BY GREGG WILLIAMS

# SOFTWARE FRAMEWORKS

*Software "toolkits" save programming time*

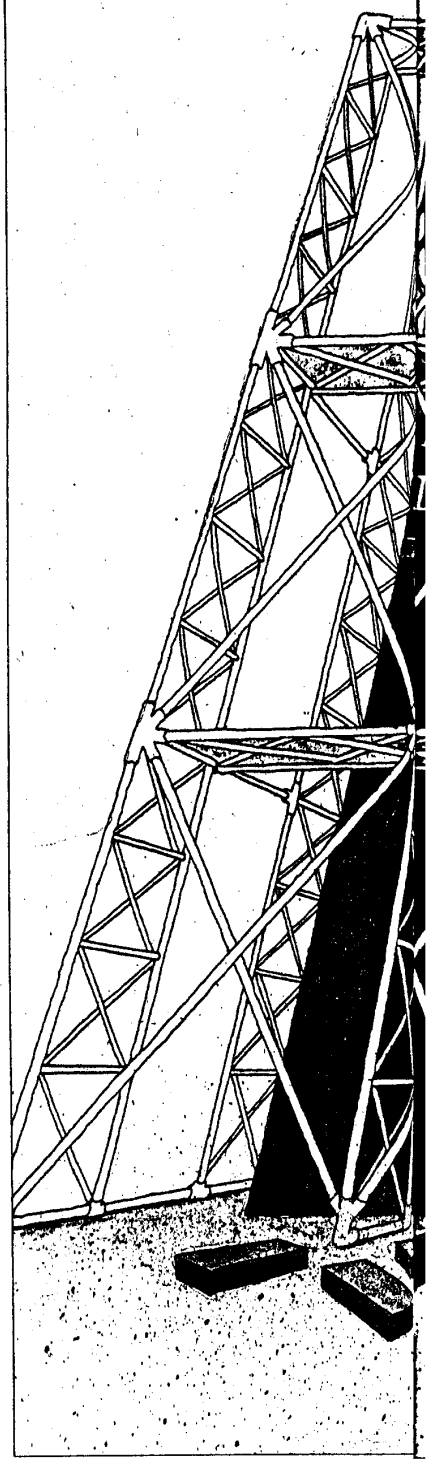
We are certainly beginning to see some wonderful software—fast, useful programs that give us color graphs, windowed information, and mouse-based cursors. Unfortunately, such software involves a tremendous amount of programming. Since more time at the programmer's computer usually translates to more dollars at the software store's cash register (in a market where software prices are high enough as it is), both you and the software publisher are part of a two-sided dilemma. On the publisher's side, the dilemma has to do with choosing between producing the more complicated software and raising its price, or *not* producing it because he believes he will not be able to sell it profitably. On the user's side, you can either buy the software you see or not—you have little direct influence on what software gets developed. Unless we can all improve our standard of living so that we won't mind spending $1200 for a spreadsheet, it looks like it's up to the software publisher to come up with an answer to this problem.

There is one sure way to keep a product's price from rising—reduce manufacturing costs. You can be sure that software publishers are looking at every phase of their operations for places to cut costs, and since development represents the largest percentage of that cost, why not start there? The quest for lower development costs has given us such things as new programming languages, productivity tools, and program generators. A promising solution being used by several software developers is that

(continued)

*Gregg Williams is a senior technical editor at* BYTE. *He can be reached at* POB 372, *Hancock*, NH 03449.

DAVID T. CRAIG



ILLUSTRATED BY RICHARD COWDREY

*A software framework defines much of an application's operating environment so that the programmer can concentrate on the application itself.*

of the *software framework*, a program (often enhanced by a programming language and/or environment) that defines much of an application's standard operating environment so that the programmer can concentrate on implementing the application itself. In this article, we will look at Apple Computer's Toolkit/32 as a representative example of this type of software. Toolkit/32 significantly reduces the time needed to create an application that runs in the Lisa's desktop-metaphor environment.

(You should not confuse my concept of "frameworks" with the Framework integrated-software package from Ashton-Tate, which was announced after I had finished writing this article.)

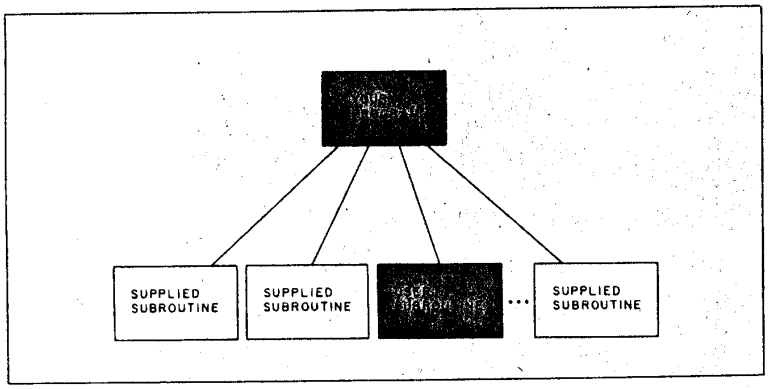



Figure 1: *Creating an application program using a subroutine library. When you must make an application given a library of useful subroutines, you must write the driver program and whatever custom subroutines (shaded) are needed to create the application. If the application is embedded in a sophisticated user interface, you will spend a long time "reinventing the wheel" (writing the code to implement an already-specified user interface).*

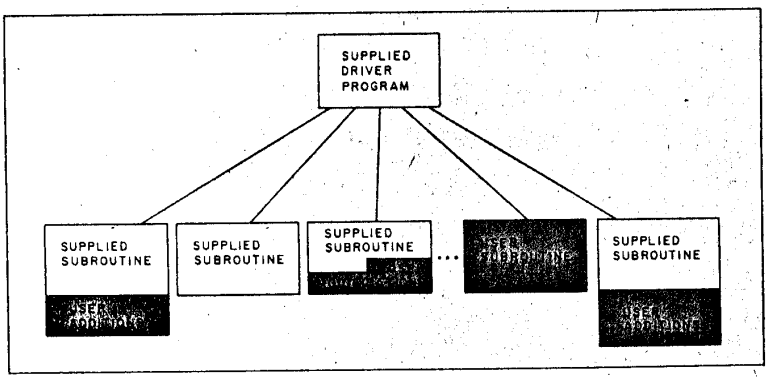



Figure 2: *Creating an application program using a software framework. If you have been given a software framework that implements the user interface but nothing else, you need only write code that implements your application. In some cases, you will write entire subroutines; in others, you will modify ones already in the software framework.*

## Conventional Program Design

Software has too often been cobbled together on an entirely custom basis—that is, each program is created without making use of any previously written code. Sometimes, a software designer who has a number of similar programs to write will create a library of useful subroutines that can be copied as needed into programs. In some cases, as with the Macintosh Application Toolbox in which a 64K-byte ROM (read-only memory) of routines is built into Apple's Macintosh computer, these routines can be quite powerful and can eliminate a large amount of needless programming. In such cases, the programmer will write a main program that makes heavy use of their own subroutines as well as those supplied (see figure 1).

What is wrong with this setup? Nothing—it's just that it often isn't enough. Much of the main program is merely program "glue" that coordinates the calling of the supplied subroutines. When a program is sophisticated enough, though, even the coordinating software is complicated. Consider the Lisa applications, all of which use the same user interface. The application program must interact quickly with user input (keyboard, mouse, and mouse button) to create complicated output in the form of graphics, text, and windows. In addition to performing its dedicated function (e.g., drawing graphs, maintaining a spreadsheet), an application must do many things that do not change from application to application: update the video cursor when the mouse is moved; display a pull-down menu when a menu title is selected (cursor on title, mouse button pressed and held); execute a given task when a menu item is selected (cursor on item, mouse button released); move a window when it is being dragged (cursor on window title, mouse button pressed and held); resize a window when its "grow box" is pulled (cursor on box, mouse button pressed and held); receive characters when they come in from the keyboard; redraw hidden parts of a window when it is

*In the toolkit approach, the supplied program is in control and coordinates system-general behavior, such as moving windows.*

brought from behind several other windows (mouse button clicked when cursor is on some visible part of window); expand a selected icon to a window (cursor on icon, mouse button pressed twice quickly); and numerous other less visible tasks. Given all this, using a set of subroutines to develop such a program is like asking for a ride downtown and being shown to a warehouse of auto parts—you don't want to build the car yourself, you just want to get downtown.

## The Toolkit Approach

Bruce Blumberg, J. Peter Young, and Larry Rosenstein of Apple's Apple 32 division have devised an alternative. Imagine a completely functional Lisa application that allows you to pull down menus, create, move, scroll, and rearrange windows, update the video display to correspond to mouse movement, and do all the other things associated with the Lisa user interface—only the windows and menus are blank. Then you (as programmer) need only add or modify code to show what's in a window and, in general, specify the behavior necessary to your application (as opposed to being generic to the Lisa user interface). Apple's Toolkit/32 gives you this capability.

The difference in philosophy is significant. In the traditional approach (figure 1), you are responsible for orchestrating correctly both system-general and application-specific behavior. In the toolkit approach (figure 2), you only have to modify and add subroutines to get your application to work. The supplied program is in control and coordinates system-general behavior (e.g., moving and scrolling windows); it calls your code when it needs to know what application-specific behavior you want (for example, what to show in a certain window).

There is nothing wrong with the toolbox (subroutine library) approach—it is certainly better than having to write everything from scratch. It's just that most software developers don't want the absolute freedom to combine library subroutines arbitrarily—they just want to adapt their program to run correctly within the standard desktop-metaphor environment and do it as quickly (and, therefore, as inexpensively) as possible.

## An Example

Let's see how a simple application can be written using an application framework like Toolkit/32. In a few hours, the people at Apple created a simple application called Lisa Boxer. In it, any window opened contains two shaded rectangles that can be moved around. The specific code needed to do this was five pages of Pascal-like code—not much at all, considering the amount of space and comments a structured program includes (see listing 1 for a segment of the code).

(continued on page 394)

Listing 1: *An example of Clascal code. This code, part of a five-page listing that creates the Lisa Boxer application, defines the methods that the class TBoxView responds to (i.e., the messages that objects of that type understand) and the code that executes when this class is created. Methods are capitalized, while fields are not. See the text for more details.*

```
METHODS OF TBoxView;
    FUNCTION (TBoxView.)NEW((itsHeap: THeap; itsPanel; itsExtent: LRect;
                itsBoxList: TList; itsPrintable: BOOLEAN): TBoxView);
    BEGIN
        ($IFC fTrace)BP(11);($ENDC)
        SELF := SubObject(TView.NEW(itsHeap, itsPanel, itsExtent, itsPrintable),
              'SIZEOF(SELF));
        SELF.boxList := itsBoxList;
        ($IFC fTrace)EP;($ENDC)
    END;

(This returns the box containing a certain point)
    FUNCTION (TBoxView.)BoxWith((LPt: LPoint): TBox);
    VAR box:    TBox;
        s:      TListScanner;
    BEGIN
        ($IFC fTRACE)BP(11);($ENDC)
        boxWith := NIL;
        s := SELF.boxList.Scanner;
        WHILE s.Scan(box) DO
            IF LPtInLrect(LPt, box.shapeLRect) THEN
                boxWith := box;
        ($IFC fTrace)EP;($ENDC)
                END;

(This draws the list of boxes)
    PROCEDURE (TBoxView.)Draw;
    VAR box:    TBox;
        s:      TListScanner;
    BEGIN
        ($IFC fTrace)BP(10);($ENDC)
        s := SELF.boxList.Scanner;
        WHILE s.Scan(box) DO
            box.Draw;
        ($IFC fTrace)EP;($ENDC)
    END;

CREATION
BEGIN


    cBox := NewClass('Apple', 'TBoxView', SIZEOF(TBoxView), 1, 1);
END;
```

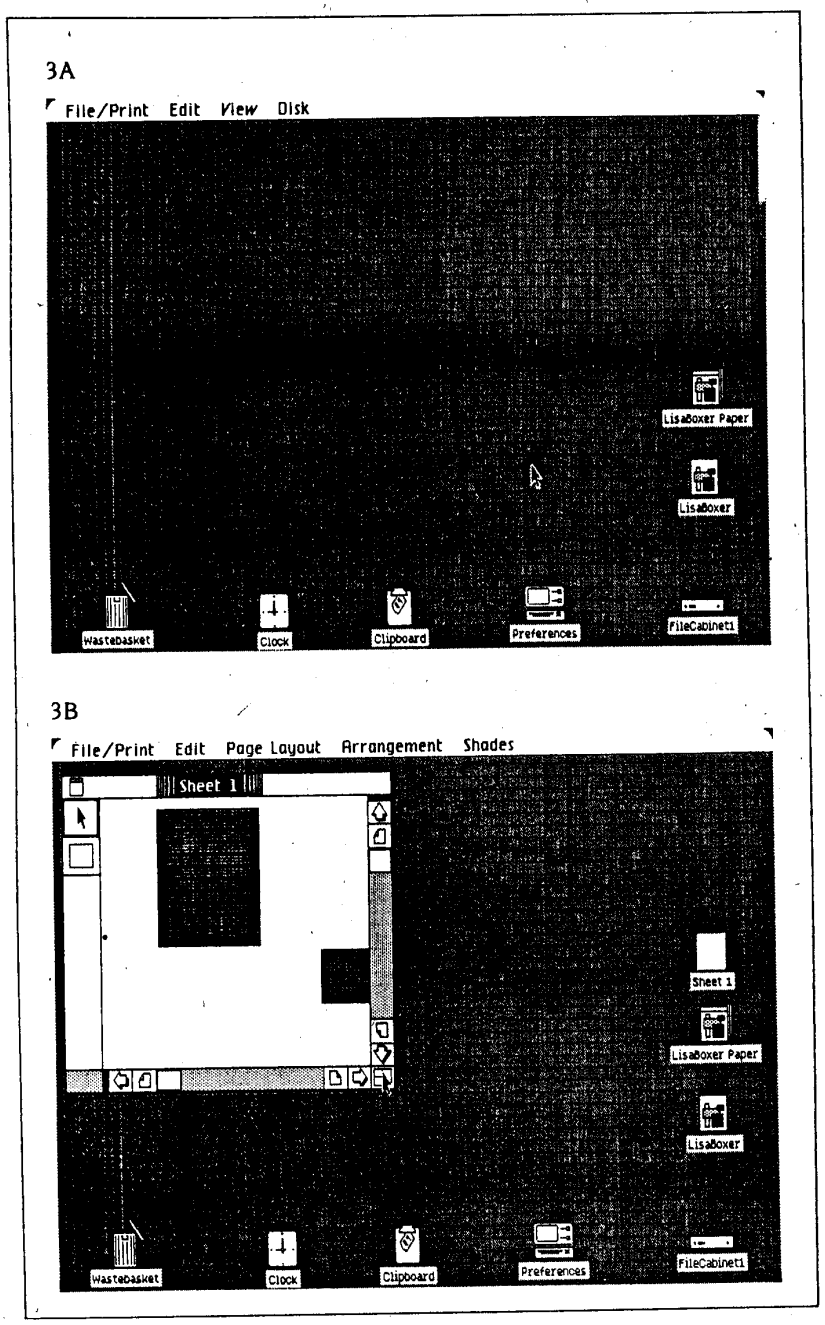



Figure 3: *How a new Toolkit/32-based application appears within the desktop environment. Once the application is linked to the Lisa Office System (figure 3a), it becomes visible on the desktop as two icons, a tool (here, Lisa Boxer) and a notepad of paper (Lisa Boxer Paper). When a sheet of paper is torn off (figure 3b), it becomes an open window used to display the application. Here, Lisa Boxer is a simple program (written in five pages of code) that puts two boxes in the window and allows you to move them arbitrarily.*

(*continued from page* 127)

We insert this code into the Toolkit/32 program, compile it, and link it to the Lisa Office System (the software that the Lisa runs to give the desktop-metaphor environment) under the name "Lisa Boxer." When we boot the Lisa Office System, the Lisa Boxer program appears on the desktop as a Lisa Boxer tool icon and, beneath it, a "notepad" of Lisa Boxer "paper" (see figure 3a). When we "tear off" a sheet of Lisa Boxer paper and "open" it, we get a window with two shaded boxes in it (figure 3b).

Now we can see how Toolkit/32 greatly simplifies the software developer's job. We find that we can change the size of the window, as shown in figure 4a; this default behavior results from the predefined Toolkit/32 code. We can also move either of the boxes, as shown in figure 4b; this behavior results from the code we added. If we try to do something that neither the Toolkit/32 code nor our code knows how to do (like selecting a box and giving it the "Gray" command from the Shades menu), the Toolkit/32 program recognizes its inability to execute the command and deactivates the command in the pull-down menu (the computer indicates this by printing the new selection in gray instead of black).

There are literally hundreds of events and interactions that you would have to orchestrate if you were writing a driver program for an application with a sophisticated user interface. Granted, it takes some effort to correctly integrate your code into a software framework like Toolkit/32, but you will still save a lot of lines of code you don't have to design, write, and debug.

## Object-Oriented Languages

Before we can take a closer look at Apple's implementation of a software framework, we must first look at object-oriented languages. If you're familiar with the phrase, you probably associate it with the language Smalltalk, on which BYTE did a special language issue in August 1981. At the

moment, Smalltalk is the best-known object-oriented language.

Most computer languages are *operator-operand languages*—that is, languages in which operators (like "+" and "/") perform predefined functions on operands (usually numbers). When we execute "5 + 3" in an operator-operand language, the operator "+" adds the operands 5 and 3. This is an orientation so widespread that most of us have trouble understanding a different one. *Object-oriented languages*, on the other hand, present the programming environment as a collection of objects that receive messages; here, when we calculate "5 + 3", the object "5" receives the message "+3" and knows what to do with it (it adds the two numbers, returning the value 8). Each object has a set of messages (called *methods*) it understands. When a message is passed to an object, the object checks the message against its list of methods. If it finds the method, it executes the associated code; if it does not find the method, it returns a message that says, "I do not understand the message 'xxx'."

Objects are grouped into classes, with the possibility that some classes may be contained within others. When this occurs, a member of a class (with some exceptions) understands not only its own methods, but also the methods of its superclass (the class that immediately contains it) and its ancestor classes (any *n*th-generation-removed superclass). (For more details, see "The Smalltalk-80 System," by members of the Xerox Learning Research Group, August 1981 BYTE, page 36.)

The object-oriented approach is too involved for an environment that manipulates numbers only, but it is very useful in environments that include different number types, windows, graphics, icons, lists, control structures, and other items. The object metaphor can encompass all these (and other) items; this simplifies the language and, therefore, makes the programmer's job easier. Also, by the careful creation of classes and subclasses, programmers can amplify

(continued)

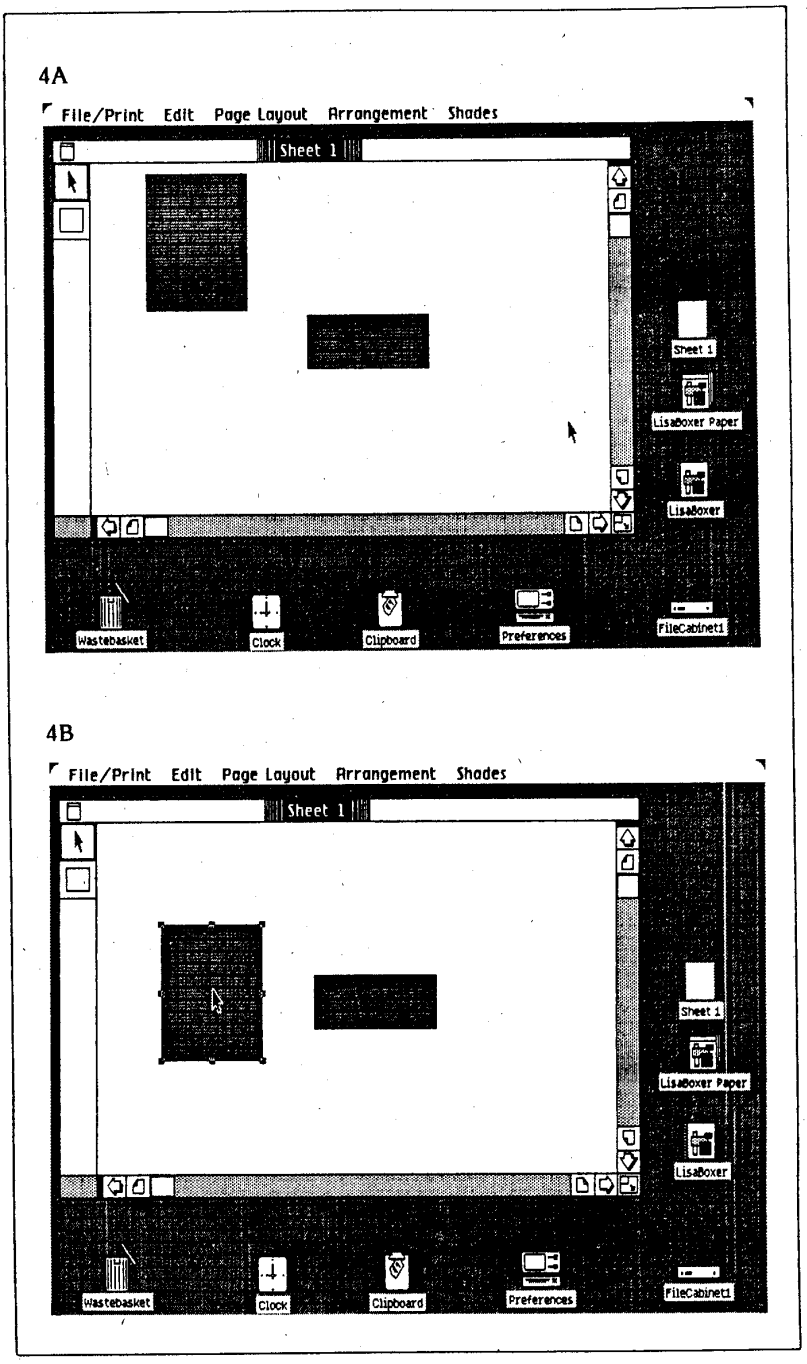



**Figure 4:** *Component of a Toolkit/32-based application's behavior. Behavior may be specified by the software framework (e.g., changing the size of a window, figure 4a) or the application (moving one box, figure 4b). In only the second case does the programmer have to add code to the software framework to get the desired result; see the text for details.*

their programming work by defining useful methods that can be used automatically by objects within a subclass. In this and other ways, object-oriented languages help programmers design and modify extremely large projects. As Bruce Blumberg puts it, "Object-oriented programming will be to the 1980s what structured programming was to the 1970s."

Object-oriented languages differ from operator-operand languages in two other important ways. First, the definition of a given operator in an operator-operand language is given in the code that defines that operation for all possible data types; to add a new data type to such a language, the programmer must add code to the definition of the numerous operators that will deal with that data type and add error-trapping code to all the operators that won't. In an object-oriented language, all such code is located in one place—the definition of the class—and operations that members of that class don't understand return error messages automatically. In addition, code that was previously written, when compiled with the information defining that new class, will run as is when it sends an old message (operator) to a new object (data type).

A second advantage of the object-oriented language is that it mirrors the subject-verb orientation of some user environments more closely than traditional operator-operand languages. Many software designers have found that the subject-verb orientation (i.e., selecting an item to be worked on, then choosing the action that will be performed on it) makes software easier to understand. For example, in Apple's Lisa Write, Microsoft's Word, and other word-processing programs, you can delete a phrase of text by first selecting it, then choosing the "delete" operation. You can see that the object-message orientation of an object-oriented language closely parallels the subject-verb orientation of the software itself; this strong parallelism makes the software easier to write and, later, to maintain.

Readers interested in learning more about Smalltalk as an example of an object-oriented language can refer to the August 1981 BYTE and to *Smalltalk-80: The Language and Its Implementation*, by Adele Goldberg and David Robson, Reading, MA: Addison-Wesley, 1983.

## PASCAL + CLASSES = CLASCAL

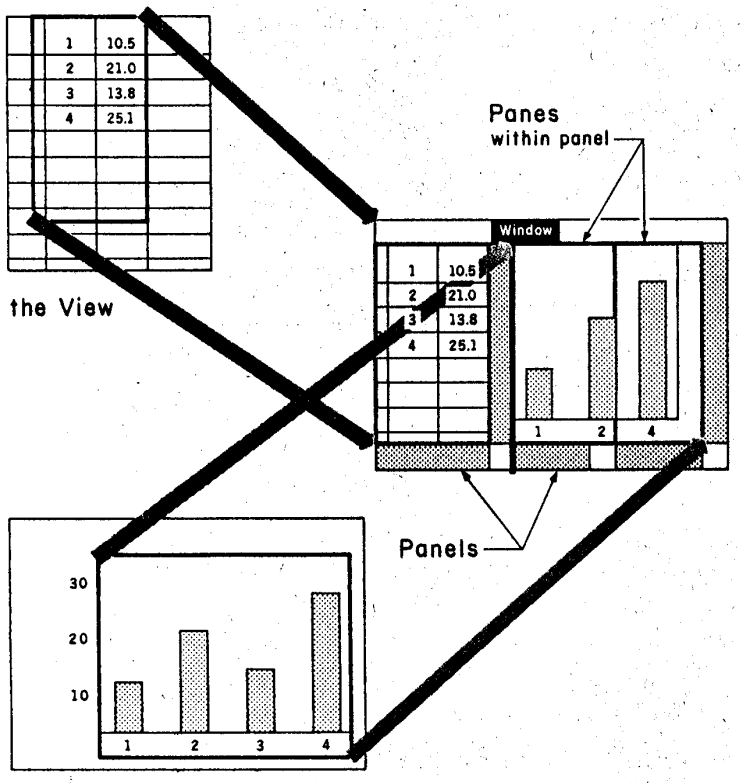
The presence of Apple's Larry Tesler in BYTE's Smalltalk issue telegraphed Apple's interest in the language (although we didn't know at the time that it was being researched in rela-

(continued)

# TOOLKIT/32 •NAMING CONVENTIONS

To understand Toolkit/32, you first have to understand several terms that have specific definitions (refer to the figure below). The *window* is the object through which you manipulate data via your application program. This is the same kind of window as is advertised in several desktop-metaphor environments; its size can be changed, it can be moved to any location on the video screen, and it can be "behind" or "in front of" other windows. Underneath the application program, invisible to you, is a data structure, and the purpose of the application program is to give you one or more *views* of the data. The actual data, which is not shown, is a collection of four ordered pairs: (1,10.5), (2,21.0), (3, 13.8), (4, 25.1). The top view shows the data as a spreadsheet, and the bottom shows it as a graph, but neither is the actual data.

A window (and the application program it represents) can show more than one view. To do this, the application program divides the window into one or more *panels*. Each panel shows part of its corresponding view and can be scrolled independently from any other panel to show any arbitrary region of its view. In many cases, a panel can be broken into *panes*, each of which can be scrolled in one dimension to show different regions of the same view. (Below, note that the right pane of the right panel has been scrolled to show only the fourth bar in the bar graph.)




tion to the Lisa). Apple implemented Smalltalk for the Lisa but never released it; "It's too slow," Tesler said, "and its syntax is too different for most people." However, designers at Apple did not lose their enthusiasm for object-oriented languages, especially after they had a taste of the problems that came with the original six Lisa application programs, which were written as standard Pascal programs.

The Toolkit/32 team created Clascal, a superset of Pascal that contains a new data type: classes. By doing so, they created what they thought contained most of the familiarity of Pascal with most of the power of an object-oriented language. The class data type is like the record data type in Pascal. Just as a Pascal RECORD statement defines the record by the fields it has, a Clascal SUBCLASS statement defines a class by the fields and methods it has. Similarly, just as a record is an instance of the record definition, an object is an instance (or member) of a class. Clascal defines one class, TObject. All other classes are subclasses of TObject, and all objects have TObject as an ancestor class. Each object can respond to the methods of its class and those of all its ancestor classes; the only exception to this is that if two of the object's classes and superclasses have the same method, the object uses the one closest to it—this allows a subclass to override the (perhaps inappropriate) methods of one of its ancestor classes.

The definition of a class consists of stating the class's name, the class of which it's a direct subclass, the new fields not inherited from an ancestor class, the functions and procedures that constitute the class's methods, the algorithms that implement those methods, and (optionally) the code that must be executed when the class is created. A class inherits the methods and fields of all ancestor classes unless they are specifically redefined within the class definition. When an object of this class is created, it is associated with a set of

(continued)

values for that class's fields; you can think of a field as a private variable each object gets upon its creation.

Listing 1 shows sample code from the application that created the Lisa Boxer application program described earlier. This segment defines the methods that the objects of the TBoxView class understand—NEW, BoxWith, and Draw—and the code (under the heading CREATION) that executes when the class is created. In this example of code, methods are capitalized and fields are not, and an object and its method are joined with a period—for example, "s.Scan(box)" sends the method Scan to the object s with the object box as an argument; also, braces delineate comments that are included only to explain the surrounding code.

**Table 1:** A *partial list of tasks that are taken care of automatically by the Standard Application within Apple's Toolbox/32. Software frameworks present a prepackaged solution to the complicated problem of creating an event-driven, window-based user environment; as such, they decrease the programmer's work by one to two orders of magnitude, thus allowing him or her to create a software application much faster and with a smaller time commitment.*

memory management
file management
printing and other I/O
communication with the desktop environment
communication with the clipboard (allows different applications to exchange data)
creation, deletion, activation, deactivation, and size and position change of windows
multiple panes within a panel
multiple panels (giving differing views of the data)
standard scrolling within a pane or panel
handling of all errors not dealt with by user code
passing of system events (e.g., mouse movement, keypresses) that the Standard Application cannot handle to user code

## INTERFACING TO THE STANDARD

Now we can examine the Standard Application in more detail. The Standard Application is a collection of classes that implement the generic behavior of a standard Lisa application program—that is, all the actions that, because of the previously de-

(continued)

fined user interface, are the same from program to program. Table 1 lists some of the functions that are handled automatically by the Standard Application. This means that programmers who are modifying the Standard Application must provide only code that implements the behavior specific to their applications (this will be referred to later as nongeneric behavior). This involves four activities:

- supplying the Standard Application code with the view or views of the data when requested (for a definition of the word "view" in this context, see the text box "Toolkit/32 Naming Conventions" on page 398)
- creating and maintaining the data structures that the application requires
- defining and implementing the actions that can be performed by the application
- adding code to modify or override the Standard Application's generic behavior

To add the nongeneric behavior of the application, the programmer needs to concentrate on the above four activities alone. In practice, this means adding new subclasses to the object classes already defined within the Standard Application.

## TOOLKIT/32 UNITS

The "driver" program (I hesitate to call it a program because of its triviality) for the Standard Application is five lines long; it initializes some variables, creates an object that belongs to a class called "process," and sends that object the method "Run." The actual behavior of the application program is determined by the definition of the process class and other classes. In the following paragraphs, I will take a look at the major units that make up the Standard Application. (The units I refer to here are like Pascal units; the units below each contain related subclass definitions.)

*UObject:* the UObject unit defines the class TObject, the universal ancestor class of all Clascal objects. This unit handles memory allocation for objects (they are given memory when they are created, and the memory is reclaimed when they are deleted); it also handles object copying and enables some optional debugging facilities.

*UList:* this unit implements dynamic arrays, indexed, linked, and blocked lists, and utilities that allow the programmer to create, modify, and scan these objects.

*UDraw:* this unit extends the Quickdraw graphics routines (at the heart

(continued)

of Lisa's graphics capabilities) to use 32-bit coordinates instead of QuickDraw's 16-bit coordinates; this eliminates possible document-size problems that could arise from using 16-bit coordinates. UDraw is used for drawing in a view.

*UABC:* the "ABC" stands for application-base classes. UABC contains most of the classes that define the standard behavior. Because of this, programmers will spend most of their time creating subclasses of UABC to implement the application's nongeneric behavior.

## Inside UABC

We will now take a brief look at which classes within UABC are usually modified and how those modifications create the application program's nongeneric behavior.

The methods of the TProcess class are those methods (or actions) that are specific to the application but are called every time the application executes. Every Toolkit/32 application creates only one process object, which implements the main program loop of the application. An application almost always follows this loop: get an event (e.g., menu selection, mouse movement, keypresses), process it by giving it to an object that understands what to do with it, and wait for the next event. The process object also handles error and warning messages and the procedure for ending the application.

An application creates one object from the ancestor class TDocManager for each document icon or open window that is associated with the application. This descendant object controls the interface to all disk files that define the document, and it manages the memory used by the document.

Objects descended from the TView class define any views needed by the application—one object for each view the application uses. The view shows the entire representation of the application's data, interpreted as the view is defined to show it. A panel object (described below) asks a view object for the view, but it is the panel object's responsibility to draw the part that is to be visible (this is one of the things the Standard Application code usually does automatically). Obviously, one of the programmer's main responsibilities is to create the code that implements the view.

Whenever the user selects an action to be performed (usually by choosing a selection from a pull-down menu), the application creates an object that is a descendant of the TCommand class and gives it the message "Perform," which causes the object to try to execute itself. Most actions should

(continued)

*TWindow, TPanel, and TPane are classes that define the behavior of windows, panels, and panes.*

be implemented so that they first change the view but not the underlying data; this allows the "Undo" message to work in most cases. If "Undo" is not the next message sent, the program automatically executes a method called "Commit," which makes the changes already shown in the view to the data itself. Each panel of the application has an object descended from the TSelection class; this selection object can handle messages and manipulate whatever objects, if any, are associated with that panel. When a command object executes itself, it gives itself to the selection object associated with the active panel.

TWindow, TPanel, and TPane are classes that define the behavior of windows, panels, and panes. Their behavior, already defined by the Standard Application, does not usually need to be changed by the programmer; each panel will ask a view object for the view and will display the appropriate part as part of its generic behavior.

## COMMENTARY

The software framework approach is an extremely useful one for programmers who want to develop sophisticated applications but who don't have the resources (or patience) to write the extremely complicated code that will bring it up in an interactive user environment. The Apple Lisa group has developed the Toolkit/32 system described in this article. The Microsoft "window" device under MS-DOS 2.0 works similarly for MS-DOS-based software. (According to Scott McGregor of Microsoft, his company has developed a "Microsoft Windows Toolkit" for software developers that

(continued)

is "identical in concept to Toolkit/32"; however, details of this were not available when this was written.)

Clascal shows great promise for extending programmer productivity when used in large projects. One question comes to mind, though: Is the software it creates fast enough? Bruce Blumberg of Apple Computer said that Clascal is "about 10 times faster than Smalltalk and only 10 percent slower than Pascal." Given that the original Lisa applications were written in Pascal and that the recently released enhanced packages added some machine-language code to speed them up, the above statement is of little comfort. We'd all better wait until the first Toolkit/32-based application programs can be examined before forming a final opinion on Clascal.

I must add to this enthusiastic description of Toolkit/32 the fact that, although it saves the programmer tremendous amounts of time, the Clascal class orientation does take some getting used to. Although a novice Clascal programmer takes about a month, according to Apple, to begin to understand Clascal well enough to take advantage of it, it still saves time in the long run to develop Toolkit/32-based application programs.

To conclude, I must make two sets of observations. First, the software framework orientation represents a subtle but important change in how we look at developing an application program—in changing the programmer's task from writing a program to writing subroutines, it clearly delineates the program's generic behavior from the nongeneric and allows the programmer to concentrate on the latter. Second, Toolkit/32 implements the software framework in an interesting and powerful way. Clascal encourages the programmer to factor out common characteristics, which are embodied in high-level classes from which much code is derived. Apple has already done this with the Toolkit/32's Standard Application code, thus allowing the programmer to write code that inherits a lot of structure and power from the previously defined ancestor classes. In any case, software frameworks may be one answer to the problem of creating sophisticated programs quickly and inexpensively. ■

[*Editor's note: As this article was going to press, Larry Tesler of Apple told us that the Toolkit/32 product was being made available to any Lisa owner (for details, write to the Software Resource Center, Mail Stop 2-P, Apple Computer, 20525 Mariani Ave., Cupertino, CA 95014). In addition, Apple is considering creating a similar product for use by Macintosh software developers.*]

↳ Will be called 'MacApp'.