Date: January 21, 1985

From: Larry Tesler LT

To: Larry Rosenstein, Ken Doyle, Scott Wallace, Gaby Hirl, Dan Ingalls, Barry Haynes, Yu Ying Chow, Mark Lentczner, Chris Espinosa, Andy Averill, Scott Knaster, Russ Daniels, Pete Cressman, Judy York, Amy Rapport, Eileen Crombie, Bud Tribble, Jerome Coonen, Rony Sebok, Steve Capps, Martin Haeberli, Rick Meyers, Al Hoffman, Gursharan Sidhu, Dan Cochran, Hasmig Seropian, Cary Clark, Bob Belleville

Subject: Expandable Application: Release Plan

This is a release plan for the Expandable Application (aka Generic Application, Standard Application, Macintosh ToolKit, Vantage).

All releases will be made through the Software Library. Non-Apple users will contact Technical Support for help. In turn, Technical Support will forward questions they can't answer to my group.

Release 0.0 ("Phi") is planned for January 23, 1985. Its authors were Larry Rosenstein, Scott Wallace, and Ken Doyle. One of two sample programs is by Gaby Hirl. Release 0.0 is intended for use by Rony Sebok (who is writing an electronic mail user interface), Russ Daniels (to prepare for eventual technical support of the product), and Kurt Schmucker and Andy Averill (to write documentation). It is missing many small but critical features; memory management is not yet smart; the Pascal syntax is not the latest Wirth-approved variety; it is virtually undocumented; it has not been used outside my group.

Release 0.1 ("Chi") is planned for February 1, 1985. The audience will be expanded to include other Apple employees who program Macintosh applications in Pascal, and a number of outside testers to be selected by Scott Knaster and myself. The outside testers will come from the ranks of university customers and certified developers with experience in both Macintosh ToolBox use and object-oriented programming (e.g., Lisa Clascal). It will be like Release 0.0 except it will have the Wirth-approved new syntax, scrap management, and various minor improvements.

We plan to teach an in-house course in February based on release 0.1.

Release 0.3 ("Psi") is planned for March 15, 1985. There will a little more documentation. The audience will be enlarged to include some outside testers with no specific prior experience. The capabilities will be expanded to include smarter memory management and to tie other loose ends.

Release 0.5 ("Omega") is planned for April 24, 1985. It will have the final architecture and fairly complete documentation. The audience will be expanded to include as many developers as we can support. Release 0.5 will never be a "Product", because the Lisa Workshop will be needed to compile, but it should make Macintosh application development easier for Lisa owners.

Release 0.7 ("Alpha") is planned for around June 1, actually, one month after the later of: Release 0.5; Alpha of the New Macintosh Development System. This version will be compatible with Objective C, support applications in assembler, and support Finder-coexistent applications. A Lisa will no longer be required.

Release 0.9 ("Beta") is planned for around July 1, at the later of: one month after Release 0.7; one week after Beta of the New Macintosh Development System.

Release 1.0 ("Official") is planned for late summer, concurrent with the Official Release of the New Macintosh Development System.

An Introduction to

MacApp™ 0.1

Larry Tesler
February 14, 1985

Motivation

There are certain features found in almost every Macintosh application. Typically, the user can scroll and resize windows, perform document operations like open, save, and print, and issue editing commands like cut, paste, and undo. In addition, a well-written program will detect and report errors and recover from them.

To implement the required features reliably and efficiently--and according to high standards of ease of use and consistency--is no mean feat. The law of conservation of complexity states that if life gets easier for the user, it must get more complicated for the programmer. No wonder many people have found that developing Macintosh software is more difficult than they had anticipated.

One way to ease the task of Macintosh programmers is to let them reuse code written by others. The Macintosh ToolBox, Operating System, and QuickDraw provide a body of shared code that reduces development time, but these packages do not go as far as they might. They provide standard routines, but no standard architecture. Every developer must work out a viable structure for each application--a surprisingly difficult task.

The omission of a standard architecture from the ToolBox was quite intentional. Apple wanted the system to be as open as possible, consistent with future compatibility and the preservation of trade secrets. People always think up ways to use a computer that we never anticipated.

But for every developer who has a non-standard requirement, there are many more who have requirements in common. For those people Apple has designed **MacApp™** .

MacApp implements a standard application architecture that we believe is well-suited to many products. If you can fit your design into the MacApp paradigm, you will save a lot of implementation time, and your customers will enjoy the benefits of greater consistency between applications.

Note: MacApp is not suitable for the implementation of desk accessories.

MacApp 0.1

MacApp 0.1 is a preliminary version of MacApp. It is intended for interim use by a handful of testers who can operate with little documentation, and for study by the people who are writing the documentation.

The Objects of a Macintosh Application

Look around the screen of a Macintosh application and what do you see? Usually, you will find a pointer, a menu bar, and one or more windows. Bordering each window is a title bar and scroll bars. In the body of each window you see a portion of a document. In one window (the active window) there is probably an indication of something that the user has selected. That indication may be a blinking insertion point, reverse video highlighting, or tiny square "handles."

If you could analyze a running program, you would find a data structure corresponding to each object mentioned above, plus many others. Fortunately, you do not have to deal with a multitude of objects to write a MacApp program. Many common objects (for example, menus) are handled for you by the ToolBox and MacApp.

The objects you do have to deal with fall into two categories: standard and particular. Standard objects are those that appear in some form in almost every application, but that vary in operation from program to program. Windows and documents are good examples. Particular objects are those that some applications need, but that many do not, e.g., spreadsheets, graphs, and slot machines.

Since standard objects recur in every application, generic definitions for them are included in MacApp. Each application customizes the generic definitions to suit its special needs. A bar-chart application might define the special behavior of x-y series windows and bar-chart windows, but it does not have to include code for generic operations like scrolling and resizing; these are already defined in MacApp.

To make it easier to customize MacApp standard types in your application, both your program and MacApp are written in an object-oriented language. At present, the language used to write MacApp programs is Object Pascal, an extension of Pascal designed with the help of Prof. Niklaus Wirth. Later this year, it will be possible to use MacApp with object-oriented versions of other languages, including Objective C from Productivity Products, Inc. You will also be able to use an assembler.

Object Pascal

A specification of Object Pascal can be found in *Object Pascal Report*, attached. The language adds to Pascal a new structured type called *object*. Members of object types are dynamically allocated on a heap, that is, they are created and destroyed during execution. Each member of an object type has its own state, together with a set of operations that can examine and change that state. The components of the state are called fields and the operations are called methods.

An example of something declared as an object type is a window. One of the fields of a window is a Boolean variable called isResizable. One of its methods is called Close.

Although you must declare windows and other standard objects as object types, you need not implement spreadsheets and other particular objects that way. However, particular objects should, in general, be allocated on the heap or on disk. The use of global variables makes it difficult to support multiple windows and documents.

Usage of the Term "Object"

In the rest of this memo, the term "object" will be used as a technical term to denote a member of an object type.

Standard Object Types

MacApp declares the following standard object types:

TApplication	TDocument	TWindow
TFrame	TView	TCommand

Every object type declared by MacApp begins with the letter "T" to avoid name conflicts with globals declared in the Macintosh ToolBox. It is not necessary for the object types you declare to begin with a "T".

Each standard object represents an important constituent of the Macintosh standard user interface. Below, we state for each type of standard object:

- **Purpose** Its purpose.
- **Fields** What some of its fields contain.
- **Methods** The names of some of its methods. Methods you are likely to customize are italicized.
- **Events** Some of the user input events the object normally handles.
- **Quantity** The number of members of the type in existence at any time during execution (except during initialization and termination).

TApplication

Purpose The application receives each event in turn from the system event queue, and either handles it itself or dispatches it to another object.

Fields Its four-character "signature" (e.g., 'MACA' for MacWrite).
A list of open documents.
A list of open windows that are not part of any document.

Methods MainEventLoop, ObeyEvent, *LaunchDocument*, *OpenToolIcon*, Terminate.

Events About..., Desk accessories, New, Open..., Quit.

Quantity One per running program.

TDocument

Purpose The document object contains the underlying data structures of a document. It also manages the user's disk files.

Fields Its four-character "file type" (e.g., 'WORD' for MacWrite).
A list of windows that belong to the document.
The file name that the user specified in "Save As...".
A tally of changes the user has made since the last Save.
The last command the user has issued to this document.

Methods *ReadFrom, WriteTo, BytesConsumed*, Save, Close.

Events Save, Save As..., Revert, Close.

Quantity One per New or Open with no corresponding Close.

TWindow

Purpose A window maintains context for the user. Windows can overlap on the screen. The user can usually move them around, change their size, and close them. The one in which the user is currently working is called the active window. Most events are channeled to the active window.

Fields The window manager's WindowPtr.
Can it have a close box? Have a resize icon? Activate?
Is it active now?
The document of which it is part (or nil if none).

Methods DrawResizeIcon, Open, Close, Activate, UpdateEvent.

Events Activate, Deactivate, Drag Title Bar, Click in Close Box.

Quantity One for each window that is open or temporarily hidden.

TFrame

Purpose A window can be subdivided into frames. A frame can contain other frames as well. Frames need not tile their container. They normally do not overlap--if they do, you must write extra code to handle the effects of overlap.

If a window has scroll bars, it has (at least) one frame whose boundary is smaller than its own. An undivided window without scroll bars can serve as its own frame. In the rest of this memo, the term "frame" designates either a window or a subdivision of a window.

Fields The window that contains it.
A list of "controls" (e.g., scroll bars) within its boundaries.
The standard amount to scroll by in each direction.

Methods ScrollBy, ScrollTo, HaveView, Focus, TrackInContent.

Events Scroll, Resize.

Quantity One for every panel or other subdivision of a window.

TView

Purpose A view is what you see inside a frame. At any moment, a view has definite boundaries. The frame can scroll over the view within those boundaries. The coordinate system of the view is relative to its own upper left corner. The units of the coordinate system need not be the same as those of the screen or page on which the view is displayed. The units will, in fact, differ while zooming and while printing in high resolution.

Sometimes, two frames or windows display different views of the same data at the same time (e.g., Series and Graph windows in MicroSoft Chart). Sometimes, a single frame or window displays different views of the same data at different times (e.g., By Icon and By Date in the Finder).

Fields The resolution in spots per inch (usually screen resolution).
The current boundaries (a rectangle <32K spots per side).
The frame (or window) in which it is shown.

Methods *DoMouseCommand, DoKeyCommand, DoMenuCommand, DoSetupMenus, Draw, Activate, HighlightSelection.*

Events Display contents, Select/Deselect, Edit.

Quantity At most one at a time per frame or undivided window.

TPrintableView

Purpose A View that can be paginated and printed is a printable view. Use a printable view when you want the user to be able to obtain hard copy by choosing the Print command.

Fields The page number for the topleftmost corner of the view.
Is page 2 below page 1 or to the right of page 1?
Are page breaks being displayed on the screen?

Methods DrawBreaks, DrawPageInterior, Print, PrintPage.

Events Page Setup..., Print..., Print One, Show/Hide Page Breaks.

Quantity At most one at a time per frame or undivided window.

TCommand

Purpose A command represents a user action, or a series of actions that would be treated as a unit by an Undo. In general, you create a command object when the user starts to type (a Key Command), when he presses the mouse button in the interior of a frame (a Mouse Command), or when he chooses a command from a menu (a Menu Command).

The command object is responsible not only for doing the specified action, but also for undoing and redoing it if the user so chooses. In the case of a Mouse Command, the command object is also responsible for giving feedback to the user while he is dragging the mouse.

Fields A command number (const cCut, cTyping, cStretch, etc.).
Is Undo supported?
Does it change the document enough that Close must Save?

Methods *TrackMouse, DoIt, UndoIt, RedoIt, Commit.*

Events Mouse motion, completion of any command, Undo.

Quantity At most one (the last either done or undone).

In summary, the responsibilities of each type of standard object include:

Type	Some of its responsibilities
TApplication	Quit, New, Open..., About..., Desk accessories
TDocument	Save, Save As..., Revert, Close
TView	Select, Edit [and Print, Page Setup, etc. for a TPrintableView]
TFrame	Scroll, Resize
TWindow	Close, (De)activate
TCommand	Do, Undo, Redo, Track mouse

The number of objects of each standard type that may exist at any moment is:

Type	Number of objects in existence
TApplication	Always one
TDocument	One for every New or Open without a Close
TWindow	One for every window on screen or hidden
TFrame	One for every subdivision of a window
TView	One per frame, plus hidden views
TCommand	One if the last command can be undone

The Ownership Hierarchy

At execution time, the standard objects in existence at any moment belong to a hierarchy called the ownership hierarchy. At the top of the hierarchy is the application object; it may own documents and/or windows, as well as a command. A document can own windows as well. A window or frame may own frames or views.

The ownership hierarchy of a running program is shown in the diagram on the following page. In the diagram, each object owns the objects below it that are connected to it by dark lines. Most windows belong to a document, but one does not. Some windows are not subdivided, others have frames that can or can not scroll. Two frames show different views of the same desert island.

When you declare and initialize a document, window, frame, or view, you can specify what attributes it has as well as what relationships it has with other standard objects.

THIS PAGE INTENTIONALLY BLANK: REPLACE IT BY THE FIGURE

An Example

We will develop a simple application to show how MacApp is used. The program will be a variation on the Puzzle desk accessory.

Imagine that you were not using a Macintosh, but rather a dumb terminal and a text-oriented operating system. You could implement a puzzle program in Pascal nevertheless. It would display the puzzle by printing a square array of numbers on the terminal:

```
 2  5  14  8
13  0   4 11
10  7   6 15
 3 12   9  1
```

Then it would prompt the user to specify which piece to move:

```
Move which piece? 13
```

The program would look through the array:

```
fPieceArray: array [0..3, 0..3] of integer;
```

for the number 13 and find that it is at [aRow,aCol]=[1,0]. It would note that the blank space at [fBlankCol,fBlankRow]=[1,1] is adjacent, and therefore allow a move:

```
fPieceArray[fBlankCol,fBlankRow] := fPieceArray[aRow,aCol];
fPieceArray[aRow,aCol] := 0;
```

It would display the puzzle in its new state and be ready for another move.

We could add a few features to this simple program. If the user typed a special command code, say, -1, the program would scramble the puzzle. If he typed -2 or -3, he would be prompted for a file name to save or load the puzzle. The command -4 would print the puzzle on the attached teleprinter, -5 would undo the last move, and 0 would quit the program and return to the shell. There could even be a command, -6, that switched to an alternate "list" view of the puzzle:

```
r0 c0  2
r0 c1  5
...
r3 c2  9
r3 c3  1
```

To make the program less machine-dependent, it would have three interacting modules: the data model, the views, and the command interface. The data model would include the puzzle array, i.e., the variables fPieceArray, fBlankCol, and fBlankRow. The views would display the puzzle on the terminal or teleprinter using WriteLn. The command interface would use ReadLn to prompt the user for a number, classify the command, and call the appropriate routines to obey the input event.

The following is a possible interface to the terminal-oriented puzzle program:

```
type    Coords = record row, col: integer end;
        TextFile = File of Char;

{The Puzzle -- A Data Model}
var     fPieceArray: array [0..3, 0..3] of integer;
        fBlankCol, fBlankRow: integer;
procedure IPuzzle; {Initialize the puzzle}
procedure Scramble; {Scramble the pieces}
function MovePiece(oldLoc: Coords; var newLoc: Coords): boolean;
procedure WriteOn(refNum: integer); {Save on an open file}
procedure ReadFrom(refNum: integer); {Load from an open file}

{The Square View}
var     fPuzzle:   TPuzzle;
        fPieceSize: Point; {a record with the piece width/height}
procedure ISquareView(size: Point); {Initialize the view}
procedure DrawSquare(outDevice: File of CHAR); {Display/Print}

{The List View}
var     fPuzzle:   TPuzzle;
        fSpacing: Integer; {space to leave between columns}
procedure IListView(size: Point); {Initialize the view}
procedure DrawList(outDevice: TextFile); {Display/Print}

{The Move Command}
var     fPuzzle:   TPuzzle;
        fOldLoc, fNewLoc: Coords;
procedure IMoveCommand(aCmd: CmdNumber; aPuzzle: TPuzzle;
   oldLoc: Coords);
procedure DoMoveIt;
procedure UndoMoveIt;
procedure RedoMoveIt; {Two undos in a row redo}

{The Save Command}
var     fPuzzle:   TPuzzle;
        fRefnum: TextFile;
procedure ISaveCommand(aCmd: CmdNumber; aPuzzle: TPuzzle);
procedure DoSaveIt; {Undo not implemented}

{and so forth}
```

To adapt the program to the Macintosh, the puzzle array would be declared as a kind of TDocument, the two views as kinds of TView, and the various commands as kinds of TCommands, all as objects in Object Pascal:

```
TPuzzle = object (TDocument)
        fPieceArray: array [0..3, 0..3] of integer;
        fBlankCol, fBlankRow: integer;
        procedure IPuzzle; {Initialize the puzzle}
        procedure Scramble; {Scramble the pieces}
        function MovePiece(oldLoc: Coords; var newLoc: Coords): boolean;
        procedure WriteOn(refNum: integer); override; {Save}
        procedure ReadFrom(refNum: integer); override; {Load}
        end;

TSquareView = object (TView)
        fPuzzle:    TPuzzle;
        fPieceSize: Point; {a record with the piece width/height}
        procedure ISquareView(aPuzzle: TPuzzle; size: Point); {Initialize the view}
        procedure Draw(area: Rect); override; {Display/Print}
        end;

TListView = object (TView)
        fPuzzle:    TPuzzle;
        fSpacing: Integer; {space to leave between columns}
        procedure IListView(aPuzzle: TPuzzle; size: Point); {Initialize the view}
        procedure Draw(area: Rect); override; {Display/Print}
        end;

TMoveCommand = object (TCommand)
        fPuzzle:    TPuzzle;
        fOldLoc, fNewLoc: Coords;
        procedure IMoveCommand (aCmd: CmdNumber; aPuzzle: TPuzzle;
           oldLoc: Coords);
        procedure DoIt; override;
        procedure UndoIt; override;
        procedure RedoIt; override; {Two undos in a row redo}
        end;

TSaveCommand = object (TCommand)
        fPuzzle:    TPuzzle;
        fRefnum:    TextFile;
        procedure ISaveCommand (aCmd: CmdNumber; aPuzzle: TPuzzle;
           aCmd: CmdNumber);
        procedure DoIt; override; {Undo not implemented}
        end;
```

The override qualifier indicates that MacApp will call that method automatically at the appropriate time, and that a default (usually a no-op) is implemented in MacApp.

The only additional requirement would be to add the following methods (parameters not all shown) to both view types:

```
function DoMenuCommand(aCmd: CmdNumber): TCommand; override;
function DoMousePress(aPoint: Point; ...): TCommand; override;
```

and to declare an application object type:

```
PuzApplication = object (TApplication)
    function NewDocument(...): TDocument; override;
    end;
```

You must of course, implement the bodies of the methods.

A different but complete version of the Puzzle program is attached.

In addition to writing the program, you must edit a so-called resource file that defines the appearance of your menus and alert messages. Then you must compile the resource file and the program itself using the Lisa Workshop.

Designing your Application

NOTE: In the code fragments used as examples in this section, identifiers that MacApp declares are italicized. The specific application would declare all other identifiers.

To design a MacApp program, you must first analyze it in terms of the standard MacApp architecture.

Will the user be able to see document icons on the desktop, open them from the Finder, and use the Open and Save commands from the application's File/Print menu to access the documents they represent? (For MacWrite™ , MacPaint™ , and MacDraw™ , the answer is yes. For Font Mover, the answer is no.) If the answer is no, the application object owns all the window objects itself. If it is yes, your program requires document objects, e.g.:

```
type
   TLedger = object (TDocument)
   ...
```

Are there windows--other than transient dialog boxes and the Clipboard--that can appear on the screen even when no documents are open? (For MacPaint, the answer is yes: the shape and texture palettes are always on the screen. For MacWrite, the answer is no: the header, footer, find, and change windows disappear when the document is closed).

Any windows that survive even when documents vanish belong to the application object:

```
type
   TPaintApp= object (TApplication)
        fShapePalette, fTexturePalette: TWindow;
        ...

type
   TWriteDoc = object (TDocument)
        fTextWindow:  TWindow;
        fHeaderWindow, fFooterWindow, fSearchWindow: TWindow;
        ...
```

Is there only one type of document, or several? That is, do all documents you support have the same data structure and operations? Would you use one icon for some of the application's documents, and a different icon for others? (Most applications handle only one type of document. Lotus Jazz™ handles several.) The answer to these questions will determine the number of object types you must declare that inherit from TDocument, e.g.:

```
type
   TSpreadSheet = object (TDocument)
   ...
   TRecordFile = object (TDocument)
   ...
```

Can the user choose to see different views of the same data? (In MacWrite, there is essentially one view, with options to display rulers, headers, and footers. In the Finder, there are two distinct views: by Icon, and as a list by Date, by Size, or by Kind.) If different views of the same data are possible, you can declare several types that inherit from TView and that have a data-reference field in common, e.g.:

```
type
   TIconView = object (TView)
        fObjList:  TObjList;
   ...
   TListView = object (TView)
        fObjList:  TObjList;
   ...
```

If multiple views of the same data are offered, can only one be seen at a time? (In the Finder, any one folder can be viewed only one way at a time. In Microsoft™ Chart, a series view and a chart view of the same data can be seen at the same time in different windows. In Microsoft Multiplan™ , the formula of one cell can be seen in the formula bar at the same time that the values in a group of cells can be seen in the worksheet.) If the views alternate in one part of a window, as in the Finder, then you have a frame that can contain different views at different times. If the views can coexist in different parts of the same window, as in Multiplan, then you have a window with multiple frames, each having a different view. If the views can coexist in different window, as in

Chart, then you have a document with multiple windows, each having a different view. No matter which case applies, you do not need to declare any special kinds of windows or frames. Your initialization and command methods simply pass parameters to generic methods to specify the relationships among your views, frames, and windows. For example, to emulate the Finder, you might launch a window with the folowing code:

```
var
   aWindow:     TWindow;
   aFrame:      TFrame;
   anIconView:  TIconView;
   aListView:   TListView;

NEW(aWIndow);
aWindow.IWindow({various parameters}); {Generic initialization method}
NEW(aFrame);
aFrame.IFrame(aWindow, {other parameters, including booleans for scrolling});

IF fCurrViewKind = byIcon THEN
   BEGIN
   NEW(anIconView);
   anIconView.IIconView(aFrame); {IIconView would call IView}
   END
ELSE
   BEGIN
   NEW(aListView);
   aListView.IListView(aFrame, fCurrViewKind ); {IListView would call IView}
   END;
```

Can a single view be split and the separate panes scrolled to show the same or different areas of the document? (The worksheet in Multiplan can be split.) If so, then the view resides in a special frame called a TSplitFrame [not yet implemented]. When you launch the window, you will create the frame using:

```
var workFrame: TSplitFrame;
NEW(workFrame);
```

Can the view be printed using the Print command? (MacPaint palettes and the Multiplan formula bar can not. The Finder's views can not either, because the Print command in the Finder causes applications to start up and print their own views.) If the view can be printed, then it should be declared as a kind of TPrintableView, e.g.:

```
type
   TWorkSheet = object (TPrintableView)
   ...
```

What commands can the user issue? Are some trivially different? (In MacDraw, the commands in the Shades menu differ only by which texture is chosen. Cut and copy differ only in whether the selection is deleted after the command, but intra-caption cut and copy are quite different from graphics cut and copy.) If commands fundamentally

identical, they can be of the same type. Otherwise, different kinds of commands should be declared:

```
type
   TShade = object (TCommand)
        fObjList:       TObjList;
        fTexture:       Color;
   ...
   TCutGraphics = object (TCommand)
        fObjList:       TObjList;
        fDelAfter:      BOOLEAN;
   ...
   TCutText = object (TCommand)
        fCaption:       TTextHandler;
        fDelAfter:      BOOLEAN;
   ...
   TStretch = object (TCommand)
        fObjList:       TObjList;
        fNumerator, fDenominator: INTEGER;
        fDirection: Direction;
   ...
```

What objects does a command affect? Most commands affect the current selection. For example, Cut deletes whatever is currently highlighted. Some commands (e.g., Print) apply to the view containing the selection or to the main view of the active window. Other commands apply to the whole document (Save) or to the whole application (Quit). Be sure each type of command object declares fields that reference the objects affected. In the examples above, fCaption and fObjList serve that purpose.

The Target

There is a global variable in MacApp called gTarget. Its value is the most specific object that contains the selection, usually a view. If there is no selection, gTarget is usually the active window. If there are no windows, gTarget is the application object. In general, your view object will have fields of your choosing that indicate what specific items are currently selected, as well as a method called *HighlightSelection* to highlight those items.

The significance of gTarget is that it gets first crack at most user events. If the user chooses a menu command (or types on the keyboard), MacApp calls the DoMenuCommand (or DoKeyCommand) method of whatever object is currently referenced by gTarget. The object inspects the command to see if it can handle it. If not, it calls its ancestor's DoMenuCommand (or DoKeyCommand) method, which is usually a generic method declared in MacApp. The syntax for invoking the ancestor's method from a DoMenuCommand function is:

```
DoMenuCommand := inherited DoMenuCommand(aCmd);
```

The generic method may handle some commands itself. Ones it can't handle it passes up the ownership hierarchy, i.e., from the view to the frame, the window, and the document, and finally to the application object itself. In each case, your override method gets a crack at handling the command before the generic method in MacApp does. If no object handles the command, MacApp displays an alert message to that effect.

For some applications, a different ordering of event handlers is needed. For example, the target view may wish to give an adjacent view a crack at the command before the frame and window get a chance. Such variations can be specified easily by varying certain parameters of initialization methods.

It should be obvious that maintaining the value of the global variable gTarget is critical. In general, it is your responsibility to update it after every user selection and command. However, MacApp does some updating of its value automatically: if a frame switches views and the old view was gTarget, the new view becomes gTarget; if a window is deactivated and later reactivated, the old value of gTarget is restored; if a new window is opened, it becomes gTarget; if the only remaining window is closed, the application becomes gTarget.

Benefits of Object-oriented Programming

You may have heard claims that, with object-oriented programming, software can be significantly improved in modularity, development time, code size, ease of maintenance, and user interface consistency. You may also have the impression that object-oriented programs suffer in the area of performance. Let us examine these issues with respect to the use of MacApp.

Modularity

Once you have analyzed your application in terms of its documents, views, commands, frames, and windows, you can divide it into modules by object type and think about each object one at a time.

For example, when you are working on the implementation of a command object, you can forget about the rest of the system and concentrate on the following issues. What state needs to be saved to undo the command? Have you declared fields to record that state, and is your initialization method passed sufficient information? Is any additional state needed to do or redo the command?

Now code the DoIt method, the UndoIt method, and the RedoIt method, being sure to leave the clipboard and the selection in their proper state. Usually you need not bother to make QuickDraw calls to update the view. If you do and are not careful, unnecessary delays and/or annoying flashes may result. Instead, simply call the Toolbox procedure InvalRect or InvalRgn to indicate what needs redrawing, and let MacApp call your view's Draw method later.

When you are working on a view object, program menu commands, mouse commands, and key commands one at a time.

For menu commands, declare overrides of DoSetupMenus and DoMenuCommand. In DoMenuCommand, return a new command object whenever the user's action causes the previous command to be too old to undo. If the document is not changed (e.g., Select All), or the previous command is being continued, return the special value gNoChanges. In DoSetUpMenus, call Enable(c, TRUE, checkMarkWanted) for every menu command c for which a case label appears in your DoMenuCommand method.

For mouse commands, determine from the mouse location, current palette choice, and other factors what action is intended. For example, in MacDraw, when the palette choice is the arrow, a button press on an object could begin either a move or select action, while a button press in the background always begins a selection. In either case, create a command to track the mouse. If it turns out to be just a selection, you can return gNoChanges; MacApp will free the unused command.

For key commands, there are several alternatives. If the entire view is a standard unformatted-text editor, declare it a kind of TTextView (you will have to USE unit UTextEdit). TTextView implements DoSetupMenus, DoMenuCommand, DoMouseCommand , and DoKeyCommand for you. If separate areas of a form or drawing can have independent editable text strings, define your own kind of view, but when the user selects one of the strings, create an object of type TTExtHandler (also defined in unit UTextEdit) [not yet implemented], and assign it to the global variable gTarget so it will handle typing.

For each document type, define its data structure and methods, including procedures ReadFrom and WriteTo to read and write data given a file number, a function BytesConsumed to calculate the approximate amount of disk space it would take to save the document , and a boolean function AcceptFile to acdept or reject file names to appear in the Open... dialog.

To retain flexibility, the document object should not make QuickDraw or InvalRect calls, or do anything that makes unwarranted assumptions about how many views exist or about what kinds of views they are. It is fine, however, for the document to associate formatting properties with its data, and for a view to consult that information as it displays.

The views and commands do make QuickDraw and InvalRect calls, extracting information from the document as needed. It is a waste for every one of them to reimplement manipulation of the document's data. To share code and improve modularity, they should call methods of the document when they need to make changes to it.

Some applications support multiple simultaneous views of the same data. When the user changes the document through one view, the other views reflect the change. One way to implement this is for every method of every command object to call a method in every affected view to tell it what part of the document changed, so that the view can invalidate that part. This is the simplest way, but it suffers from lack of modularity. A more modular way is to have the document keep track of what parts of itself changed,

and for it to notify the other views after every command. This works well for some commands, but it is difficult in this scheme to achieve mouse-tracking feedback in all views concurrently. The most flexible solution is for the owning frame, window, or document to create a special TMulVwCommand object [not yet implemented] to intercept every call to a command method. After letting the real command object do its thing, the multi-view command asks the document what changed, and tells the other views to invalidate affected parts.

If you adhere to modular design, it will be easy later to modify a view, add new views, or define additional commands. It will be a trivial matter to plug a view into a splittable or scrollable frame or to make it print.

Development time

MacApp implements many functions for you, saving you months of effort. They include:

- All user actions outside your views (e.g., title bar, scroll bars, resize icon, menu bar) .
- Apple-menu commands (you simply edit the About message in your resource file).
- File-menu commands, including Print commands (you define the TDocument methods AcceptFile, ReadFrom, BytesNeeded , and WriteTo).
- Window opening, closing, moving, scrolling, resizing, splitting, and updating (you supply a Draw method for each view).
- Page break display (you can override page break placement if you wish).
- Automatic scrolling when the mouse is dragged past the outer edge of a frame.

MacApp also does all of the following for you:

- It determines whether the user started the application by selecting the application icon or one or more document icons, and whether the command he issued to the Finder was Open or Print. It then calls appropriate application methods.
- It runs the event loop that receives each user input and serial i/o event, classifies it, and dispatches to appropriate methods. [Serial port events not yet implemented.]
- It counts multiple mouse button clicks in rapid succession (double click).
- It calls special methods during idle time when the queue of user input is empty. Thus, menu titles can be disabled (handled by MacApp), insertion points can blink (handled by UTextEdit), and the cursor shape can be changed [not yet implemented].
- It makes sure no menus are enabled except those that are enabled by gTarget.DoSetupMenus and by methods it calls.
- It keeps track of the command object that will handle the next Undo or Redo, and gets rid of it when the next command comes along.
- It the disk is too full to save the document without deleting the old version, it asks the user whether it may overwrite the old version.
- It handles Reduce To Fit and other zooming operations [not yet implemented].
- It keeps the elevator in the correct place in the shaft at all times.
- It lets your DoMouseCommand and Draw routines deal in local view coordinates.
- It takes care of the details when you tell a frame to switch views.
- It tells you when to highlight and unhighlight selections so as to minimize flashing.
- It makes it easy to resize your view when the document grows, and to make its size an integer multiple of the page size if you desire, even if the page size changes.

And when you are debugging...

- It provides a window for your use to WriteLn debugging information.
- It provides a debugger that can get control at every procedure entry and exit, offering: symbolic trace, keyboard interrupt, breakpoint, single step, symbolic display of the call stack, and hexidecimal display of parameters, locals, object fields, and arbitrary locations in RAM.

Code size

There is no question that the amount of code you write yourself will be much less--in both source and object form--if you use MacApp. A more difficult question is what will happen to the total size of your product, including the code of MacApp.

There are two versions of MacApp: one with debugging features enabled and one without. The former is approximately 60K bytes and the later 30K bytes [numbers subject to change]. Because the debugging version is so large, you will want to use a 512K Macintosh or a Macintosh XL to debug, unless your application is very small.

Of the 30K or so in the end-user version, one-third is printing code that is only swapped in during printing, another third is resident at almost all times, and the rest swaps in small pieces as needed for time-consuming and infrequent operations like initialization, termination, and filing. [Segmentation is currently poor--to be improved.] {Numbers subject to change.]

It is true that you could probably implement your application in a bit less space without MacApp than with it, because you could take advantage of special circumstances to shorten or omit certain routines. However, it is unlikely you would save enough space to warrant the extra development effort, especially if you work for yourself or a small company, or if you are developing for a relatively small market, or for your own use.

In the future, we may provide one or more of the following ways to reduce the code size of your MacApp program:

- We may recode all or parts of MacApp in assembler.
- We may provide a way to strip methods you don't use out of object files.
- We may provide a way for MacApp object code to appear just once on a disk.

Ease of maintenance

Pertinent comments, well-named identifiers, and internal documentation are as necessary as always. But if the recommendations about modularity are followed, MacApp programs should be easier to maintain than programs not written in an object-oriented style.

User interface consistency

Because so much of the standard Macintosh user interface is implemented by MacApp, it is easier for a programmer to conform with the standard than to violate it. However, it is always possible for you to override anything that you find necessary to make your user interface fit your problem better.

Performance

If a MacApp program is written in Object Pascal, its performance will usually be within 10% of what it would be if it ran the same code in straight Pascal. There is some slowdown due to run-time dispatch to methods, but method calls need not occur much in inner loops.

Some programs may actually run faster on MacApp than if you wrote them from scratch. In creating MacApp, we are taking advantage of the experience of a large number of programmers who have developed various performance techniques over the last few years. (If you know of ways to further improve MacApp's speed, please let us know!)

Documentation Plans

Hayden Press is planning a textbook to appear late this year called *Object-oriented Programming for the Macintosh*. It will describe MacApp and include examples in several languages, including Object Pascal, Objective C, assembler, and Smalltalk.

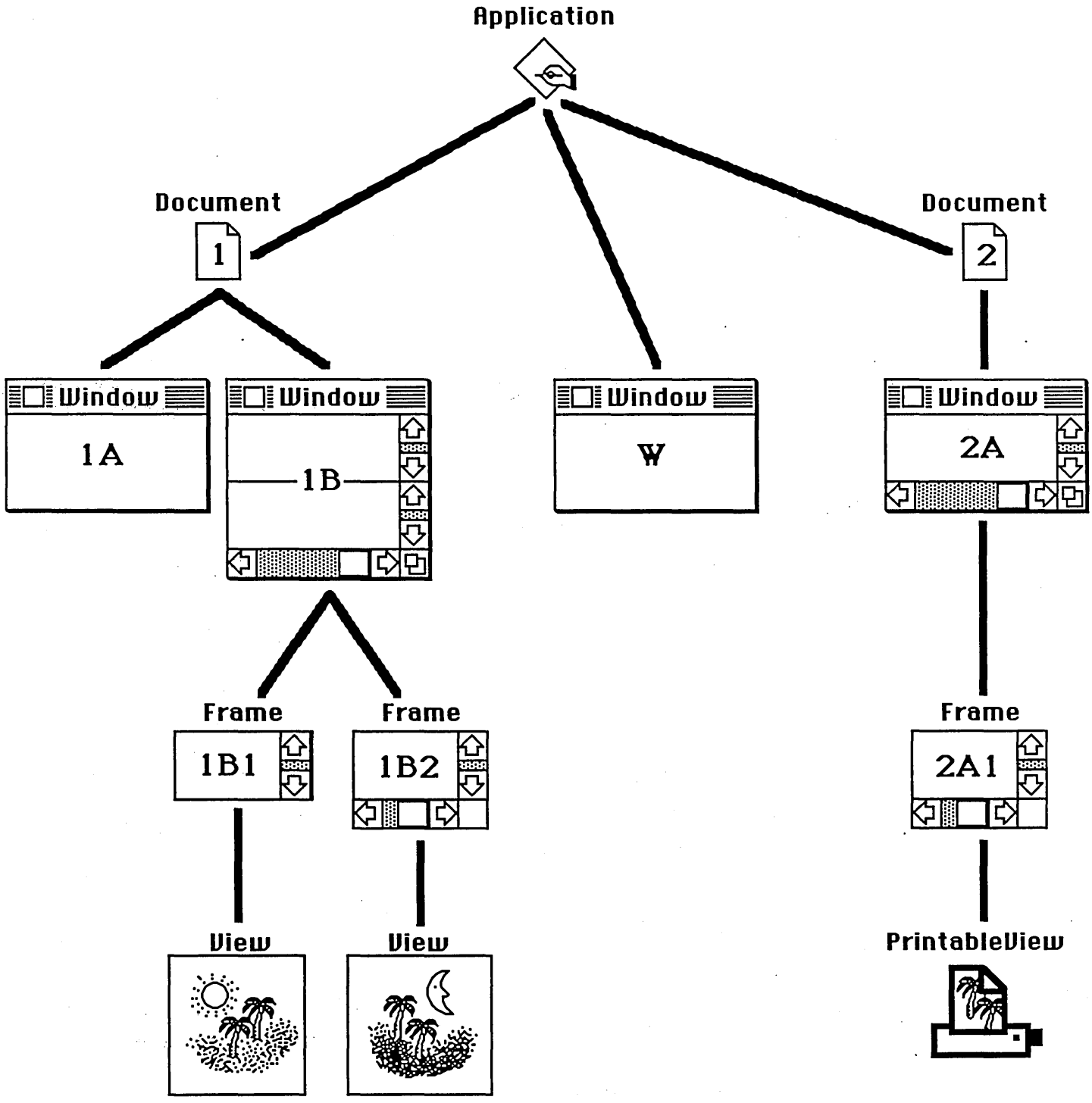
Apple is producing a manual for MacApp that will include:

- Documentation of every class, field, and method.
- Flow diagrams that show what methods are called for every user action.
- A cookbook that lists common things that programmers want to implement or change in MacApp. For each one, it tells you what sections to read in the MacApp manual and *Inside Macintosh*, and what methods you will need to override in MacApp.

The source code of MacApp will be available to study and to edit. However, whenever possible, it is recommended that you override methods rather than edit them.

At this time, all of the promised documentation is in progress. Early users will have to work with inadequate documentation, and suffer some changes of interface from one version to the next. The reason we expect to be be making changes is that we hope early users will tell us what areas of MacApp require improvement.

(To be continued)


Application
Document
1
Document
2
Window
1A
Window
1B
Window
W
Window
2A
Frame
1B1
Frame
1B2
Frame
2A1
View
View
PrintableView