Building Extensible Applications Using Object-Oriented Methadologies

(Version 1.0)

by Shane D. Looker

Abstract

Over the past few years object oriented programming has been the source of countless papers and books. The promise of object oriented programming is appealing to the software author because of the ease of new code development. However, it's benefits have not been directly available to the end user

This paper presents a way for application writers to allow extension modules written outside of the application development environment to be utilized by end users. This method allows users to select custom modules they desire in a product, and create a "roll-your-own" application more directly suited to their needs.

The methods presented in this paper build upon ideas already familiar in the Macintosh environment: WDEF, CDEF, cdev, etc. A discussion of both the application design and the module design is given, along with suggestions for potential applications and modules

Introduction

The Macintosh is evolving continually. From its humble beginnings as the 128K Mac, up to the newest machine, the Macintosh IIfx it has evolved through hardware, and its software has evolved alongside that hardware growth, from its original System with MacPaint and MacWrite, up to System 7.0 and thousands of applications available.

Yet in some ways, the evolution of the Macintosh has diverted from its original intent. Instead of hundreds of applications being used in conjunction to form a whole, we have recently be inundated with megalithic programs which "do it all." Along with these programs come a multitude of bugs and a steep learning curve.

It is time to return to the original concepts of the Macintosh, though those concepts have also evolved. With the onset of object-oriented programming in the past few years, it has become possible to combine several of the concepts behind object-oriented programming with the single-function applications found originally on the Macintosh. By combining these we can create applications which are extensible by the users who actually know what they need to perform their tasks.

This paper presents an approach to building extensible applications which utilizes some of the methodologies from object-oriented programming. Object modules which can be loaded at application run-time are described along with their functional requirements. The interface of the application to the modules is presented along with the details on how to manage objects in an application in a generic fashion.

Macintosh users should be able to buy an application which provides base functionality for their work. Then they should be able to enhance their application with modules which mesh into the software they already have, expanding it beyond the bounds of a traditional single application. A word processor should not just format text, it should allow the user to create drawings and illustrations for a document, add charts, sounds, and even animation or video, *without* being a different application. Graphics users should be able to add new drawing and painting tools to fit the needs of a job, not be forced to purchase a new application this week, and another next month. A spreadsheet user should be able to take data and transform it without spending days writing a macro to perform an esoteric function. A communication program user should be able to add new terminal types, connection abilities, and file transfer protocols without changing programs. And all users should be able to share files and information through integrated mail in any application.

By writing applications which don't have to "do it all", software developers benefit from smaller applications, which can result in a faster development cycle, and the ability to add new features to an application *after* the product is released.

End users also benefit from this approach with easier to use products, the flexibility to select needed tools for any specific task, and a much smaller learning curve, since tools can be taken from one program to another.

So the question is, how can this be done? The answer is that it can be accomplished by applying some of the methodologies developed for use in object-oriented programming and applying them not just to source code, but to the entire application concept.

The important concepts to be brought over to extensible applications are close data and function coupling along with messaging. Using data/function coupling, we can hide all the details behind an object and its methods from the user, while delivering functionality. Messaging is used to communicate between the application and the individual objects being used.

Even though the concepts being presented here are based on the object-oriented model, they do not require an object-oriented language for implementation. Because of the lack of dynamic binding available in the object-oriented languages currently in vogue, all the interfaces between applications and modules are described in terms of procedural programming.

Some Examples of Modules

I am going to deviate from the standard order of a paper and give a partial list of uses for modules as described in this paper. This will give the reader an idea of where the work below is heading, and allow a deeper understanding of the material.

Consider writing a piece of software, a page layout program for instance. The software is completed and then a decision is made that the software should be marketed as "groupware". Do you as the developer (or as part of the development team) want to spend several months re-writing the software so that it works over a network, sharing information?

The answer to this is probably, "Of course not!". Using conventional methods, this is exactly what would have to be done. But what if you could take an already existing "groupware" module and sell it on the same disk as the software? Instant solution, requiring no programming beyond what has already been done in the original program. So instead of spending months re-writing code and learning about all the different communication possibilities, the project is still finished and can be shipped immediately.

In reality, the "groupware" module will probably have to be written or customized for the product, but the main application will not have to be changed.

As a second example, consider a program to supply UNIX[1†] style tools. A separate module could be written for each Unix utility to be available. Then using graphical links between module icons, each module could be called in a chain giving the same flexibility and functionality as a Unix command line. If new tools are desired, they can be written and dropped into the application folder.

Now that you have seen two potential application, think about what could be done with extensible spreadsheets, word processors, databases, games, or any other type of application.

The Module Concept

Added functionality should be available through add-on modules. A module is a single file which is loaded by an application and becomes available to the user. What should a module do? This is a tricky question to answer. The obvious response is "anything". The next question immediately following this question becomes, "How?"

Functionality can be broken down into several categories of application. In this paper, the categories being considered are: Word Processing/Desktop Publishing, Graphics, Communications, and Spreadsheets. These categories can be further broken down into specific functional capabilities, with emphasis being placed on the word processing/DTP and graphics areas.

Word Processing/Desktop Publishing

While word processing and desktop publishing are different categories, they merge towards the high end of the word processing spectrum. Both are primarily concerned with putting words and images on a page and formatting them. Some defined functions are: text formatting (justification, indentation, margins, line spacing, etc.), page layout (columns, headers, footers, footnotes), graphics (tables, pictures), and extended (index, table of contents, word counting, spelling, etc.).

[1†] UNIX is a Trademark of Bell Laboratories. It is important that you never, ever forget this.

Graphic Programs

Graphic programs are conceptually simple currently. Any given program has a set of defined tools. These tools then perform a clearly defined function. Functionality can be split into: shaped object manipulation, text manipulation, and attribute setting (line style, patterns, color.)

The problem with this is that these tools can not be extended at all beyond their current function. Ideally, a user should be able to buy a new tool and add it to a drawing program, just as a traditional artist might buy a new brush or pen to perform a specific function. It would also be nice to allow graphic tools to be included in any other program where an illustration or annotation of an idea or data could be included quickly and easily.

Communication Programs & Spreadsheets

Communications programs can be split into several categories: terminal emulation, file transfer, and mail transfer. Functionality defined by these categories include: terminal protocol (VT100, 3278, etc.), connection path (AppleTalk, Ethernet, serial), file transfer protocols (Kermit, Xmodem, Ymodem, FTP, etc.), and mail (addressing, message delivery.)

Spreadsheets are great for organizing data and performing simple calculations on that data, but performing higher mathematical functions not defined by the spreadsheet are tedious to impossible in a macro. By adding modules, it becomes possible to write functions in *compiled* languages, then make them available inside a spreadsheet, thus allowing the ease of data manipulation, with the number crunching power of compiled and optimized code.

The Module File

A module is a file which contains executable 'oCOD' resources plus any other resources needed by the code. Any type of resource found in a normal Macintosh application can be stored in the resource file. These resources will become available to the module during its execution phase. The oCOD module is similar to a standard driver (or DA), in that it is based off of register A4, instead of register A5.

A module file can have only one distinct module in it. Modules always start in oCOD resource 0. If a module has more than one oCOD resource, it must load and link to the other oCOD resources on its own.

A module can act as either a single instance of an object (like a traditional function which performs a single task), or it can create multiple instances (true objects) which will be manipulated by the module code.

In theory it is possible to write a module which will operate on any common data structure without the user being cognizant of the internal data manipulations. In reality, such a piece of code would be too unwieldy to write. Instead modules are currently defined in distinct categories.

Type 1 Modules

The type 1 module is both the least general and the most powerful module type. This type of module posses specific knowledge of an application's data structures, global variables, and internal application assumptions.

Examples of type 1 modules would be spelling checkers, grammar checkers, etc. that could access a word processor's RAM based formats and so on. These must be application or data structure specific, as there is no standard mechanism defined for keeping track of what is in a window or how to manage formatted text (as opposed to simple TextEdit records).

The interface to this module type is not standardized as in the other types. Instead, the application/module interface is designed by the application programmer and must be published to allow third party developers to implement modules for the application using the modules. The modules are also handled by the application, instead of using the methods described in this paper.

Type 2 Modules

The type 2 module is the opposite of the type 1 module. It is a complete black box to the application. These modules are typically general utility functions which may not have any interaction with the application, per se, but allow the user to perform a function such as sending a file to another machine or renaming/moving a file.

Type 2 modules exist independent of specific or proprietary data structures. An examples of a type 2 modules would be file I/O routines to read and write specific disk formats. Data would be presented in one of several standard formats, ASCII text, QuickDraw picture, BitMap or PixMap and the module would handle any particular or specific processing, I/O, etc. requested. A type 2 module can be thought of as an object or self contained set of routines which perform their function without requiring additional communication or interaction with the application.

Type 3 Modules

The type 3 module is used to implement general tools such as those used for drawing or data manipulation. A module of this type can interact not only with the user, but with the application as well. This means that a type 3 module must be able to handle a variety of input types and return a desired output type on demand. Not all permutations are possible of course, and modules will not need to handle even the possible permutations. If one module can not perform a desired function, it returns a message indicating this, and an attempt will be made with a different module. Most modules will fall into the type 3 category.

Type 3 modules can involve aspects common to either type 1 or type 2, but require an additional element of integration into the event handling and processing routines of the application. Examples of type 3 modules include graphics or text tools which needed to represent themselves in the interface via icons, menu items, etc. and may need to perform mouse tracking, updating, activation/deactivation and other event triggered or related behaviors.

The scope of this paper is such that only type 2 and type 3 modules will be discussed in detail. Type 1 modules will be detailed at a later date.

Apple's Objects

Apple defined several object interfaces when the Macintosh was developed. The most commonly used are CDEFs (controls), WDEFs (windows), LDEFs (lists), and cdevs (Control Panel devices).

While most readers are familiar with writing some of these, a brief review of their design is in order.

WDEF and CDEF

Both the window definition and control definition function show some of the necessary parameters for object creation. (See Figure 1 for the function declarations of both types.)

```
FUNCTIONMyWindow  (varCode : INTEGER; theWindow : WindowPtr;
                   message : INTEGER; param : LONGINT) : LONGINT;
FUNCTIONMyControl (varCode : INTEGER; theControl : ControlHandle;
                   message : INTEGER; param : LONGINT) : LONGINT;
```

Figure 1. Window and Control Definition Function Declaration

The parameters for the definitions are defined as:

varCode	–	indicates which variation on the control to use.
theControl	–	the handle to the previously allocated and initialized window or control storage.
message	–	what function to perform for this invocation of the CDEF or WDEF function.
param	–	auxiliary data depending on the value of message.

LDEF and cdev

Both the List Definition (LDEF) and Control Panel Device (cdev) are later introductions to the stable of Macintosh objects. They are, at the same time, both more general and more narrow in focus. The LDEF has only one purpose, to display and manipulate a list or array of data. Yet it can be written such that any type of data can be manipulated (within limits.) The cdev allows the use to control a single aspect of the Macintosh (with the exception of General.)

Figure 2 shows the definitions of these two object types. The parameters for each are described below.

```
PROCEDUREMyList (lMessage : INTEGER; lSelect : BOOLEAN; lRect : Rect;
                 lCell : Cell; lDataOffset, lDataLen : INTEGER;
                 lHandle : ListHandle);
FUNCTIONcdev    (message, Item, numItems, CPanelID : INTEGER;
                 VAR theEvent : EventRecord; cdevValue : LONGINT;
                 CPDialog : DialogPtr) : LONGINT;
```

Figure 2. List and cdev Function Declarations

LDEF

lMessage	–	indicates which operation is to be performed.
lSelect	–	indicates whether the indicated cell should be selected.
lRect	–	the rectangle the cell should be drawn in.
lCell	–	the cell being used on this invocation.
lDataOffset	–	offset into the cell data area where this cell's data is located.
lDataLen	–	the length of the data for this cell.
lHandle	–	the handle to the previously allocated and initialized list storage

cdev

message	–	indicates which event has just occurred.
Item	–	the dialog item number which has just been hit.

numItems	–	number of items preceding the cdevs items in the dialog item list.
CPanelID	–	base resource ID of the Control Panel.
theEvent	–	pointer to the event record of what caused the invocation.
cdevValue	–	the value returned by the cdev after the last invocation.
CPDialog	–	the pointer to the dialog box the cdev is in.

Analysis

As we can see from an examination of the above calling sequences, there are a few common parameters which are needed. Other parameters depend upon the functionality desired by the object.

WDEFs and CDEFs are the most closely related. They share the same concepts for all four of their parameters. Both require a variation code because they are graphical objects which may take on multiple visual forms, they share the concept of using pre-allocated storage for their data areas, and they require a message and additional data to indicate what action to perform.

The LDEF has two parameters in common with the WDEF/CDEF style of object. A parameter specifying the function to perform (lMessage) and the previously allocated storage (lHandle). Along with these parameters, there is information on what to draw (lCell, lDataOffset, lDataLen), and how to draw it (lSelect, lRect).

The cdev shares only the message parameter with any of the other three object types. Where other objects have access to their drawing information (the GrafPort) from their own internal data structures, the cdev is dependent on the CPDialog parameter for it's drawing information. The cdev also implies directly, for the first time, the concept of an object being a finite state machine. By sending the last returned value back to the cdev object, a sequence of states can be implemented.

With the concept of extension modules, we do not need the variation code. It is necessary for flexibility, but our concept is a specific function only. If module configuration is necessary, it can be handled by one of the methods described later.

Since it is desirable to have multiple instances of certain object types, some objects can be initiated and terminated as well as perform specified functions. Therefore a function selector with predefined messages is necessary.

Internal data storage in an object, as used by the cdev, is not a good idea. Unless the data is static configuration information, this will prevent reentrancy in the module. If the module is reentrant, multiple instances of it may be used without needing multiple copies of the executable code. As it will often be desirable to have a reentrant module, external data storage becomes required, as is done in a WDEF or CDEF. The module must acquire this storage and fill it with its own data structures, thereby allowing the object to truly act as a black box.

If the module is to perform as a finite state machine, it needs to be given its last state on each invocation. If multiple instances of the object are needed, then the state information must be external as with a cdev. Instead of making the application keep track of state information (which may or may not be present), it will be stored in the module instance data.

The cdev and LDEF both take external data which affects their operations: theEvent for the cdev and lCell, lDataOffset and lDataLen for the LDEF. New information is placed into the WDEF and CDEF data areas by the Mac OS, and then the object is invoked to operate on that data. So another necessary parameter is some type of external data. Along with the external data, we need a flag to indicate what type of data is being

supplied.

If the module or object is to perform any drawing, the GrafPort where that drawing is to be done must be available to it. Since most modules/objects will perform some graphic action (see **Common Functions**, below), a parameter should be used to indicate where to draw.

The Module/Object Concept

Now we have an idea of the essential parts of an object on the Macintosh: a function selector, external instance data storage, drawing information, and additional data used by the object. The object must also be able to return an error code, object handle, or function result as well as an indication of what kind of data is being sent and/or returned.

A specific subclass of module is the graphic module. A graphic module is defined as any module which creates objects with a visible marking in a window or port. These modules must show an icon. These icons are placed into a palette window or region of a window. If a module icon is selected and a mouseDown event occurs in an active window, a new object is created by tracking the mouse. The module may deselect itself or stay selected. Call InModuleIcon to determine if the mouseDown event is in another module icon.

Declaration

The basic declaration of an external module is always the same, no matter what the module will do. It can be declared in Pascal as:

```
FUNCTION    MyObject(   objHand : Handle; oPort : GrafPtr;
                        oMessage : INTEGER; oData : LONGINT;
                        inputDataKind : ModDataKind;
                        VAR reqDataKind : ModDataKind) : LONGINT;
```

objHand	–	the handle to the object's instance.
oPort	–	the pointer to the GrafPort to draw into. This can also be a WindowPtr or DialogPtr.
oMessage	–	the function to perform.
oData	–	any additional data required for the selected function.
inputDataKind	–	the structure for the input data.
reqDataKind	–	pointer to the structure specifying the return data type.

The Object Instance

An instance of an object (created by the module) is defined by identification and state information in an object header, followed by data specific to the instance. The header information is in a fixed format for each object. The Pascal declaration for the object header and associated types looks like:

```
TYPE
    ModDataKind = RECORD
                      highFlags      : LONGINT;
                      lowFlags       : LONGINT;
                  END;
```

```
ModFlagKind = RECORD
                    eventMask      : INTEGER;  { EventMask for object }
                    flags          : INTEGER;
                END;

ObjectHeader = RECORD
                    objectType     : INTEGER;  { Type identifier }
                    objectClass    : INTEGER;  { Class identifier }
                    objectCreator  : OSType;
                    inputFlags     : ModDataKind;
                    outputFlags    : ModDataKind;
                    objectFlags    : ModFlagKind;
                    objectName     : STRING[32];
                    moduleHandle   : Handle;
                    objectPort     : GrafPtr;  { Port drawing occurs in }
                    objectRect     : Rect;     { Rect holding icon }
                    objectRgn      : RgnHandle;{ Region used by object }
                    { ... }                    { Internal Object Data }
                END;
```

Each instance of an object contains a 4 byte type identifier. This is an integer value indicating what type this object is. Valid values are 1, 2, and 3. Following the type identifier is a 4 byte Creator field. This is defined as an OSType and is used like the Creator of a file.

Following the Creator field are two Flag_Structures each consisting of two 4 byte variables. The first structure contains the flags declaring which variable types are acceptable to the object. The second structure indicates what types of data may be returned by the object. ObjectFlags is a set of flags which the module or object can set to indicate some of its requirements and abilities such as needing background time, or handling events. Following the input and output flags is the object name. The moduleHandle is the handle to the executable code for the object's module.

The objectPort is a pointer to the GrafPort where the object is displayed. The objectRect is the bounding rectangle for the object and the objectRgn is the region the object occupies.

Details of ModDataKind and ModFlagKind are discussed in Appendix A.

Defined Functions

All objects or modules should be able to perform or understand a standard set of functions. Basic functions include: loading, object initialization, object termination, unloading, icon drawing, and object mapping. Optional functions include: menu handling, event handling, and background processing. This is not an exhaustive list of defined functions, but a beginning of a list. (More details of functionality are discussed in Appendix B.)

Common Functions

Each module must meet minimum functional requirements. Each of the above defined messages must be implemented within the following guidelines.

Load Routine

The load routine is called after the module has been loaded into memory from the disk. It will then be called with the oLoad message. If the module can not run on the specific machine or for any other reason, return

oGenError, otherwise allocate and initialize any static storage during this phase.

This is also the time to install any menus or menu items that the module will need. (See Module Menus, below.)

Init Routine

This is the object creation message. Initialize the external storage for an instance of the object. Allocate the storage by calling NewHandle if external storage is used. Initialize the allocated memory, then return the handle as the function result. On subsequent calls, this handle will be returned in objHand. DO NOT SAVE A COPY OF THIS HANDLE! A saved copy will destroy the object's reentrancy.

Draw Icon Routine

Each module should have an icon it will display on request. Graphical modules are required to have an icon. There should be a black and white icon and an optional color icon if desired. Draw the icon in the oPort within the rectangle pointed to by oData.

Terminate Routine

This message tells the module to terminate the instance pointed to by objHand. Validate the memory being passed as being an object of the module, and if the module does own it, release the acquired memory and return. If the data does not belong to an object of the module, return oGenError.

Unload Routine

This routine is called when the module is going to be unloaded from memory. This can happen in low memory situations or if a program is finished with the module. The module may be reloaded at a later time. You must deallocate all internal storage and return noErr (unless an error occurs).

Event Routine

This message is passed to the object when an event has occurred in the program. This allows the object module to monitor for a click on an object or the module icon or do any special event filtering. Null events are not passed to objects. If the object/module handles the event, noErr should be returned to prevent the event from being passed to other objects or the application. If a double-click occurs on an object or module icon the object/module can interact with the user. A dialog box is usually used to set or change internal parameters.

Menu Routine

This message is passed to the module when an unknown menu or menu item has been selected. Objects can create menus and interact with the user through them (see Module Menus, below). The Menu ID and item are passed in oData in the same format used by MenuSelect. Make sure that the menu and/or menu item are yours. Multiple modules may install menu items. If the menu item is yours, return oMenuDone, otherwise return oNotMyMenu.

If the object is currently selected (isSelected flag), and an oCut, oCopy, oPaste or oClear message is passed to the object, the effect must be appropriate for that menu selection.

MapObject/ReadObject Routine

The oMapObject message tells the module to take the object pointed to by objHand and map it into a flat storage space allocated by NewPtr(). This pointer is returned as the function result. The oReadObject message tells the module to take the pointer stored in oData and return a handle to the object represented by that handle. It is the inverse of the oMapObject message. If the pointer is a reference to incorrect data, return oGenError.

Background Routine

The oBackground message is used to allow an object or module to execute periodic tasks as requested by the needTime flag.

The Life of a Module

While it may seem like modules and objects are almost identical there are major and subtle differences between the two. Modules and objects can also share some common assumptions. There are things a module must do that are not applicable to objects and vice versa, however.

Both modules and objects can count on the fact that their resource file is available to them as the top resource file. Therefore, any of the single resource file management functions (e.g. Get1Resource()) will work for their own resources.

Even though modules and objects are treated as separate types, in reality one piece of code handles functions for both. In object-oriented terms, the modules which create objects function as a class and the object is an instance of that class. Modules without objects can be thought of as a class with a maximum of one instance.

A module can install a new menu or append a menu item to an already existing menu. If at all possible, a module should install its menu items in the Modules menu which is supplied by the Module Manager. Modules which are defined as graphical must display an icon and react to clicks on that icon.

An object for the most part is passive. It can be selected, modified or dragged around in a window. Objects do not create windows as such, but they may create modal dialogs for user interaction. Objects must respond to mouse clicks and drag actions. They must be able to moved around in a port by the user. Graphical objects must also be able to respond to changes to such parameters as font, size, style, line style and thickness, fill patterns, line patterns, and color changes. Not only must they be able to messages to change those attributes, they must be able to report those attributes so the Module Manager can set menu items appropriately in the drawing portion of the Modules menu.

The Module Manager

The application interface to modules and objects is through the Module Manager. The Module Manager is responsible for loading, organizing and managing modules and objects. The application does not handle modules or objects directly. This simplifies application coding considerably. Also, not all defined messages are passed to objects. Some messages, such as oGroup/oUngroup are handled directly by the Module Manager. Instead, the application communicates with the Module Manager through the following routines.

```
PROCEDURE   LoadModule (      searchProc:ProcPtr; filterProc:ProcPtr;
                              oPort : GrafPtr);
```

This procedure is called during program initialization. LoadModule looks in the current directory, then in the System Folder for any files of type 'oCOD'. oPort is the GrafPort the module icons are to be drawn in. If filterProc isn't NIL, LoadModule executes the function it points to for each possible module file, to determine whether that file should be loaded. The filterProc has one parameter and returns a Boolean value. For example:

```
FUNCTION myFilterProc ( resFileNum : INTEGER) : BOOLEAN;
```

LoadModule passes this function the resource file number of the already open module file. The module file type is defined by the resource 'TYPE' N, where N is the module's type as defined above. The function returns TRUE for each file which is to not be shown (i.e. filtered out.)

The searchProc function is called after the standard directories are searched. When it is called, the current directory will be the original application directory. The searchProc function is defined as:

```
FUNCTION mySearchProc ( invocCount : INTEGER) : BOOLEAN;
```

The searchProc procedure can then set the current volume to other directories to search. This allows an application to look in particular subfolders for modules. invocCount is the current iteration through this routine. On the first invocation, invocCount = 1. Return TRUE if a new directory has been set and FALSE when all directories have been exhausted.

```
PROCEDURE ModuleTask();
```

This procedure is called at WaitNextEvent() for applications not using object scheduling or at event scheduling time for object oriented languages or at WaitNextEvent() under MultiFinder applications.. It allows modules or objects time to perform background tasks.

```
FUNCTION IsModuleEvent( theEvent: EventRecord, VAR objHand : Handle;
                        VAR objReturned : BOOLEAN;
                        VAR newObject : BOOLEAN) : BOOLEAN;
```

This procedure is called when an event is returned from GetNextEvent() or WaitNextEvent(). The Module Manager passes the event to objects and modules which can handle events. If the event is handled internally, then IsModuleEvent returns TRUE, and the application should disregard the event. If FALSE is returned, the event must be processed normally. If the event is handled by an object (not a module), the object handle is returned in objHand and objReturned is set to TRUE. If the event caused a new object to be created, the new object is returned in objHand and newObject is set to TRUE.

```
FUNCTION IsModuleMenu(  menuSelection : LONGINT) : BOOLEAN;
```

IsModuleMenu is called after MenuSelect() (or MenuKey()) returns a menu selection. If the menu item belongs to a currently active module or object, then IsModuleMenu returns TRUE. If FALSE is returned, the menu selection must be handled as a normal menu selection.

```
FUNCTION ModuleMessage( objHand : objectHandle; oPort : GrafPtr;
                        oMessage : LONGINT; oData : LONGINT;
                        inputDataKind : ModDataKind;
                        VAR reqDataKind : ModDataKind) : LONGINT
```

The ModuleMessage function allows a specific message to be sent to a specific object. objHand is the handle to the object instance or module. If objHand is set to NIL, this forces a brute force search of all modules, looking for any module to perform the requested data transformation. oPort is the GrafPtr, WindowPtr, or DialogPtr where any drawing is to occur. oMessage is the function selector as described in Appendix A. inputDataKind and reqDataKind are flag structures indicating the input and requested output formats. They may be set to NIL if no data is being used. The function returns a LONGINT which may be a pointer, handle, or result code, depending on the requested function and reqDataKind.

```
PROCEDURE UnloadModules ();
```

Call UnloadModules before exiting the application. This allows all modules a change to clean up after themselves nicely. This is very important, since some modules may have installed background tasks which need to finish.

Module Manager Utilities

```
PROCEDURE GetObjectRect (     objHand : Handle; VAR objectRect : Rect);
```

This procedure returns the bounding rectangle of the object. If the object has no visible representation, objectRect is set to (0, 0, 0, 0).

```
PROCEDURE GetObjectRgn  objHand : Handle; VAR objectRgn : RgnHandle);
```

GetObjectRgn returns the region representing the area possessed by the object. If the object has no visible representation, objectRgn is set to NIL. Do not dispose of this region or modify it in any way.

```
FUNCTION InModuleIcon ( theEvent : EventRecord) : BOOLEAN
```

InModuleIcon is used internally for a module to determine if a mouseDown event is occurring in the icon of another module. It returns TRUE if so and FALSE otherwise.

```
PROCEDURE ObjectModifyRegion (  objHand : Handle; theRgn :  RgnHandle;
                                oMessage : LONGINT);
```

This routine is called when a two objects may overlap that shouldn't. oMessage indicates whether the region should be deleted from an object's region space. This can be used to implement tagged fields in something like a text object. (For instance, a sound bite might be included in a text block, but text should not be overlaid on the tag area.) oMessage can be oSubRgn when an area is to be removed for a tag, or oAddRgn when the tagged area is being removed.

The Application Interface

By using the Module Manager, an application can be insulated from the details involved in using modules. At some point, however, an application must know that some objects are involved with the application windows or documents. To allow an application to work with objects, object handles are returned from IsModuleEvent() if an object does handle the event. If IsModuleEvent returns TRUE, check the newObject and objReturned parameters to decide what to do next.

When a new object handle is returned, it should be stored with some relationship to the current graphic port. (This is the only place where these objects can be placed.) When objReturned is TRUE, check the object region to see if it is visible in the current port. If it is not, scroll the port until the region is visible. If the region is larger than the visible section of the port, it application must decide what to do. In general, scrolling should be attempted, since if the object is a text handling object, the bottom of the region is where any active text will appear. The routines GetObjectRect() and GetObjectRgn() can be used to access the object bounding rectangle or the object region.

When it is time to save or close the window, call ModuleMessage(), passing the object handle with the oMapObject message for saving, or the oTerminate message for simple window closing. This allows all memory for objects to be deallocated.

Conclusion

This paper has presented a method to implement extensible applications in any compiled language using object-oriented methodologies, but not requiring object-oriented programming. The interface between the module and the Module Manager has been defined as well as the interface between the application and the Module Manager.

Details of the module header and flags are discussed in Appendix A. Information on the currently defined messages and functions they implement is noted in Appendix B.

There is still work to be done. Additional functions need to be defined as well as extending the input/output types of modules and objects. I would suggest that a group be formed to finalize the interface specifications for both modules and the Module Manager.

Not only is it necessary to finalize the definitions started in this paper, but the work describe here needs to be implemented in a number of applications, and module tools must be written and made available to users.

Appendix A
Defined Data Types

The data types defined for use with modules are defined in this appendix. Not all values are currently defined for some data types such as ModDataKind.

```
TYPE
    ModDataKind = RECORD
                      highFlags      : LONGINT;  { Reserved for expansion }
                      lowFlags       : LONGINT;  { Currently used flags }
                  END;

    ModFlagKind = RECORD
                      eventMask      : INTEGER; { EventMask for
                                                  module/object }
                      flags          : INTEGER;
                  END;


    ObjectHandle = ^ObjectPtr;
    ObjectPtr    = ^ObjectHeader;
    ObjectHeader = RECORD
                      objectType     : INTEGER;   { Type identifier }
                      objectClass    : INTEGER;   { Class identifier }
                      objectCreator  : OSType;
                      inputFlags     : ModDataKind;
                      outputFlags    : ModDataKind;
                      objectFlags    : ModFlagKind;
                      objectName     : STRING[32];
                      moduleHandle   : Handle;   { May be dropped }
                      objectPort     : GrafPtr;  { Port drawing occurs in }
                      objectRect     : Rect;     { Rect holding icon }
                      objectRgn      : RgnHandle;{ Region used by object }
                      { ... } { Internal Object Data }
                  END;
```

Defined Errors

The following are defined as error codes which may be returned by the object.

```
CONST
   oGenError          = -1; { General Error Message  Object unable to
                              complete }
   oNotMyObject       = -2; { Module doesn't know about this object.}
   oNotMyMenu         = -3; { The menu selection did not belong to
                              this object/module }
```

Defined Flags

The following flags are defined for the ModFlagKind.

Module Flags

The following values are defined for the flags in the module/object header.

```
singleInstance      = 1;   { Module has single instance only              }
needTime            = 2;   { Module or object runs periodic functions     }
hasMenu             = 4;   { Module or object has menu or menu items      }
hasGraphicObject    = 8;   { Module can create graphic objects            }
hasWindow           = 16;  { Module has a non-modal window                }
isSelected          = 32;  { Module or object in window has been selected }
```

Defined Data Flags for ModDataKind

The following values are defined for the lowFlags in the ModDataKind type.

```
fPtr              = 1;     { Generic Pointer }
fHandle           = 2;     { Generic Handle }
fInteger          = 4;     { Pascal integer (2 bytes) }
fLongint          = 8;     { Pascal long integer (4 bytes) }
fStr              = 16;    { Pointer to a Pascal String }
fGrafPtr          = 32;    { Pointer to a GrafPort }
fWindowPtr        = 64;    { WindowPtr }
fDialogPtr        = 128;   { DialogPtr }
fTEHandle         = 256;   { TextEdit Handle }
fPicHandle        = 512;   { PicHandle }
fRgnHandle        = 1024;  { Region Handle }
fSFReply          = 2048;  { Pointer to an SFReply structure }
fResFile          = 4096;  { Resource File Number }
fHParmBlkPtr      = 8192;  { Pointer to an HParamBlockRec }
```

This is not an exhaustive list of values. More values will be defined at a later time.

Appendix B
Defined Messages

The following are currently defined messages for modules/objects.

```
CONST
   oLoad              =  0;   { Initialize any internal storage           }
   oInit              =  1;   { Allocate external storage space for object }
   oDrawIcon          =  2;   { Draw Icon at rect stored in oData         }
   oTerminate         =  3;   { Deallocate and free external object storage}
   oUnload            =  4;   { Release any internal storage for unloading }
   oEvent             =  5;   { An event occurred.  Attempt to handle it  }
   oMenu              =  6;   { The object's menu was selected            }
   oMapObject         =  7;   { Create a flat representation of the object
                                for storage in a file                     }
   oReadObject        =  8;   { Create object from flat map               }
   oRun               =  9;   { Perform any background task as per object
                                request                                   }
   oAddRgn            = 10;   { Add region in oData to object region      }
   oSubRgn            = 11;   { Subtract region in oData from object region}
```

Message	Function	Module/Object
oLoad	Initialize Module	Module
oInit	Initialize Object	Object
oDrawIcon	Draw Icon in Port	Module
oTerminate	Destroy Object	Object
oUnload	Deallocate Module Storage	Module
oEvent	Handle Event Record	Module, Object
oMenu	Handle Menu Selection	Module, Object
oMapObject	Change Object to Flat Storage	Object
oReadObject	Change Flat Storage to Object	Module
oRun	Execute Periodic Action	Module, Object