Design and Implementation of ET++, a Seamless Object-Oriented Application Framework[1]

André Weinand,
Erich Gamma,
Rudolf Marty

Abstract: ET++ is a homogeneous object-oriented class library integrating user interface building blocks, basic data structures, and support for object input/output with high level application framework components. The main goals in designing ET++ have been the desire to substantially ease the building of highly interactive applications with consistent user interfaces following the well known desktop metaphor, and to combine all ET++ classes into a seamless system structure. Experience has proven that writing a complex application based on ET++ can result in a reduction in source code size of 80% and more compared to the same software written on top of a conventional graphic toolbox. ET++ is implemented in C++ and runs under UNIX™ and either SunWindows™, NeWS™, or the X11 window system. This paper discusses the design and implementation of ET++. It also reports key experience from working with C++ and ET++. A description of code browsing and object inspection tools for ET++ is included as well. ET++ is available in the public domain.[2]

Key Words: application framework, user interfaces, user interface toolkits, object-oriented programming, C++ programming language, programming environment

1 Introduction

Making computers easier to use is one of the reasons for the current interest in interactive and graphical user interfaces that present information as pictures instead of text and numbers. They are easy to learn and fun to use. Constructing such interfaces, on the other hand, often requires considerable effort because they must not only provide the functionality of conventional programs, but also have to show data as well as manipulation concepts in a pictorial way. Handling user commands from input devices such as a mouse or a keyboard in order to build an event driven application complicates the programmer's task even more.

Much of the user-friendliness of applications comes not only from an iconic user interface but also from a uniform user interface across applications. This leads to a significant amount of development redundancy because most of the code required by the user interface has to be reengineered for every new application.

---

1 Reprinted from *Structured Programming*, Vol. 10, No. 2, 1989. ©1989 Springer-Verlag New York Inc.
2 The ET++ project was partially supported by the Swiss National Science Foundation.

A first solution to reduce this complexity has been the invention of so called *toolboxes*, rich collections of library functions that implement the low-level components of the user interface like windows, menus, and scrollbars. The toolbox may be part of the system software for a particular computer or of specific window system software. The biggest problem of most conventional toolboxes is their lack of flexibility and extensibility paired with a considerable overall complexity. Quite likely, it is not possible to upgrade their functionality or to add new components without modifying or duplicating source code.

Recent toolbox implementations like the *xt* toolbox for the X window system [Rao87] and *atk* for the Andrew system [Pal88] use object-oriented programming to improve flexibility and extensibility by dynamic binding and inheritance. But even the functionality of these toolboxes is inadequate for substantially easing the application building process.

Much of a typical application's main program built on top of a toolbox is merely program “glue” that manages the calling of toolbox subroutines or, in object-oriented toolboxes, the message passing between objects. The major drawback of the toolbox approach is that it does not define an overall structure for an application. This application structure is therefore often given as a program skeleton that can be copied and modified to fit the application's requirements. But skeletons are not the optimal solution because they duplicate code which should go into a library and because they make the application code more complex and less manageable.

A promising solution is that of an (object-oriented) *application framework* that defines much of an application's standard user interface, behaviour, and operating environment so that the programmer can concentrate on implementing the application specific parts.

Prominent examples for application frameworks are Smalltalk-80 for the Smalltalk user interface, the Lisa Toolkit [Wil84] for the Lisa, and MacApp [Sch86, Ros86] for the Macintosh user interface [App88].

An application framework allows reusing the abstract design of an entire application, modelling each major component with an abstract class [Joh88]. In a graphical application, for example, these components are documents, windows, commands, and the application itself.

While the framework approach is useful for the development of any software, it is especially attractive if a standard user interface should be encouraged, for it is possible to completely define the components that implement this standard and to provide these reusable components as building blocks to other developers. This is an advantage over the toolbox approach where user interface “look-and-feel” standards are explained textually rather than being “wired” into the software.

A *User Interface Management System* (UIMS) is an alternative approach which has attempted a strong separation of application components and user interface components. This allows developing and changing each part independently of the others and also is the foundation for multiple interfaces for the same application. UIMSs are well suited for applications dealing with simple interaction only. But direct manipulation interfaces, for example in a graphic editor, require that semantic information be used extensively for controlling feedback, which is in contrast to the initial goal of strongly separating application and user interface parts. As a consequence, current UIMSs are typically fairly limited in the types of interfaces they support [Mye87].

This article presents the design, architecture, and construction of ET++, an object-oriented application framework implemented in C++ for a UNIX environment and various standard window systems.

ET++ combines the functionality of MacApp II [Bia88] with an object-oriented library of user interface components. In addition, ET++ contains many data structures that are not only useful for the implementation of the user interface part, but for the application part as well. All system dependencies are encapsulated in abstract classes. This allows easy porting of the system to another environment.

Providing a class library with such a rich functionality as ET++ considerably increases the learning time of novice users to exploit its full functionality. This becomes particularly evident with deep class hierarchies where many programmers have trouble grasping all the inherited behaviour of a subclass. To tackle this problem a programming environment is integrated into ET++.

2 An Example of an ET++ Application

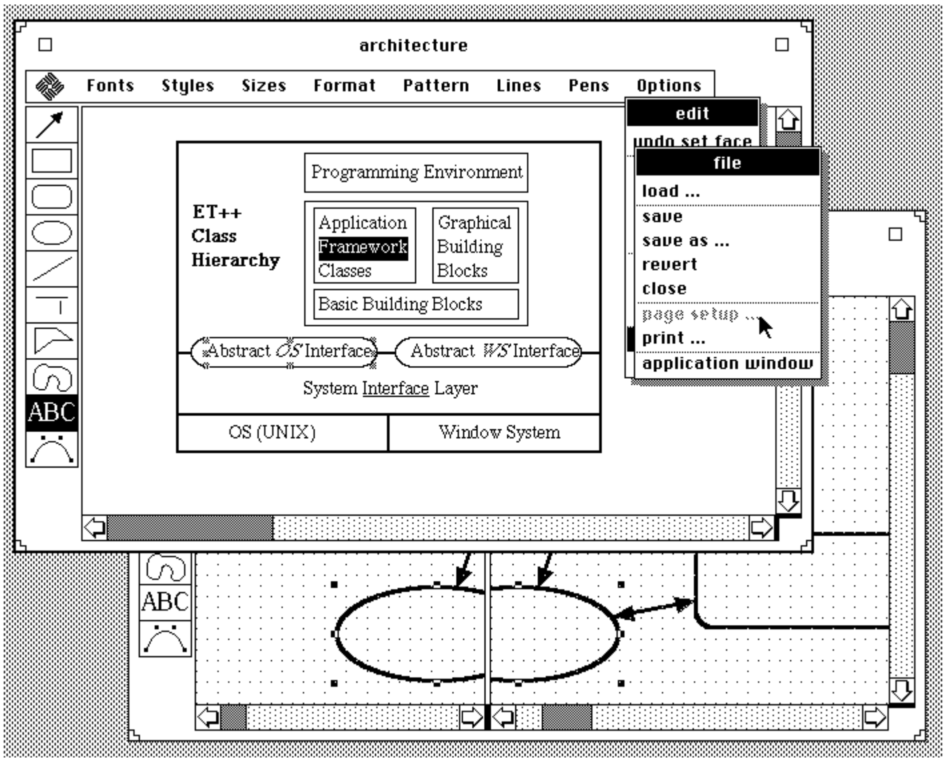


Figure 1. An Example of an ET++ Application.

Figure 1 shows a screen dump of ET++Draw in order to give an idea of what kind of applications ET++ supports. ET++Draw is a drawing program similar in functionality to MacDraw™. The main difference is that the implementation of ET++Draw required some 4000 lines of code whereas the implementation of MacDraw based on the Macintosh Toolbox required almost 8 times as much. The following list highlights some tasks ET++ takes care of without any special effort by the programmer when building applications such as ET++Draw:

- concurrent editing of several drawings in several windows,
- moving, stacking, and resizing of windows,
- scrolling of the window contents (including *auto scrolling* and *real-time scrolling*),
- displaying disconnected portions of the drawing by using several panes,
- file and dialog management for loading and storing a document,
- flicker-free screen update based on double buffering, and
- device independent hardcopy output of the drawing, for example in PostScript™.
- As a benevolent side-effect of the abstract system interface instantiated to a real one at run-time, the application runs on all window systems supported by ET++ without recompilation.

Other parts of the implementation that are not handled automatically but are supported by components of ET++ include:

– data structures underlying the draw application (lists, sets, dictionaries, etc.),
– input/output of the data structures used in the application (even data structures containing cycles, because ET++Draw supports arbitrary visual connections between shapes,
– undoable commands,
– support for transferring a selection of shapes to the clipboard or to duplicate any shapes, a feature which substantially simplifies the implementation of undoable commands),
– layout of a group of graphical objects, for example in dialog boxes, and
– feedback for operations like dragging objects (this mechanism is not based on inherently non-portable XOR raster operations).

3 Design Principles of ET++

The designer of a class hierarchy working with C++ has to choose between a *single rooted* or a forest approach for structuring the class library. In a single rooted library such as known from Smalltalk-80, all classes are derived from a common root class. In the forest approach as used in *libg++* [Lea88], the library consists of a collection of almost independent classes. For ET++ a single rooted approach was chosen because the resulting system is more homogeneous and provides some valuable basic functionality inherited by all classes.[3]

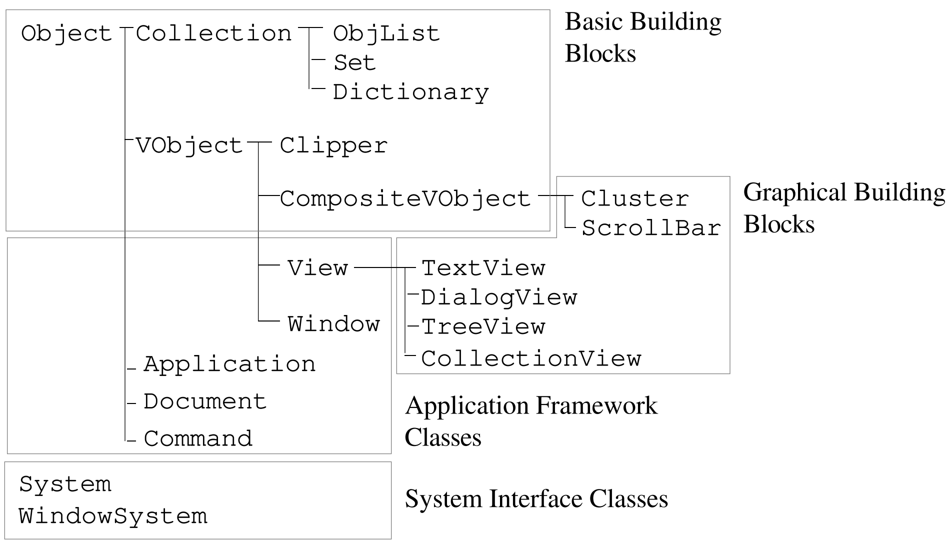


Figure 2. Excerpt of the ET++ Class Hierarchy.

Deriving all classes from a common root class has never been observed to result in increased overhead of ET++ applications. As can be seen in Figure 2, the only classes not derived from `Object` are the classes modelling the system interface of ET++. The ET++ architecture is the result of several design and redesign cycles. Reorganizing the class hierarchy was a common activity during the evolution of ET++. Intermediate designs of ET++ were always verified with existing applications. The principle of *Promotion of Structure* [Ste86] was often applied during the evolution of ET++, i.e.

3 C++ does not support a complete single root approach as in Smalltalk-80 because built-in types are not classes and are therefore not part of the class hierarchy.

methods were moved up in the class hierarchy in order to increase the sharing of code. The structure of the final class hierarchy is rather deep, some metrics are given in section 10.

```
AbstractClassOfDialogComponents
    Button
    Slider
    CollectionOf
```

Figure 3. Introducing Recursion.

Another ET++ design goal is derived from the Smalltalk philosophy [Ing83]: “Choose a small number of general principles and apply them uniformly.” The implementation of ET++ is based on a small set of basic mechanisms. The introduction of recursion follows the same principle. In object-oriented systems recursion can be introduced by using inheritance as illustrated in Figure 3. This paper employs the convention of indenting class names for presenting subclass relationships.

This example illustrates how primitive dialog components can be combined to implement more complex composite items. Primitive components and composites can then be freely exchanged.

Another Smalltalk-80 influence is the so-called *Model-View-Controller* (MVC) paradigm [Kra81]. The MVC paradigm is an approach to modularize the structure of a user interface. MVC strictly separates interactive behaviour of an application from the underlying data structure (model). The interactive behaviour is split into rendering a data structure (view) and reacting on user input (controller). The MVC paradigm was used in ET++ primarily where different implementations of a data structure (model) should have the same interactive behaviour. Our experience has shown that it is not convenient to apply MVC everywhere. Using MVC to implement a menu composed of several menu items, for example, is more of a nuisance than a help. The data structure of menus is so simple that a separation into a view and a model does not improve modularity.

The ET++ class hierarchy and the ET++ programming environment are strongly influenced by the design of the Smalltalk-80 system. The primary reason why the ET++ work was not built directly upon Smalltalk-80 is that a more evolutionary approach to object-oriented systems as stated in Cox [Cox86] was preferred. From a software engineering point of view, a major drawback of Smalltalk-80 is the lack of strong static type checking as found in languages like C++ or Eiffel™ [Mey88].

In a language with *strong static type checking*, the programmer can define in a type declaration which operations are allowed on a variable. This mechanism allows control of how restricted a variable is for a specific algorithm. In ET++ instance variables or method arguments are always declared as a class (type) that is as high as possible in the class hierarchy. The optimum is reached when this is an abstract class. Applying this rule consequently results in algorithms being independent of a specific implementation of a class. A further result is that implementations of an abstraction can be exchanged with minimal effort.

Another consideration during the design of ET++ was the idea of a *narrow inheritance interface* of a class. Behaviour that is spread over several methods in a class should be based on a minimal set of dynamically bound methods. This allows a client deriving from an existing class to override just a few methods in order to adapt its behaviour. Not adhering to the *narrow inheritance* principle often means that too many methods have to be overridden resulting in ugly and bulky code.

4 Architectural Overview

The backbone of the ET++ architecture is a class hierarchy with about 234 classes and a small device dependent layer mainly mapping an abstract window and operating system interface to an underlying real system (Figure 4).

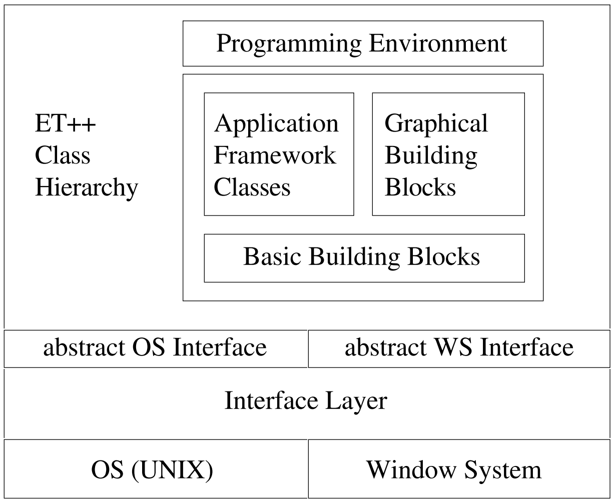


Figure 4. ET++ Architecture.

The *Basic Building Blocks* contain the most important abstract classes of the ET++ class hierarchy: the class `Object`, the root of the overall class hierarchy, implements the common behaviour for all ET++ classes; the class `VObject` (visual object) implements the common behaviour for all graphical classes. In addition, the *Basic Building Blocks* define general useful basic data structures, like arrays, lists, sets etc. which are used heavily for the implementation of ET++ itself.

Application classes are high level abstract classes that factor out the common control structure of applications running in a graphic environment. They define the abstract model of a typical ET++ application and together form a generic ET++ application.

The *Graphic Building Blocks* contain all the graphical and interactive components found in almost every user interface toolbox, such as menus, dialogs, or scrollbars. In addition, it defines the framework to easily build new components from existing ones.

The *System Interface Layer* provides its own hierarchy of abstract classes for operating system services, window management, input handling, and drawing on various devices. Subclasses exist for implementing the system interface layer's functionality for various window systems and the UNIX operating system.

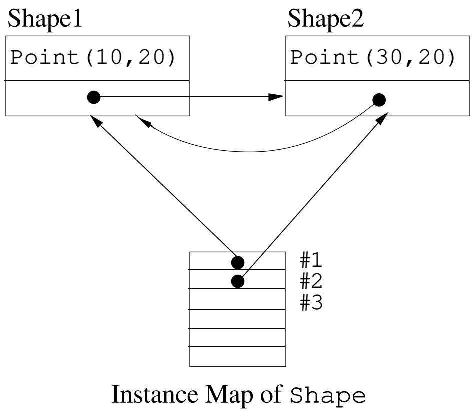
The *Programming Environment* part of the ET++ system is in fact a set of small ET++ applications which are automatically included in every ET++ application. They implement tools for inspecting the running application and browsing through its source code to help the developer understand the program.

5 Basic Building Blocks

5.1 The Class `Object`

The ET++ class hierarchy is single rooted and most classes are derived from the class `Object`. `Object` defines abstract methods for comparing objects and for object input/output to name just two.

The object input/output facility of ET++ supports the transfer of arbitrarily complex dynamic structures from an external medium to memory and vice versa. This functionality is based on the abstract methods `PrintOn` and `ReadFrom`, which are overridden in subclasses to store and read an object's instance variables. The power of the ET++ object input/output facility lies in the fact that the programmer does not have to distinguish between transmitting a pointer to an `Object` and an ordinary scalar typed variable. Moreover, circular structures are linearized transparently to the programmer. Storing pointers is internally implemented by a map per class which assigns a unique identifier to each transmitted instance (Figure 5). This identifier can be transferred to other address spaces or to disk.[4]



Output of `Shape1.PrintOn`:

```
{Shape #1
    {Point 10 20}
    {Shape #2
        {Point 30 20}
        {Shape #1}
    }
}
```

Figure 5. Object Input/Output of a Circular Structure.

Object input/output needs some information about the type of an object at run time, because not only the state of an object but also its corresponding class have to be transmitted. ET++ run-time support would even provide enough information about an object's instance variables to implement the `PrintOn` and `ReadFrom` methods generically in the class `Object` (similar to Objective-C™). But we preferred the approach of a programmer selectively deciding which instance variables should be written to disk. Instance variables caching some state of an object that can easily be reconstructed in the `ReadFrom` method do not even have to be transferred to disk. In [Gor87] Gorlen gives the example of a hash table that compacts itself, i.e. ignores empty slots, when it is saved.

The case of encountering an unknown class while reading back an object structure leads to the discussion of dynamic linking. To handle this case gracefully ET++ includes a mechanism to load a new class and link it to a running application. The implementation of dynamic linking in ET++ exploits the virtual function mechanism of C++ to provide type-safe incremental linking as described in [Str87]. This dynamic linking support can be further used to extend a running system. In the ET++Draw application, e.g., a new kind of shape could be implemented and incrementally linked while the application is running.

The object input/output facility together with the flexible stream classes of C++ allowed the implementation of a generic `DeepClone` method for objects in the class `Object`:[5] The stream classes (the C++ standard classes for input/output) support object transfer not only to disk files but also to a buffer in memory. To do so, the `PrintOn` method is simply invoked to write an object to a dynamically growing buffer in memory; it is followed by `ReadFrom`, which creates the duplicate object by reading the buffer. Experience with ET++ has shown that the `DeepClone` method is hardly ever overridden in subclasses.

The object input/output facility is also used as the standard format to transfer an object to the clip-board. The transparent integration of dynamic linking into the object input/output mechanism allows the copying of instances of classes from the clipboard that are not known in the running application.

---

4 It is interesting to see that a pointer does not identify an object uniquely because an object can be deleted after a transfer and another one allocated at the same address. Consequently, some precautions have to be taken to handle this case properly, otherwise the same identifier would be assigned for different objects.

5 A deep clone of an object consists of its instance variables plus a deep clone of all objects referenced by it.

Another general mechanism provided by `Object` is *change propagation*. The basic idea is to give some support to the synchronization of objects, for example, a model with its associated view in an MVC-design. Providing this mechanism at the root class of ET++ allows the synchronization of completely independent objects. Change propagation is modeled after Smalltalk-80's support for dependencies [Gol83]. There is a method of registering an object as dependent on another object. Modifications of the state of an object are announced with a `Changed` method, triggering a call of the `Update` method for all dependent objects. To react to a change notification, this `Update` method has to be overridden. Change propagation has proved to be a very useful mechanism and is not only used in MVC-designs. ET++Draw, for example, uses change propagation to maintain visible connections between arbitrary graphical elements. Another good example is an inspector tool used to view the current state of an object. Based on change propagation, the inspector can easily update the display whenever the object changes without the inspected object having to know anything about the inspector or the fact that it is currently being inspected.

In conjunction with object input/output, whether a transferred object's dependents should also be output or not has to be decided. Including all the dependents in the transfer has the disadvantage that a lot of non-persistent objects like an MVC view are needlessly output. Consequently, the class `Object` provides a hook allowing the filtering out of dependencies that should not be transferred.

5.2 Container Classes

The basic building blocks of ET++ include abstract data types often referred to as *container classes*. Included are `ObjList` (linked lists), `OrderedCollection` (dynamically growing arrays), `Set` (hash tables), and `Dictionary` to name a few. They are modeled after the *Smalltalk collection classes*. ET++ still lacks a general class for strings, because we still could not find a consensus about its interface. Albeit some extended library functions for string handling are included in ET++.

5.3 Dynamic Type Checking

The container classes all deal with any instance derived from the class `Object` which excludes more specific type checking at compile time. Their implementation has, therefore, to build upon some kind of run-time type checking, eg. an `IsKindOf` method testing the class of an object. In order to perform an equality operation, e.g., it is necessary to dynamically test the types of the objects for compatibility. Depending on the type of an object to perform an operation is considered bad practice in object-oriented systems, but this example illustrates that it cannot be completely avoided in the container class approach followed in ET++.

Parameterized types are an alternative approach allowing more type checking at compile time. C++ currently has no parameterized types but offers only a “poor man's” version thereof based on macros. For this reason the approach relying on dynamic type checking was preferred.

5.4 Robust Iterators

Container classes require a mechanism often referred to as *iterators* [Lis86] to inspect their elements one by one. One way to implement iterators is to store the state of the traversal in the container class itself. The disadvantage of this approach is that only one iterator can be active for an instance of a container class at any time. For this reason ET++ implements a companion class for each container class storing the traversal's state in instances of it. This approach allows several active iterators at the same time.

A problem that has to be considered is what happens when the underlying collection of objects is modified during a traversal. CLU[Lis86], a language with built-in support for iterators, requires that while an iterator is active the collection should not be modified. It is up to the implementor of an iterator to handle this case properly. This restriction cannot be enforced when working with an application framework because a lot of the control-flow resides in the framework. A client can hardly decide whether at a certain point an iterator is active and whether it is secure to remove an object from a collection. ET++ container classes take care of this problem by introducing robust iterators. The basic idea is that deletion of an object is delayed until no more iterators are active. Due to this kind of iterator some hidden bugs and *memory leaks* have been eliminated from ET++. Their implementation profited from the fact that a lot of the code has been factored out and realized in the common superclass of all container classes (`Collection`).

5.5 Run-Time Support for C++ (*Metaclasses*)

One problem of C++ is that its run-time system does not provide any information about the class structure or the instance variables of an object. Consequently, an additional mechanism has to be introduced to gather this information in order to support an `IsKindOf` method and for the object input/output facility.

ET++ uses the approach of associating with each class a special object describing its structure. These descriptors are instances of the class `Class` which is itself a subclass of `Object`. In analogy to Smalltalk-80, they are called *metaclasses* (strictly speaking this is a misnomer because these descriptors are not classes but only instances).

Because the C++ run-time system gives no access to type and structure information, it cannot be circumvented that the programmer has to provide some of this information manually. Nevertheless the principle “do not bother the programmer” was always a consideration in designing the metaclass mechanism. For this reason, preprocessor macros extract as much as possible automatically from the source code. See [Gam89] for a more detailed description of this approach.

Metaclasses store the following information about a class:

- the associated superclass,
- the name of the class,
- the size of an instance in bytes,
- the types of its instance variables, and
- a source code reference to the definition and implementation part of the class.

In the current version of metaclasses only a list of the instance variables' types has to be provided manually, the rest is extracted automatically. The types of the instance variables and the source code reference were introduced in order to give support for the ET++ programming environment.

5.6 Graphical Foundation Classes

`VObject` (visual object) is the most general graphical class in ET++ and, in this respect, corresponds to the class `Object` in the overall class hierarchy. It defines an abstract protocol for managing the size of graphical shapes, for input handling, and for drawing graphical shapes on the screen.

The relatively complex interface of `VObject` mirrors the fact that it should be possible to design all high-level algorithms dealing with graphical objects in terms of the abstract protocol of `VObject` alone. This approach automatically results in the algorithms working on any kind of graphical object. `VObject`, as an example, in addition to its origin and extent, defines the abstract protocol to maintain a baseline, which is essential for the alignment of `VObjects` representing text.

Most of the interface of `VObject` are simple utilities which are implemented in terms of a few dynamically bound methods. In order to create a new subclass, it is only necessary to override a small number of methods. `VObject` provides quite a number of methods to change its origin and extent or the x or y component thereof (`SetExtent`, `SetOrigin`, `SetContentRect`, `SetWidth`, `SetHeight`, `Align`, and `Move`). It would be a pain had all these methods to be overridden in every subclass. But with the *narrow inheritance interface* it is in fact only necessary to override `SetOrigin` and `SetExtent`.

Besides its rendering on the screen every graphical object must include a mechanism to react to input events. `VObject` defines methods for various input events like key and mouse button presses which are called when the corresponding input event occurs. In order to react to a specific event, the corresponding method must be overridden. The default implementation of these methods is to propagate the event to another `VObject` referenced by an instance variable.

A design goal for `VObjects` was to keep them small and lightweight. This means that `VObjects` have very little space or performance overhead even when using thousands of them (for example as cells of a spreadsheet). Consequently, `VObjects` have no built-in coordinate transformation and establish no clipping boundary, which makes it unnecessary to base their internal implementation on the clipping machinery of an underlying window system. Our experience shows that this is rather an asset than a burden because most of the simple graphical objects (e.g. the items in a menu or buttons) do not need a clipping boundary anyway and take no profit from having their own coordinate system.

An interesting mechanism of the graphical foundation classes is their ability to combine several `VObjects` (e.g. a `Collection`) into a single, composite object which can be treated as one `VObject`. The abstract class `CompositeVObject` applies methods executed on itself to all of its components and forwards input events to one of them. The layout management of composite `VObjects` is the responsibility of a subclass. With the introduction of the `CompositeVObject`, `VObjects` are most easily arranged in a tree-like fashion which allows a reasonable default implementation of all event handling methods: the event is propagated through its container `VObject` upward to the root of the tree.

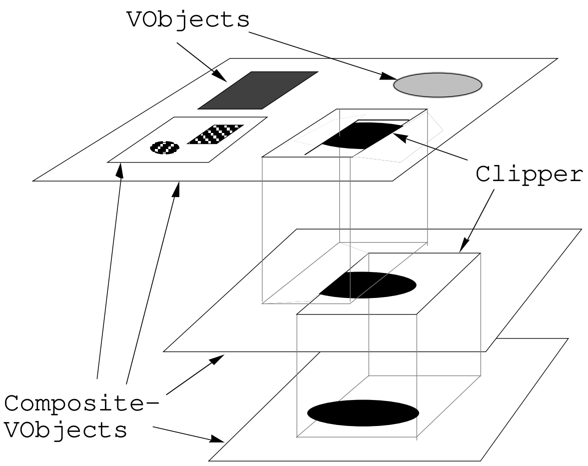


Figure 6. A Hierarchy of Nested Clippers.

The class `Clipper`[6] is a subclass of `VObject`. It defines an independent coordinate system and clips the graphical output of a `VObject` to a rectangular area. It is kind of a “hole” through which another `VObject` or a part of it can be seen and scrolled. The implementation of scrolling is based on this class. Because a `Clipper` is a subclass of a `VObject`, a `Clipper` can again be installed within a `Clipper`. This results in the concept of hierarchies of independently scrollable `VObjects` nested to arbitrary depth (Figure 6). An application for this will be given later in the general discussion of the graphic building blocks.

6 In earlier papers on ET++ (e.g. [Wei88]) `Clipper` was called `ViewFrame`.

6 Application Framework Classes

The most important principle embodied in an application framework is to express an abstract design for a particular kind of application with a collection of abstract classes. These classes define the application's natural components and how they interact. This interaction is the main difference between an application framework and a collection of abstract but not strongly related classes. The former allows the factoring out of much of the control flow into the class library, while the latter requires that the developer know exactly when to call which method.

Because ET++ was designed for the same kind of highly interactive applications as those known from the Macintosh, the natural components of typical ET++ applications are not too different from those of the Macintosh. The classes `Application`, `Document`, `View`, and `Command` are basically derived from MacApp [Sch86] and therefore have a similar behaviour and similar interfaces.

An abstract ET++ application consists of a single instance of the class `Application` which controls the application as a whole and manages any number of `Doc uments`.

A `Document` is another abstract class that encapsulates the data structure or the model of an application and “knows” how to open and close documents (e.g. files) and how to save them on disk. The implementation of opening and closing documents is a very good example of factoring out the common control flow: consider, for example, an end user who intends to open a new file without saving the old modified file to disk. The abstract class `Document` manages all necessary dialogs to ask the user if and where to save the old file and what file to open next.

The main task of graphical applications is rendering the document's data structures on the screen. `VObjects` or subclasses thereof are the basic components for implementing the entities of the model as graphical elements.

In addition, interactive applications have the notion of a current selection on which some operation triggered by the user will be performed. The class `View`, another subclass of `VObject`, represents an abstract and possibly arbitrarily large drawing surface. Its main purpose is to factor out all control flow necessary to manage rendering and printing as well as maintaining a current selection. A `Document` can have any number of `Views`, all showing the same model in various representations. This closely corresponds to the MVC model of Smalltalk.

The application framework class `Window` defines the standard user interface of an ET++ window and implements window-related methods like moving, resizing, and closing. It also implements the mechanism to ease and optimize the screen update.

It is sometimes hard to determine when it is necessary to redraw an object in order to update its image on the screen in response to changes in its internal state. This is due to the fact that visual objects may be obscured partly or completely by other objects which have to be redrawn as well. The ET++ application framework makes use of an indirect drawing scheme which makes it completely unnecessary to call the `Draw` method of a `VObject` directly because all control flow is factored out into the class library. An ET++ application simply announces which objects must be redrawn by calling a method which adds the region occupied by the object to a single update region per window. Whenever ET++ is idle, it requests the application to redraw the update region. Because invalidation is cheap and redrawing expensive, this delayed update mechanism optimizes the redrawing on the screen without further help from the application.

Since redrawing is completely under the control of the framework, it is possible to integrate further optimizations that are transparent to the application. *Double buffering*, for example, provides for flicker free screen update by collecting the output of a sequence of drawing requests in a memory (shadow) buffer, which is copied to the screen in a single operation. This substantially simplifies the implementation of the text handling classes because it is no longer necessary to minimize the update region by sophisticated incremental strategies.

Figure 7 shows an example (snapshot) of the dynamic relationship between instances of the application framework classes for an application with two open documents. Each arrow denotes a reference to another object. Most relations are automatically built and maintained by ET++.

A very important issue of user friendly applications are undoable commands because they allow novice users to explore applications without the risk of losing data. Implementing undoable commands, unfortunately, is a pain unless there is some support from a framework. One approach for their implementation is to collect enough state before executing the command in order to be able to reverse its effect when the user selects “undo”. For a single level “undo” this state can be discarded whenever the next command is performed. Clearly, it is very difficult to build a totally automatic but efficient framework for undoable commands without further support by the programmer. But it is possible to design a framework that factors out the flow of control, leaving only the decisions regarding what state to save and how to “do” and “undo” a command to the programmer.

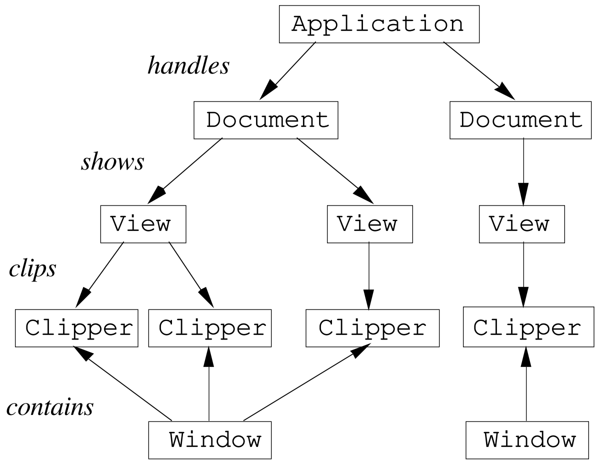


Figure 7. Links between Application Classes at Run Time.

The abstract class `Command` defines the protocol while the class `Document` implements the control flow for dealing with a `Command` object. To implement an undoable command, a subclass of `Command` has to be derived. Such a subclass defines the necessary state variables and methods for doing and undoing the command. ET++ applications never perform commands directly but simply instantiate command objects and pass them to ET++. The framework calls their methods and frees command objects when they are no longer undoable.

`Command` classes, which were first introduced in EZWin [Lie85], are a very elegant example of the reuse of an abstract design and not only ease the implementation effort substantially but also help to modularize complex applications into small and more manageable pieces.

7 Graphic Building Blocks

Graphic building blocks, like menus, buttons, scrollbars, and editable texts, are the “Lego bricks” of an interactive user interface and are available in almost any user interface toolbox. But usually there is only a fixed set of them and no simple way to modify existing ones or to construct new ones from predefined lower level components.

Inheritance is one possibility to modify existing components or add behaviour to them. An example is adding a borderline by overriding the `Draw` method of a `VObject`. But if all predefined items should have a borderline, it becomes necessary to override all corresponding draw methods which results in duplicated code. As another example, a button may consist of an image or text, a single or double borderline, and a special behaviour to react to 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 even 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 thought it became obvious that multiple inheritance was not the ultimate solution. As an example, multiple inheritance does not allow the combination of a `TextItem` and two `BorderItems` in order to get a `Double-BorderedTextItem`.

Another observation was that dialog items most often come in groups. The Macintosh printing dialog, to take just one complex dialog box, consists of about 30 different items which are placed nicely in a dialog window. On the Macintosh the placement of dialog items can be handled 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 led to the integration of some layout management in ET++ which is based on a hierarchical and high level layout description rather than on the explicit placement of items.

An almost perfect approach for two-dimensional hierarchical composition of visual elements is the UNIX text processing tool *eqn*, a troff-preprocessor for typesetting mathematics [Ker75]. 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.

As a result a design idea for all kinds of user interface building blocks is to provide a small number of basic items which can be pieced together with a number of layout operators to form more complex items following the principle of recursion introduced in a previous section. This approach has been used to implement the layout management in ET++. All graphical elements visible on the screen are bound into a tree of `VObjects`, whose root is installed in a `Window`.

The simplest items, such as the classes `TextItem`, `LineItem`, and `ImageItem`, are direct subclasses of the `VObject`. In contrast to more complex components, they do not separate a model and a view because the underlying data structure and its rendering is very simple and interaction behaviour (e.g. editing operations) is typically not needed.

```
VObject
    simple
    TextItem
    ImageItem
    LineItem
    composite
    CompositeVObject
        Slider
        BorderItem
        Cluster
            OneOfCluster
            ManyOfCluster
            ScrollBar
        Button
            ActionButton
            OnOffButton
            RadioButton
            ToggleButton
    complex(MVC)
    View
        StaticTextView
            TextView
        DialogView
        TreeView
        CollectionView
```

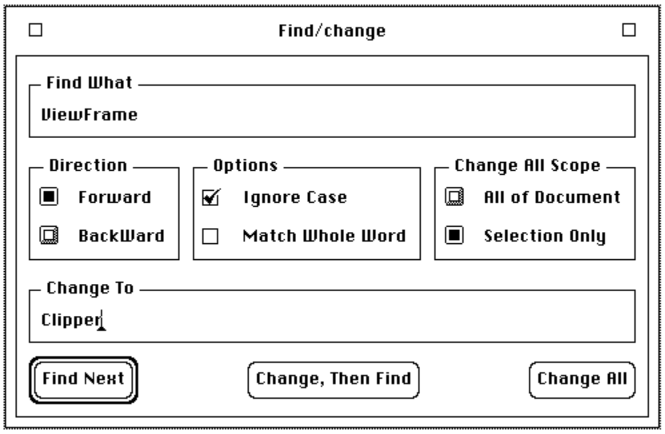
Figure 8. User Interface Components.

Figure 8 shows some of the user interface components of the class hierarchy.

7.1 Composite Objects

The layout operators as subclasses of `CompositeVObject` are responsible for controlling both the communication among their components and the relationships among the locations of these components, i.e. the layout management.

A `BorderItem`, for example, draws a borderline around its contents and displays an optional title aligned above its contents in a number of ways. The contents as well as the title are in turn instances of `VObject`.




```
new Cluster  (VCenter,
     new BorderItem("Direction",
          new OneOfCluster(cIdMode, HLeft,
               new Cluster(VBase,
                    new RadioButton,
                    new TextItem("Forward"),
               0),
               new Cluster(VBase,
                    new RadioButton,
                    new TextItem("Backward"),
               0),
          0)
     ),
     new BorderItem("Options",
          new ManyOfCluster(cIdOpt, HLeft,
               new Cluster(VBase,
                    new ToggleButton,
                    new TextItem("Ignore Case"),
     // more statements
0);
```

Figure 9. Part of a Dialog Box and its Defining Statement.

A `Cluster` implements a tabular layout of its component `VObjects`. The commonly used horizontal or vertical lists of items are special cases of a general layout: Each `Cluster` item can be aligned horizontally as well as vertically in a number of ways (left, right, center, top, bottom, base). The `Cluster` implements a very powerful mechanism which fits the needs of most complex dialog layouts without having to position items explicitly (Figure 9).

The `OneOfCluster` (`ManyOfCluster`) is a subclass of `Cluster` that implements the one-of (many-of) behaviour of several on/off buttons.

```
new Cluster(VCenter,
      new BorderItem("Direction",
            new OneOfCluster(cIdMode, HLeft,
                  "Forward",
                  "Backward",
            0)
      ),
      new BorderItem("Options",
            new ManyOfCluster(cIdOpt, HLeft,
                  "Ignore Case",
                  "Match Whole Word",
            0)
      ),
      ...
0);
```

Figure 10. Building the Dialog with Clusters (Short Form).

Any single item in the statement of Figure 9 could be replaced by an arbitrarily complex composite item. Because this generality is not always needed, other constructors exist for often-used dialog patterns. This allows a much simpler description (Figure 10).

7.2 Complex Objects

More complex user interface building blocks with the notion of a current selection and clipboard support are derived from the class `View` rather than from the class `CompositeVObject` in order to design a framework which can be used in a general way, ET++ strictly separates classes for managingthe data structures and classes for rendering them. A `View` acts as controller and view in the MVC paradigm; the data structure class represents the model.

```
StaticTextView
      TextView
            CodeTextView
            RestrictedTextView
```

Figure 11. The `TextView` Hierarchy.

Text handling classes illustrate this separation very well. A hierarchy of `Views` (Figure 11) incrementally implements the basic text editing and formatting operations on an abstract text data structure. `StaticTextView` renders (displays) an instance of a text class while `TextView` implements basic text editing operations. `CodeTextView` further adds auto indenting, *find-matching-bracket* features, and pretty printing of program text 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 modification of text (there is a class `RegularExpression` in the foundation classes). A typical application of `Restricted TextViews` are dialog items to enter floating point or integer numbers.

```
Text
      CheapText
      GapText
            StyledText
                  VObjectText
```

Figure 12. The Text Hierarchy.

Classes for managing the data structures of a text are descendants of an abstract class `Text` defining a standardized protocol for all subclasses (Figure 12.)

The most important (abstract) methods of the class Text are `Cut`, `Copy`, `Paste`, and `GetIterator`. The method `GetIterator` returns an instance of the class `TextIterator`. 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.

`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` is used for larger texts and implements the text abstraction as a character array with a gap as known from the text package of the Andrew system [Han87]. A subclass of `GapText`, the class `StyledText`, supports multifont text.

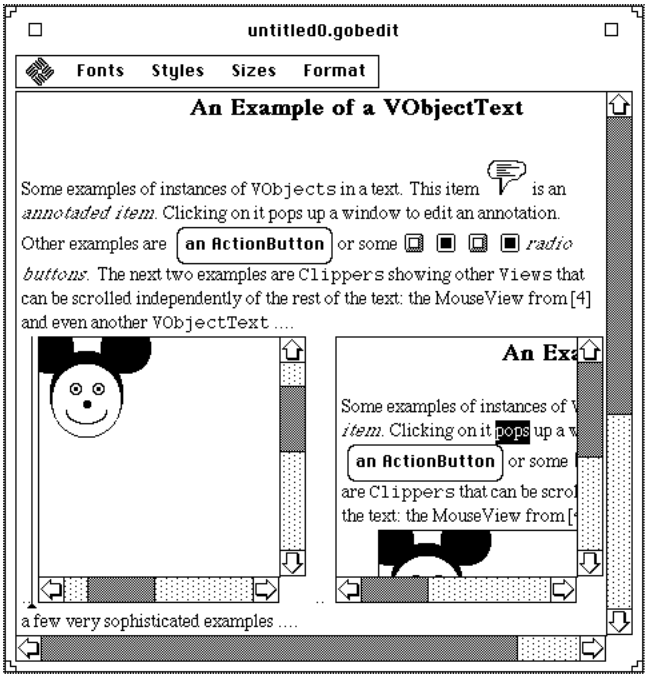


Figure 13. An Example of a `VObjectText`.

An interesting subclass of `Text` is `VObjectText`. The protocol supported by visual objects makes it possible to consider a `VObject` as a glyph that can be integrated into text and that behaves 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. Figure 13 shows an instance of a `VObjectText` rendered by a `TextView`.

Applications of inserting instances of `VObjects` into a text are dialog items like buttons or annotation marks as found in hypertext systems. In order to make dialog items within a `VObjectText` responsive to user input, the `TextView` methods interpreting input events have to be overridden to propagate the input event to the corresponding `VObject`. `TextView` itself is a subclass of `VObject` and it is thus possible to nest instances of the class `TextView` recursively.

`Clippers` are subclasses of `VObject` and their instances can be integrated into a text, too. Since all applications use `Clippers` to display their `Views` and since the class `VObjectText` establishes the ground to integrate them into a text, ET++ provides a flexible yet simple framework to integrate text and graphics. In Figure 13 the `MouseView` from [Sch86] is installed in a `Clipper` and integrated into the text. Associated with the `Clipper` is its scrolling mechanism. This means that it is possible to scroll a `View` shown in the `Clipper` independent of the rest of the text. The insertion of `Clippers` into text is an example of a view hierarchy. Given the general abstraction of `VObjects`, the implementation of this special kind of text structure was very straightforward.

The text building block described so far is flexible enough to be used in many different contexts such as dialog boxes (dialog items), diagrams (annotations), and editors or browsers (program text). 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 `Clipper` (Figure 14).

Following the goal of 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 section 6 . Due to double buffering the screen is updated flicker-free, even for text displayed against arbitrary backgrounds.

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, tool palettes, or scrollable lists of dialog items. Due to the fact that menus are built on top of the very general abstraction of a `View`, a menu 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 behaviour to open the respective submenu.

A `DialogView` implements a standard behaviour for modal or modeless dialog boxes. A single method must be overridden to create the dialog, another to set up the initial state, and a last method to react to all dialog interactions. In addition, the `DialogView` registers all editable text items in order to maintain an active insertion point and to allow cycling through these items with cursor or tabulator keys. Figure 14 shows a dialog and its underlying hierarchy of views.

The `TreeView` renders a tree data structure (in fact a `Collection` of `Collections`) and implements simple editing operations, for example moving nodes and collapsing/expanding subtrees.

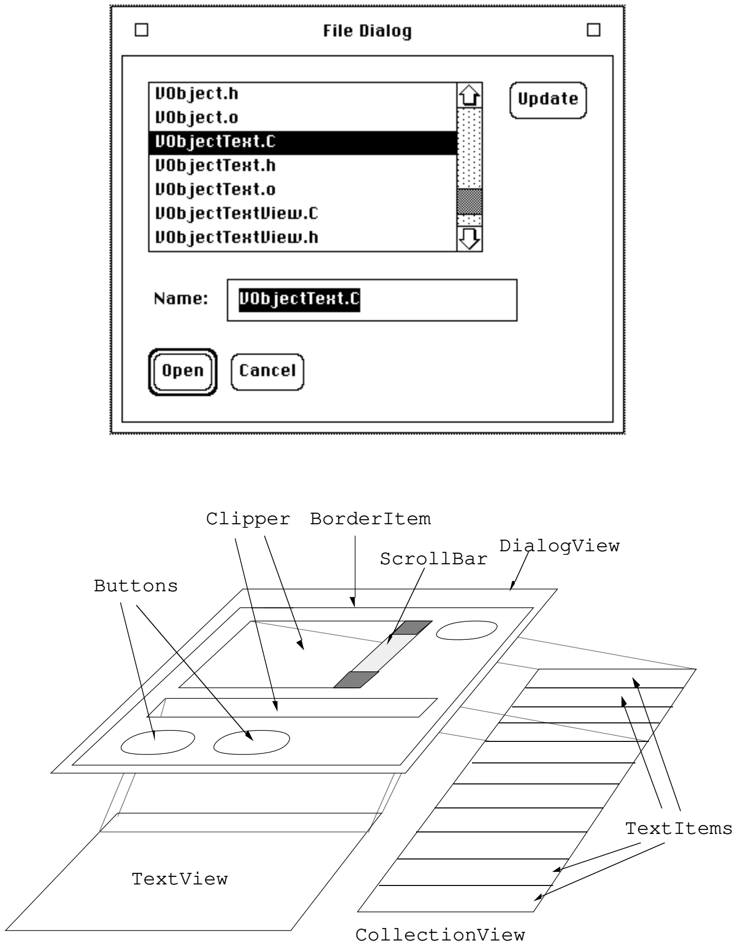


Figure 14. A Dialog and its Underlying `TextView` and `CollectionView`.

8 Object-Oriented Modelling of System Dependencies

Portability was a major issue in the design of ET++. In contrast to the Macintosh, a UNIX environment lacks an established window system standard and even the operating system interface varies among different UNIX implementations. In order to be independent of a special environment, all system dependencies were encapsulated by introducing an abstract system interface defining a minimal set of low-level functionality 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; and
– operating system services.

Every category is defined as an abstract class which has to be subclassed for a specific environment or output device. These subclasses are considered the *system interface layer* of ET++. As a consequence, ET++ classes and applications do not contain any UNIX or window system specific calls, which makes it possible to port ET++ to other environments with a minimum of effort.

The two abstract classes `System` and `WindowSystem` define the entry point into the system interface layer. Their responsibility is to instantiate new objects representing operating system resources like files and directories or window system resources like ports (windows), font managers, images (bitmaps) and cursors.

In addition, the class `System` defines an abstract interface for multiplexing system events coming from different sources. In a terminal emulator, for example, it is necessary to send and receive data to or from another process as well as from the window system.

Due to the clean and abstract interface between `System` and `WindowSystem`, they are completely decoupled, which allows the combination of any window system with any operating system.

```
Port
    PicturePort
    PrinterPort
        PicPort
        PostScriptPort
    WindowPort
        SunWindowPort
        ServerPort
        XWindowPort
        NeWSWindowPort
```

Figure 15. Port Hierarchy.

The hierarchy of port classes and their interfaces is another illustration of appropriate object-oriented modelling of system dependencies (Figure 15).

The root of this hierarchy is the abstract class `Port`, defining the graphical output primitives common to all output devices. Subclasses of a `Port` override the abstract output primitives with a device dependent implementation or they add device specific methods.

The `PicturePort` collects all drawing requests in a compact data structure which represents the standard ET++ exchange format for images.

A `PrinterPort` adds abstract methods for dealing with an abstract printing device. Its subclasses `PicPort` and `PostScriptPort` are implementations for generating *PIC* [Ker81] commands and PostScript.

The abstract class `WindowPort` extends the output interface of a `Port` with methods for input handling and window management. In essence every instance of `WindowPort` represents a single window of the underlying window system. The window system actually only needs to provide mechanisms for the management of overlapping rectangular areas on the screen. Adornments like window borders and title bars are completely under the control of ET++, which results in windows with exactly the same appearance and behaviour across different window systems. This approach is in contrast to some other user interface toolkits which build their own window system which either occupies the entire screen or lives inside a native window of the host environment.

The subclass `SunWindowPort` is an implementation of `WindowPort` for SunWindows™, Sun's standard library-based window system; the class `ServerPort` for a very simple server-based implementation of SunWindows (see below); the `XWindowPort` for X-Windows (X11R3); and the `NeWSWindowPort` for Sun's network window system (NeWS™).

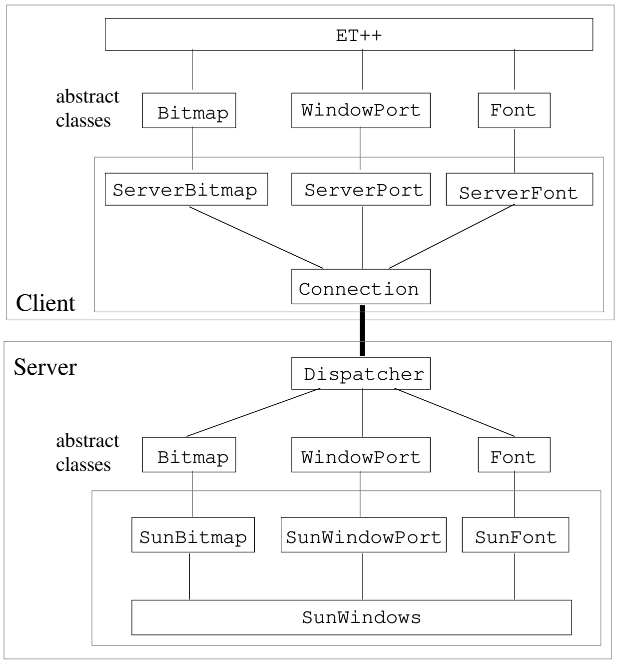


Figure 16. Structure of a Server Based Implementation of ET++.

Before porting ET++ to a standard server-based window system like X or NeWS, the authors wanted to prove the usefulness of the ET++ system interface approach by implementing a simple server-based version of the SunWindow device interface. The basic idea was to split the functionality of the SunWindow classes into a client and a server process (Figure 16).

In the server classes (`ServerPort`, `ServerBitmap`, `ServerFont` etc.) about 35 methods were simply overridden with a remote procedure call interface. The classes `Connection` and `Dispatcher` are abstractions for a *tcp/ip* stream connection between a client and a server process. The implementation (about 1000 lines) showed that ET++ applications can very well run on server-based systems with acceptable performance. After having finished the ET++ port to X and NeWS, it was interesting to note that the first prototype of a server-based implementation is still faster than both the X and NeWS implementation.

Due to dynamic binding of the device dependent methods, ET++ is able to concurrently run applications on different window systems without any recompiling. It is even possible to switch the window system at run-time or open windows on different window systems at the same time.

Usage of the port classes is straightforward: ET++ maintains a current output port to which all drawing requests are automatically directed. Because updating of this current port is completely under the control of the application framework, application-transparent printing is just a matter of switching the current port to a printer port.

The design of ET++'s imaging model was influenced by two conflicting goals: it should be sophisticated enough to ease its use in ET++ applications considerably but at the same time simple and device independent enough to allow an implementation on top of various window systems with minimum effort.

A consequence of the second goal was the decision of not supporting the inherently device dependent and nonportable raster operations found in most current window systems. These operations are difficult to emulate (for example on a printing device like a PostScript printer) and their visual effect on color systems is often unpredictable or unsatisfactory (albeit the current version of ET++

does not support color). Therefore, a subset of the stencil model found in the PostScript imaging language was adopted. This model can be emulated on most window systems with little effort.

In PostScript all drawing is performed by first constructing an arbitrarily shaped path which is then used for filling and stroking with a color, or to define a clipping region. This general concept is hard to implement on top of an existing window system and, more important, it is not really necessary for typical ET++ applications. Consequently, ET++ imaging model has only some predefined paths like rectangles, round corner rectangles, ovals, arcs, and polygons which can all be filled or stroked. Lines can only be stroked, characters only filled and a clipping region is restricted to having a rectangular shape.

This imaging model is embodied in 35 primitive methods which must be implemented for any particular output device. The (stateless) interface to these primitives is cumbersome to use because they all take their drawing attributes as parameters. A second (stateful) interface has been added which maintains attributes such as fill and stroke 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. But, in the spirit of narrow inheritance interfaces, all these alternative interfaces are based on the primitives defined in the abstract `Port` and thus do not enlarge the device interface.

A problem with the implementation of this simple and abstract device interface is that it is sometimes difficult to use functions provided by an underlying window system suitable for optimizing some special cases. Passing every single character through the device interface and the clipping machinery of the window system just to get displayed, for example, is inefficient. This is why most window systems provide special functions for drawing more than one character at a time (a batch) in order to internally use one single optimized operation to clip and display the entire batch. Because such optimizations are inherently window system dependent and cumbersome to use, ET++ only provides methods to output a single character or a string, but implements an abstract mechanism to collect all characters in a window system dependent data structure which automatically is flushed by ET++ at appropriate times.

Due to the low level nature of the ET++ output primitives some information about the abstract high-level structure of a sequence of graphic primitives is lost. With this information at hand some drivers would be able to optimize their internal behaviour. ET++ defines a primitive for specifying additional information to a driver. This information is of advisory nature only and can be safely ignored by the driver. There is also no need for an application to provide any information. Most calls to this primitive come in pairs, bracketing a sequence of graphic requests. Examples are the *double buffering* mechanism of the `Window` class and the *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. Consequently, additional information about synchronization points within a line is given to the PostScript driver.

9 ET++ Programming Environment (ET++PE)

First experiences with ET++ have shown that providing a class library with rich functionality considerably increases the learning time for novice users to fully exploit this functionality. For this reason the design and implementation of a programming environment including browsers for ET++ was initiated. Browsers and browser-like tools make inheritance more accessible and easy to use. ET++PE includes a source code browser to access class definitions and their structure as well as an inspector to view object structures at run-time. Every ET++ application has these tools automatically built in; they execute in the application's process. The browser or inspector can be invoked either at startup time or at any time while the application is running by pressing a special key.

9.1 The Inspector

The state of two objects can be viewed concurrently in the two panes of the inspector (Figure 17). After startup the left pane displays the values of the application's global variables. Pointer variables indicate their hexadecimal value and the dynamic type of the pointer. Clicking on a pointer variable dereferences the pointer and shows the corresponding object in the other pane of the inspector. There is a special item to retract to the previously inspected object. The name of the classes from which the variables are inherited are indicated together with the values of the instance variables.

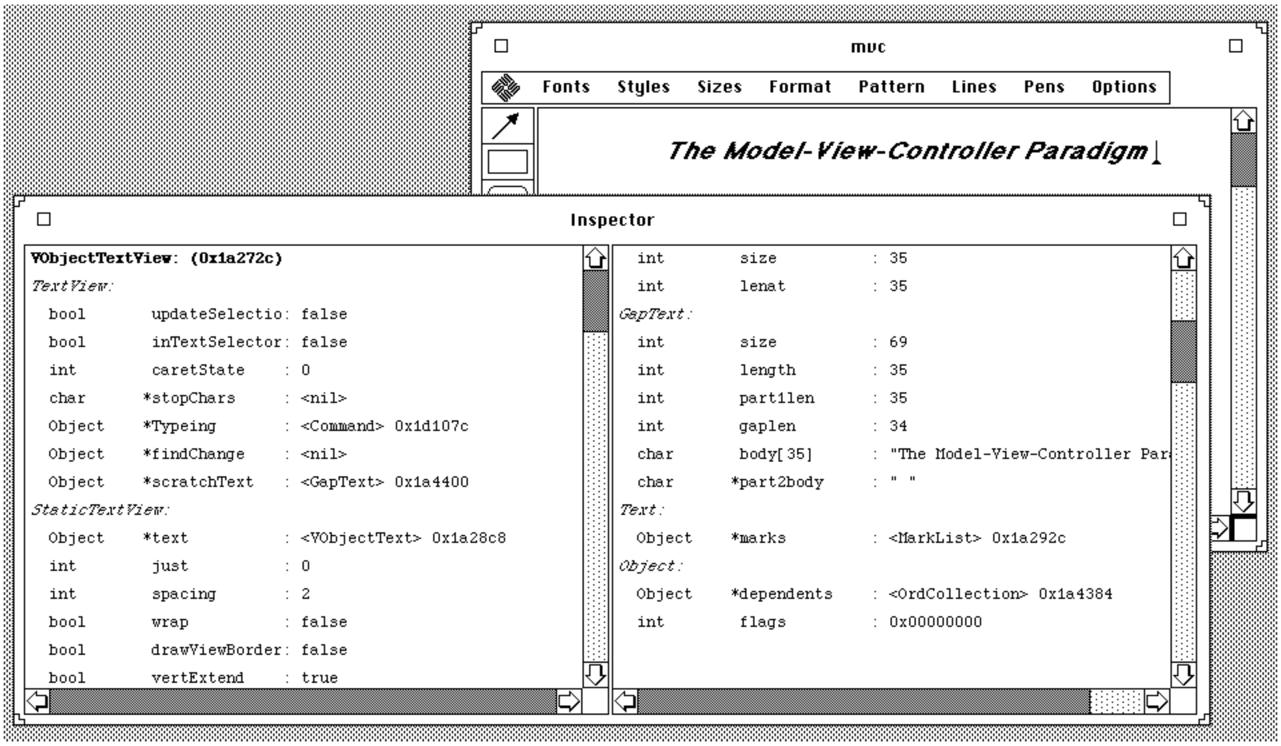


Figure 17. The Inspector.

The inspector always provides instant access to the source code. A menu allows the selection of the classes' definition or implementation to be displayed in the Source Code Browser (see below).

Navigation through a chain of pointers starting at some global variable for inspecting an object can be cumbersome. For this reason, a so called *inspect-click* has been implemented. An inspect-click is just a mouse click together with some modifier keys. It allows clicking on any visible object on the screen for inspecting it. This feature is especially helpful for students to “learn by example” about the implementation of a certain part of an application. To understand how the *find/change* dialog in a text editor is implemented, e.g., the user does an inspect-click over the dialog box. Once in the inspector, the user has access to the actual values of instance variables and to the corresponding source code.

9.2 The Source Code Browser

Figure 18 shows a screen dump of the Source Code Browser. Its left pane displays an alphabetically sorted list of the classes known in an application. The right pane provides a full-fledged text editor showing the pretty-printed source code in different font styles. Built into the text editor are also some browsing features to edit the super class or the class in the current selection.

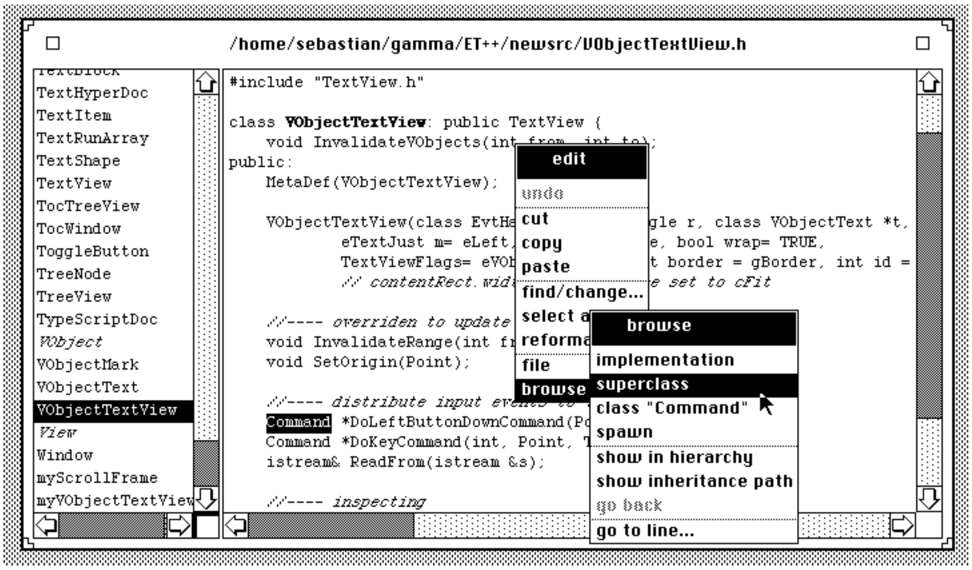


Figure 18. The Source Code Browser.

An alphabetically sorted list does not reveal the structure of a class hierarchy. This is even a problem in the standard system browser for Smalltalk-80 [Gol84], where no graphical display of the subclass relationship but only a narrow view of the class hierarchy is available. For this reason, the Source Code Browser optionally provides a tree representation showing the inheritance relationship between the classes (Figure 19). Clicking on a class name in the tree loads the corresponding source code into the editor.

The hierarchy viewer is based on the class `TreeView` and therefore allows collapsing and expanding a subtree. Notice that ET++ is based on a version of the C++ compiler not including multiple-inheritance, therefore a simple tree suffices. The layout algorithms for graph structures to display, for example, the structure of a multiple inheritance class hierarchy have been explored in another ET++ based project [Sch89]. An alternative way to display the class structure is a *flat inheritance view*. For each class this representation shows the associated inheritance path in a tabular form (Figure 19). The flat inheritance view allows quick exploration to determine where some behaviour is inherited from.

In an application framework the abstract classes are particularly important. That is why they are highlighted with an italic styled font in all representations.

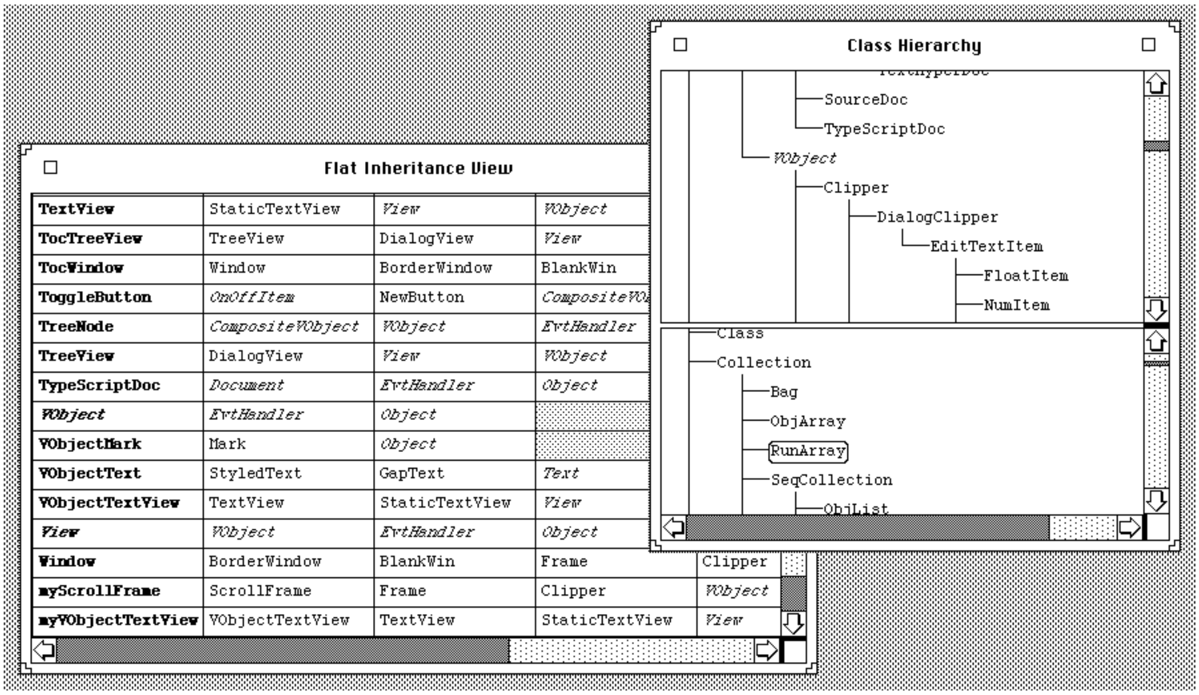


Figure 19. The Hierarchy Viewer.

9.3 Beyond Browsing

Browsing in a class library is one way to come to understand an object-oriented system. But novice users of an object-oriented application framework need some additional support to see how the different classes fit together. In order to reduce the high learning effort of an object-oriented application framework, it became desirable to have a *cookbook* for ET++ which has some hypertext facilities. The cookbook should include a collection of recipes illustrating the usage of the framework together with a rich collection of full applications.

To implement a first prototype of such a cookbook for ET++, a hypertext layer similar to Intermedia [Yan88] was added to the existing ET++ classes. The basic functionality of this prototype consists of editors for textual descriptions, for graphics, and for source code, allowing the creation of links between these types of documents. Links from a textual description to the corresponding piece of source code in an application are especially helpful for a cookbook.

In order to be able to experiment with a sample application referenced in a descriptive text, so called *action links* were introduced. When traversed, an action link does not display another document but rather invokes an application as a separate process. This application can then be further explored with the help of inspect clicks as described before.

To give some support to the hierarchical organization of a cookbook, a special hierarchical link type was defined. Based on these links a table of contents similar to the hierarchy viewer can be generated and displayed in an ET++ `TreeView` .

Manually linking all the references of classes in a text to their source code would be a nuisance. Therefore, the cookbook has an implicit referencing facility: after selecting a class name in a text, the user can invoke a menu command to display the code of the class.

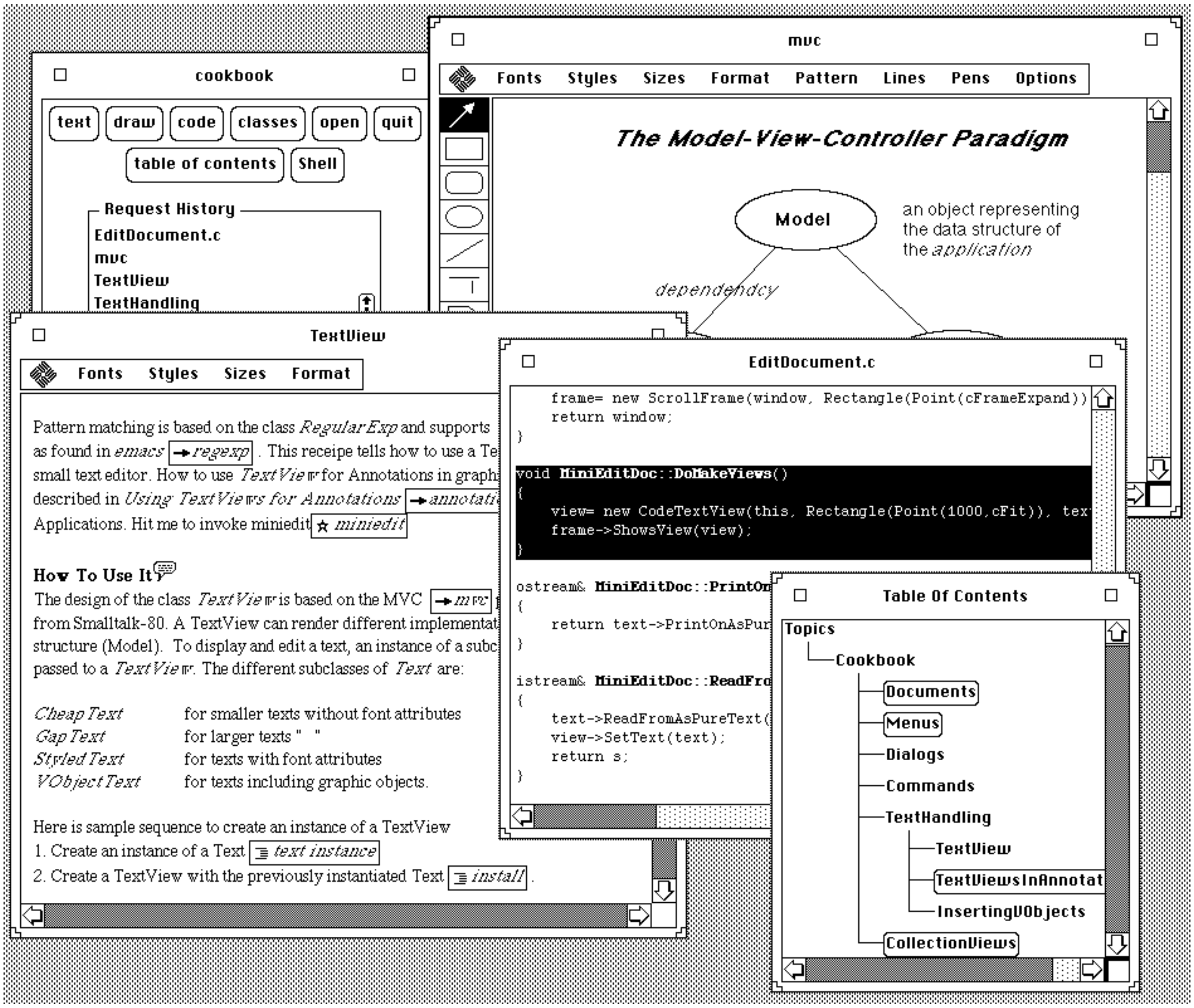


Figure 20. The ET++ Cookbook.

Figure 20 gives an example of a session with the ET++ cookbook. The links in the text are marked with a button consisting of an icon indicating the type of the link together with a descriptive name. Clicking on a link button follows the link. In Figure 20 the user first followed a link describing the MVC-design graphically and then further followed a link from a textual description to a piece of code of a simple text editor showing how an instance of a `TextView` is created. The link button marked with a star icon represents an action link. When activated it invokes the example application miniedit. The application window titled “Cookbook” shows the list of documents already visited.

10 Metrics

The current implementation of ET++ consists of 234 classes with 2343 methods. Only a subset of these classes and methods has to be known for using ET++.

	classes	methods	lines of code
Basic Building Blocks	60	604	11157
Appl. Framework	16	228	3792
Graphic Build. Blocks	103	908	14632
Prog. Environment	9	75	1472
System Dependent Parts			
Abstract Interface	16	273	4270
SunOS	5	38	1435
Postscript	1	17	586
NeWS	6	51	575
SunWindows	6	35	3284
X11.3	6	57	1300
Server	6	56	991
Total	234	2342	43494

Figure 21. Source Code Statistics.

Around 30-40 classes are used to develop a typical application. The ET++ class library comprises 43494 lines of C++ and 1200 lines of PostScript (Figure 21). Figure 22 shows the distribution of the ET++ classes (excluding the classes modelling the system dependencies) over the different levels of the hierarchy.

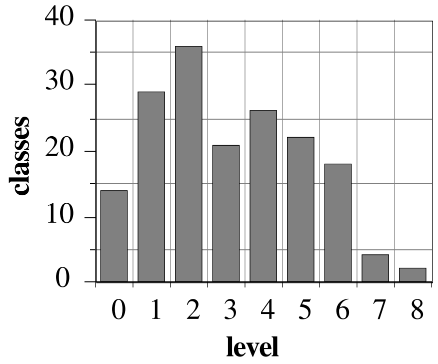


Figure 22. Distribution of Classes in the Hierarchy.

(Level 0 contains the class `Object` and other classes not derived from `Object`)

11 Experience

This section describes the experience gained while working with ET++ and its implementation language C++.

11.1 Experience with ET++

During the two years of ET++ development, several applications evolved which were used to test the functionality of ET++. The most interesting application of ET++ is certainly the ET++ programming environment (ET++PE) described in section 9 . The implementation of the ET++PE user interface illustrates the high reusability of the ET++ classes. All of the inspector's user interface components, the source code browser, and the hierarchy viewer had already been part of the ET++ class library or were implemented as very simple subclasses thereof. The views showing the instance variables, the sorted list of all classes in the system and the flat inheritance view are just instances of the `CollectionView`. The source code editor is an instance of the `CodeTextView`. The view for a tree display is an instance of the class `TreeView`. Due to the high reusability of the ET++ classes, the final version of ET++PE was finished after an effort of two person months.

An object-oriented application framework transfers a lot of control flow from client code into the class library. This characteristic allows the

addition of new functionality without any modifications in existing applications and with little programming effort. Especially when a system is well factored, a large amount of leverage is available. The implementation of the inspect-click (see section 9.1) is a good example: implementing this mechanism only required the addition of a few lines of code to the existing event distribution mechanism centralized in the class `VObject` (the root of all graphical classes). The new code filters out the special key combination and calls the `Inspect` method of the underlying object. This code is automatically inherited by all applications and implies no programming effort at all.

A well factored system with no code redundancies offers a lot of benefits during the tuning of a final release. Each small improvement in efficiency disperses throughout the entire system. By following the design rule of declaring instance variables types as high in the hierarchy as possible, it was very easy to substitute a class with another more adequate one. The data structure underlying the `Com-positeVObjects`, for example, was an `ObjList` in a first version. Based on the fact that these clusters are accessed much more than modified, the `ObjList` (a pointer-based implementation of a list) was replaced with an array-based version provided by the class `Ordered-Collection`. This change required only the modification of the source line where a new instance of a collection is created. Since the layout mechanism of `CompositeVObjects` is used extensively in ET++ this small modification improved the overall performance of ET++, applications.

Using ET++ in student projects revealed that they need a substantial amount of training until they come up with good reusable classes. But how to teach and train students to design reusable classes remains an open issue.

From a project management point of view it seems that the small ET++ project team (two computer scientists) was a big help in reducing code redundancies and toward the construction of a homogeneous system.

The evolution of the classes `CollectionView` and `Menu` illustrates a very interesting example of the "Promotion of Structure" principle 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 into the menu class, but after second thoughts the linked list was replaced 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 ET++Draw. The next logical step was to factor out this mechanism into a class called `CollectionView` .

Simultaneously, the layout algorithm for showing a `Collection` was extended from a vertical-only to a two-dimensional style. With this extension the `CollectionView` became one of the most reusable parts of ET++. It is now used not only to build menus but also for Macintosh-like menu bars, for scrollable lists of arbitrary items, and for tool palettes. All that had to be done to achieve this was adding or changing a few lines of code.

In yet another step the `CollectionView` was modified to work with the general `VObject` instead of a more specialized `MenuItem`. This opened the road for implementing the graphics part of a simple spreadsheet application with each cell containing a full-fledged text editor.

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` .

11.2 Experience with C++

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 easy-to-use but still flexible method interfaces. *Operator overloading* was very useful for implementing the two classes `Point` and `Rectangle` and helped to simplify the code considerably and thereby also improved its readability.

C++ lacks support for garbage collection, which was sometimes a burden during the development of ET++. A general garbage collector, on the other hand, would impact the efficiency of C++ programs. Due to the single-rooted hierarchy structure of ET++, the class `Object` was instrumented to gather some statistics about memory allocation. These statistics were indispensable to find memory leaks. Some experiments have been made with the conservative garbage collector described in [Boe88]. This garbage collector has not been designed specificly for C++ and therefore does not call the destructors of an object when it is freed. For this reason further investigations in the conservative garbage collector have been relinquished; the garbage collector is only invoked after an application runs out of memory. For the allocation of small objects like iterators, ET++ provides a class for implementing class-specific allocators. Objects of these classes are allocated and freed in chunks and thereby reduce the overhead of free store management dramatically. Measurements have shown that in a typical run of an application only 10% of all dynamic objects are allocated on the heap, the other 90% by class specific allocators.

The major benefit of C++ is its strong static type checking. Especially while reorganizing the class hierarchy, the strong static type checking of C++ proved to be very valuable and detected inconsistencies at compile time.

Working experience with C++ has shown that having information about the classes and their structure available at run-time is not just convenient but even required. The approach of building additional descriptor objects manually or with the help of some tricky preprocessor macros is not very elegant. All of the information collected in such a metaclass object is in fact a subset of the C++ translator's compile time information (e.g. its symbol table). As a logical consequence it would be very easy to automatically generate the necessary structures containing the meta-information. The difficult part is to agree upon what information is really needed. But without a consensus every library builder will most likely invent some new tricks and programming conventions making the exchange of classes between libraries much more difficult.

Executable modules developed with ET++ are quite large (> 1 MByte). The high reusability of classes results in almost every class statically depending on a large number of other classes although most of their methods are never called in a given application. Building a library with every method as a separate library component is not a decent solution because of the way the AT&T C++ translator handles dynamically bound functions (an array of function pointers). This creates many external references and outperforms even smart linkers.

A better solution is dynamic demand loading and linking of classes into running applications. Using this approach results in very small ET++ applications (5k – 100k). A dynamic linker for a C++ environment has to provide a hook that allows the calling of the constructors of the object file's static objects after a class has been linked to the application. The programming interface of the dynamic linker of SunOs 4.0 [Gin87], however, lacks such a hook and thus cannot be used for the dynamic linking of ET++ classes. For this reason dynamic linking support for ET++ had to be reimplemented.

ET++ was developed with a C++ version not supporting multiple inheritance. The current class hierarchy of ET++ is easy to understand and the authors doubt whether ET++ would have the same clear class structure had multiple inheritance been used.

12 Overview of Other Class Libraries

This section gives a brief overview of other class libraries and compares them with ET++. The feature list in Figure 23 can be considered as the union of the functionality provided in the class libraries for conventionally compiled languages.

Class Library	**MacApp II**	**Atk**	**Xt**	**Inter views**	**Eiffel-Library**	**OOPS**	**ET++**
Implementation language	ObjectPascal C++	Class	C	C++	Eiffel	C++	C++
Operating System	MacOS	Unix	Unix	Unix	Unix	Unix	Unix
Window System	Macintosh	X	X	X	X	N/A	SunWindow, X, NeWS
Basic Data Structures	+	–	–	–	–	++	++
Object Input/Output	–	+	–	++	++	++	++
Application Framework Classes	++	+	–	–	–	N/A	++
User Interface Building Blocks	++	++	++	++	++	N/A	++
Layout of Composite Objects	–	–	++	++	–	N/A	++
Dynamic Linking of Classes	–	++	–	–	–	–	++
Abstract Device Interface	++	–	–	–	–	N/A	++
Browsers, Inspectors	+	–	–	–	++	N/A	++

– no support
+ basic support
++ support
N/A not applicable

Figure 23. Overview of Related Class Libraries.

12.1 The Andrew Toolkit (Atk)

Atk [Neu88], developed at the Information Technology Center (ITC) of Carnegy Mellon University, is implemented with a custom object-oriented environment called *Class*. Class is a preprocessor for C which supports object-oriented programming and integrates dynamic linking of classes. Atk was originally built on a custom window system and has been ported to X. The architecture of Atk applications is influenced by the Smalltalk-80 MVC paradigm. Every toolkit component consists of a data object (model) and view (view+controller) pair synchronized by a change propagation mechanism inherited from a common superclass (`Observer`). The motivation for the view/data object distinction is that a data object can be displayed simultaneously in several views. In ET++ the functionality to show different parts of the **same** view is provided generically by the framework based on the class `Clipper`. Only when different views of the same model are displayed does the programmer have to use change propagation to synchronize them: an example of this is a tabular (numeric) and a pie chart view of a list of numbers.

The Atk class library includes a text building block with the capability to integrate view/data object pairs like a spreadsheet into the text. In this respect, it is similar to ET++'s `VObject` `Texts`.

Looking at the code of some Atk based applications reveals that a lot of common behaviour of these applications has not been factored out into the class library. To support scrolling or auto-scrolling, for example, Atk provides only a scrollbar class but the scrolling behaviour has to be reimplemented in every application. In the same sense, support for mouse tracking is insufficient resulting in code redundancies in all applications. Atk provides no support for undoable commands, and indeed all Atk based applications provide no undo facility.

Object input/output is handled by defining a standard ASCII format for objects but no support for linearizing circular pointer structures is provided. In contrast to ET++, the structure of the Atk class hierarchy is flat; just some subclasses of the text classes can be found below level 3.

12.2 X-Toolkit (Xt)

Xt [McC88] for the X window system provides a higher level programming interface to the underlying X window system. It is based on the abstraction of a *widget*. A widget can be considered as an abstract class from which specific user interface components can be derived. Every widget is based on one X window. Widgets were originally designed to be a lightweight abstraction. Considering their internal structure, it is questionable whether the implementation of a spreadsheet, for example, could use a widget for each cell without substantial performance penalty.

Instances of widgets are buttons, text editor windows, forms, or scrollbars. Widgets are organized in a hierarchy and the layout of the children of a widget is controlled by composite widgets. The functionality provided by widgets can be compared with the ET++ classes `VObject` and `CompositeVObject`. Xt is coded in pure C and based on conventions (known as the *Xt Intrinsics*) to get an object-oriented flavor. The lack of a real object-oriented programming language is the reason that deriving a new class from an existing widget is rather complicated; all of these conventions have to be taken care of.

12.3 The Object-oriented programming support library (OOPS)

OOPS [Gor87] was not designed to provide specific support for graphical applications. It implements most of the collection classes known from Smalltalk and similar to ET++ but without robust iterators. Included is an object input/output facility that is less transparent concerning the storing of pointers than ET++. We were not able to base ET++ on OOPS because both projects started at about the same time.

12.4 The Eiffel library

The Eiffel library [Mey88] covers general data structures like search tables or trees. The implementation of these classes does not rely on dynamic type checking but uses genericity as provided by the Eiffel language. Object input/output of any class can be achieved by deriving from the class `STORABLE` (Eiffel supports multiple inheritance). The graphic classes of the Eiffel library are implemented for the X window system and provide abstractions for windows, menus, or simple graphical shapes like polygons or circles. Higher level application framework components are missing. In contrast to ET++, the Eiffel library lacks an abstract device interface supporting transparent output to a window or to a PostScript printer.

12.5 MacApp (I+II)

The architecture of MacApp I [Sch86] provided the base for ET++. MacApp II [Bia88] is a major redesign of MacApp I with the goal of unifying its graphic model and easing the construction of view hierarchies. In a parallel development effort, the ET++ graphics model was extended in the same direction. In both MacApp II and ET++ all graphic objects are derived from a common root class. The major difference is that the root class of ET++ is more lightweight and, contrary to MacApp II, it does not include clipping. This has the advantage of ET++ `VObjects` having very little space and performance overhead. MacApp II provides an interactive tool to layout groups of graphical objects and is not based on declarative lay out specifications like in ET++.

A major drawback of MacApp is its hybrid structure. MacApp is only a thin layer on top of the nonobject-oriented Macintosh toolbox. The implementation of an application, as a result, always requires some intimate knowledge of both MacApp and the Macintosh toolbox. Experience shows that especially user interface components like menus profit extensively from the polymorphism offered by object-oriented languages.

12.6 Interviews (Interactive Views)

Interviews [Lin87] is a user interface toolkit written in C++ running on the X window system. The design of Interviews is centered around a class `Interactor` defining the behaviour of all interactive objects. Interviews includes classes supporting a simplified version of the TEX [Knu86] *box and glue* model to lay out interactors side by side. Interviews provides typical user interface components like buttons, scrollers and text buffers as found in other toolkits. Interviews is quite a rich class library consisting of around 130 classes. The class library has been kept intentionally shallow (most classes are at level 2 or 3) based on the experience that many levels of subclasses overwhelm programmers and because there are no integrated browser and inspector tools. The imaging model of Interviews is similar to that of ET++, but other major application framework components are missing.

13 Concluding Remarks

A first version of ET++ including the programming environment without the cookbook is distributed in the public domain. The current release of ET++ runs under SunWindow, X11.3, and NeWS. A port to Apple's A/UX™ is in progress.

References

[App88] Apple, *Human Interface Guidelines: The Apple Desktop Interface*, Addison-Wesley, Reading, MA, 1988. (1)

[Bia88] C. Bianchi and D. Goldsmith, *MacApp 2.0 Display Specification*, Apple Computer, Inc., Cupertino, CA, 1988. (1, 12.5, 1, 12.5)

[Boe88] H. Boehm and M. Weiser, “Garbage Collection in an Uncooperative Environment,” *Software—Practice and Experience*, Vol. 18, No. 9, September 1988, pp. 807-820. (11.2)

[Cox86] B. J. Cox, *Object Oriented Programming – An Evolutionary Approach*, Addison-Wesley, Reading, MA, 1986. (3)

[Gam89] E. Gamma, A. Weinand, and R. Marty, “Integration of a Programming Environment into ET++ – A Case Study,” In *ECOOP 89, Proc. of the Third European Conference on Object-Oriented Programming, (Nottingham, UK)*, S. Cook, ed. Cambridge University Press, Cambridge, 1989, pp. 283-297. (5.5)

[Gin87] R. A. Gingell et al., “Shared Libraries in SunOS,” In *USENIX Association Conference Proceedings (Atlanta, Georgia, June 9-13)*, USENIX Assoc., El Cerrito, CA, 1987, pp. 131-145. (11.2)

[Gol83] A. Goldberg and D. Robson, *Smalltalk-80, The Language and its Implementation*, Addison-Wesley, Reading, MA, 1983. (5.1)

[Gol84] A. Goldberg, *Smalltalk-80, The Interactive Programming Environment*, Addison-Wesley, Reading, MA, 1984. (9.2)

[Gor87] K. E. Gorlen, “An Object-Oriented Class Library for C++ Programs,” *Software—Practice and Experience*, Vol. 17, No. 12, December 1987, pp. 899-922. (5.1, 12.3, 5.1, 12.3)

[Han87] W. J. Hansen, “Data Structures in a Bit-Mapped Text Editor,” *Byte Magazine*, Vol. 12, No. 1, January 1987, pp. 183-189. (7.2)

[Ing83] D. H. H. Ingalls, “The Evolution of the Smalltalk Virtual Machine,” In *Smalltalk-80, Bits of History, Words of Advice*, G. Krasner, ed. Addison-Wesley, Reading, MA, 1983. (3)

[Joh88] R. E. Johnson and B. Foote, “Designing Reusable Classes,” *The Journal Of Object-Oriented Programming*, Vol. 1, No. 2, 1988, pp. 22-35. (1)

[Ker75] B. W. Kernighan and L. L. Cherry, “A System for Typesetting Mathematics,” *Comm. Assoc. Comp. Mach.*, Vol. 18, March 1975, pp. 151-157 (May 1974, revised April 1977). (7)

[Ker81] B. W. Kernighan, *PIC — A Crude Graphics Language for Typesetting*, Bell Laboratories, Murray Hill, New Jersey, 1981. (8)

[Knu86] D. E. Knuth, *The Texbook*, Addison-Wesley, Reading, MA, 1986. (12.6)

[Kra81] G. Krasner, “The Smalltalk–80 Virtual Machine,” *Byte Magazine*, Vol. 6, No. 8, August 1981, pp. 117-124 (reprinted in Tutorial on Object-Oriented Computing, Vol. 2 Implementations, IEEE Computer Society Press, 1987, pp. 117-124). (3)

[Lea88] D. Lea, “libg++, The GNU C++ Library,” In *USENIX Proceedings C++ Conference (Denver, CO)*, USENIX Assoc., El Cerrito, CA, 1988, pp. 243-257. (3)

[Lie85] H. Lieberman, “There's More to Menu Systems Than Meets the Screen,” *ACM Computer Graphics (San Francisco, July)*, Vol. 19, No. 3, 1985, pp. 181-189. (6)

[Lin87] M. A. Linton and P. R. Calder, “The Design and Implementation of InterViews,” In *USENIX Proceedings and Additional Papers C++ Workshop (Santa Fe, NM, 1987)*, USENIX Assoc., El Cerrito, CA, 1987, pp. 256-268. (12.6)

[Lis86] B. Liskov and J. Guttag, *Abstraction and Specification in Program Development*, McGraw-Hill, New York, 1986. (5.4)

[McC88] J. McCormack and P. Asente, “Using the X Toolkit,” In *Proc. USENIX '88*, 1988. (12.2)

[Mey88] B. Meyer, *Object–Oriented Software Construction*, Prentice Hall, Englewood-Cliffs, New Jersey, 1988. (3, 12.4, 3, 12.4)

[Mye87] B. A. Myers, “Gaining General Acceptance for UIMSs,” *ACM Computer Graphics*, Vol. 21, No. 2, April 1987, pp. 130-134. (1)

[Neu88] C. M. Neuwirth and A. Ogura, *The Andrew System – Programmer's Guide to the Andrew Toolkit (Volume I: Theory and Examples)*, ITC Carnegie Mellon University, Pittsburgh, 1988. (12.1)

[Pal88] A. J. Palay et al., “The Andrew Toolkit – An Overview,” In *USENIX Tech. Conf. Winter 1988 (Sunset Beach, CA, September)*, USENIX Assoc., El Cerrito, CA, 1988, pp. 9-21. (1)

[Rao87] R. Rao and S. Wallace, “The X Toolkit: The Standard Toolkit for X Version 11,” In *USENIX Association Conference Proceedings (Atlanta, Georgia, June 9-13)*, USENIX Assoc., El Cerrito, CA, 1987, pp. 117-129. (1)

[Ros86] L. Rosenstein, K. Doyle, and S. Wallace, “Object–Oriented Programming for Macintosh Applications,” In *ACM Fall Joint Computer Science Conference (Dallas, Texas, November 2-6)*, 1986, pp. 31-35. (1)

[Sch86] K. J. Schmucker, *Object Oriented Programming for the Macintosh*, Hayden, Hasbrouck Heights, New Jersey, 1986. (1, 6, 7.2, 12.5, 1, 6, 7.2, 12.5)

[Sch89] D. Schmidt, *Das TOPOS-Component Management System (to be published)*, Institut für Informatik der Universität Zürich, Zürich, 1989. (9.2)

[Ste86] M. Stefik and D. G. Bobrow, “Object-Oriented Programming: Themes and Variations,” *The AI Magazine*, Vol. 6, No. 4, Winter 1986, pp. 40-62 (reprinted in Tutorial on Object-Oriented Computing, Vol. 1 Concepts, IEEE Computer Society Press, 1987, pp. 182-204). (3, 11.1, 3, 11.1)

[Str87] B. Stroustrup, “Possible Directions for C++,” In *USENIX Proceedings and Additional Papers C++ Workshop (Santa Fe, NM, 1987)*, USENIX Assoc., El Cerrito, CA, 1987, pp. 399-416. (5.1)

[Wei88] A. Weinand, E. Gamma, and R. Marty, “ET++ – An Object Oriented Application Framework in C++,” In *OOPSLA'88 Conference Proceedings (September 25-30, San Diego, CA)*, published as *Special Issue of SIGPLAN Notices*, Vol. 23, No. 11, November 1988, pp. 168-182. (5.6)

[Wil84] G. Williams, “Software Frameworks,” *Byte Magazine*, Vol. 9, No. 13, December 1984, pp. 124-127, 394-410. (1)

[Yan88] N. Yankelovich et al., “Intermedia: The Concept and the Construction of a Seamless Information Environment,” *IEEE Computer*, Vol. 21, No. 1, January 1988, pp. 81-96. (9.3)