# ET++ – An Object-Oriented Application Framework in C++

*André Weinand*
*Erich Gamma*
*Rudolf Marty*

Institute for Informatics
University of Zurich
Winterthurerstr. 190
CH-8057 Zurich, Switzerland
E-mail: {weinand,gamma,marty}@ifi.unizh.ch
{weinand,gamma,marty}@unizh.uucp

**Abstract**

ET++ is an object-oriented application framework implemented in C++ for a UNIX† environment and a conventional window system. The architecture of ET++ is based on MacApp and integrates a rich collection of user interface building blocks as well as basic data structures to form a homogeneous and extensible system. The paper describes the graphic model and its underlying abstract window system interface, shows composite objects as a substrate for declarative layout specification of complex dialogs, and presents a model for editable text allowing the integration of arbitrary interaction objects.

## 1. Introduction

Since 1984 our research group worked on a UNIX based toolkit to support high-level dialogs on bitmap-oriented workstations. Our toolkit, called ET [Mar86], has been used in several projects. It proved to be useful but appeared to us as too rigid, especially after we started to think in object-oriented terms.

In early 1987 we initiated a follow-up project to design and implement a fully object-oriented application framework for UNIX environments. The implementation language is C++, our development and target systems Sun workstations running Sun's enhanced version of 4.2BSD UNIX and SunWindows™. ET++, as we call the object-oriented application framework, essentially combines the functionality of MacApp™ [Sch86, Ros86] with an object-oriented flavour of the Macintosh toolbox [App85] while enhancing and generalizing the class hierarchy. Besides its uniform object-oriented implementation, the novel aspects introduced in ET++ are:

- Its declarative layout specification for composite objects as found in dialog boxes modeled after a system to typeset mathematical formulas.
- The possibility to build up a hierarchy of independently scrollable views.
- Flexible classes to handle text as a base to integrate arbitrary objects.

Fig. 1 shows a screen with applications developed with ET++.

## 2. ET++ – Technical Objectives

The principle aim in developing ET++ was to design and implement an object-oriented application framework for a UNIX environment with roughly the functionality of both MacApp and the user interface elements of the Macintosh toolbox.

Another goal was to unify the hybrid structure of MacApp and the non object-oriented Macintosh toolbox into a single object-oriented application framework. We believe that a major drawback of MacApp is the fact that MacApp is only a thin layer on top of the Macintosh toolbox. The implementation of complex applications, as a result, always requires some intimate knowledge of both the internal structure of MacApp and the Macintosh toolbox interface. We hoped that a single homogeneous

† *UNIX is a registered trademark of AT&T.*

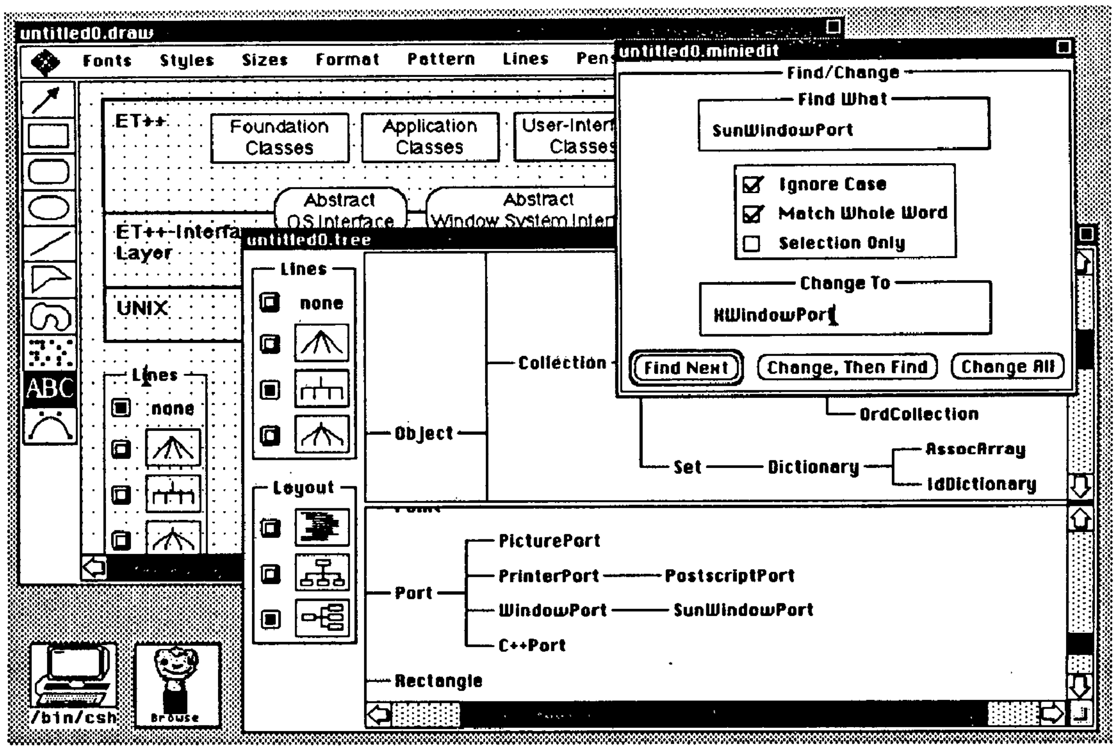


Figure 1: Some applications developed with ET++

system would strongly improve the efficiency of building complex applications, especially for non-expert programmers. This would allow students develop CASE (Computer Aided Software Engineering) applications in short term projects.

We planned to realize the building blocks of the user interface exclusively with concepts known from MacApp and to unify all parts of the implementation of ET++ with a small set of basic mechanisms. If extensions should become necessary they should never make the overall structure of ET++ more complex and less elegant.

The integration of *foundation classes* like those known from the Smalltalk programming environment [Gol83] was another goal. The term *integration* not only means installing these classes in the class hierarchy but also supporting them with mechanisms for interaction and rendering on the screen.

Portability was a major issue in the design of ET++. In contrast to the Macintosh, a UNIX environment lacks an established window system standard. In order to overcome these problems when building portable applications, some implementors of user interface toolkits and application frameworks build their own window system which either occupies the entire screen or lives inside a native window of the host environment. This was not acceptable to us for a number of reasons: first, we wanted to use all tools available on our Suns, second, we did not want to implement a new window system, and third, we believed that it is possible to find a small set of window system functions that is available in most window systems or that can be adapted in a small interface layer between ET++ and an underlying system.

## 3. The ET++ – Architecture

The basic building blocks of the ET++ architecture are a class hierarchy and a small device dependent layer mainly mapping an abstract window system interface to the underlying real window system (Fig.2).

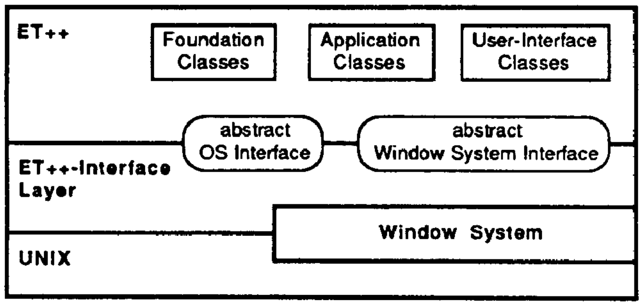


Figure 2: ET++ – architecture

*Foundation classes* represent basic data structures (array, list, set etc.). The *user interface classes* implement graphical elements found in almost every user interface toolkit such as menus, dialogs, or scrollbars. *Application classes* are high level classes that factor out the common control structure of applications running in a highly graphic environment. The *interface layer* provides a small set of functions for window management, input handling, and drawing on various devices.

The basic application classes are derived from MacApp and therefore have a similar behavior and similar interfaces. These classes are `Application`, `Document`, `View`, `ViewFrame`, `Window`, and `Command`. They provide methods for the management of documents, windows, and user commands. For a detailed description of their structure and functionalities we refer to [Sch86a, App86].

## 4. Design and Implementation Aspects

We will focus on the most relevant issues here, in order to show the main differences from other systems and to illustrate the benefits of an object-oriented approach.

### 4.1. Foundation Classes

*Foundation classes* are generally useful data structures. These classes are not strictly coupled with the rest of ET++ and can be used autonomously. ET++'s foundation classes include a simplified version of the Smalltalk-80 *collection classes* [Gol83]. Examples are `ObjList` (doubly linked lists), `SortedObjList` (sorted lists), `OrderedCollection` (dynamically growing indexed arrays), `Set` (lookup tables based on hashing) and `Dictionary` (tables storing a set of associations between keys and values). These data structures are used heavily in the implementation of ET++ itself. Other useful data types included in the foundation classes are `Point` and `Rectangle` which are frequently used in the graphics building blocks.

*Change Propagation:* Tools for graphical software design methodologies often require a mechanism to maintain the connectivity between individual graphical symbols. ET++ supports the coordination among different objects by a change propagation mechanism realized at the root of its class hierarchy (by the class `Object`). Change propagation is modeled after Smalltalk–80's *changed and update principle*.

*Input/Output of Objects:* The foundation classes provides the framework to transfer arbitrarily complex objects from disk to memory and vice versa. This functionality is based on the methods `PrintOn` and `ReadFrom` which may be overridden to store or read an object's instance variables. Input/output of pointers or references to other objects are handled properly by the framework. Although C++ does not provide automatic *activation/passivation* as found in Objective-C™ [Cox86], when working with ET++ object input/output requires only a small programming effort.

The input/output facility of objects together with the flexible *stream* classes of C++ allowed a general implementation of a `DeepCopy` method. These stream classes support not only the transfer of data to disk files but also to a buffer in memory as well. The `DeepCopy` method simply invokes the `PrintOn` method to write an object to a buffer in memory, followed by `ReadFrom` that creates the duplicate object out of this buffer.

Another application of the object input/output facility is a kind of *inspector* for an object. The method `Inspect` defined in class `Object` uses the `PrintOn` method to display the state of an object in an inspector window. This window registers itself as one of the object's dependents. If the inspected object follows the change propagation protocols and announces its changes, the inspector window will be updated accordingly and thus will always show the actual state of its object.

### 4.2. The Graphics Model of ET++

ET++ supports a simple 2-D graphics model which lacks complex operations like scaling and rotating (we plan to integrate zooming into the next release). The model is based on the classes `View`, `VObject`, `ViewFrame` and `BlankWindow`. Unlike MacApp, all user interface elements like menus, dialogs and scrollbars are implemented on top of these classes.

#### 4.2.1. Abstract Drawing Surfaces (`View`)

The view represents an abstract and possibly arbitrarily large drawing surface with its own coordinate system. It is subclassed to add a data structure which reflects the position of graphical elements. Its most important method (`Draw`) is called by ET++ whenever the `View` or a part of it needs to be redrawn. This method is never called directly by an application. Instead, the application informs the view on which rectangular part of its drawing surface has changed and thus needs to be updated. The next time the application is idle, ET++ will call the `Draw` method for the union of all invalidated rectangles, thereby avoiding unnecessary redrawing. In addition, the `Draw` method is also called to print the `View` on other devices (e.g. printers) without need for additional code in the application.

As a benevolent side-effect of this indirect drawing scheme it is always possible to simultaneously show a `View` or different portions of that `View` at several places on the screen without any need for additional support by an application. When an area of the `View` is invalidated ET++ will redraw all places on the screen where that area is visible. Feedback functions like rubberbanding are not based on the invalidation mechanism for optimization reasons. They rather use direct drawing with an exclusive-or raster operation. ET++ provides support for automatically replicating this drawing to all places where a view is visible.

### 4.2.2. Visual Interaction Objects

Drawing into a View can be accomplished by using the graphic primitives provided by the ET++ window system interface (section 4.3). But most graphical operations are basically executed to visually represent some conceptual entity which has its specific interaction behavior (e.g. buttons, menus, etc.). In an object-oriented environment these entities are best represented as objects. We decided to introduce the abstract class VObject (visual object) with a standard protocol for size management, for input handling, and for the object's rendering on the screen.

VObject is the most general graphical class in ET++ and, in this respect, corresponds to the class Object in the overall class hierarchy. The VObject is the result of an evolutionary process in which we tried to factor out the common behavior of all visual interaction objects.

An instance variable of type Rectangle holds the VObject's origin and extent which both can be modified and retrieved by a number of methods. In addition, a VObject defines an abstract protocol for maintaining a basepoint which is essential for the alignment of VObjects representing text. Furthermore, a VObject is also a subclass of the EventHandler, the root of all classes handling input. To react to a certain input event a corresponding method must be overridden. A default implementation forwards the event to another EventHandler.

### 4.2.3. ViewFrame

The most important subclass of VObject is the ViewFrame. An ET++ ViewFrame is kind of a rectangular "hole" in a View through which another View or a part of it can be seen and scrolled. In other terms, a ViewFrame establishes a clipping boundary and has its own coordinate system to show that portion of a View which reflects the current scrolling position.

Unlike a *Frame* in MacApp a ViewFrame has no controls, no border, and no direct support for subframes (ViewFrames inside ViewFrames). But the lack of these elements is not a deficiency of a ViewFrame because a ViewFrame is itself a VObject and therefore can be installed in a View like any other visual object. This simple model allows us to build arbitrary hierarchies of Views as shown in figure 3.

In order to realize subframes in a ViewFrame an intermediate View is necessary to install ViewFrames and other graphical objects (VObjects). Although this might appear as complex it dramatically simplifies both

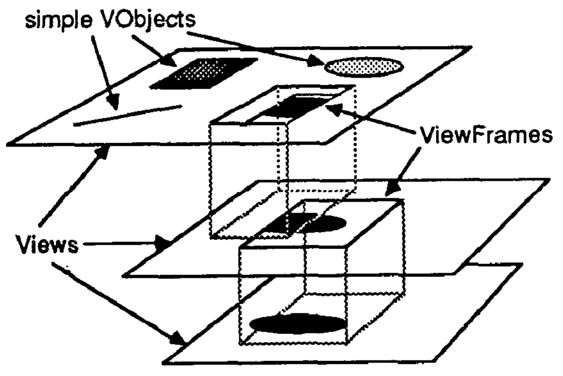


**Figure 3**: View-hierarchy

the implementation and the conceptual graphics model without sacrificing the concept of subframes. Moreover, the model allows the contents of the sub-ViewFrame as well as the View showing these sub-ViewFrames to be independently scrolled.

An example for an application of this mechanism is a View showing several dialog items. With our model it is possible to install an arbitrary View as a scrollable dialog item (text, item lists) in a dialog view.

### 4.2.4. Windows

Up to this point we have only discussed Views and mechanisms to render portions of Views in other Views. But Views are abstract; in order to make a View visible it must be connected with some real area on the screen. This leads us to the class BlankWindow, the root of all window classes.

In ET++ a real window (as provided by the underlying window system) is considered as a "hole" in the screen desktop and thus corresponds closely to a ViewFrame which represents a "hole" in a View. In other words a window is a subclass of a ViewFrame (called BlankWindow) which implements the binding to the window interface. Like the ViewFrame, a BlankWindow has neither a border nor a title bar; it is completely blank. Its main purpose is to implement the update mechanism (section 4.2.1) of the drawing model. A BlankWindow maintains a list of rectangular areas which represent the area on the screen that needs to be updated. Rectangular areas are added to this list whenever a part of the window becomes damaged or exposed by some external event, or whenever the application invalidates an area of a View shown in that window.

In addition, the BlankWindow implements the event loop. Unlike the Macintosh toolbox, ET++ has no notion of an active window. All events are forwarded from the window system interface to the visible window enclosing the current mouse position. Consequently, an ET++ application does not have a single event loop (located e.g. in the `Application` object) but an event loop for every window. The window, in turn, forwards the input events to the interaction objects it contains (`Views`, `ViewFrames`, `VObjects`, etc.).

Windows with borders and title bars are implemented as subclasses of `BlankWindow` by adding a `View` which contains the appropriate graphical elements and a `ViewFrame` representing the contents of the window.

## 4.3. The ET++ Window System Interface

In order to be independent from a special environment (e.g. a certain window system), system dependencies were encapsulated by introducing an abstract system interface defining the minimal set of low-level functions necessary to implement ET++. These functions can be subdivided into the following categories:

- graphical functions, window management, and input handling
- font, cursor, and bitmap management
- operating system services.

All categories are defined as abstract classes which are to be subclassed for a specific environment or output device. These subclasses are considered as the *system interface layer* of ET++.

As an example we describe the structure of the class hierarchy of the first category and discuss their interface (Fig. 4).

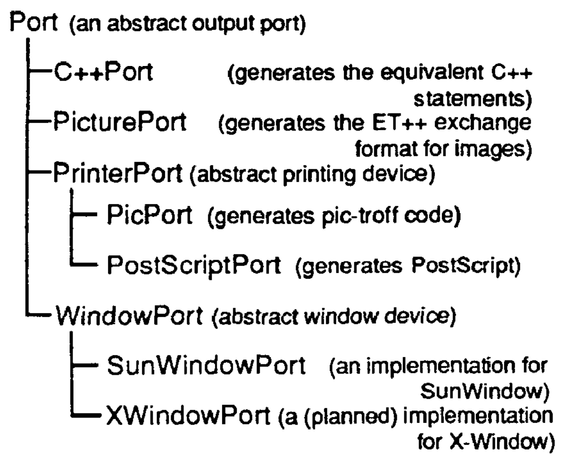

Port (an abstract output port)
C++Port (generates the equivalent C++ statements)
PicturePort (generates the ET++ exchange format for images)
PrinterPort (abstract printing device)
PicPort (generates pic-troff code)
PostScriptPort (generates PostScript)
WindowPort (abstract window device)
SunWindowPort (an implementation for SunWindow)
XWindowPort (a (planned) implementation for X-Window)

**Figure 4:** `Port`-hierarchy

The root of this hierarchy is the abstract class `Port` defining the graphical output primitives common to all output devices (Fig. 5). Subclasses of a `Port` override the abstract output primitives with a device dependent implementation or add device specific methods. The abstract class `WindowPort`, for example, extends the output interface of a `Port` with methods for input handling and window management. The underlying window system must actually only provide mechanisms for the management of overlapping rectangular areas on the screen. All drawing including window borders and title bars is performed completely under control of ET++. The subclass `SunWindowPort` is an implementation of `WindowPort` for Sun workstations; an `XWindowPort` (for X-Windows [Sch86b]) will soon be added.

```
Open, Close
Stroke{Line,Rect,Oval,RoundRect,Arc,
                                  Polygon}
Fill{Rect,Oval,RoundRect,Arc,Polygon}
ShowBitmap
ShowPicture
ShowChar, ShowString
SetClip, ResetClip, SetOrigin
GiveHint
```

**Figure 5:** Graphics primitives

Usage of these classes is straightforward: ET++ maintains a variable `port` that holds a reference to the current output port. All drawing primitives used in an application are automatically applied to this current port.

The device dependent implementation of the graphic primitives of Fig. 5 is stateless, i.e. all primitives take their drawing attributes as parameters. Because this interface is cumbersome to use we added a second interface which maintains state for attributes such as fill pattern, fill mode, pen pattern, pen position, etc. thereby providing an alternative set of graphic functions with less parameters. Other interfaces exist to further reduce the number of parameters for common usages. All these alternative interfaces are based on the stateless primitives implemented in the device independent portion of the abstract `Port` and, as a consequence, do not enlarge the device interface.

Except `GiveHint` all methods from figure 5 have an evident functionality. `GiveHint` provides a mechanism to let drivers optimize their internal behavior by giving them some hints about the high-level structure of a sequence of graphic primitives. A certain value out of an open ended set of constants specifies the additional information for a specific driver. Most calls to `GiveHint` come in pairs, bracketing a sequence of graphic primitives. Notice that all calls to `GiveHint` are of advisory nature only and can be safely ignored by the driver. There is also no need for an application to provide hints by calling `GiveHint`. Three examples illustrate the usage of `GiveHint`:

*Double Buffering.* Double buffering provides for flicker free screen update. This simplifies the implementation because it is no longer necessary to minimize the update region by sophisticated and complex strategies. All drawing requests between a pair of `GiveHint` calls are done in a memory bitmap whose size is given as an argument. At the end of the sequence the bitmap is copied to the screen in a single operation.

*Character Batching.* The text classes display text essentially by calling the method `ShowChar`. It may seem inefficient passing every single character through the device switch and the clipping machinery of the window system in order to get displayed. On the other hand, the complexity of the text manipulation classes would increase substantially if only the method `ShowString` would be used just to optimize speed. To overcome this dilemma, the method `GiveHint` is used to inform the driver when a new line starts and again when it ends. With this information at hand the driver is able to collect single characters of a line into some data structure (a *batch*) and use one single optimized operation to clip and display the entire batch.

*High Resolution Printing.* In this context `GiveHint` is used for high resolution printing of text, e.g. on a PostScript printer. Usually, all text positioning is based on screen pixel coordinates. But a PostScript printer is able to adjust characters much more precisely. `GiveHint` may be used to give additional information about synchronization points within a line to a PostScript driver.

## 4.4. Text Handling

Manipulating text is an essential part of most applications. Text editing features are typically needed in many different contexts such as dialog boxes (dialog items), diagrams (annotations), and browsers (program text). Our goal was to design a framework that can be used in a general way. To achieve this goal we strictly separated between a class to render and format text (`TextView`) and classes for managing the data structures to store the text. These classes are descendants of an abstract class `Text` defining a standardized protocol for all `Text` classes (Fig. 6). To display and edit a text, an instance of a subclass of `Text` is passed to a `TextView`. The `TextView` acts as *Controller* and *View* in the model-view-controller (MVC) [Sch86a] paradigm, the instance of a `Text` class represents the *Model*. The most important (abstract) methods of the class `Text` are `Cut`, `Copy`, `Paste`, and `GetIterator`. The method `Get-Iterator` returns an instance of the class `Text-Iterator`. This iterator retrieves a sub-sequence of text character by character, word by word, or line by line together with the bounding box and the baseline.

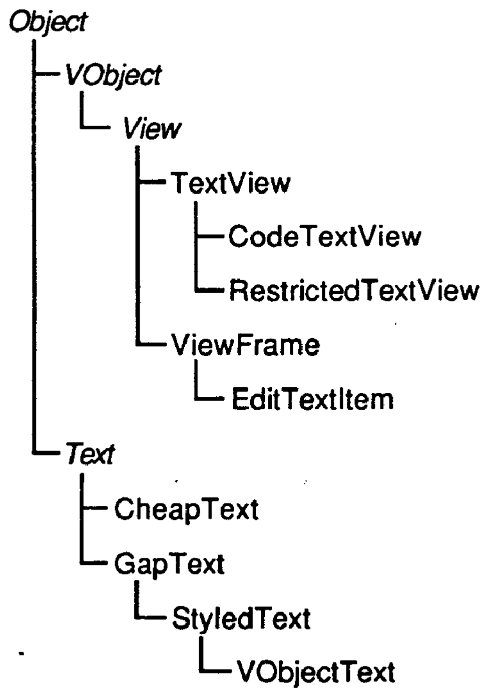
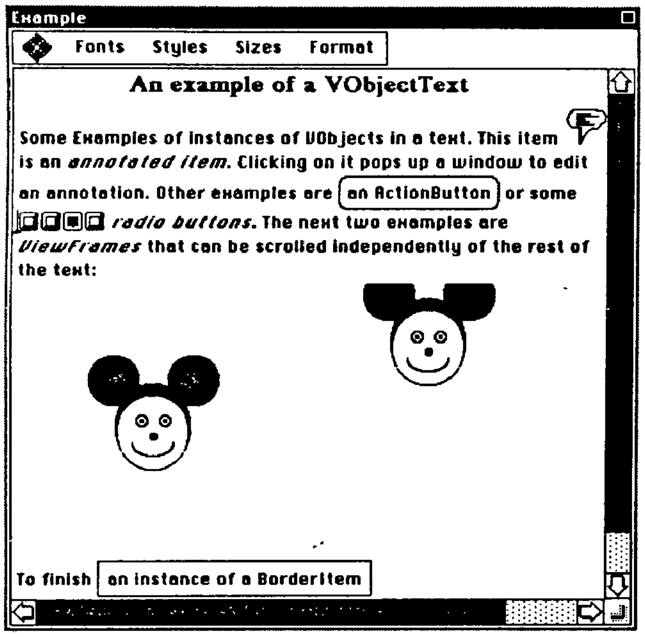


**Figure 6**: Text Classes

`CheapText` is the simplest implementation of a text data structure and is typically used by dialog items. The underlying data structure is a dynamic character array. `GapText` should be used for larger texts and implements the text abstraction as a character array with a gap such as found, e.g., in the text package of the Andrew system [Han87]. A subclass of `GapText`, the class `StyledText`, supports multifont text.

An interesting subclass of `Text` is `VObject-Text`. The protocol supported by visual objects (section 4.2.2) makes it possible to consider a `VObject` as a glyph that can be integrated into text and behaving as an ordinary character. This integration of `VObjects` into text is realized by the class `VObjectText` which extends the methods for cutting and pasting text intermixed with visual objects. Fig. 7 shows an instance of a `VObjectText` rendered by a `TextView`. Applications of inserting instances of `VObjects` into a text are dialog items such as buttons or annotation items as found in hypertext systems. In order to make the dialog items in the text responsive to user input, the methods interpreting input events of the class `TextView` have to be overridden to call the corresponding method of the `VObject`. Remember that visual objects are a subclass of `EventHandler` and therefore support these methods. `TextView` itself is a subclass of `VObject` and it is thus possible to nest instances of the class `TextView`



An example of a VObjectText

Some Examples of instances of VObjects in a text. This item is an *annotated item*. Clicking on it pops up a window to edit an annotation. Other examples are an ActionButton or some *radio buttons*. The next two examples are *ViewFrames* that can be scrolled independently of the rest of the text:

To finish an instance of a BorderItem

Figure 7: An example of a VObjectText

recursively. ViewFrames are a subclass of VObject and their instances can be integrated into a text, too. Since all applications use ViewFrames to display their Views and since the class VObjectText establishes the ground to integrate them into a text we have build a flexible yet simple framework to integrate text and graphics. In Fig. 7 the MouseView from [Sch86a] is installed in a ViewFrame and integrated into the text. Associated with the ViewFrames is their scrolling mechanism. This means, that it is possible to scroll the View shown in the ViewFrame (the mouse picture) independent from the rest of the text. The insertion of ViewFrames into text is an example of a view hierarchy (section 4.2.3). Given the general abstraction of VObjects, the implementation of this special kind of text structure was very straightforward.

A graphical editor, e.g., uses TextViews for annotations and installs them directly into its own view. In order to get scrollable text in dialog boxes or in a program editor a TextView has to be installed in a ViewFrame. The predefined class EditTextItem used in dialog boxes is simply a subclass of ViewFrame showing a TextView. The class CodeTextView adds auto indenting and find-matching-bracket features to the functions provided by a TextView. Instances of the class RestrictedTextView are used whenever the edited text has to conform to a client specified format. This format is specified with a regular expression and checked upon every insertion or deletion of text (there is a class RegularExpression in the foundation classes). A typical application of RestrictedTextViews are dialog items to enter floating point or integer numbers.

Following our goal to use uniform mechanisms wherever possible, the implementation of the class TextView uses the same mechanism for invalidating a region of a view in order to update the screen as described in a previous section. The possibility of double buffering considerably reduced the implementation effort for the text building block. We did not have to design an incremental update algorithm to reduce the amount of flickering on the screen. With double buffering we achieved a flicker free screen update, even for text displayed against arbitrary backgrounds.

## 4.5. Dialog Classes

Dialog items like menus, buttons, scrollbars, and editable texts are the most basic elements of an interactive user interface and are available in almost any user interface toolkit. But usually there is only a fixed set of them and no simple way to build new dialog items from existing ones or to construct them from predefined lower level components. A button, for example, may consist of an image or text, a single or double borderline, and a special behavior to react on mouse clicks. A scrollbar typically consists of an up and a down button together with an analog slider which itself may be a filled rectangle, an image, or even a number reflecting its current value. All these parts may be useful for other kinds of dialogs or in a completely different context.

At first sight multiple inheritance seemed to be a possible way to combine various kinds of basic classes to form the complex items mentioned above. But on second thoughts it became obvious that multiple inheritance was not the mechanism we were looking for. As an example, multiple inheritance does not allow combination of a TextItem and two BorderItems in order to get a DoubleBorderedTextItem.

Another observation was that dialog items most often come in groups. The Macintosh printing dialog, for example, consists of about 30 different items which are placed nicely in a dialog window. On the Macintosh the placement of dialog items can be done interactively with the resource editor. But if the size of a single item changes, the overall layout of the dialog has to be redone. Moreover, the precise horizontal and vertical alignment of text items is a tedious task if done interactively. This lead us to integrate some layout management based on a hierarchical and high level layout description rather than on the explicit placement of items.

We found an almost perfect approach in the UNIX text processing tool *eqn* [Ker75], a *troff*-preprocessor for typesetting mathematics. *Eqn* translates a simple description of a formula into a sequence of typesetting commands. The basic items of *eqn* are characters or strings which can be pieced together with a number of layout operators to form more complex items. Repeated grouping of items finally leads to a tree representation of the formula.

We used this approach to implement all our dialog classes in the following way: a dialog is considered as a tree of `VObjects`, with simple `VObjects` (`TextItems`, `ImageItems` etc.) as leafs and composite `VObjects` (`DialogNodes`) as inner nodes. A `DialogNode` is a object that allows several `VObjects` (e.g. a `Collection`) to be combined into a single, composite object which can be treated as a unit. The class `DialogNode` is abstract because it does not know anything about the layout of the `VObjects` it contains. Its main purpose is to apply methods executed on itself (such as `Draw`, `Highlight`, `Move` etc.) to all of its components and to forward input events to one of them. Subclasses of `DialogNode` are responsible for controlling both the communications between their components and the relationship between the location of these components.

Fig. 8 shows the dialog portion of the class hierarchy.

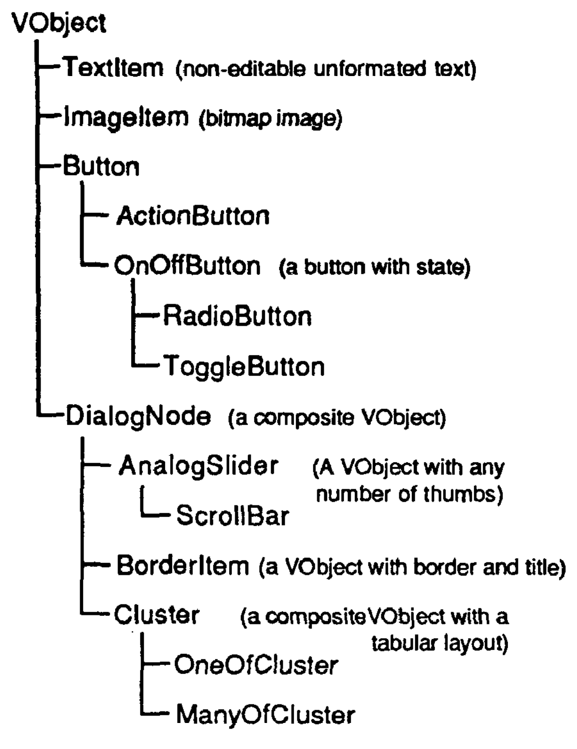
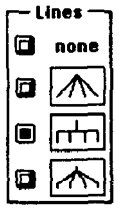

VObject
TextItem (non-editable unformated text)
ImageItem (bitmap image)
Button
ActionButton
OnOffButton (a button with state)
RadioButton
ToggleButton
DialogNode (a composite VObject)
AnalogSlider (A VObject with any number of thumbs)
ScrollBar
BorderItem (a VObject with border and title)
Cluster (a composite VObject with a tabular layout)
OneOfCluster
ManyOfCluster

**Figure 8:** Excerpt of the dialog classes

A `BorderItem`, for example, draws a borderline around its contents and displays an optional title centered above its contents. The contents as well as the title are considered as `VObjects`.

The most important subclass of a `DialogNode` is the `Cluster` which implements a tabular layout of the contained `VObjects`. The commonly used horizontal or vertical lists of items are special cases of a general layout. Each item can be aligned in a `Cluster` horizontally as well as vertically in a number of ways (left, right, center, top, bottom, base).

The `Cluster` presents a very powerful mechanism to fit the needs of complex dialog layouts without having to position items explicitly (Fig. 9).



```
new BorderItem(new TextItem("Lines"),
  new OneOfCluster(HLeft,
    new Cluster(VBase,
        new RadioButton,
        new TextItem("none"),0),
    new Cluster(VCenter,
        new RadioButton,
        new ImageItem(image1),0),
    new Cluster(VCenter,
        new RadioButton,
        new ImageItem(image2),0),
    new Cluster(VCenter,
        new RadioButton,
        new ImageItem(image3),0),
    0
  ),
);
```

**Figure 9:** A dialog and its defining statement

The `OneOfCluster` used in Fig. 9 is a subclass of `Cluster` that implements the one-of behavior of several on-off-buttons.

A `DialogView` implements a standard behavior for modal or modeless dialog boxes. Its method `DoCreateDialog` must be overridden to create the dialog as a tree of dialog items. In `DoSetupDialog` all initial settings of dialog items should be done. The method `Control` must be overridden to react to all dialog interactions.

In addition, the `DialogView` registers all `EditTextItems`, maintains an active insertion point, and allows cycling through these items with a key.

### 4.6. CollectionView and Menus

Another specialized view is the CollectionView, which displays any collection of VObjects as provided by the foundation classes in a tabular format. In fact its implementation is based on the class Cluster of the dialog classes. It also takes care of selecting and deselecting single items as well as contiguous and non-contiguous areas of items. The CollectionView is a basic building block for all user interface objects which have to present a collection of selectable items. It is the root class for menus, menu bars, tools' palettes, or scrollable lists of dialog items. Due to the very general nature of CollectionView a Menu, for example, can always scroll and show items not only as lists but also in a tabular style. Hierarchical popup menus are implemented with items of class PopupItem, which contain a submenu in an instance variable and implement the special behavior to open that submenu.

The evolution of the classes CollectionView and Menu illustrates a very interesting example of the principle "Promotion of Structure" as defined in [Ste86].

In a first version of the class Menu, the item list was implemented as a simple linked list for efficiency reasons. Later it became necessary to have the items of a font menu sorted alphabetically. The first idea was to integrate a sort method, but on second thoughts we decided to replace our linked list in the implementation of the class Menu with the more general data structure Collection found in the foundation classes. To show a sorted menu it was now only necessary to use a SortedObjList rather than an ObjList.

At that time the class Menu was basically a View which could render a Collection of special MenuItems as a vertical list from which one item could be selected with the mouse. This mechanism seemed useful in itself e.g. for implementing a palette of tools like the one found in MacDraw. Consequently the next step was to factor out this mechanism into a class called CollectionView.

Simultaneously, we extended the layout algorithm to show a Collection not only vertically but also in a two-dimensional style. With this extension the CollectionView became one of the most reusable parts of ET++. We became able to build not only menus but also menu bars for Macintosh-style pulldown menus, scrollable lists of arbitrary items, and tools' palettes with a few lines of code.

In yet another step we extended the CollectionView to work with the general VObject instead of a more specialized MenuItem which enabled us to implement the graphics part of a simple spreadsheet application with each cell containing a full-fledged text editor with just a couple of lines.

The last step was to replace the special layout algorithm by a dialog item of type Cluster to further reduce the source bulk of CollectionView.

## 5. The Anatomy of an ET++ – Application

This section tries to give an impression of how the different classes of ET++ fit together and what they contribute to a typical application. We briefly discuss some structural and functional properties of the tree editor displaying the class hierarchy in Fig. 1. This tree editor is a very simple application of some 500 lines of code but nevertheless shows the most important characteristics of an ET++ application.

The following features are automatically provided by the application framework classes and need no special programming effort.

- scrollable views (including autoscrolling)
- splittable views allow several panes (e.g. ViewFrames) looking on disconnected portions of the tree view
- multiple documents in multiple windows and window management dialogs
- automatic file and dialog management for loading and storing a tree onto disk
- printing of a view and dealing with multiple pages
- generating the ET++ exchange format of images which can be read by other ET++ applications, for example a graphics editor
- double buffering for flicker-free screen update.

The implementation of the tree editor reveals the high reusability of the ET++ classes and centers around the DialogNode which provides a convenient framework to handle the layout management of several subtrees. The subdivision of the window into a dialog and a tree part is another application of the layout mechanism provided by the dialog classes.

The visual representation of inner nodes and leafs of a tree are just instances of the very general class VObject. The current implementation uses simple static TextItems, but the modification of a single line would suffice to replace them with a full-fledged text editor (e.g. an EditTextItem). Forwarding of input events is implemented in the DialogNode. Maintaining an active insertion point is handled by the DialogView.

A DialogNode manages a Collection of arbitrary VObjects. Thus, the resulting data structure underlying the tree editor is basically a Collection of Collections. Transfering this data structure to disk and reading it back into memory is almost completely handled by the foundation classes. As long as no new subclasses of VObjects are inserted into the tree no additional code for input/output has to be written.

The implementation of undoable commands, e.g. relinking of leafs and subtrees is substantially simplified with the class `Command` and the possibility to make *deepcopies* of arbitrary objects. This provides an easy way to make a copy of a subtree before executing a command.

## 6. Current Status

The ET++ class library is currently in use by three groups at our institute. Recently, we started to use ET++ in student projects. During the development of the class library several applications evolved which were used to test the functionality of ET++: A drawing program comparable with MacDraw which also supports connections between graphical elements, a tree editor used to browse in the class hierarchy (Fig. 1), and a spreadsheet, just to name some.

### 6.1. Some Statistics

The current implementation consists of 140 classes containing 1653 methods. 309 of these methods are C++ inline functions typically used to access instance variables. Only a subset of these classes and methods has to be known for using ET++.

**Table 1**

| | lines of code | classes | methods |
|---|---|---|---|
| Foundation Classes | 6000 | 34 | 403 |
| Application Framework | 13200 | 97 | 1071 |
| Device Interface | 6800 (3000 C) | 9 | 179 |

We estimate that around 30-40 classes are used to develop a typical application. Our class library comprises 23000 lines of C++ and 3000 lines of C code (Table 1). C has only been used in the interface code to the window system. Table 2 shows the distribution of the classes among the different levels of the hierarchy giving an indication of the degree of subclassing used in ET++.

(Level 0 contains the class `Object` and other classes not derived from `Object`). These numbers give an idea of the high-reusability of the classes for the implementation of ET++ itself.

**Table 2**

| level | classes | % of all classes |
|---|---|---|
| 0 | 13 | 9.1 |
| 1 | 24 | 17.1 |
| 2 | 33 | 23.6 |
| 3 | 19 | 13.6 |
| 4 | 16 | 11.4 |
| 5 | 18 | 12.9 |
| 6 | 11 | 7.9 |
| 7 | 6 | 4.3 |

## 7. Related Work

The architecture of MacApp provided the base for ET++. There already was an effort in the Intermedia project [Mey86, Yan88] to port the MacApp framework to non-Macintosh environments such as Sun workstations or IBM RT PC's. Their approach differs from ours in that they based their MacApp framework on a port of the Macintosh toolbox, CadMac done by Cadmus Computer Systems. CadMac is implemented in conventional C and does not use object-oriented techniques. Intermedia extended the object structure of MacApp by introducing special subclasses for their hypermedia system. The resulting system is not integrated into a standard window system. Intermedia is currently one large integrated process with the disadvantage that it is impossible to run several tools simultaneously. Due to its smooth integration into an existing window system, tools developed with ET++ can run in parallel with already existing ones.

Another user interface toolkit available for UNIX environments is IC-PAK™ 201 [Cox86]. The IC-PAK 201 class library from Stepstone™ is a collection of Objective-C classes which they call *user interface software-IC's*. The main point where ET++ and IC-PAK 201 differ is the user interface architecture they are based upon. IC-PAK 201 uses Smalltalk–80's model-view-controller (MVC) paradigm whereas ET++ is based on MacApp. Beyond this architectural difference ET++ provides a richer set of user interface classes. Unlike MacApp or ET++, IC-PAK does not support automatic transparent printing.

Other object-oriented class libraries for Unix environments typically address only a subset of the needs of graphical applications. For example the object-oriented program support (OOPS) class library [Gor87] corresponds roughly to the foundation classes provided in ET++ but provides no classes except `Point` and `Rectangle` for building graphical applications. We were not able to base our work on OOPS because it was not available when we started. OOPS offers some interesting classes not provided by our foundation classes especially the ones to program with coroutines similar to those of Smalltalk–80.

The X Toolkit [Rao87] coded in conventional C with some object-oriented conventions is based on the abstraction of a *widget*. This is an object providing a user interface abstraction roughly corresponding our `VObjects`. The X Toolkit includes a set of implementations of this abstraction, for example, buttons, labels, forms, or scrollbars. The functionality provided by these classes is comparable with the user-interface classes of ET++. The main difference between ET++ and the X Toolkit or other frameworks for X windows such as InterViews [Lin87] or the Andrew Toolkit [Pal88] is the homogeneous integration of foundation and application classes providing, for example, the base for undoable commands or mouse tracking. Another difference is that these systems are not based on a graphical model separating between `Views` and `ViewFrames` allowing the transparent integration of features like scrolling, autoscrolling, or splittable views into a framework. The X Toolkit and InterViews are tightly coupled with the underlying X window system. An exception is the Andrew Toolkit which is based on a similar abstract window system interface as ET++. Another feature of the Andrew Toolkit is the integration of dynamic linking into an object-oriented environment. Recently we started experimenting with dynamic linking in ET++, too and we integrated this feature transparently with our object input/output facility.

## 8. Implementing an Application Framework in C++ under UNIX

Using C++ as the implementation language of ET++ has worked extremely well. The well known efficiency of C++ was a very favorable background for the implementation of ET++. Indeed, we never experienced efficiency problems due to dynamic binding. In addition to the object-oriented concepts C++ provides some other features that improved the programming interface as well as the code of ET++: *Default arguments* are often used in the constructors of ET++ classes and support both easy to use but still flexible method interfaces. *Operator overloading* is another feature of C++ that was very useful for implementing the two classes `Point` and `Rectangle`. Operator overloading helped us to reduce the bulk of code considerably and also improved its readability. The related feature of *function overloading* was used mainly in constructors and is an elegant way to provide alternative interfaces to create instances of a class.

The stream classes as provided by C++ proved to be a very powerful abstraction. As shown in the foundation classes, streams provided for a straightforward implementation of a `DeepCopy` method. Another application of streams is a text window with a stream interface that is used to display debug messages. Its implementation consisted of overriding one method of a stream class.

The strict compile time checking of C++ was a big help for the parallel development of ET++. When merging our development efforts, the new version was in most cases up and running as soon as it compiled error-free.

ET++ has been developed with a C++ version not supporting multiple inheritance. The current class hierarchy of ET++ is easy to understand and we are not sure if we would have arrived at the same clear class structure had we begun with multiple inheritance in the first place. We are looking forward to experimenting with the new version of the C++ compiler supporting multiple inheritance.

In contrast to Objective-C, C++ does not provide any information about the class structure or the field layout of an object at run time. Having some of this information available, input/output of objects could be provided at the root of the class hierarchy for all classes. C++ has no built-in facility to check the type of an object at runtime. With some programming conventions it is possible to implement an `IsKindOf` method in the class `Object`. This is done in the OOPS library, and in ET++ we use a similar scheme. But this leads to different roots of the class hierarchies and to different conventions.

Executable modules produced with the ET++ library are quite large (0.8-1 MByte). For our *draw* application, for example, the binary bulk can be divided into: 150 KByte application code, 300 KByte ET++ library, 400 KByte SunWindow libraries. When running several ET++ tools in parallel, the memory requirements are high. These memory problems are drastically reduced with the possibility to create shared libraries as included, e.g., in Sun's new operating system release.

## 9. Conclusions and Summary

ET++ is an application framework based on the architecture of MacApp. It is smoothly integrated into a Unix environment with a conventional window system. All levels of the implementation use object-oriented techniques. The novel aspects described in this paper were an approach for handling composite objects and their declarative layout specification, classes for editable text allowing the integration of arbitrary interaction objects, and finally a model to build hierarchies of scrollable views.

We believe that without applying object-oriented techniques it would not have been possible to implement a system that comprises the functionality of the Macintosh toolbox and of MacApp by only two programmers in just one year.

## 10. Future Work

The starting point for our effort was to develop CASE-tools. We are now starting to work on tools for well known design methods such as entity-relationship modeling. In the near future we are particularly interested in tools and methods for object-oriented design.

Another aspect that we want to study is the integration of the ET++ application framework into a server-based window system such as NeWS™ [Mic86] or X-Windows. One benefit of a server-based window system will be a further reduction in the size of the executable modules. NeWS supports dynamic loading of client code into the server process, and we are interested in what parts of an application framework can be moved into the server.

## Acknowledgements

The authors would like to thank Peter Schnorf, Bruno Schäffer, Christoph Draxler, Karin Imholz and Anna Schlosser for their helpful comments on earlier versions of this paper. We are grateful to our early users — Wolfgang Pree, Hanspeter Mössenböck and Duri Schmidt — for their tolerance working with an evolving experimental system.

## References

[App85] Apple Computer, *Inside Macintosh Volume I*, Addison-Wesley, Reading, Mass., November 1985.

[App86] Apple Computer, *MacApp Programmer's Manual*, Apple Computer, Inc., Cupertino, CA, November 1986.

[Cox86] Brad J. Cox, *Object Oriented Programming*, Addison-Wesley, Reading, Mass., 1986.

[Gol83] Adele Goldberg and David Robson, *Smalltalk-80, The Language and its Implementation*, Addison-Wesley, Reading, Mass., November 1983.

[Gor87] Keith E. Gorlen, "An Object-Oriented Class Library for C++ Programs," *Software—Practice and Experience*, vol. 17, no. 12, pp. 899-922, December 1987.

[Han87] Wilfred J. Hansen, "Data Structures in a Bit-Mapped Text Editor," *BYTE*, vol. 12, no. 1, pp. 183-189, January 1987.

[Ker75] B. W. Kernighan and L. L. Cherry, "A System for Typesetting Mathematics," *Comm. Assoc. Comp. Mach.*, vol. 18, pp. 151-157, Bell Laboratories, Murray Hill, New Jersey, March 1975.

[Lin87] Mark A. Linton, "InterViews Reference Manual (Version 2.1)," Computer Systems Laboratory, Stanford University, September 1987.

[Mar86] Rudolf Marty and Erich Gamma, *ET – An Editor Toolkit for Bitmap-oriented Workstations*, 86, Institut für Informatik der Universität Zürich, Zürich, 1986.

[Mey86] Norman Meyrowitz, "Intermedia: The Architecture and Construction of an Object-Oriented Hypermedia System and Applications Framework," *OOPSLA'86, Special Issue of SIGPLAN Notices*, vol. 21, no. 11, pp. 186-201, Portland, Oregon, November 1986.

[Mic86] Sun Microsystems, Inc., *NeWS Preliminary Technical Overview*, October 1986.

[Pal88] Andrew J. Palay, Wilfred J. Hansen, Michael L. Kazar, Mark Sherman, Maria G. Wadlow, Thomas P. Neuendorffer, Zalman Stern, Miles Bader, and Thom Peters, "The Andrew Toolkit - An Overview," in *Proc. EUUG, London*, 7-9 April, 1988.

[Rao87] Ram Rao and Smokey Wallace, "The X Toolkit: The Standard Toolkit for X Version 11," in *USENIX Association Conference Proceedings (Atlanta, Georgia, June 9-13)*, pp. 117-129, USENIX Assoc., El Cerrito, Calif., 1987.

[Ros86] Larry Rosenstein, Ken Doyle, and Scott Wallace, "Object–Oriented Programming for Macintosh Applications," in *ACM Fall Joint Computer Science Conference, Dallas Texas*, pp. 31-35, November 2-6, 1986.

[Sch86b] Robert W. Scheiffler and Jim Gettys, "The X Window System," *Transactions on Graphics*, vol. 5, no. 2, pp. 79-109, April 1986.

[Sch86] Kurt J. Schmucker, "Macapp: An Application Framework," *Byte*, vol. 11, no. 8, pp. 189-193, August 1986.

[Sch86a] Kurt J. Schmucker, *Object Oriented Programming for the Macintosh*, Hayden, Hasbrouck Heights, New Jersey, 1986.

[Ste86] Mark Stefik and Daniel G. Bobrow, "Object-Oriented Programming: Themes and Variations," *The AI Magazine*, vol. 6, no. 4, pp. 40-62, Winter 1986.

[Yan88] Nicole Yankelovich, Bernard J. Haan, Norman K. Meyrowitz, and Steven M. Drucker, "Intermedia: The Concept and the Construction of a Seamless Information Environment," *IEEE Computer*, pp. 81-96, January 1988.