulting code is better designed and more readable, and thus more maintainable. Maintaining software is the most expensive part of the software business.

Figure 13 illustrates the complete object-oriented nature of NeXTstep. On a NeXT computer, all programming can be done using Objective C and C++ objects. Since the user interface objects

Sun

Figure 14 *On the Sun, only the programmer's own code can be objects, the rest of the application is a mixture of C language modules and custom widgets.*

of the AppKit are simply Objective C objects, they are extended using Objective C subclassing (as the Custom UI Object in Figure 13 on the previous page illustrates). All of the engine code—the portion of the application that's independent of the user interface—can be coded in either Objective C or C++. And all interactions between objects in the programmer's code and either the AppKit or other applications are merely object-oriented messages, not C Library calls.

How Well Integrated are the Sun Toolkits With Object-Oriented Techniques?

On Sun, toolkits are not written using object-oriented technology. The interface between the programmer's code and the toolkit consists of a set of C function calls, illustrated in Figure 14.

XView and OLIT use C function callbacks to communicate with other modules within the application. These toolkits contain no notion of sending a message to a given object. This flat namespace of C functions causes large programs to have problems with modularity and maintainability (spaghetti code).

The NeWS Toolkit is similar: a callback is permitted to be a PostScript procedure. This can be a call to a method in a PostScript object, or it can send a token back to the C client (which effectively allows for a C language callback). There is no support for object-oriented C callbacks in TNT.

Interfaces between the objects in the programmer's engine code can be messages, and code can be added to the DevGuide C stubs module so messages can be sent to these objects—but all other transactions are simply C Library calls. The interface to the NeWS Toolkit, as well as access to other applications through ToolTalk, is all done using C, not an object-oriented language.

Customization

As illustrated Figure 14, the Sun Toolkits extend objects not through Objective C or C++ subclassing, but through techniques unique to each toolkit. XView lets the programmer supply a custom XView C Library, OLIT allows the developer to write Custom Widgets, and TNT uses PostScript, rather than Objective C, subclassing.

In addition, none of the Sun toolkits allows customized objects to be used with the layout tool, DevGuide.

The difficulty of customizing makes the idea of component software, where a third party sells custom user interface elements or a department creates a set of standard user interface elements, very unattractive using these toolkits. In contrast, a number of firms are doing just this for NeXT.

NeXT's AppKit

Background

NeXT's AppKit is a complete object-oriented application framework that provides everything common to applications, including:

- A complete environment for Printing.
- Standard dialogs for most of the standard functions needed by applications, such as opening and saving files, choosing colors and fonts, printing, and faxing.
- Built-in support for TIFF and EPS standards, including imaging files on the screen as well as on the printer.
- A scrolling text object that reads and writes Microsoft RTF files. This object is almost a complete word processor, and includes all of the user interface required to change the font, manipulate ruler settings, etc.
- Methods for interapplications messaging, including standard messages recognized by all applications.
- Support for standard pasteboards for exchanging data between applications, including ASCII, multifont text, TIFF, and sound.

Programming Interface

NeXT's AppKit is based on the Objective C language. All user interface elements (such as buttons and windows) are implemented as Objective C objects. This means that the syntax used to create the other objects in a program is the same syntax used to manipulate AppKit objects.

Let's look at a simple example of this. Objective C denotes a message by putting it in square brackets ([ ]). The message new is sent to a class to generate a new instance. In the code below, an AppKit listener object is generated; then a delegate is assigned to be a new instance of my own custom class called MyObject.

```
id listener = [Listener new];
[listener setDelegate:[MyObject new]];
```

Although this is a very simple demonstration of the subject, it does illustrate another point also: since the AppKit is built on an object-oriented language, it is built to take advantage of objects. In the example above, an AppKit object was sent a message that set its "delegate" object. The AppKit objects will send a variety of status messages to their delegate objects. To accomplish this in a non-object oriented system, you would have to set a callback function for each of these status changes. AppKit gains these advantages easily through its object-oriented approach:

- A single call sets up an object (delegate) which is informed of all status changes
- Setting up a plethora of callback functions is missing all of the modularity that is a key advantage of object-oriented systems.

This approach also offers the possibility of a richer protocol. For instance, the delegate for a TextField receives messages when the text is about to change, after it changes, after each key stroke, and when the user is finished with the field.

Feature Set

One of the most distinguishing features of AppKit are the built-in standard dialogs, which exist for all aspects of functions such as printing, file operations, and choosing fonts and colors. For example, here is the code for a program that wishes to show the user a dialog for saving a file:

```
id save = [SavePanel new];
char resultFile[MAXPATHLEN];

[save  runModalForDirectory:aDirectory
       name:""];
strcpy(resultFile, [save filename]);
```

The resulting panel is shown on the left of Figure 15 as it would actually look on the user's screen. All of the setup of this SavePanel was accomplished with four lines of code, yet this is a relatively sophisticated object: capable of reading the contents of the filesystem, allowing for incremental file completion, etc. All of this is so easy on NeXT, while the same is very difficult on Sun. NeXT achieves a richer and more consistent application, implemented with fewer lines of code to be maintained.

While the panel on the left was a standard SavePanel, the panel on the right side shows a slightly modified version of the same SavePanel. Here the programmer needed to let the user specify the format in which the file would be saved: Draw, EPS, or TIFF. The AppKit allowed the programmer to attach additional controls to the standard dialogs by merely adding an Accessory View with the radio buttons shown. The programmer was thus able to give the user finer control over the operation, while keeping the same overall appearance within the application.

Figure 15 *Examples of NeXT SavePanels. On the right is the standard SavePanel. On the right a SavePanel with an Accessory View added*

Customizability

The AppKit provides a number of ways to customize behavior. Here are some of the ways that AppKit objects can be modified to suit a particular application:

Subclassing all AppKit objects can be subclassed, since they are simply Objective C objects. Subclassed objects can completely replace some methods, or add some functionality and call the method in the original superclass.

Delegates many AppKit objects have delegates that offer some level of control over them. A delegate is a separate object that is notified when an AppKit object changes its state. Delegates can modify the way an AppKit object behaves without subclassing it. This is one of the unique features of the NeXT system.

Accessory as illustrated above, an Accessory View can modify any standard panel to give it additional controls.

Services offers a way for programs to embed functionality into other programs without prior knowledge. It is possible to buy a third-party shrink-wrapped program and add a feature to it without changing the program. You merely write a program that accepts input via a pasteboard (ASCII, Rich Text Format, EPS, TIFF, or Sound) and return another pasteboard with the result. Through Services, your new functionality is added to the menu of other programs without their being changed. This makes it possible to write a grammar-checking algorithm, or any program that accepts and returns these datatypes. One developer (HSD) wrote an optical character recognition system that could produce editable ASCII text from any incoming fax or TIFF file.

XView

The XView consists of C function calls with a large number options. XView uses attribute-value lists to specify these options. The lists contain a set of pairs: the name of the option and the requested setting. You don't need to mention any options unless you are changing them from their default setting. Using attribute-value lists and defaults, XView avoids the problem that some user interface toolkits have had with functions requiring large numbers of positional parameters that are hard to remember. Here is an example of how you would create a slider in XView:

```
slider = xv_create(controls,
     PANEL_SLIDER,
     PANEL_LABEL_STRING, "Send:",
     PANEL_SLIDER_END_BOXES, FALSE,
     PANEL_SHOW_RANGE, FALSE,
     PANEL_SHOW_VALUE, FALSE,
     PANEL_MIN_VALUE, 0,
     PANEL_MAX_VALUE, 20,
     PANEL_TICKS, 0,
     PANEL_NOTIFY_PROC, sl_changed,
     NULL);
```

In XView programming, user interface elements are created and then C function callbacks are associated with them. A slider may have a callback associated with an action such as the user moving it to select a new value. Each element may have one or more of these handlers. The handlers are passed relevant information about the user action that caused them to be called.

Feature Set

The biggest deficiency in XView (as with all of the Sun toolkits) is its lack of standard dialogs. XView has many of the base user interface elements that the AppKit has, but a few of the sophisticated elements of the AppKit are either missing or are present in a rudimentary form. For instance, XView has a scrolling list, where the AppKit has a multi-level browser. In the ESL study, the XView scrolling list was found lacking, causing the ESL developers to start from the very beginning:

For example, the NeXT version was initially able to display sample HyperNotes FileBins (the contents display used a standard NXBrowser object), while the Sun developers could only later create a custom canvas based object to display FileBin contents (the standard XView Scrolling List was too limited in functionality).

The AppKit text object is another clear example. ESL had problems implementing its own HyperNotes™ information system on XView:

On the NeXT, the standard text object easily provided necessary features like the display of glyphs within the text (for Link and Highlight icons) and multiple fonts and colors (to emphasize highlighted regions). This functionality was not provided by the Sun XView textsw object. An ESL developer spent approximately two months to extend the Sun text object so that it could provide the basic icon and highlighting capabilities.

To summarize XView:

- XView lacks many of the features of the AppKit—features which make delivering a product-quality product easy with the AppKit and difficult with XView.
- XView offers none of the customization capabilities mentioned in the previous section for the AppKit, since subclassing and delegation require an object-oriented language, and XView doesn't use one. Accessory Views require built-in dialogs, which are lacking in XView; and Services require exchanging multimedia data via standard pasteboard types, also lacking in the Sun system.

The Open Look Intrinsics Toolkit (OLIT)

OLIT's interface, called the Intrinsics, is the same interface that Motif and many other X Windows toolkits use. Developed at MIT, the Intrinsics consist of a number of C calls that allow access to a Widget Set.

A Widget is a piece of code that defines a user interface element. The actual Widget is coded as Xlib drawing calls, and a Widget syntax defines how various events are handled.

On the positive side, OLIT has the same functions that Motif has, such as XtCreateManagedWidget(). Motif programmers only need to pass the different Widgets to these functions. The intrinsics style is different from that of XView, in that arg-lists are constructed with XtSetArg rather than with attribute-value lists. The result is quite similar to XView: user interface items are created, and C callbacks are associated with them, as shown here:

```
Arg wargs[2];
...
XtSetArg(wargs[n], XtNsliderMin, 0);
XtSetArg(wargs[n], XtNsliderMax, 20);
w = XtCreateManagedWidget(name,
       sliderWidgetClass,
       parent, wargs, 2);
XtAddCallback(w, XtNsliderMoved,
       sl_changed, data);
```

Like XView, OLIT uses defaults to reduce the number of different options that need to be supplied by the programmer. And like XView, OLIT's text view only allows a single font to be used at any time, and does not have facilities for embedding graphics.

Feature Set

OLIT elements are slightly different in both appearance and operation from those in XView. Users will find that OLIT draws elements differently than XView. OLIT uses the keyboard to operate switches that can be manipulated only with a mouse under XView. The result is a different look: some controls are painted a different color to indicate they have the keyboard focus. Even in feel they are dissimilar because the keyboard is handled differently.

In Open Windows version 2.0, cut and paste operations between XView and OLIT applications did not work. The new version 3.0 fixes this, so now applications written in any of the Sun toolkits should be able to cut and paste ASCII text between each other.

Conclusion

All of the analyses of XView are valid for OLIT also. More third party companies are using OLIT than XView, so it is felt to be of a higher quality than XView; but the feature set is not actually appreciably different. Both toolkits use the C function callback style, and both lack advanced features such as a multifont text view or a hierarchical browser.

The only difference is that applications written in OLIT will look and work slightly different than the system tools, which are all written in XView.

The NeWS Toolkit

The NeWS Toolkit is different from these other toolkits in that it is written completely in PostScript. PostScript is a powerful, flexible interpreted language, and to it Sun has added object-oriented support. TNT uses the PostScript dictionaries as objects and classes, and a single operator: send has been added.

Here's an example of TNT PostScript code to create a slider that ranges over the numbers from 0 through 20:

```
    % Make a slider.
/slider framebuffer /new ClassHSlider
        send def

    % Turn on the end boxes.
true /setendboxes slider send

    % start out with a range of 0..20
0 20 /setrange slider send
```

The slider is created by sending the message "/new" to ClassHSlider, and supplying a number of arguments on the stack, the result is stored in the PostScript variable *slider*. After the slider is created, messages can be sent to it, above the range of values is set. Notice that the syntax is in reverse order (like the rest of PostScript):

```
<args ...> object send
```

TNT sliders range over float values by default (while XView sliders only range over integers). The programmer can make the slider send only integers by setting a normalizer procedure, which is a piece of PostScript (shown in braces below). In this case, the procedure truncates the float value:

```
    % Make values integers.
{ round cvi } /setnormalizer slider send
```

It is possible to write a complete program in NeWS PostScript. In fact, that's the way most of the demo applications shipped by Sun with Open Windows were written. Most complete applications, however, will require some access to C. To enable this, Sun provides the CPS, which is an interface between a PostScript program and a C program, much like the DPS wrap concept. In addition, Sun as added Wire Service™ to TNT, a library that implements an XView-like notifier for TNT programs. Wire Service allows events on the PostScript side to trigger C function calls on the client side.

UI	Written in TNT PostScript
C Interface	Written with CPS and the Wire Service library

A TNT program consists of some PostScript code that is loaded into the server by a C program. A special syntax is used to create C interfaces which cause the C program to initiate functions in the window server. In addition, user interface elements in the server must inform the C program when something occurs, such as when the user presses a button.

Maintainability

Unfortunately, PostScript was never intended to be a language in which large applications would be written. The Reverse Polish notation is hard to read

Table 4: Sun Toolkits versus NeXT AppKit

	NeXT AppKit	Sun XView	Sun OLIT	Sun NeWS
Extension language	Obj C	XView Internal	Widget	PostScript
Uses same extension language as rest of code	√			
Integrated with object oriented C	√			
Display custom objects in layout tool	√			
Primary tool for vendor	√	√		
Primary tool for third parties	√		√	
Standard dialogs	√			
Support for TIFF and EPS	√			
Multifont text support (RTF)	√			3.0 Library
Delegates	√			
Services (ability to add features to programs without prior knowledge)	√			

and maintain. Since the language is not compiled, there are the obvious performance problems. For reasons such as these, NeXT decided to use a C-based language (Objective C) as the basis for the windowing system of the AppKit. NeXT uses PostScript only to draw on the screen, and uses C drawing functions for most operations so the programmer is not forced to program directly in PostScript.

Security and Robustness

Implementing objects as dictionaries exposes all instance variables (and methods) for reading and writing. Programmers can easily manipulate TNT internal data structures, thus causing not only their own but also other programs to crash, since the TNT base classes are shared between all applications and are changeable at runtime. TNT is also delivered in source code form, and is loaded into the server at runtime.

Summary of TNT:

- User interface code is written in a completely different language than the rest of the application.
- PostScript was never intended to be used as a programming language for large programs. The resulting code is difficult to read, and thus hard to maintain.
- Exposing this much of the client-server connection to the programmer adds an unnecessary level of complexity to TNT programs.

Summary of Toolkit differences

A summary of differences between the Sun toolkits and the NeXT AppKit is shown on Table 4.

VIII.Conclusion

Sun's tools seem, at first glance, to be similar to their NeXTstep counterparts. When you go through the checklist of tools (e.g., a window system, a toolkit, and a layout tool), Sun has a product for each of these.

The differences become apparent, however, when the programmer attempts to produce a real application with the tools. Here the Sun tools fail.

The ESL team's experience when they deployed the applications written for both Sun and NeXT are typical. They found:

Analysts received one half to one full day of training before receiving their new workstation. This training went very smoothly on the NeXT—users learned how to operate the NeXT environment quickly, and became rapidly proficient using ESL applications. On the Sun, it took longer for the analysts to understand how to manipulate the OpenWindows environment. Analysts also had more trouble learning to use the custom applications. In part, this was because the tools were less polished and debugged. As a result of this training experience, the Sun delivery was delayed by several weeks to solve some of the problems encountered.

Within the same amount of time, the NeXT team at ESL deployed an application that was judged to be richer in features, more polished, and more bug-free. The Sun applications lacked niceties such as multiple fonts, and were delayed due to bugs found during initial deployment. Since the two teams were given the same amount of time to complete their projects, ESL's study found that:

Developers found the NeXT development environment more supportive of complex tasks. The completed applications were generally of higher quality on the NeXT, and users found the NeXT easier to begin using.

Summary

- The Sun toolkits are lower quality, less robust, and lack essential time-saving features.
- None of the Sun toolkits contain standard dialogs for opening and saving files, choosing fonts and colors, or any of the other dialogs that are part of the NeXT AppKit. Thus significant amount of extra time must be spent developing (and debugging) applications on Sun.
- The Sun product lacks an application-based printing architecture. Every application must write its own dialogs to access available printers, and paginate, then generate the PostScript for itself, and spool the result off the printer. NeXT does all of this for the application, and provides the ability to fax and preview on-screen with no extra work.
- Applications written in NeXTstep are usually richer in features such as image handling and multi-font capability due to the almost automatic nature with which these are implemented with the AppKit.

Finally, and perhaps most importantly, the Sun environment completely lacks an object-oriented base:

- None of its toolkits are constructed for use with an object-oriented C language.
- DevGuide does not allow program modules to be designed logically, since it enforces module breaks on user interface boundaries.
- Customization of toolkit elements is difficult, and requires a completely separate language than the rest of the application uses.
- DevGuide does not support hooking user interface elements to code objects (written in an object-oriented language) in the rest of the application.
- By not supporting the loading of customized toolkit elements, DevGuide makes it impossible to lay out these custom objects. In Interface Builder, custom objects written by anyone else (another department or another company) can be used by the programmer, and treated as though they were NeXT AppKit objects.