EXOTIC LANGUAGE OF THE MONTH CLUB

Clascal—An object-oriented Pascal

By Tim Endres

In 1983 Apple Computer set into motion a new ideal in the personal computer industry. With the introduction of the Lisa, Apple gave the computer public its first inoculation of user-friendly, fully integrated software.

No other personal computer has quite matched the innovation of the Lisa. Yet very little is known about the heart of this computer. Besides some of the most impressive hardware specifications for a computer in its price category, the Lisa has some of the most sophisticated software ever packed into a personal computer.

Included in this software is a very fast and powerful graphics driver (QuickDraw), a standard operating system, a hardware interface, and a development environment called the Workshop—with Pascal, Assembler, Linker, COBOL, BASIC, and a command list processor.

Apple used these standard building blocks, along with several custom utility libraries designed to drive the windows and folders of the desktop, to create the Office System environment. The Office System is a collection of integrated software aimed at office and business automation.

To top the package off, Apple spent a great deal of effort to develop a system that would allow future programs written for the Office System to be fully integrated. This system, called the Toolkit, was built on an object-oriented language named Clascal.

Clascal was an extension built into Apple's Pascal that gave it an object-oriented syntax. Clascal and the Toolkit provided me, an independent developer, with the resources to develop an application for the Office System in approximately half the development time I would have expected for any other PC. This application was fully integrated into the Office System and provided a user-friendly interface consistent with all other Office System applications.

To realize the significance of the Toolkit, one must first appreciate Clascal. Clascal is an extension of Pascal that makes it object oriented, which means that the syntax of the program is based on the idea of objects instead of procedures and object parameters instead of variables.

With regular Pascal, you program using procedures, functions, and variables. With Clascal, you program using objects, which have methods (procedures and functions) and data fields (parameters).

To illustrate the difference between Pascal and Clascal, consider the code written to display and control the window in which my application executed (Figure 1). The window had three different views in which user interaction occurred: the status view on top, calendar view bottom left, and appointment view bottom right.

Let's consider writing the code that would be responsible for merely drawing the window. In Pascal, I would call the procedure that draws the window:

```
Draw_Window;
```

The *Draw_Window* procedure would call the procedures responsible for drawing each of the views and highlighting selections:

```
PROCEDURE Draw_Window;
    BEGIN
        Draw_Status_View;
        Draw_Appointment_View;
        Draw_Calendar_View;
        Hilite_Selections;
        END;
```

In Clascal, the window is an object. Objects are referenced with symbolic variables the same way variables are referenced in Pascal. To draw the object, you must invoke its method, called *Draw*:

```
myWindow.Draw;
```

Here *myWindow* is a symbolic reference to the window object. *MyWindow*'s *Draw* method would first call each view's *Draw* method, then invoke its own highlighting method. In Listing 1, notice how the data field *views* is a reference to an object that is a list of objects. Each object in the list is a view of the window. The method *Each* invokes the method in parenthesis for each object in the list. Lists are important classes in the Toolkit.

The keyword *SELF* is essential to Clascal. When any method in Clascal uses this keyword, it is referencing the object that was asked to perform the method (*myWindow* in the preceding example). To

```
PROCEDURE TStaWindow.Draw;    {TStaWindow is myWindow's class name}
   BEGIN
      SELF.views.Each(Draw); {Causes each object in list "views" }
                             {to invoke its DRAW method          }
      SELF.HiliteSel;        {Invokes the window's own HiliteSel
                             {method                             }
         END;
```

Listing 1.

understand this a little more, we must discuss classes, the foundation of Clascal.

Objects are the functional building blocks for Clascal. Every tangible piece of the program is represented by an object, and every tangible action in the program is the method of an object or a function or procedure called by a method.

Thus, when the software developer writes code, he or she is creating objects that represent the different pieces of the application (for example, *Window, View, Appointment, Calendar, Day, Clock, Folder*) and causing these objects to act upon each other by invoking their methods.

Classes are the conceptual building blocks of Clascal. Every object created in a program is defined to be in a particular class. Classes are similar in syntax to types in Pascal but function considerably differently. They create one of the most powerful aspects of Clascal.

Classes describe the types of objects used in a program in terms of the parameters of each object and the methods that each one can perform. More importantly, each class is a subclass of some other class. This is a required syntax and gives the language a hierarchical structuring.

This hierarchy is one of the most powerful aspects of Clascal. It accounts for a very critical quality called extendibility. Extendibility is the ability to take a functioning block of code and extend its capabilities through the mechanism of subclassing.

For instance, again in Listing 1, I used the object *myWindow* to control the window of my application. This object belongs to the class *TStaWindow*, a subclass of *TWindow*, which is a class provided in the Toolkit. Any object in the class *TWindow* can perform the following methods:

- Create a new object of the class *TWindow*
- Delete itself
- Clone or duplicate itself
- Draw itself
- Perform commands and handle mouse and keyboard events
- Open, close, suspend, resize, and refresh itself.

Because of inheritance, every object in the class *TStaWindow* can perform any of the preceding methods. The objects of the class *TStaWindow* also inherit the parameters defined for objects of the class *TWindow*. Thus, by merely coding the line that states *TStaWindow* is a subclass of *TWindow*, I can create my own windows that do everything Apple has coded for objects of the class *TWindow*.

Better still, I can add to the parameters and methods that are defined for *TWindow* objects to make my windows more functional. Plus, I may redefine the methods that are already defined for *TWindow* objects to do something different. For example, Apple's *TWindow* objects, when

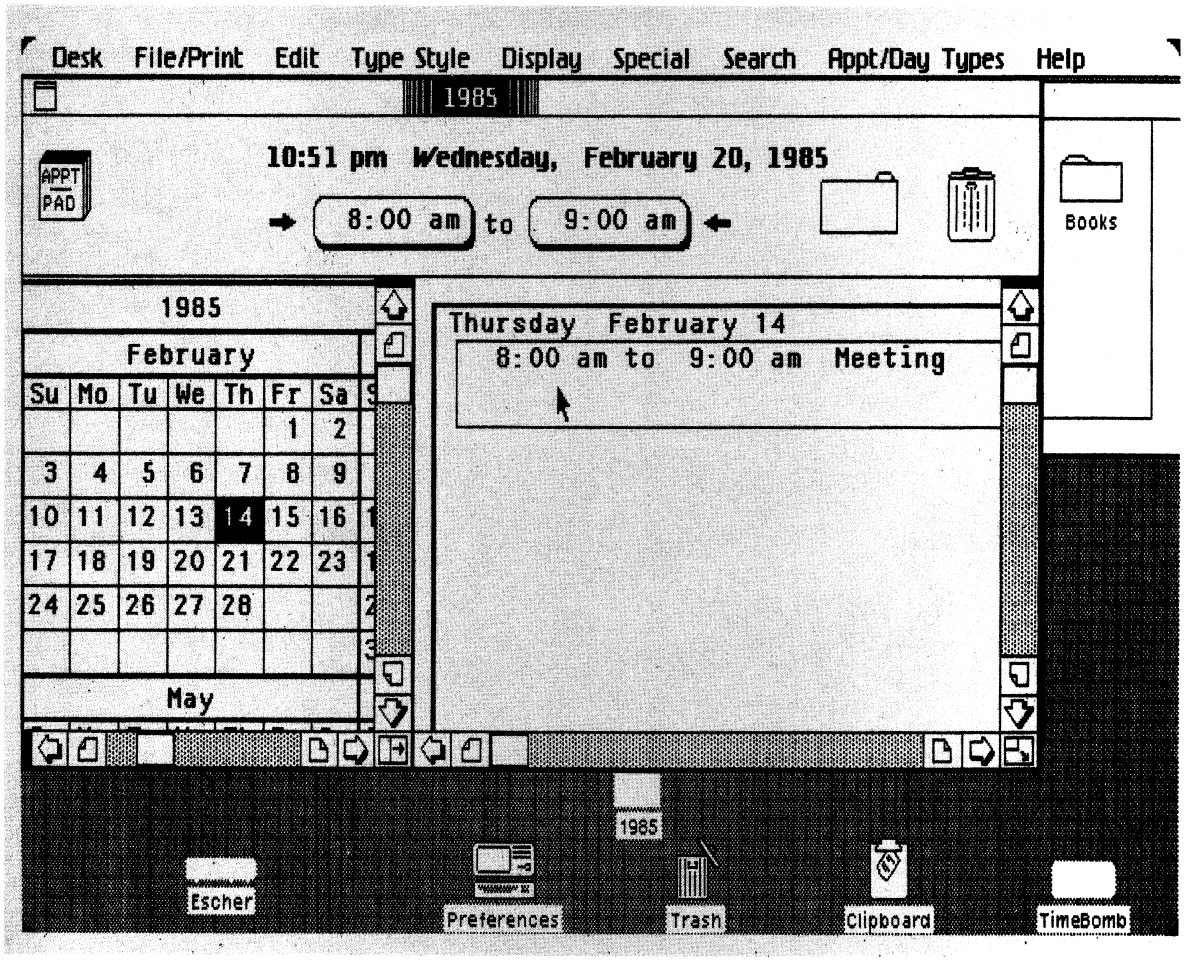

Figure 1.

activated, would do some default highlighting of selected objects. This was fine except that the default highlighting logic caused a glitch in my program's display in certain cases. To fix this problem, I simply redefined the method that activated the window to use a different highlighting logic. I did this by defining a method for the class *TStaWindow* with the same name as the method defined for *TWindow*.

Why did this override the activation method performed by *TWindow*? This brings us back to the keyword *SELF*. Whenever a Clascal method uses the keyword *SELF*, it is referencing the object that was asked to perform the method.

Notice the number of occurrences of the word "itself" in the list of the previous methods. Consider, from the example method *TStaWindow.Draw*, the line *SELF.HiliteSel*. This line states that the window object should perform its method called *HiliteSel*. When the keyword *SELF* is encountered, a search beginning with the object performing the method is conducted for the method definition. If the method is not defined for the object's class, the search continues up through the superclasses of the object's class, until the definition of the method is found.

If I define my own method called *HiliteSel* for the *TStaWindow* class, and a *TStaWindow* object is asked to perform its *Draw* method, then that method in turn will invoke its own *HiliteSel* method. If I had not defined this method for *TStaWindow* objects, then the *Draw* method would invoke the *HiliteSel* method defined for *TWindow* objects. This situation is illustrated in Figure 2.

Just as methods are inherited, so are data fields. The methods defined for *TStaWindow* objects can reference the same data fields defined for *TWindow* objects. The only difference is that you do not override data fields. Inheritance is collective. Classes inherit the methods and parameters of their superclass, which in turn inherit the methods and parameters of their superclasses.

The Toolkit is a library of software provided to support Office System application development. It accomplishes this through the mechanism of extendibility provided by Clascal.

The Toolkit is a collection of class definitions that together perform all of the basic Office System functions. It defines windows, views, panels, scroll bars, documents, document managers, processes,

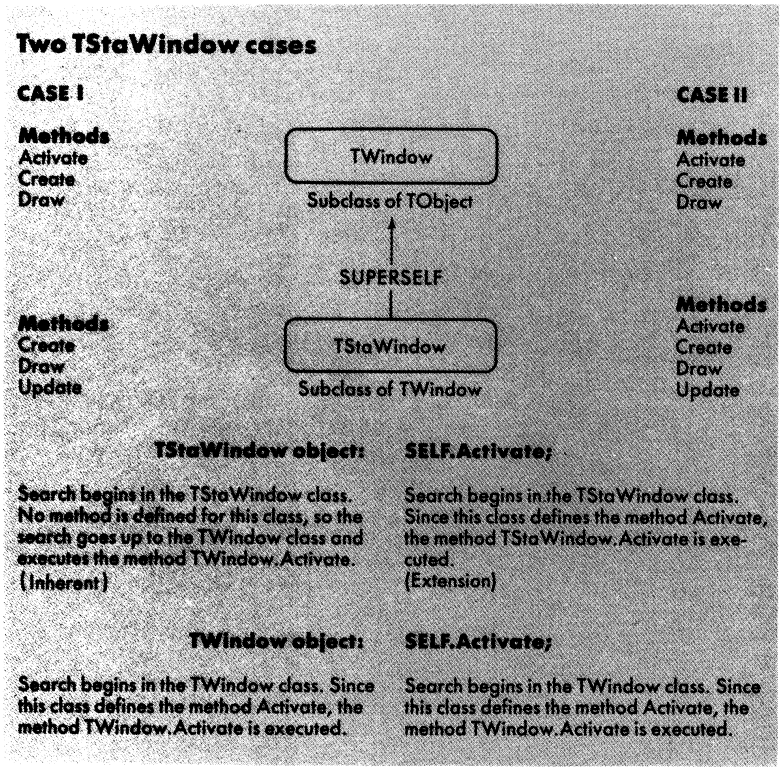

Figure 2.

selections, clipboards, dialog boxes, text and much more. These definitions provide the software developer with an application skeleton, called the generic application.

By coding no more than 100 lines, I can develop an application that will display a window with scroll bars, a view with my name drawn in it, and menus. It would allow moving and sizing of the window, picking of commands from menus, scrolling, view splitting, and document creation, saving, and deleting. Then, by overriding defined methods and creating my own classes, my application slowly develops into its own unique definition.

Since Toolkit classes were provided for cutting, pasting, printing, mouse handling, and text, my application was fully integrated with all other Office System applications. The application also utilized similar mechanisms for the user interface.

The amount of coding I was saved by not having to develop Toolkit classes is quite significant. The source code of the Toolkit contains approximately four times the number of lines that my application contains. Since development took me nine months, I figure the Toolkit saved me at least two years. It also provided the less exciting, low-level code and allowed me to develop at a higher, more creative level.

Clascal has brought object-oriented languages into a new arena. For the first time, general PC programmers can develop software with an object-oriented language. For the first time, they might learn what SmallTalk is. Apple has just released a language called Object Pascal for the Macintosh and created MAC App, the equivalent of the Toolkit. Cross your fingers in the hopes that more businesses will see that object-oriented languages are the next language in the hierarchy.

This language also provides an inherent structuring. The different conceptual objects of an application must belong to classes. Each class has a set of methods, which perform functions unique to the class. Each class has a set of parameters. These classes must be hierarchical and inherit methods and parameters from parent classes.

Clascal, being an object-oriented language, reduces development time and increases integration, due to extendibility. Extendibility also provides a convenient mechanism to provide generic expert systems that users can customize through subclassing.

Clascal is truly an unsung hero. Of all the fanfare the Lisa received, only a handful of articles even mentioned the Toolkit, and fewer yet talked about Clascal. If more exposure is given to this language, perhaps this situation will change.

Tim Endres is responsible for advanced planning and technology transfer for GM/EDS at Buick-Oldsmobile-Cadillac in Lansing, Mich. He has a B.S. in electrical engineering from General Motors Institute and is a licensed developer for Apple's Lisa and Macintosh.