Cognits: A Portable Library of Intelligent Classes

Steven M. Lewis, Ph.D.
Member of the Technical Staff
Computer Systems Division
The Aerospace Corporation, El Segundo, Calif. 90245

Abstract

Cognits is a C++ class library designed to facilitate the development of large object-oriented applications. The library includes classes that parallel many of the functions available with MacApp, and other classes are available for scientific graphics and image processing. Unlike MacApp, however, with Cognits both data and screen objects are built from the same object classes, allowing the use of a model-controller-view paradigm. In this schema, the responsibility for the construction and maintenance of displays is given to the display objects, rather than to the represented objects. The library consists of approximately 80,000 lines of code defining about 200 classes. Approximately two-thirds of the classes are widgets responsible for displaying specific data on the screen. The use of controls and dialog boxes is generally replaced by the use of objects that are given responsibility for specific sections of a window.

Applications developed with Cognits come with built-in object and class browsers, which allow developers to examine any object in the system and trace interconnections between networks of objects. Most windows in Cognits applications are dynamic, with the type and placement of objects determined at run time. Most programs allow the code to generate default object placements. Static windows, whose contents are known at compile time, are developed by writing the code to generate the resulting objects. Resources representing size, location, color, and font may be altered from within the application by using click-and-drag techniques to rearrange the window. The delivered application offers the same capability to the user.

Cognits code is written to run in Macintosh, X Windows, and MS Windows environments. Portability is accomplished by generating a virtual graphic interface layer for each target machine. Events on each platform are read, transformed into canonical events on the virtual machine, and passed to system-independent software. Drawing and windowing calls are converted from calls to the virtual machine to calls on the underlying hardware.

The library is currently being used to support five different applications: a medical application running on the Mac and the PC; two satellite-scheduling programs running on the Mac and the Sun; a decision-support tool running on the Mac, PC, and the Sun; and an image-processing tool running on the Mac.

Cognits illustrates the use of intelligent objects as an aid to the management of large applications. The virtual machine demonstrates one approach to the generation of system-independent code for the Macintosh and other platforms.

Introduction

Graphic windowing systems are complex. The development of a user interface may represent from 60 to 80% of the time involved in the development of a complex project. The full management of a windowing system is a complex process requiring considerable expertise on the part of the programmer. The Cognits project aimed at giving the programmer access to the power of modern windowing systems without requiring windowing expertise on the part of the programmer. Cognits allows for development at different levels, ranging from a rapid prototyping level that allows the programmer to specify the bare minimum, to full control of the graphical environment.

In implementing Cognits on the Macintosh the goal was to develop a widget set that sufficiently abstracted away the Mac toolbox so that, except for a small, system-dependent library, the same code could run on a number of different platforms. The classic

Mac widget sets, MacApp and TCL, are too closely tied to specific features in the Macintosh toolbox to support this degree of portability.

Another goal was to build into the basic objects methods that would aid developers in debugging applications. Tools such as object and class browsers are normally built into external tools. In Cognits these were made part of every application. Adding these tools adds approximately 50k bytes of code which makes the standard Hello World application approximately 300k bytes of code. There is negligible speed penalty for this and most other functionality.

Design Objectives

Philosophy

The major innovation in object-oriented programming is the notion of dividing both code and data into independent modules that can be instantiated as objects. The development of an object-oriented program involves the assignment of responsibilities to specific objects, as well as the establishment of relationships between objects to allow messages to be passed to objects having specific responsibilities.

Views

The idea underlying Cognits is that a display on the screen represents a view or representation of data within the program. Objects on the screen may represent *actions*, as in the case of buttons, or *data*, as in the case of, say, text fields and graphs. Objects on the screen may allow the user to manipulate the data within the program or merely observe them. This approach, pioneered in Smalltalk, is called the model-view-controller approach, where the internal data are the model, on-screen representations are the views, and user interactions with these views constitute the controller. In Cognits, views and controllers are the same object and are treated as a unary concept.

Cognits significantly improves the model-view-controller approach by assigning responsibility to objects in the system, so that models are responsible for the maintenance of their own data. They have little knowledge of displays on the screen and no knowledge of pixels, colors, or mouse clicks (which are handled by the view-controller objects known as "Cognits"). Views, on the other hand, must understand a great deal about the objects they intend to represent. For almost every object on the screen, a corresponding data item of the user's program is represented, and Cognits must understand that item. When user interaction is enabled, Cognits supports user modification of the program's data. As objects are developed to represent data within the program, corresponding Cognits will be developed to allow the representation of these data. Just as complex structures may be built from simple substructures, complex Cognits may be built by aggregating simpler ones.

A major problem in the development of interfaces is the specification of the details of managing the user's interaction with objects on the screen. These details include determining when the object should be drawn, what to do when a mouse click occurs on the object, what to do when the object's window is closed, and so on. The interface objects, the Cognits, handle the details of this interaction for the user. In general, the declaration of a Cognit contains all the information required to manage the object. As is illustrated in the examples below, the developer needs to know very little about the underlying Cognit structure to create reasonable interfaces.

Two classes of data may be represented by Cognits—"smart" and "dumb". Classical data, *ints floats*, and *structs* are "dumb." For example, unless there is external coding, the address of an *int* cannot tell a representing object what type of data are represented or when the data are altered. This places a burden on the widgets and the programmer to keep the image on the screen consistent with the data in memory. In contrast, objects are "smart". Any object in the class library has methods for dealing with representations on the screen. Objects are "aware" that they are represented on the screen and can thus update the display when they are altered. The display itself is also capable of sending messages to an object, including requests for the type of object and the requirements of a display. The most sophisticated displays represent objects.

While the models, the underlying data, have little responsibility to understand the details of the interface in terms of window placement, color, or font, they do have responsibilities to cooperate with the interface. Models whose data have changed are required to request the interface to update its display. When a data object is disposed of, it must also guarantee the disposal of all views. The management of these responsibilities falls under the

broader heading of *interobject connectivity*, discussed in detail below.

System Architecture

Connectivity

A major issue in object-oriented design concerns the maintenance of relationships among objects. Any time two objects establish a persistent relationship, that is, one that continues beyond the scope of immediate execution, a number of issues arise. Foremost among these is how to maintain the validity of the relationship, as the following example illustrates. Suppose that object A contains a reference to object B. If at some time in the future object B is destroyed, any attempt to access it through that reference will have disastrous consequences for the system. It is therefore necessary to guarantee that such references cannot take place. A "safe" relationship between A and B is one whereby the destruction of object B requires the immediate destruction of object A. While this sounds Draconian, we can use this approach to maintain any possible relationship.

In a "parent-client" relationship, the client is dependent on the parent for its continuing existence. Clients have only one parent and maintain a pointer to that parent. When the client is destroyed, a message is sent to the parent for it to delete all references to the client. When a client's parent is destroyed all clients are destroyed, so no dereferencing is possible. Keeping a pointer to the parent's parent or any ancestor is also safe.

A second relationship is maintained by a *connection*. A connection is an object that holds pointers to a target and a source object. It is a client of both target and source, so the destruction of either will destroy the connection. When a connection is destroyed, both target and source receive messages indicating the loss of the connection, and either may take appropriate actions. In some cases, for example when the target is a view of the source, the destruction of the source will also cause the destruction of the view.

Representation

Any object in the data space may build one or more views on the screen. The relationship between an object and a view on the screen is maintained by a special class of connections called a views. Every widget that represents an object maintains a pointer to the represented object. Widgets without direct representation, such as a text field representing a float within the object, or a button in a window representing an object, are treated as views of the object represented by the enclosing widget.

The major method relating a view to its object is *Reconcile*. Reconcile causes the view to examine data in the represented object and adjust the window to correspond to the current state of the model. The model is never altered during this operation. The responsibility for developing the display rests entirely with the widgets. The model merely has the responsibility to use the *UpdateDisplay* method, which sends *Reconcile* messages to all displays.

When an object is disposed of, it disposes of all connections. The disposal of the view connection will kill the corresponding widget.

Implementation Decisions

Design Decisions for Controls

Three possible models have been used for controls. (1) Under most X Widgets systems and also under MS Windows, every control, text field, button, and scroll bar is a separate window. Events such as mouse clicks are received by the system and sent directly to the control's window. (2) The default behavior in the Macintosh makes each control an active region within a window. The operating system provides a means to locate the mouse click within a particular control. However, it is still the responsibility of the application to call this routine and then generate the code to handle the event. (3) A third model treats a window as a canvas on which controls are painted. In this model the operating system has no knowledge of the structure of the window. Events are sent to the window, which in turn is responsible for determining which region was selected and responding appropriately. In this approach the underlying system sees no physical structures for controls. Rather, controls are a logical construct maintained by each application.

Cognits uses the latter model for its widgets. Widgets within a window are arranged in a hierarchy, with the window as the root. Every widget defines a region (not necessarily a rectangle)

under its influence. Child widgets lie above the parent, and their visible region of influence is the intersection of their own region with that of all ancestors. The method *PointWithin* takes a point in a window and returns the highest widget enclosing that point. The widget receives the mouse event and is responsible for handling it. The declarations of logical regions are the responsibility of the application.

The choice of drawn widgets increases portability, since it minimizes the responsibilities of the underlying GUI (graphical user interface) to those of providing a canvas and basic drawing routines. It means that the application is responsible for handling problems resulting from overlapping widgets. In Cognits, widgets are drawn from back to front, ensuring their proper appearance when an entire window is drawn. When a lower widget is updated, it is responsible for updating all overlaying widgets.

Cognits does not support standard Macintosh controls such as buttons, text fields, and scroll bars. Instead, objects with the look and feel of the Macintosh resource controls are drawn by Cognits objects. All active objects are drawn within a window, with the window object distributing events to the appropriate active object. This architecture allows event handing to be achieved by a method within each object. The high-level program need only send an event to the target window, receive the target object, and actuate the proper method within that object. All event handling is then local within each widget. As noted earlier, the decision not to use controls on any system was critical to the maintenance of portability. Nevertheless, the objects used by Cognits are designed to retain the look and feel of the original controls.

Cognits Controls

Cognits buttons were designed to support all button types supported by HyperCard: Shadow, RoundShadow, transparent, and also several buttons with a three-dimensional look similar to those used by Motif and MS Windows. Scroll bars are laid out in a manner similar to those of the Macintosh scroll bars, but with a drag button that indicates the fraction of the visible section.

Text fields do not use the Macintosh text field control but instead are drawn objects. Two classes of text field are supported: output-only and editable. Output-only text fields do not allow editing. While in some classes of output-only text field, mouse clicks will initiate specific actions (e.g. sending the line number to a completion routine), users may never alter or even drag and select the underlying text. Output-only text fields have been subclassed to support a variety of different formatting, such as an ability to set tab stops and draw boxes around portions of each line. Subclasses of output-only text fields can support the selection of multiple, separated lines and may be used as a text list-processor object.

Editable text fields are objects that support the general behavior of the original text fields but are restricted to a single font, style, size, and color. By retaining all of the code within the editable text field class, interesting behaviors can be supported. For example, as with Macintosh text, double-clicking selects the current word and allows the user to draw, to select more words. Triple-clicking selects the current line and allows the user to drag, to select more lines. Quadruple-clicking selects all text.

Windows Defined in ResEdit

Cognits does not support standard resources for the specification of windows, menus, and other controls. The decision not to support resources was made for three reasons. First, the use of resources requires a significant separation of functional specifications. The resources state which objects are created within a window, and the code for managing the window states which objects will be supported. However, there is no guarantee that these be the same set of objects—i.e., that code will be available for an object in a window resource or that objects referenced in the code will be present in the resource. Anyone who thinks resources are "safe" should try using ResEdit to add an item as the first item in the menu of an existing application. Second, many windows are dynamic, with the objects present in the window unknown until run time. In Cognits, this type of window is treated as the rule rather than the exception. Third, resources are difficult to port across systems. If much of the information about a window is held in resources, it will be difficult to achieve the goal of supporting the same screen appearance across multiple platforms.

The Cognits approach is to allow objects described in the code to create and manage resources describing size, color, and location. This forces resources to correspond to objects referenced in the program. The

resource editor is not a separate program but is built into all objects capable of supporting resources.

Portability

Portability is a major design driver. It was the desire of the author to write portable code, which motivated the development of his own class library, rather make use of existing libraries. Cognits achieves portability in three ways. First, no object interacts directly with the underlying graphic system. Instead, all objects interact with a virtual GUI through a series of defined subroutine calls. Most of the drawing calls have the same names and arguments as the QuickDraw traps. File access follows POSIX conventions. Macintosh file-access structures are on alternative systems versions of QuickDraw. Calls are written with the underlying draw commands. Second, much of the more sophisticated functionality is not utilized. For example, the ListManager is not available; instead, applications are expected to develop equivalent functionality by means of lower-level Quickdraw routines. Third, high-level objects do not include system-specific header files such as Quickdraw.h, which forces all operations to occur through system independent low-level calls.

The Cognits system recognizes a collection of system-dependent opaque types. These structures represent, for example, native windows, graphports, and fonts. Each type may be referenced in system-independent code only as an argument to a collection of system-dependent subroutines. These routines create, destroy, manipulate, and access key data from these structures.

The system-dependent structures of Cognits include the following:

WINDOWPtr: A window on the native system. Windows may be native, headerless, or temporary.

BITMAPPtr: A structure that may be selected for drawing. This is a Mac GraphPort or an X Drawable. Every WINDOWPtr contains a BITMAPPtr. BITMAPPtrs may be either on screen (in a window) or off screen.

PICTUREPtr: A structure that can store drawing commands. On the Mac this is a Pict structure, and on other systems it is a collection of postscript text.

MENUPtr: A structure representing a pulldown menu. This is defined only on systems (such as the Mac and MS Windows) that supply native menus.
CURSOR: A structure that represents a system cursor.

Drawing Model

Cognits supports a drawing model similar to that used in the Mac toolbox. Drawing commands such as DrawLine and PaintRect are called only with coordinates. The system maintains a current port (BITMAPPtr) and current pen colors, style, and size, as well as the current font. Characteristics of this implicit port are altered in separate calls to set color, font, and port. This differs from the model used by MS Windows and X Windows, where the port and in some cases the pen are part of the call to draw.

Events Model

Cognits translates events from the underlying system into its own event record. The structure of the events model is similar to that of the toolbox but the event types are different. Mouse clicks are translated into generic clicks of a virtual mouse with a large number of buttons (e.g. a single mouse click is called a BlackMouseClick, a click while the command key is held is called a RedMouseClick, and a double click is called a PinkMouseClick). This frees the widget from the need to determine the type or number of mouse clicks, by placing all of this code in the event manager. It also allows clicks on a multiple-button mouse to be mapped in the event manager without affecting the rest of the program's code. In addition, the event manager takes care of events within window controls such as the go-away box.

Updates

A traditional approach in Macintosh programming is to handle all drawing through update events. An update may thus occur for two reasons. First, the system may determine that a region of the window has become invalid; this occurs when the window is first exposed or when an overlaying window is removed, revealing a previously obscured section of the window. An application may use this invalidation when the data represented by a section of the window are invalid and require updating.

Cognits uses only the first mechanism. In Cognits, when an object such as a text field wants to update itself, it marks itself as "dirty" and sends messages to enclosing objects that a subobject requires updating.

As part of the event loop, a Redraw command is issued to redraw all dirty objects. This mechanism is used for two reasons. First, it separates what are two distinct operations: one in which an object requires redrawing because a region of the object's window has been obscured, and another in which an object requires redrawing because the data that object represents require a change in the screen image. In the former case a current Picture or offscreen BitMap of the window may be used as an efficient alternative to redrawing. In the latter case not only will objects be required to redraw, but they will also have to alter any internal variables to represent the current state of the data. These are very different operations.

Another reason for using an internal update mechanism is that if only a portion of the window is invalid, an efficient program will redraw only those objects lying within the invalid region. If redrawing is performed by sending messages to invalidated widgets, the application will first have to find those widgets that lie within the invalid region. If a widget were to invalidate a region rather than setting an internal invalid flag, on receipt of an update event it would have to test whether it lies in the invalid region and requires updating. If the update is an internal request, the latter test is redundant, because the requesting widget "knows" it is invalid.

Cognits Classes

The following are some of the major classes of controls contained in the Cognits class library:

WindowWidget: The basic window widget is a standard Macintosh window. The root class generates the system window with all default controls. Subclasses include the **HeaderlessWindow,** which generates a plain window with no title bar, and the **NextWindow,** which is a headerless window with a titlebar and grow box added as Cognits widgets rather than as Macintosh controls.

ButtonWidget: This is a standard button. Buttons represent a callback function that may be activated when the button is pressed. The look of the button may be altered by setting internal flags as shown below. Button subclasses include **IconButtonWidget**, which displays an icon instead of text, and **MenuButtonWidget,** which brings up a pop-up menu when pressed.

TextField: Textfields are capable of displaying text. Cognits supports two broad classes of TextField. **OutputOnlyTextWidgets** can display text but do not support text editing. Users may select text by line or select the entire field, but other granularity of selection is not supported in this class. Removing the need to edit and maintain a correspondence between position and characters allows richer output styles. Subclasses support alignment to tab stops and different coloring and fonts in different columns.

The **EditableTextObject** represents an object similar to early TextEdit fields. Only a single font and color for all text are supported. Cut and paste, and selection by character, word, or line are also supported. Text length is not limited to 32K but rather is constrained only by system memory.

EditableTextObjects are subclassed according to the type of data represented. The text object has the responsibility of representing the following: **FloatWidgets** represent doubles, **IntegerWidgets** represent ints, **BooleanWidgets** represent doubles. Other specialized widgets represent Points, pointers to Objects.

GraphSurface: This is the fundamental object for scientific graphics. The GraphSurface defines a mapping from x and y values to screen coordinates. GraphSurfaces may autoscale to guarantee that all objects within the surface are visible. When drawn, the surface is responsible for adding titles, labels, and a grid to the resultant graph.

GraphSurfaceObject: This is an object representing some data given as floating-point x,y pairs. GraphSurfaceObjects are responsible for drawing their data in a coordinate system defined by the GraphSurface, and for telling the surface what the limits of their represented values are. **GraphSurfaceObject**s may represent a single point, a collection of points, or a polygon.

ScrollBars: Scrollbars are not treated as an independent object. They are always created programatically by the object requiring scrolling and are a view of that object. Scrollable objects support two methods, one to indicate their state of scroll and one to set their scroll to a new state. Scrollbars support user interaction to alter the object's scroll state.

Dynamic Windows

Cognits recognizes two types of windows: static and dynamic. Static windows have resources describing the size and placement of all objects within the window. Dynamic windows are created at runtime with no knowledge of the window's contents. Cognits places objects within windows as the windows are created. A series of procedures is used to place widgets automatically within a developing window. When objects are generated they may be sized and placed by passing in a Quickdraw Rect. If no Rect is passed, the objects themselves perform default placement and sizing. The precise algorithm is class-dependent. Buttons arrange themselves in a row until the next button will not fit in the enclosing object, then they begin a new row. Many larger objects will place themselves at the left of the enclosing space. Resizable objects such as graphs will place themselves below the last object and occupy the lower portion of the window. Hints may be passed to objects for better placement. The EndRow hint causes the next object to be placed at the left edge, below other objects.

A particularly useful method is SizeToFit. This sets the size of an object to the smallest size that will enclose all subobjects. A typical window is built by creating a window of some default size, placing a number of objects within the window, and finally calling SizeToFit to set the window size to that required to hold all subobjects.

The advantage of dynamic placement and sizing is flexibility. While windows may not look as good as they would when objects are handplaced, they allow windows to maintain a reasonable look as objects are added or deleted. In most phases of code development, the ability to generate reasonable windows with minimal coding is an advantage. Once the contents of a window have gelled, the window may be made static and handcrafted as described below.

Static Windows

Static windows allow the window's characteristics to be stored in resources. The resource does not describe a Macintosh control but rather a collection of parameters for each widget in the window. Parameters include foreground and background colors; font type and size; object placement; properties such as border (or no border) rounded (or square) corners, and shadow. The properties describe the way a widget is drawn, but they do not specify what the widget is, what data it represents, or how it interacts with the user. Those details are specified in the code that created the widget.

Static windows differ from windows created from Resedit in that the objects within the static window are created by the code rather than the resource. With Cognits static windows, widgets are created that search for resources to tell them their look and feel. With resource dialogs, controls are created to have a particular a look and feel, then look for code to tell them their functionality.

Static windows may enter a ResEdit-like mode where users may use the mouse to size and located all objects. Double-clicking on objects in this mode opens an editor window, allowing the modification of color, font, and style. Users can then remember the new configuration by copying a resource description. This capability moves the responsibility for the window configuration from a separate editor to the application, and allows end users to reconfigure displays easily.

Portable Dialogs

The proper operation of the Macintosh depends on a number of standard dialogs, which are used to open files, select colors, and control printing. Some systems use similar dialogs; for example, MS Windows has dialogs for file selection. It is uniformly true that the call and the arguments differ radically. The solution is to wrap system- dependent dialogs in a wrapper that offers the application a consistent view of the transaction. For example, the boolean GetExistingFile(char TheName[]) takes a pointer to a character buffer that may contain a candidate file name. After the dialog, TheName contains a string containing the file name, including directory and volume (if needed). A return is true if a valid selection is made. An earlier call can restrict choices to a specific creator or extension.

The strategy is to determine the nature of the final product of the dialog, then to default most arguments and allow a simple call to proceed. When no equivalent dialog exists, a window is generated that presents the user with the appropriate choices, and events are processed until the dialog is dismissed.

Cognits does not support the concept of modal dialogs. Instead, users may always access other applications or desk accessories from within a dialog; in most cases, users may quit the application or access other windows without leaving a dialog. A "nag" feature

keeps popping the current dialog to the top at regular intervals so the user cannot simply ignore it.

Cognits Paradigms

Macintosh programmers are very familiar with the code required to handle significant Macintosh events such as key presses or mouse clicks. In the Cognits environment, events are translated into methods that are invoked. The way Cognits handles a mouse click is shown below.

Button Press

Normally buttons are created with the following code:

SimpleButton("ButtonName",ButtonWindow,ButtonCallback)

where ButtonWindow is the window or containing widget, and ButtonCallback is a callback function taking a single argument, the button's address. Normally developers will not have to deal any more extensively with buttons. The discussion below details the way the underlying system handles a button click.

• WaitNextEvent returns an event.

• The event is sent to the event manager.

• The event manager determines that the event is a mouse click and sends it to the mouse manager.

• The event manager determines that the click occurs in the window's content region. If not it is passed to other handlers.

• The mouse manager tests for modifier keys and multiple clicks. It then classifies the event. For example, an unmodified click is a BlackMouseDown, a command-click is a RedMouseDown, and a double click is a PinkMouseDown.

• The event manager finds the WindowWidget that is managing the affected window. The address of this object is the Window's Refcon.

• The Event manager asks whether the WindowWidget is the currently active window. If not, the Widget is activated and no further action is taken.

• The Event manager then invokes the appropriate method for the window. For example, for an ordinary click it will call the Window's BlackMouseDown method. BlackMouse is a click of a virtual mouse button representing an unmodified mouse click.

TheWindow->BlackMouseDown(ThePoint);

• The default window's mouse-click handler locates the actual widget clicked on. It calls

ActualWidget =
TheWindow->PointWithin(ThePoint);

(This method tests the point against the active region of all objects in the window. The highest object that includes the point is returned. If *TheWindow* is returned, then the click occurred outside any active widget and no action is taken.)

• The mouse handler for the active widget is invoked:

ActualWidget ->BlackMouseDown(ThePoint)

The default action is to track the mouse:

if(this ->TrackMouse())

(TrackMouse will call Hilite when the mouse is within the Widget, and UnHilite when the mouse leaves. The method continues tracking the mouse until the mouse is released. It then returns true if the release was within the Widget, and false if not.

• A button will then invoke the Callback method. Widgets hold the address of a callback routine that is invoked at the end of the interaction with the user:

this ->Callback();

Callback by default does the following:

(MyCallbackRoutine)(this);*

where *MyCallbackRoutine* is the address of a routine that takes the widget as an argument.

The above approach allows many levels of flexibility: a widget that wants to pop up a menu will override the BlackMouseDown handler to pop

up a menu as soon as a click is received; a widget that is to be dragged by the mouse will override the TrackMouse method to support dragging; and CheckBoxes override Callback to toggle their data before invoking the callback routine. Users are rarely required to deal with the code at this level, and only when designing special-purpose widgets that have interesting interactions.

Display Update

Normally the way a field is updated is that an object executes the method UpdateDisplay() without "worrying" about the resultant effect. The following discusses what happens when the display is updated.

The object sends an Update message to each View Object. The view then passes the update to the target, the affected widget. The widget in turn will send Reconcile to all subwidgets, and this will be passed recursively throughout the structure.

On receiving a Reconcile message a FloatWidget (that is, a widget that represents a floating-point number) will look in its attachment. The attachment is the address of a float that the widget represents; it compares the current value of the represented float with an internal float representing the value associated with the current text. If the two values are different, it replaces the internal value with the updated value, formats the appropriate text, and sets its internal text.

Next the widget calls SetDirty(). This sets a flag, the Dirty Flag, indicating that the next time the window is redrawn the widget must redraw itself. It also sets a flag in all ancestors requesting a redraw. The redraw method causes all widgets with dirty subwidgets to send Redraw to all subwidgets. Widgets with the Dirty Flag set Redraw themselves. Redraw may be issued under program control but is usually issued as part of the event loop.

Programming in the Cognits Environment

Programming in Cognits uses the following paradigm. A certain, frequently small collection of objects is created. At creation time these objects are given instructions as to how to behave and what they represent. Users rarely need to deal with the event handler; they merely provide callback routines to handle the results of pushing buttons and setting values. The programmer need not deal with issues of placement and sizing. The only statement referring to sizing and placement in the examples below is the method SizeToFit, which makes a window or other widget large enough to hold all the objects it encloses.

Note in the first example the line SimpleButton, which creates the buttons and gives all the required information as to how to draw and manage the buttons. Once the buttons are created the program need not worry about them.

While the actions of New, Open, Save, and Save As under the file menu may be altered by the application, the default actions are illustrated in the second example. The statement *SetDocumentClass("Patient")* says the application will be dealing with objects of class Patient. This will cause New to create an instance of the class Patient and make this the current document. It will also cause Open to open text files and seek instructions to create objects of this class. Save and Save As request the current document to write a description of itself to disk.

Programming Examples

The following are simple examples that illustrate principles of Cognits Programming. The first shows the basic program skeleton and details how the program is set up and events are handled. The second example shows how a new class is built and how an application may be developed to deal with that object.

Button Window Example

This creates and displays a window called "Button Window". It adds two buttons called Beep and Quit to the window, then sets up callback functions that define the action of each of the buttons. Buttons are representations CallbackFunction, which is discussed below.

// DoWidgetQuit is a built-in callback which does

```
// what WIDGETs programs normally do when QUIT is selected
from the file menu  extern void DoWidgetQuit(WIDGET
*Ignore);

// ***********************************************
//        This is a simple Callback function - executed when the
Beep                                    //      Button is pressed.
Callback functions have no return and take
//      a single argument:  the affected Widget
// ********************************************** **

void DoBeep(WIDGET *Ignore)
{
        Beep();
}

     /* System dependent main declaration */
DECLARE_MAIN
{
        WIDGET *TheWindow;   /* This is the window we will
create                  */
        /* Initialize - This should be the first executable
statement */
        INITIALIZE_MAIN
        /* Make theWindow  */
        TheWindow =
             SimpleWindow("Button Window");
       /* Makes a button which Beeps   */
        SimpleButton("Beep",TheWindow,DoBeep);
      /*  make  button which Quits   */
        SimpleButton("Quit",TheWindow,
                DoWidgetQuit);
      /* Resize TheWindow */
        TheWindow->SizeToFit();
       /* Show TheWindow */
        TheWindow->Activate();                  /* Handle
events     */
        SWEventLoop();
}
```

Code Comments

DECLARE_MAIN: This macro declares the main program. A macro was used to accommodate differing styles of main declaration: under Unix, main is declared with two arguments: argc and argv; on the Mac, main has no arguments; and under Windows, main has four. The solution is to make the main declaration a system-dependent macro, and leave the job of processing input arguments to another system-dependent macro, *INITIALIZE_MAIN*.

INITIALIZE_MAIN: This should be the first executable statement in the main after the declaration of variables. It is responsible for handling any command-line arguments, for initializing the Cognits, and for initiating the connection with the server. The code called up by this macro is system-dependent and may change in later versions of Cognits. The variables accessed are hidden from the user. INITIALIZE_MAIN may declare local variables and thus needs to be placed in a position to do so.

SWEventLoop(): This handles all events. In an event-driven program, the art of programming is in developing a structure capable of handling all allowed events. This loop waits for events and then processes them. If no events are generated by the user, the system will periodically (ideally about 5 to 10 times per second) generate NullEvents, initiating background processing. The program executes until the user selects Quit under the file menu or other events cause the routine *DoWidgetQuit()* to be called.

Callback Functions

A CallBackFunction is a function that is called following the interaction of a user with a widget. In the case of Buttons, MenuItems and several other classes of Cognit, the sole function of the widget is to present a CallbackFunction to the user. CallbackFunctions take a single argument, the address of the calling widget, and they return nothing. In the functions used in the following example, the calling widget is ignored.

Example: ACallbackFunction That Does Nothing but Beep

```
void DoBeep(WIDGET *Ignore)
{
        Beep();
}
```

A major topic is that of access to data from a calling function. Because the only argument is the calling widget, only the following data are accessible to a CallbackFunction:

(a) Global variables. Good practice minimizes the use of these.

(b) Data pointed to by the widget itself. Widgets have a pointer to an object represented and may have other internal pointers to data.

(c) All ancestor and child widgets. Actually, one may traverse the widget tree and access any widget in the system.

(d) Any data pointed to by any accessible widget.

(e) Any data pointed to by any accessible object.

The techniques for accessing data from CallbackFunctions will be discussed in more detail below.

A Discussion of Functions Called in the Code

This section discussed in detail the calls used in the ButtonWindow example.

*void *SimpleButton(char *Name,WIDGET *Parent,CallbackFunction TheAction)*

The above routine creates a ButtonWidget. ButtonWidgets are simple objects that represent an action to the user. When the user clicks on a button, the button remains highlighted as long as the mouse remains within the button. If the mouse is released within the button, the CallbackFunction is executed.

*void DoWidgetExit(WIDGET *TheWidget)*

This routine exits the program after performing cleanup. It is the routine called by the Quit selection on the file menu. The widget passed in is ignored and can be NULL. The details of what is done on exit and how to modify this will be discussed below.

void Beep(void)

Do whatever is needed to beep the standard system beep in the standard manner.

Note that we did not remember the address of the buttons. Once a button was created, the program specified all the information required to manage it. Also note that the CallbackFunctions do not access external data. Later, the ways that CallbackFunctions access data will be a major topic for discussion.

Example: A Program to Represent a Class of Objects

The following example shows how an application can be designed to deal with a specific class of data. First, the class Patient is derived from the class ParentObject. The parent class already has and supports a name field. Then fields are added for Age, Height, Weight, and Sex. The macros CLASS_ATOM and START_METHODS declare auxiliary structures and methods. OBLIGATORY_METHODS declares these methods and creates the structures. These macros are deliberately designed to allow flexibility in their actions. START_VARIABLES is a macro that declares methods related to new fields. When new fields are added, the BROWSEDECLARATION macro and subsequent lines are needed to support browsing of the new fields.

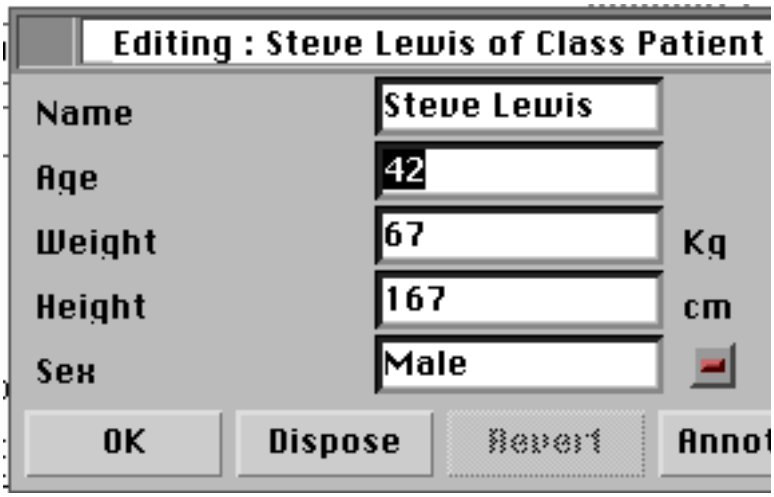
The class declares a single method—BuildEditor. This adds widgets into an editor window. One line per field is required to add views of each of the fields. Note that in the declaration SimpleLabeledBoolean for Sex, we declare the user's view of false as Female and true as Male. The rest of the editor window (see illustration below) is set up in higher-level methods. The Type argument is ignored but may be used to support different editor windows.

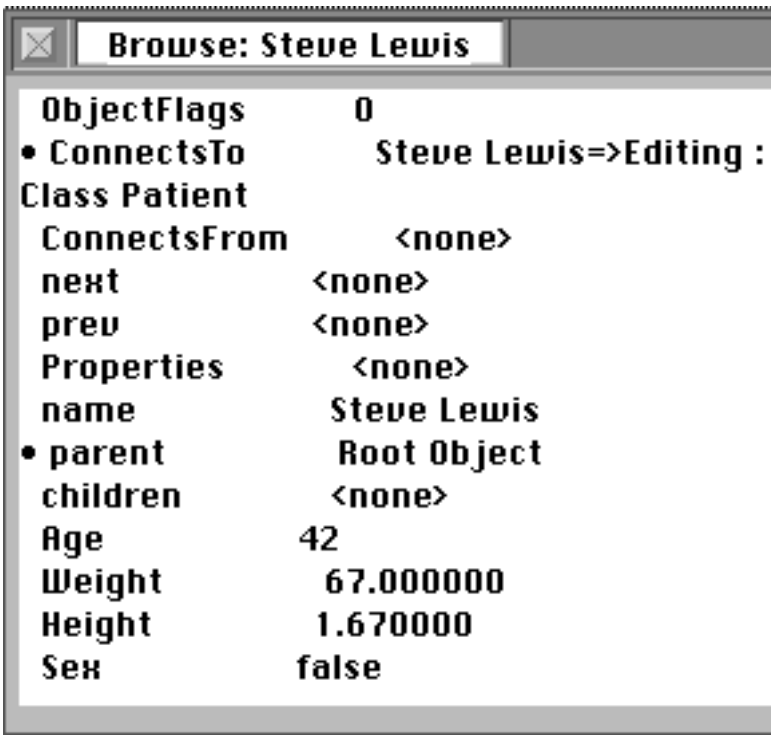
The main program has three setup calls. First, it assigns the text strip to a global related to text. Second, it tells the system that the current document will be of class Patient. This will cause New to create an instance of Patient and call that object's Edit method, opening up an edit window. It also says that only a PatientObject is acceptable as the CurrentObject. If we had overridden two more methods in PatientObject: WriteSelf and KeyWordRead, we could support saving and opening files describing these objects. Finally, ShowAboutWindow brings up the About window as the application opens.

```
CLASS_ATOM(Patient);

class Patient : public ParentObject
{
        // declare methods for new fields
        START_VARIABLES
        int     Age;
        FLOAT   Height;
        FLOAT   Weight;
        boolean Sex;
        /* ************************ */
        /*      Methods         */
        /* ************************ */
```

```
        // declare methods for all classes
        START_METHODS(Patient)
        virtual WIDGET *BuildEditor(WIDGET
*TheDisplay,Atomic *Designator);

};

/* ******************** */
/* Class Macros                    */
/* ********************  */
#undef THISCLASS
#define THISCLASS Patient
#undef SUPERCLASS
#define SUPERCLASS ParentObject

OBLIGATORY_METHODS(Patient,ParentObject);

/*  ************************
        Code for supporting the browser

  ******************** */

void THISCLASS::BROWSE_DECLARATION    /* Obligatory
Boilerplate          */
{
        BROWSE_VARIABLES;         /* Obligatory Boilerplate
*/
        SUPERCLASS::BROWSE_CALL;   /* Obligatory
Boilerplate                */

         // Tell the Browser about Age
        INTEGER_FIELD(Age);
        FLOAT_FIELD(Weight);  // Weight
         FLOAT_FIELD(Height);  // Height
         BOOLEAN_FIELD(Sex);   // sex
}

/* ********************************************

        WIDGET *BuildEditor(WIDGET *TheWidget,Atomic
*Designator)
                Extend adds new fields
  ******************************************** */
WIDGET *Patient::BuildEditor(WIDGET *TheDisplay,Atomic
*Designator)
{
        // SUPERCLASS adds field for name - an inherited
field
        SUPERCLASS::BuildEditor(TheDisplay,Designator);
        // add widgets to represent each of the new fields
        SimpleLabeledInteger("Age",   TheDisplay,&Age,
NULL);

NewLabeledUnitWidget("Weight",TheDisplay,&Weight,"Kg",NU
LL);

NewLabeledUnitWidget("Height",TheDisplay,&Height,"cm",NU
LL);
        SimpleLabeledBoolean("Sex",   TheDisplay,&Sex,
"Male","Female",NULL);
}

// ******************************************
// Text for the About window
// ******************************************
char *AboutPatients  = "Welcome to Patients\r\r"
"Select New or Open from the File Menu\r\r"
"Developed by Steven M. Lewis,Ph.D.\r"
"The Aerospace Corp".;

DECLARE_MAIN
{
        INITIALIZE_MAIN
      // what the About window says
        AboutText = AboutPatients;
     // we are dealing with Patients
        SetDocumentClass("Patient");
   // Start with an About window
        ShowAboutWindow();      SWEventLoop();          //
Handle events
}
```

Selecting New from the file menu brings up the window shown below. Initially, Name is "Untitled" and all fields are 0 (zero). After the window is filled in it, it looks like the illustration below. The units Kg and cm are associated with Popup menus of alternative units. The button for Sex will toggle Male to Female.


Editing : Steve Lewis of Class Patient
Name
Steve Lewis
Age
42
Weight
67
Kg
Height
167
cm
Sex
Male
OK
Dispose
Revert
Annot


Browse: Steve Lewis
ObjectFlags 0
• ConnectsTo Steve Lewis=>Editing :
Class Patient
ConnectsFrom <none>
next <none>
prev <none>
Properties <none>
name Steve Lewis
• parent Root Object
children <none>
Age 42
Weight 67.000000
Height 1.670000
Sex false

Clicking on the editor window while holding Option and Shift brings up a browser of the Patient represented by the window. This is scrolled to show the lines added to the browser by the *BROWSE_DECLARATION* method.

Code

Most of the code in this example created an Object editor. The BuildEditor command takes a window and adds specific fields for the particular object. The Baseclass Edit command creates the window, adds buttons at the end, resizes it, and displays it. Note that the only code we needed to supply to manage the class was the code needed to add the new fields to the editor and the browser

*WIDGET*SimpleLabeledInteger(char *Name,WIDGET *Parent, int *TheNumber, CallbackFunction TheAction)*

This routine creates a LabeledInteger widget, a composite widget consisting of a label and a text field representing an integer. TheAction is called if the user alters the value by editing the text field.

*WIDGET*SimpleLabeledBoolean(char *Name,WIDGET *Parent, boolean *TheNumber, char *FalseText,char *TrueText,CallbackFunction TheAction)*

This routine creates a LabeledBoolean widget, a composite widget consisting of a label, a text field representing a boolean, and a toggle button. The text will either be *FalseText* or *TrueText*, depending on the boolean's value. The toggle button will toggle the field's value, as will clicking on the text. The toggle is thus a visible indicator that clicking on the text acts like a button. TheAction is called if the user alters the value.

*WIDGET*NewLabeledUnitWidget(char *Name,WIDGET *Parent, double *TheNumber, char *UnitName,CallbackFunction TheAction)*

This routine creates a LabeledUnit widget, a composite widget consisting of a label, a text field representing a boolean, and a Unit widget. The text will either be the value of a floating-point number represented in the indicated units or blank if there is no value For example, if the text is "lbs" the text will show the value multiplied by 2.54, the number of kg per pound. All internal values are in MKS units. Clicking on the units brings up a popup menu of alternative units. For example, for lbs the alternatives would all be units of weight. All user input is interpreted in the alternative units and converted to MKS units for internal storage.

ShowAboutWidget(void)

This routine shows the AboutWindow. There is a default About window that shows *AboutText*. The routine to create this window may be replaced in user code to achieve different effects.

SetDocumentClass(char ClassName)

This routine shows the AboutWindow. There is a default About window that shows *AboutText*. The routine to create this window may be replaced in user code to achieve different effects.

Examples Summary

The examples show the ease with which fairly sophisticated applications can be developed with Cognits. The developer is not concerned with details such as management of or even placement of widgets. These are handled by the run time system. The ease with which interfaces can be developed is so powerful that throwaway interfaces can be created in the course of program development merely to give developers better access to a developing system.

Future Directions

Future plans involve expansion in two directions. First, the number of systems which are supported will be increased. The Cognits package currently runs on the Mac, X Windows and DOS using Borland graphics. In the near future, a version will be available under Microsoft Windows. A display postscript version is contemplated. Another path is to use the XVT™ toolkit to generate a generic version which relies on XVT™ for portability.

A second area of expansion is functionality. Future versions will employ automatic code generators which parse header files and automatically write the more routine code functions such as browsers and simple displays. Persistence is currently supported by writing descriptions to ASCII files. Future versions are planned to be able to read and write objects to relational data bases. Portable use of sound and animated images is also under consideration.

Summary

The Cognits package is a class library that simplifies programming on the Macintosh. Cognits combines the functionality of MacApp with added features, including scientific graphics and built-in object and class browsers. In addition, by making all calls to a virtual GUI, the package is portable and allows developers to port well-behaved Macintosh applications cleanly to other operating systems. Cognits distributes responsibility to objects allowing good interfaces to be developed with little more that a statement of the types of data requiring representation. The ease with which interfaces can be developed is so powerful that throwaway interfaces can be created in the course of program development merely to give developers better access to a developing system. We believe that these object oriented tools represent a powerful new paradigm for program development.