ET++ – a Portable, Homogenous Class Library and Application Framework

André Weinand, Erich Gamma

Taligent, Inc.[1]
10201 N. De Anza Blvd
Cupertino, CA 95014-2233
Andre_Weinand@taligent.com
Erich_Gamma@taligent.com

ET++ is a portable and homogenous object-oriented class library integrating user interface building blocks, basic data structures, and high level application framework components. ET++ eases the building of highly interactive applications with consistent user interfaces following the direct manipulation principle. The ET++ class library is implemented in C++ and can be used on several operating systems and window system platforms. Since its initial conception the class library has been continuously redesigned and improved. It started with an architecture which was close to MacApp. During several iterations a new and unique architecture evolved. A byproduct of the ET++ project is a set of tools, which were designed to support the exploration of ET++ applications at run-time.

1 From Toolbox to Application Framework[2]

Making computers easier to use is one of the reasons for the 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. It is not uncommon that up to 50-80% of the code are devoted to user interface aspects.

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.

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 considerable overall complexity. It is not easily possible to upgrade their functionality or to add new components without modifying or duplicating source code.

---

1 Work performed while at UBILAB, Union Bank of Switzerland, Zurich, Switzerland

2 Published in Procs. of UBILAB Cnference '94, Universitätsverlag Konstanz, 1994

Modern toolbox implementations like the *Motif toolkit* for the X window system [Ber91] use object-oriented programming techniques to improve flexibility and extensibility by dynamic binding and inheritance.

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, implementing the callbacks invoked by the toolkit. The major drawback of the toolbox approach is that it does not define an overall structure for an application. In other words it provides architectural guidance for how to structure the application. In addition toolboxes can only ensure user interface consistency at the level of individual user interface widgets and not at the application level. A partial solution is to provide developers with a program skeleton which 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) framework in general and application frameworks in particular. An application framework defines much of an application's standard user interface, behavior, 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, MacApp [Sch86, Ros86] for the Macintosh user interface, and *Unidraw* an application framework for structured graphics editors [Vli89].

An application framework allows reusing the abstract design of an entire application, modeling 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" guidelines are by prescription rather than implementation.

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.

2 An Example of an ET++ Application

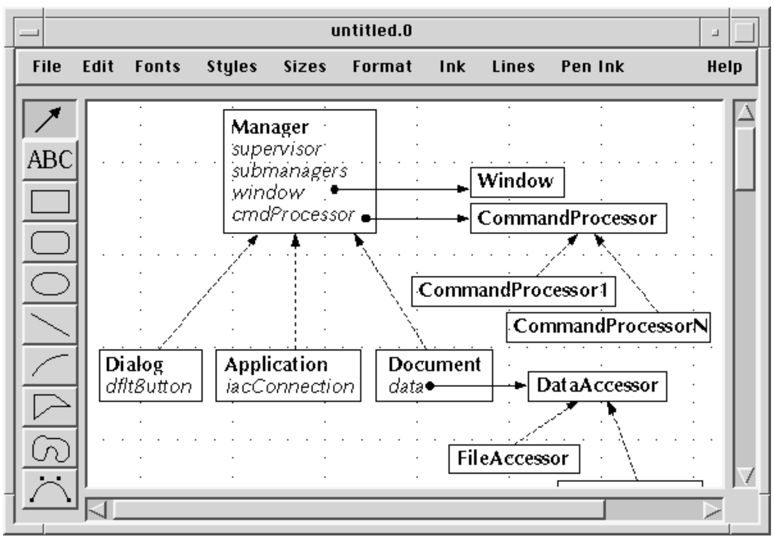


Figure 1. ET++Draw an Example of an ET++ Application.

Figure 1 shows a screen dump of ET++Draw to give an idea of what kind of applications ET++ supports. ET++Draw is a classical drawing program. The following list highlights some tasks ET++ takes care of without any special effort by the programmer when building applications such as ET++Draw:

- editing of several drawings in several 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,
- incremental flicker-free screen update based on double buffering, and
- device independent hard copy output of the drawing, for example in PostScript. The printing of the drawing can always be previewed on the screen.
- the application runs on all window systems supported by ET++.
- the appearance or the look of user interface widgets like scrollbars and buttons can be switched dynamically at run-time. Currently a Motif and a Macintosh inspired ET++ look are supported.

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.),
- a powerful text building block which supports editing of rich text, that is, text with different character and paragraph attributes,
- 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 (multi level),
- a notification mechanism for maintaining dependencies among objects. ET++Draw uses this mechanism to implement the visual connections between shapes. A connection depends on the positions of the shapes it connects,
- 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),
- import and export of graphics, images, and text in standard exchange formats (e.g. TIFF, PICT, PBM, RTF) is facilitated by a converter framework,
- automatic layout of a group of graphical objects, for example in dialog boxes.

3 Architectural Overview

The backbone of the ET++ architecture is a layered class hierarchy with about 300 classes (Figure 2).

ET++ is organized as a “mostly” single rooted class library, that is, most of the classes descend from a common base class `Object`. While this organization is not very popular in C++ it has proven to be well suited for our needs. It enabled the implementation of some valuable infrastructure inherited by all descendants of `Object`. For example, run-time access to information about the classes of an application (meta information) could be smoothly integrated into the class library by defining the corresponding infrastructure at the root class.

The *Toolkit* layer contains the most important low level building blocks of the ET++ class hierarchy: the class `Object`, the root of the overall class hierarchy, implements the common behaviour for all ET++ classes. *Data Structures* are general purpose classes, like arrays, lists, sets etc. which are used heavily in the implementation of ET++ itself. The *View System* contains a small number of graphical building blocks which implement the common behavior of all graphical classes and defines a framework to easily build new components from existing ones. The *User Interface Toolkit* contain all the graphical and interactive components found in almost every user interface toolbox, such as menus, dialogs, or scrollbars.

The *Framework* layer contains a small number of high level frameworks. The *Application Framework* 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 other frameworks provide high level and extensible building blocks for rich text, tables and spreadsheets, and trees and graphs.

On top of this core system are more domain specific frameworks. These are part of the ET++ distribution but not topic of this paper.

The *ET++ Exploration Environment* is a set of exploration tools which are automatically included in every application. The tools allow inspecting the running application to help the developer understand the program.

The *Environment Abstraction Layer* provides its own hierarchy of abstract classes for operating system services, inter application communication, window management, input handling, and drawing on various devices. Subclasses exist for implementing the system interface layer's functionality for various native window and operating systems.

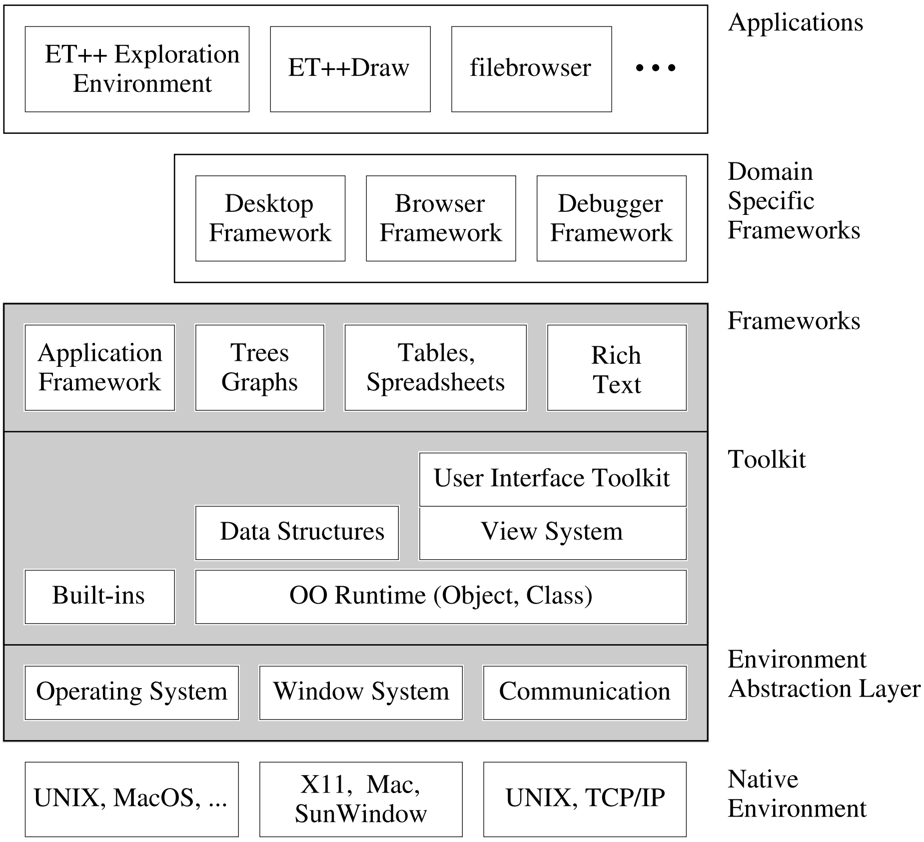


Figure 2. ET++ Architecture.

4 The Toolkit

4.1 Built-in Classes

ET++ is not strictly single rooted and not all class library descend from `Object`. Classes which are used like built-in types do not descend from `Objects`. Examples are the geometry classes `Point` and `Rectangle` or the class `String`. These classes are not used in a polymorphic way and are treated as built-in types.

4.2 The Class Object

Most classes of ET++ are derived from the class `Object`. `Object` defines protocols (abstract methods) for comparing objects, for notification between objects, and for object input/output to name just the most important ones.

The object input/output facility of ET++ supports the transfer of arbitrarily complex polymorphic data structures from memory to an ET++ stream and vice versa. This functionality is based on the abstract methods `PrintOn` and `ReadFrom`, which are overridden in subclasses to stream 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 and multiple references to the same object are restored properly. Storing pointers is implemented by a map per stream which assigns a unique identifier to each transmitted instance. This identifier can be transferred to other address spaces or to permanent storage.[3]

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`. 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. Another example is 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 loading and 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 ET++ allowed the implementation of a generic `DeepClone` method for objects in the class `Object`:[4] The stream classes 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 arbitrary data structures to other ET++ applications via the clipboard. 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.

Another general mechanism provided by `Object` is *change notification.* 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

---

3 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.

4 A deep clone of an object consists of its instance variables plus a deep clone of all objects it references.

of completely independent objects. Notification is modeled after Smalltalk-80's support for dependencies [Gol83]. `AddObserver` is the method for 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 `DoObserve` method for all dependent objects. To react to a change notification, this `DoObserve` method has to be overridden. The default implementation of notification in `Object` uses a space efficient global dictionary to store the `Object`'s observers. Subclasses are free to provide implementations with better performance.

Notification has proved to be a very useful mechanism and is not only used in MVC-designs. ET++Draw, for example, uses notification to maintain visible connections between arbitrary graphical elements.

4.3 Portable Run-Time Support for C++

Even with the upcoming *run-time type information* standard for C++ (RTTI) the 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).

Metaclasses store the following information about a class:

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

The types of the instance variables and the source code reference were introduced in order to give support for the ET++ exploration environment.

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.

4.4 Container Classes

The basic building blocks of ET++ include abstract data types often referred to as *container classes*. Included are `OrdCollection` (dynamically growing arrays), `ObjList` (linked lists), `Set` (hash tables), and `Dictionary` to name a few. They are more or less modeled after the *Smalltalk collection classes*.

```
Object
   Container
      Collection
         SeqCollection
            ObjArray
            ObjList
               SortedObjList
            OrdCollection
         Set
      Dictionary
```

Figure 3. The Container Classes.

The container classes all deal with any instance derived from the class `Object` which excludes more specific type checking at compile time. Recent implementations of C++ support parameterized types (templates) as an alternative approach allowing more type checking at compile time. But because the support for templates and maturity varies between different compilers the authors are still reluctant to use templates.

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 (`Container`). Kofler [Kof89] is a detailed description of the design and implementation of robust container classes for ET++.

4.5 The View System

`VObject` (visual object) is the most general graphical class in ET++. It defines protocol for drawing graphical objects on the screen, for input event handling and distribution, and for managing their size and position.

A design goal for `VObjects` was to keep them small and lightweight. This means that `VObjects` have very little memory or performance overhead even when using thousands of them (for example as leafs and nodes in a large tree view). Consequently, `VObjects` have no built-in coordinate transformation and establish no clipping boundary. Our experience shows that this is an asset rather 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 gain no benefit from having their own coordinate system. A more detailed discussion of these design decisions is [Wei92].

The relatively large 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, a so called *bottleneck interface*. 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 bottleneck 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.

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 `CompVObject` 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 `CompVObject`, `VObjects` are most easily arranged in a tree-like fashion which allows a reasonable default implementation of all event handling methods: the event is dispatched downwards and the calls to handle an event are by default propagated up to its container.

The class `Clipper` 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 4). Applications for this will be given later in the general discussion of the graphic building blocks.

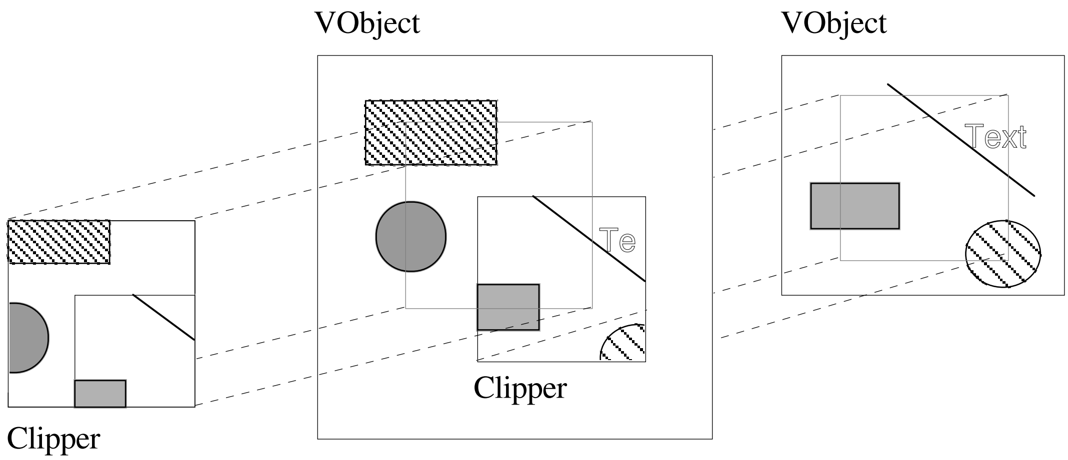


Figure 4. A Hierarchy of Nested Clippers.

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++ view system makes use of an indirect drawing scheme (*invalidation*) which makes it completely unnecessary to call the `Draw` method of a `VObject` directly because all control flow is factored out into the framework. An ET++ application simply announces which `VObjects` (or part thereof) must be redrawn by calling a method which adds the region occupied by the `VObject` to a single update region per window. Whenever ET++ is idle, it requests the application to redraw the update region by calling the `Draw` method of the top most `VObject` (e.g. a view installed in a window). Because invalidation is cheaper than redrawing, 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 view system 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 screen flicker by sophisticated incremental update strategies.

This indirect synchronous drawing scheme works very well with most classical application types but has its limitations when dealing with *animation* or *digital motion video* because it is not feasible to invalidate and redraw regions of the screen at real-time video rates. However[Sch93] shows how support for motion video can be integrated into ET++ with very little modification.

User Interface Elements

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 behavior 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 behavior 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 or mixins 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 by hand. This led to the integration of some automatic 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. 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 behavior (e.g. editing operations) is typically not needed. Figure 5 shows some of the user interface components of the class hierarchy.

```
VObject
   TextItem
   ImageItem
   PictureItem
   Clipper
      Window
      Zoomer
   CompVObject
      Slider
      Matte
      BorderItem
      Box
         Expander
         Form
         OneOfGroup
         ManyOfGroup
      Button
         ActionButton
         RadioButton
         ToggleButton
```

Figure 5. User Interface Components.

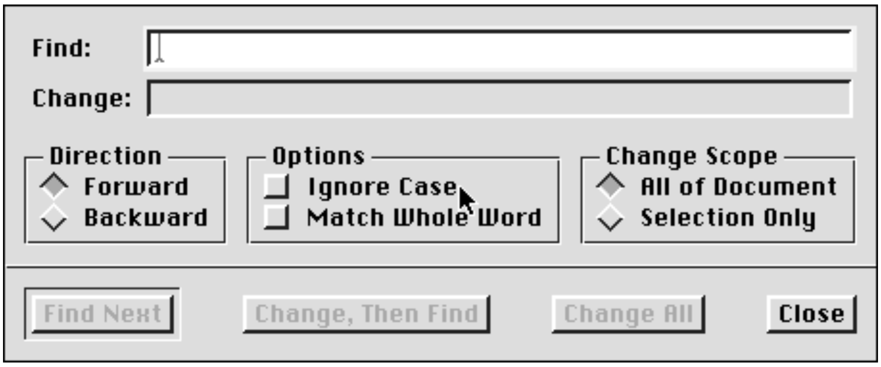
The layout operators as subclasses of `CompVObject` are responsible for controlling both the communication among their components and the relationships among the locations of these components, i.e. the layout management.

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`.

A `Box` 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 `Box` item can be aligned horizontally as well as vertically in a number of ways (left, right, center, top, bottom, base). The `Box` implements a very powerful mechanism which fits the needs of most complex dialog layouts without having to position items explicitly (Figure 6).

The `OneOfGroup` (`ManyOfGroup`) is a subclass of `Box` that implements the one-of (many-of) behavior of several on/off buttons.

Any single item in the statement of Figure 6 could be replaced by an arbitrarily complex composite item. Because this generality is not always needed, convenience constructors exist for often-used dialog patterns. This allows a much simpler description (Figure 7).




```
new VBox(
  new Form(
     new TextItem("Find:"), new TextField,
     new TextItem("Change:"), new TextField,
  0),
  new HBox(
     new BorderItem("Direction",
       new OneOfBox(eLeft,
          new HBox(eBase,
             new RadioButton,
             new TextItem("Forward"),
          0),
          new HBox(eBase,
             new RadioButton,
             new TextItem("Backward"),
          0),
       0)
     ),
     new BorderItem("Options",
       new ManyOfBox(eLeft,
          new HBox(eBase,
             new ToggleButton,
             new TextItem("Ignore Case"),
     // ...
  0),
  new LineItem,
  new HBox(
     new ActionButton("Find Next"),
     new ActionButton("Change, Then Find"),
  // ...
  0)
0);
```

Figure 6. Part of a Dialog Box and its Defining Statement.

```
new VBox(
    new Form(
      "Find:",  new TextField,
      "Change:",  new TextField,
    0),
    new HBox(eCenter,
      new BorderItem("Direction",
        new OneOfBox("Forward", "Backward", 0)),
      new BorderItem("Options",
        new ManyOfBox("Ignore Case", "Match Whole Word", 0)),
    // ...
0);
```

Figure 7. Building the Dialog using Convenience Constructors.

5 Application Framework

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 of much of the control flow into the class library, while the latter requires that the developer know exactly when to call which method.

5.1 Managing Presentations

The ET++ Application Framework centers around a hierarchy of so called *presentations.* A presentation is a collection of (mostly) visual components, like menubars and menus, dialogs, views and windows that make up one aspect of an application. Every presentation is managed by an instance of class `Manager` or a subclass thereof.

An application consists of a hierarchy of `Managers`: the top-level `Manager` is an instance of the specialized subclass `Application`, which controls the application as a whole and manages any number of sub-`Managers`.

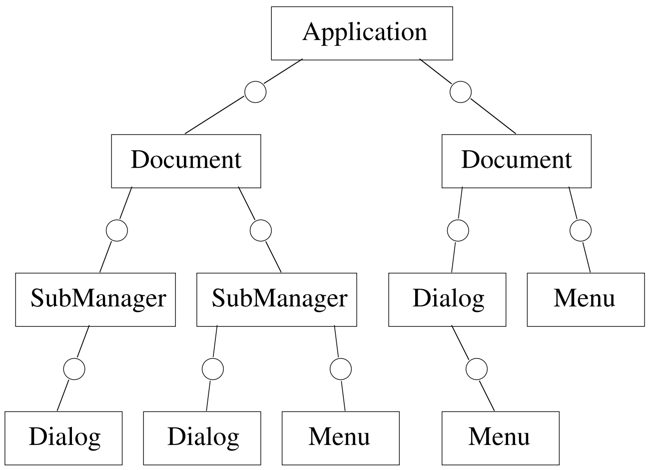


Figure 8. A Typical Manager Instance Hierarchy.

Figure 8 shows an example (snapshot) of the dynamic relationship between instances of `Managers` and their subclasses. The small circles denote methods (*factory methods*) clients have to override in order to create specific sub-`Managers`. After their creation ET++ takes care of them.

The most important subclass of `Manager` is `Document`. In addition to its responsibilities as a `Manager` a `Document` encapsulates the data structure (the model) of an application and collaborates with another abstraction (`Data`, see below) in order to load and later store and close the model.

The implementation of loading and closing documents is a 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.

If a presentation is not document based but, for example, a data base front-end or a simple control panel or dialog, its sufficient to just create a subclass of `Manager`. ET++ dialogs and even menus are subclasses of `Manager`.

A Dialog implements a standard behavior for modal or modeless dialog boxes. A single method must be overridden to create the dialog's VObject tree, another to set up the initial state, and a last method to react to all dialog interactions.

```
Manager        // manages a presentation
  Application  // manages an application as a whole
  Document     // manages model based presentation
  Dialog       // manages dialog box presentation
  Menu         // a presentation of a one-of-many selection
```

Figure 9. Manager Classes.

Managers are a good example of a simple yet powerful abstraction that is applied in different areas and greatly reduces the number of concepts a developer has to learn.

5.2 Data and Converter Framework

The abstract class Data defines an interface for transparently accessing and converting external or remote data, like files, components embedded in documents or data bases, or the clipboard. Clients typically use subclasses of Data to extract an object that will be used as a model. The Data abstraction tries to hide details like source and original format of the external data by always providing an object that is convenient for the client to work with. A text editor for example will ask a Data object to convert the data source to a subclass of Text, a graphics editor will try to get a Picture. After the client is done with the object, the object can be converted back to its original form and saved in its original place.

The most important subclass of Data is FileData, a data source connected to a file stream of the underlying file system. Another subclass is ObjectData which wraps a given Object. It is used for implementing the ET++ clipboard and drag-and-drop mechanism.

The implementation of the Data uses the *converter framework* of ET++. This framework first uses TypeMatchers to determine the type of an external data source. It then tries to find a Converter object from the set of available Converters which can convert the external data to a requested class type. If more than one conversion sequence is possible a selection dialog is presented to the user.

The converter framework can be easily extended by clients with new TypeMatchers and Converters. Because existing and new applications work on Data only, they automatically profit from any new converters by being able to exchange and import data formats previously unknown to them. There are converters provided for *Rich Text* (RTF), Macintosh PICT2, TIFF, *Encapsulated PostScript* (EPSF), *X11 Bitmap Format* (XPM), and *Portable Bitmap Format* (PBM).

5.3 Viewing

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.

In addition a View adds the ability to be visible in multiple Clippers, that is different portions of the same View can be shown in different places on the screen. This powerful concept is used in ET++ for the implementation of the classes Splitter and PageView. A Splitter shows disconnected portions of an underlying `View` in several panes. A `PageView` breaks an underlying continuous `View` into separate pages and displays each of them with border and optional header and footer as a matrix of pages. Because this functionality is totally transparent to the original `View`, `Splitter` and `PageView` can be used with any existing and future `View` subclass.

5.4 Undoable Commands

An 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 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.

The abstract class `Command` defines the protocol while subclasses of the abstract class `CommandProcessor` implement the control flow for executing and undoing a command. Simple non-model based `Manager` like `Dialogs` have an `SingleCommandProcessor` which allows a single level undo. Documents typically have an `Unbounded-CommandProcessor` which keeps all command objects since the last load or save operation.

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.

6 High Level Building Blocks

In addition to the basic user interface elements described above ET++ provides higher level building blocks for textual, tabular and hierarchical data.

6.1 Rich Text Building Block

A text building block has to support a spectrum of different applications. At the lower end are text entry fields in dialog boxes. A drawing editor uses editable text for annotations. A program editor needs efficient text for editing and displaying pretty-printed source code. Finally at the higher end are on-line documents which require support for multiple character and paragraph styles with embedded graphics and hypertext links.

To cover this entire spectrum the text building is not implemented as a monolithic object but as a separate framework. The key abstractions of this framework and their responsibilities are:

- `Text` maintains the text data structure and sends out a notification whenever the text changes.
- `TextView` renders the text, handles input, and change notifications from its `Text` object. Maintains the text composition, that is how the text is broken into lines. The responsibility for finding the breaks is delegated to a `TextFormatter` object. The rendering of a single text line is delegated to a `TextPainter` object.
- `TextFormatter` finds the line breaks in a paragraph.
- `TextPainter` renders a single line of text and is responsible for handling typographic effects like justification and ligatures.

Each abstraction is implemented by a family of classes. To create an editable text object the client selects a `TextView` class and combines it with a `Text`, a `TextFormatter` and a `TextPainter` object. This approach enables the creation of editable text objects with different functionality by combining different text objects in different ways. `TextView` acts as the controller of an editable text object and mediates between the other objects.

Figure 10 illustrates the hierarchy of `TextView` classes. `StaticTextView` renders text while `TextView` also handles input an implements 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`. A `ShellTextView` is connected to a shell and supports to enter commands and to log their outputs.

```
VObject
   View
      StaticTextView
         TextView
            CodeTextView
            ShellTextView
```

Figure 10. The `TextView` Hierarchy.

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 4.5. Due to double buffering the screen is updated flicker-free, even for text displayed against arbitrary backgrounds.

The separation of the text storage from the `TextView` and the use of notification to communicate changes enables to view the same `Text` object in multiple `TextViews`.

Classes for managing the data structures of a text are descendants of an abstract class `Text` (Figure 11.)

```
Object
   Text
      CheapText
      GapText
         StyledText
            VObjectText
```

Figure 11. The `Text` Hierarchy.

Text defines a bottleneck interface for its subclass. Editing operations like Cut, Copy, Paste are implemented in terms of a `ReplaceRange` primitive. Subclasses only have to implement this method. Efficient access of the text contents is achieved by iterators which retrieve a sub-sequence of text character by character, word by word, or paragraph by paragraph.

`CheapText` is the simplest implementation of a text data structure and is primarily used for testing purposes. 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]. The class `StyledText`, supports the assignment of different character and paragraph attributes to a text range.

Another subclass of `Text` is `VObjectText`. `VObjectText` enables to embed arbitrary objects into a text which behave as ordinary characters. The protocol for such character like objects is defined by the class `VisualMark`. It defines the interface for measuring and rendering character like objects. An example subclass is `VObjectMark` which implements the `VisualMark` interface for `VObjects`. It enables the embedding of arbitrary `VObjects` into text. Figure 12 shows the ET++Write application. It uses `VisualMarks` to embed pictures and the running clock.

A `TextFormatter` is responsible for finding the line breaks in a text. The line breaking functionality is factored into its own object to enable the use of different line breaking strategies. ET++ provides two implementations, `SimpleTextFormatter` and `FoldingTextFormatter`. `SimpleTextFormatter` breaks the text only at explicit line breaks whereas `FoldingTextFormatter` breaks the text into lines of a given width.

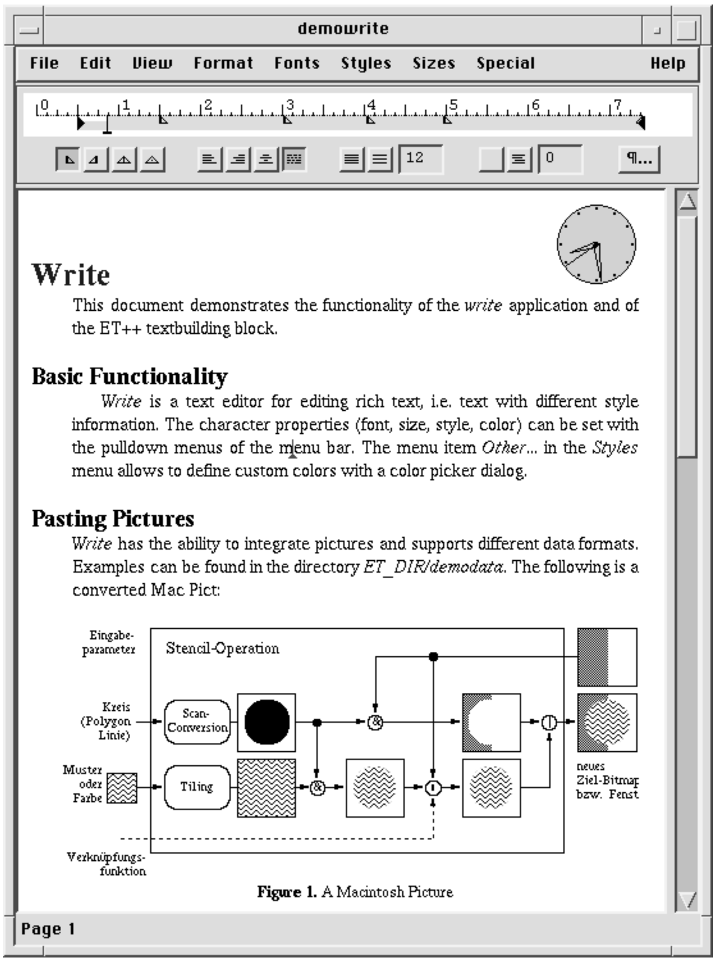


Figure 12. ET++Write.

`VisualMarks` turned out to be a powerful extension mechanism for text, they are not limited to embedding pictures. In ET++Write `VisualMark` subclasses were used to implement:

- *markers*, zero width characters with a symbolic name, they are used to define *sticky* targets for hypertext links,
- *hypertext links*, the link target is referenced as document name, marker name pair,
- *action links*, display a little icon and store a shell script which is executed when the user clicks on the icon,
- *annotations*, shown as an icon and when clicked pop-up a dialog for entering an annotation,
- *counters* for headings and figures, this counters look like ordinary characters but they are objects with the additional behavior to update a counter.

Another example is the *FileBrowser* application which uses `VisualMarks` to enable the inclusion of pictures as comments in source code.

The above architecture has proven to cover a wide range of application needs. For example, ET++Write uses a custom `TextView` subclass with a `VObjectText` and a `FoldingFormatter`. Simpler text-entry fields use the standard `TextView` class with a `GapText` and a `SimpleTextFormatter`. More specialized text entry fields which need to support data validation are covered by a family of `TextField` classes. Multiple `TextField` classes share the same `TextView` and thereby reduce the memory requirements.

In addition to the core text classes described above the text building block includes several `Command` classes for manipulating text in an undoable way.

The text building block leverages the converter architecture for supporting import and export of text in different formats, like the RTF (rich text format) interchange format.

6.2 Grid Views for Tabular Data

Another important class derived from `View` is the `GridView`. A `GridView` is a flexible framework for displaying cells in a matrix and forwarding input to them. It also takes care of selecting and deselecting single cells as well as contiguous and non-contiguous areas of cells. The `GridView` is a basic building block for many user interface objects which have to present a collection of selectable cells.

A `GridView` makes no assumptions about the underlying data structure of a cell but clients are required to override at least a `DrawCell(`column, row, rectangle`)` method to draw a cell's content. The fact that a cell is not an object appears to be not very object oriented but our experience has shown that in many cases its much easier to implement a single method by accessing an existing data structure than to convert the data structure to a class `GridView` requires. A benevolent side-effect is that not each element in a table has to be allocated as object, which can result in a major performance improvement for larger tables.

A typical example from the ET++ exploration environment is a scrolling list containing the names of all available classes. The ET++ run time system already maintains a collection of meta classes. Converting them to `TextItems` (a subclass of `VObject`) would be much more cumbersome than to override `DrawCell` and accessing the `Name` method of the meta class.

`StringGridView` is a subclass of `GridView` for manipulating a matrix of cells containing `Strings`. Again `StringGridView` makes no assumption about where the `Strings` are stored but clients have to implement a `GetString(`column, row`)` and a `SetString(`column, row`)` method. In the same way as the `TextFields` described above `StringGridView` maintains a single instance of a `TextView` for implementing a memory efficient floating string editor. Whenever keyboard focus is transferred to a cell (e.g. by clicking into it) the cell is overlaid with the `TextView` which in turn allows editing of the cells contents.

Another specialized `GridView` is the `CollectionView`, which displays any collection of `VObjects` as provided by the foundation classes in a tabular format. `CollectionViews` are used for displaying tables with heterogeneous contents which require some polymorphic cell behavior (Figure 13).

```
View
   GridView
      CollectionView
      ObjectGridView
      StringGridView
```

Figure 13. The `GridView` Hierarchy.

6.3 Tree Views for Hierarchical Data

The `TreeView` renders a tree data structure (in fact a `VObject` tree) and implements simple editing operations, for example moving nodes and collapsing/expanding subtrees. Different layouts are supported. Each layout is represented as a separate object which can be plugged into a `TreeView` (Figure 14). The `TreeView` class itself can be easily adapted to show any kind of hierarchical data. To do so a client overrides the `MakeChildrenIterator` and the `NodeAsVObject` methods. `MakeChildrenIterator` is used to traverse the hierarchy. `NodeAsVObject` defines the visual representation of a node in the hierarchy.

`GraphView` is a subclass of `TreeView`. It adds support for the layout of acyclic graph structures. It uses a simple layout algorithm which builds on top of `TreeView`. `GraphView` supports that the user can interactively rearrange a generated layout.

```
Object
   TreeLayout
      TopDownLayout
      LeftRightLayout
      IndentedLayout
      CollapsedLayout
      OptimizedLeftRightLayout
```

Figure 14. The `TreeLayout` Hierarchy.

7 Object-Oriented Modeling of System Dependencies

Portability was a major issue in the design of ET++. In order to be independent of a specific environment, all environment 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:

- window management and input handling,
- graphic system, graphic resource management (fonts, images, cursors),
- inter application communication,
- operating system services and resources.

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 operating 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 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.

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

```
Port
   PicturePort
   PrinterPort
      MacPictPort
      PostScriptPort
      // ...
   WindowPort
      XWindowPort
      SunWindowPort
      MacWindowPort
      // ...
```

Figure 15. Port Hierarchy.

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 structured graphics.

A `PrinterPort` adds abstract methods for dealing with an abstract printing device. Its subclasses `MacPictPort` and `PostScriptPort` are implementations for generating Macintosh PICT2 files 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 subclass `XWindowPort` is an implementation of `WindowPort` for X11, `SunWindowPort` is for SunWindows, Sun's now obsolete library-based window system; and the `MacWindowPort` for the Macintosh. Due to dynamic binding of the device dependent methods, ET++ is able to run applications on different window systems.

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 non portable 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. Therefore, a subset of the stencil/paint 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, pattern, or more complex graphics. ET++ uses a subset of this general concept.

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, 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.

8 Run-time Object Inspection[5]

The run-time structure of an ET++ application bears little resemblance to its static code structure. The static code structure is frozen at compile-time and it consists of fixed inheritance relationships. The run-time structure consists of a network of communicating objects. It is difficult to understand the dynamic structure based on the static class hierarchy. Understanding the dynamic structure is even more complicated in ET++ applications since the heavy use of object-composition.

To address this problem we investigated in exploration tools to support the analysis of the dynamic object structure of an ET++ application. Every ET++ application has these tools optionally built in; they execute in the same address space as the application.

8.1 Inspect Clicking

When exploring an application at run-time the first problem is how the user can define the focus of interest. In other words there has to be convenient entry point into the object network. A graphical application can profit from the fact that a lot of interesting objects are visible on the screen. We therefore added a so called *inspect click* feature. 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. An inspect click helps to answer the question “what is this object on the screen” from an implementation point of view.

Figure 16 shows the dialog displayed after inspect clicking an object in an color picker dialog box.

The list on the left shows the path up the visual hierarchy starting at the clicked object and ending at the window object of the dialog. An object is identified by its class name and an optional identifier which defaults to the object’s address. Enumerating the chain of parent objects enables to easily navigate to an object’s parent. Clicking the Inspect button brings up the Inspector on the selected object.

---

5 ET++ also includes some browsers for the static analyisis of an application, that is the source code and the class hierarchy. However, since this functionality is currently subsumed by Sniff (see below) we focus on the run-time object inspection support.

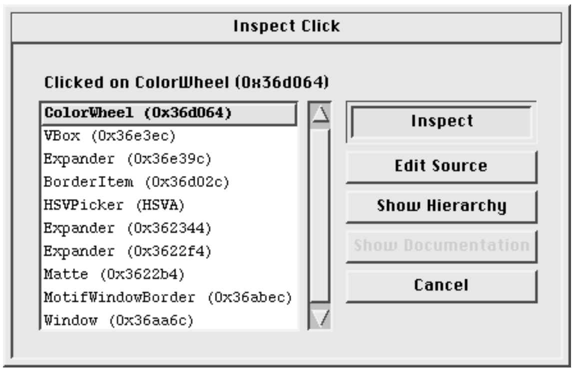


Figure 16. The Inspect Click Dialog.

8.2 The Inspector

The inspector has five panes (Figure 17). The top left pane shows the list of the application classes. The number in parenthesis shows a snapshot of the number of allocated instances. Selecting a class name of this pane fills the middle pane with the instances of the class. An object is identified by its class name and an optional identifier which defaults to its address.

Selecting an object shows the object's state in the bottom pane. This pane shows the instance variables including the ones inherited from base classes. Pointer variables indicate their static type. If the dynamic type differs from the static type it is shown as well (see for example, the `container` instance variable in Figure 17). Clicking on a pointer variable dereferences the pointer and shows the corresponding object. The buttons in between the top and the bottom panes support to navigate back and forth along the path of visited objects.

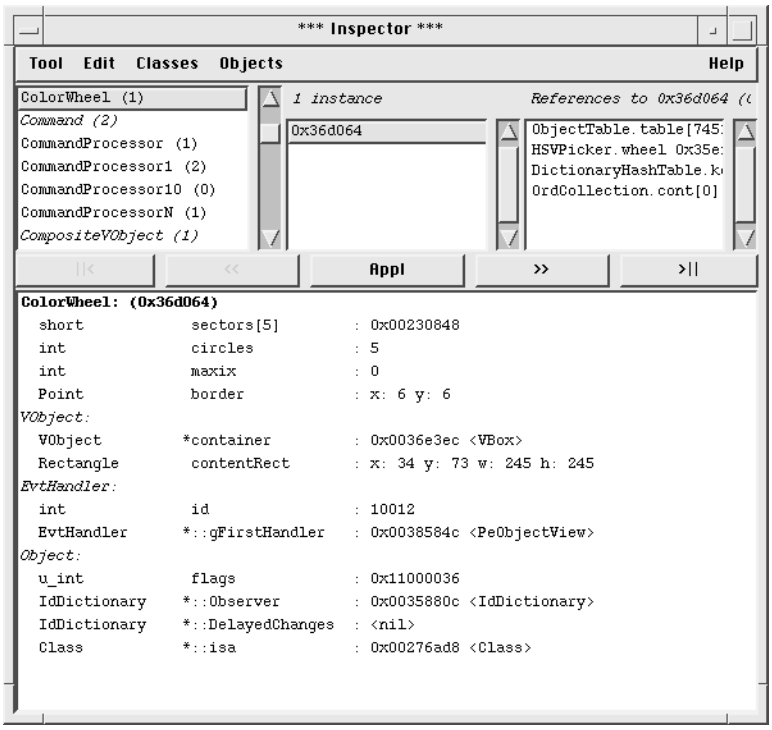


Figure 17. The Inspector.

Another interesting feature of the inspector is the possibility to query for other objects having a reference to the currently inspected one. The result of such a query is shown in the top right pane. Selecting an object from this list shows its state in the bottom pane.

The Inspector is a helpful tool for navigating back and forth in the object network. However it can only give a keyhole view since it can only show one object at a time. To get a better understanding of the objects and their relationships the exploration environment also includes an object structure browser.

8.3 The Object Structure Browser

The *Object Structure Browser* visualizes run-time relationships between objects. It shows an object's part-of hierarchy. For example, figure 18 shows the part-of hierarchy of a color picker dialog. A node represents an object and the thin lines connect the object with its parts. The object structure browser can be invoked from the Inspector on the inspected object. Double clicking a node in the object structure browser shows the corresponding object in the Inspector.

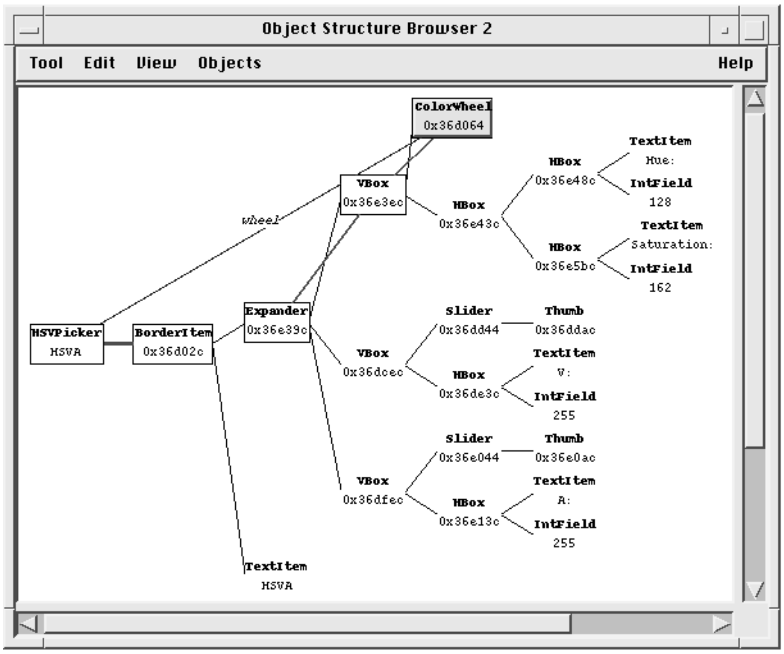


Figure 18. The Object Structure Browser.

The part-of hierarchy serves as a starting point for additional queries on object relationships. The "Points To" and "Referenced From" menu items can visualize the outgoing and incoming references of a selected object. The references are shown as additional (colored) lines between the objects. The object structure has also some built-in knowledge of the ET++ class library and it can also display event distribution paths and notification relationships.

8.4 Run-time support for the exploration environment

The exploration tools described above require some additional run-time support. The functionality to find all the instances of a class is based on an object table. This table is maintained by the root class `Object`. Its constructor adds an entry to this table and its destructor removes it. The maintenance of this table can be turned of for applications which are in production use. The object table is also used to find objects pointing to a specific object.

The determination of an object's parts requires some help from the class implementor. The root class `Object` defines a `CollectParts` method which can be overridden by subclasses to let the tools know which object references point to an object's parts. Another instrumentation method is `InspectorId` which is overridden to define an additional textual identifier for the instances of a class. This identifier is derived from the values of an object's instance variables. It is shown in both the Inspector and the Object Structure Browser.

9 ET++ Applications

A class library in general and a framework in particular has to be validated by real applications. During the development of ET++ we switched back and forth between the roles of class library developers and class library clients. The result of this is a set of samples which are distributed together with ET++. This section gives a short overview of the samples followed by two more comprehensive applications .

9.1 Samples

The samples address two different needs. First they are used by us to validate the design of the class library and second they are used by developers to learn the class library.

The samples include both simple entry-level applications and more comprehensive ones. It was an important goal of ET++ to cover the entire spectrum. During the evolution of the class library we always focused on making the applications simpler and reduce their code bulk.

The entry-level applications include:

- *Nothing*, the minimal ET++ application,
- *Hello*, a graphical Hello World program,
- *Calculator*, a simple calculator,
- *TwoShapes*, *ThreeShapes*, minimal drawing applications,
- *Miniedit*, a simple text editor.

The more advanced applications are:

- *FileBrowser*, a program editor combined with support for navigating in the directory hierarchy,
- *Spreadsheet*, a simple spreadsheet which uses the TCL interpreter [Ous94] for expression evaluation,
- *Er*, a simple diagram editor for entity-relationship diagrams,
- *IconEdit*, an icon editor which understands a large number of image formats,
- *Draw*, a drawing application,
- *Write*, a simple word processor.

9.2 Sniff

Sniff is a pragmatic C++ programming environment providing browsing, cross-referencing, design visualization, documentation, and editing support. The main goal in developing Sniff was to create an efficient and portable C++ programming environment which makes it possible to edit and browse large software systems textually and graphically with a high degree of comfort, and without using large amounts of system resources [Bis92].

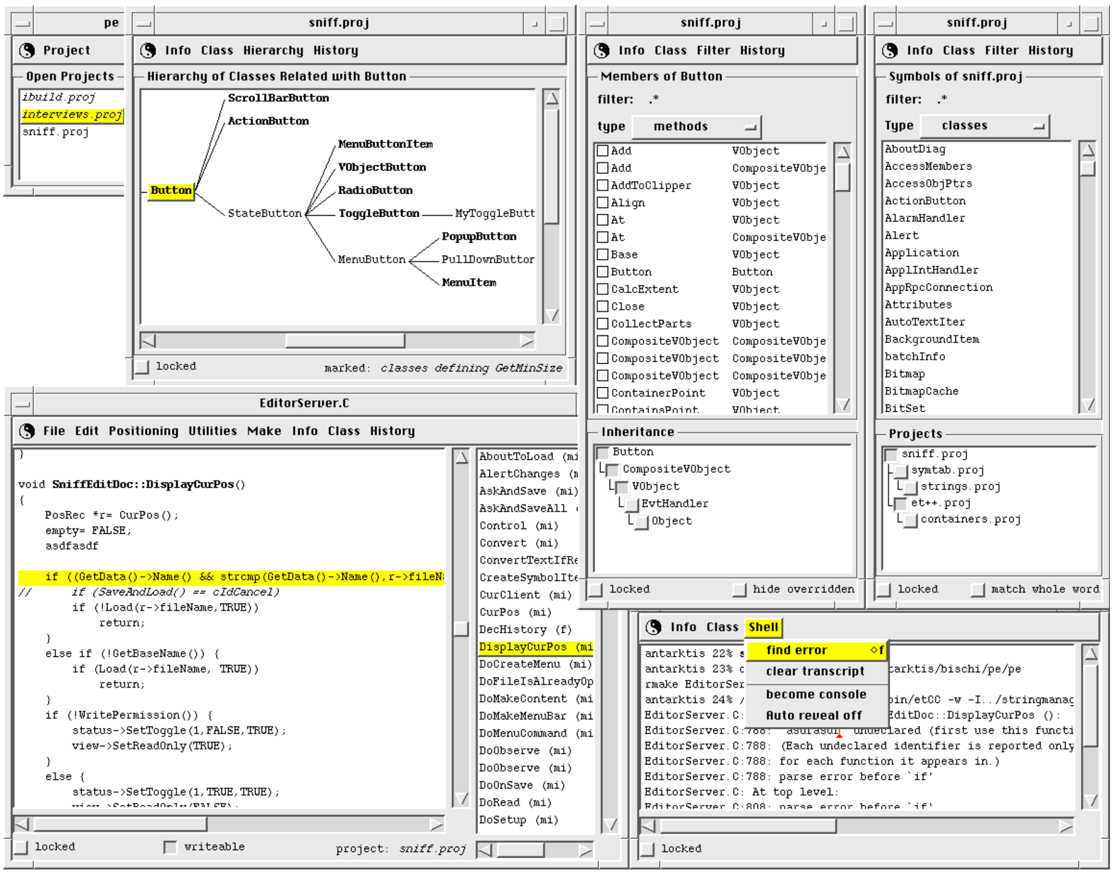


Figure 19. A Screen showing different windows of the C++ Programming Environment Sniff.

Sniff also integrates a front-end for command line debuggers such as *gdb*. The integration of etgdb was achieved by using the ET++ provided inter-application-communication services.

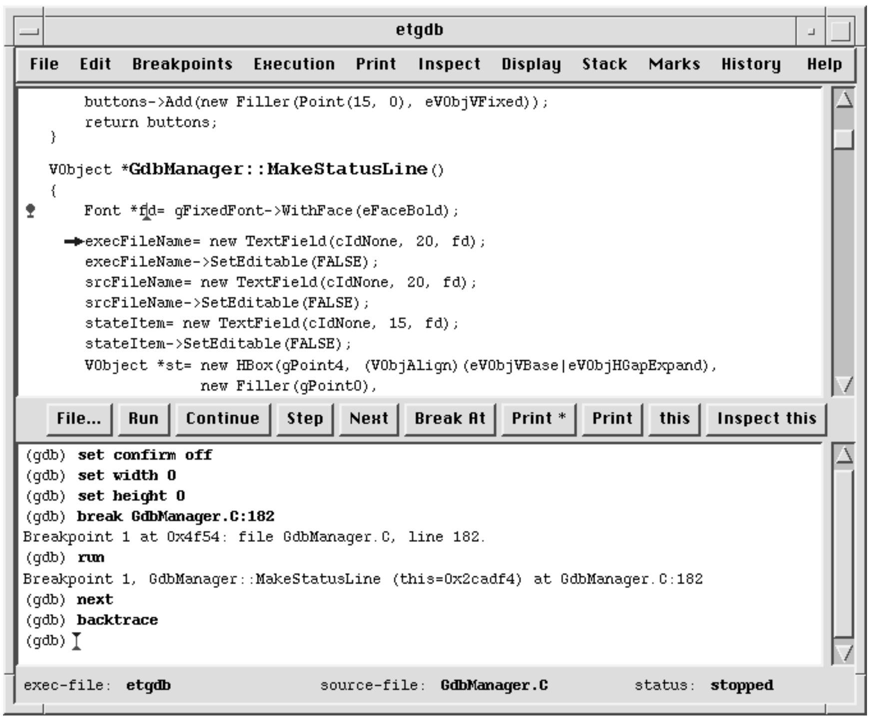


Figure 20. An ET++ Front-end for *gdb*.

9.3 SwapsManager

The SwapsManager project [Egg92] used ET++ to implement a custom application which helps a trader in pricing a financial instrument called a *swap*. The application is built around the notion of a *domain specific desktop*. In contrast to more conventional desktop environments which only present icons for documents and folders this desktop consists of iconic presentations from the trading domain. The top right window in Figure 21 illustrates the iconic representations for swaps, portfolios (collections of swaps), yield curves (yields for a set of maturities), and scenarios (possible interest changes applicable to a yield curve). The icons can be manipulated by drag and drop. For example, to add a swap to a portfolio the trader drags the scenario icon onto a yield curve.

As in common desktop environments an iconic representation can be opened and inspected in a window. The bottom right window shows a more conventional form based interface for editing the data of a swap object. The left window shows a graphical editor opened on a yield curve object. This editor provides a palette of tools for changing and inspecting the curve. Manipulating the curve creates an interest rate scenario object which shows up on the desktop. In addition to changing the curve the trader can also apply any existing scenario by dragging a scenario icon onto a yield curve icon. Each of these manipulations triggers a recalculation of all dependent data which enables the trader to easily perform a risk analysis for different scenarios.

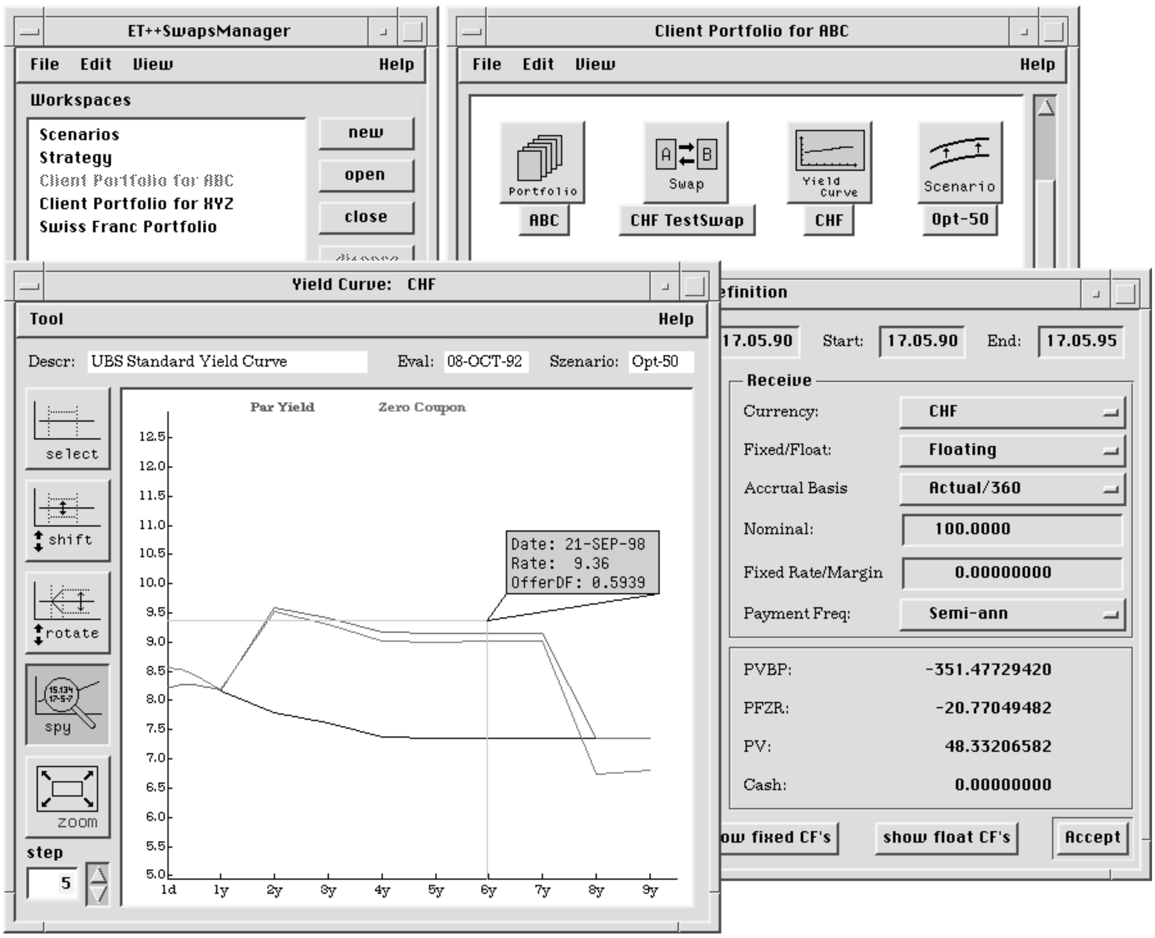


Figure 21. The swaps trading workspace of the ET++SwapsManager

A swap is just one of many financial instruments. However, each of these instruments requires a similar calculation engine to analyze the price and value. To cover more than one of these instruments a framework for building calculation engines evolved [Bir93].

10 Experience

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

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 8.1) 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.

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.

Using C++ as the implementation language of ET++ has worked 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. However, we are very conservative in using C++ language features. We avoid virtual base classes and multiple inheritance altogether. Learning a comprehensive class library is already hard enough and we don't want to increase the complexity by using all possible language features. In addition, since portability was an important constraint reducing the number of language features facilitates switching between different C++ compilers.

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.

11 Availability

ET++ 3.0 and its tools and documentation is distributed in the public domain. It is available by anonymous ftp from ftp.ubilab.ubs.ch in the directory pub/ET++. The current release of ET++ runs under X11, SunWindow, and the Macintosh.

References

[Ber91] Berlage T: *OSF/Motif™: Concepts and Programming*, Addison-Wesley, Wokingham, England, 1991.

[Bir93] Birrer A, Eggenschwiler T: Frameworks in the Financial Engineering Domain: An Experience Report, In *Proceedings ECOOP '93 (Kaiserslautern, Germany, July 26-30)*, LNCS 707, O. Nierstrasz, ed. Springer-Verlag, Berlin, 1993, pp. 21-35.

[Bis92] Bischofberger WR: Sniff—A Pragmatic Approach to a C++ Programming Environment, In *USENIX Proceedings C++ Conference (Portland, OR)*, USENIX Assoc., El Cerrito, CA, 1992, pp. 67-81.

[Egg92] Eggenschwiler T, Gamma E: ET++SwapsManager: Using Object Technology in the Financial Engineering Domain, In *OOPSLA'92 Conference Proceedings (October, Vancouver, Canada)*, published as *OOPSLA'92, Special Issue of SIGPLAN Notices*, Vol. 27, No. 10, October 1992, pp. 166-177.

[Gam89] Gamma E, Weinand A, Marty R: Integration of a Programming Environment into ET++ – A Case Study, In *Proceedings ECOOP '89 (Nottingham, UK, July 10-14)*, S. Cook, ed. Cambridge University Press, Cambridge, 1989, pp. 283-297.

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

[Han87] Hansen WJ: Data Structures in a Bit-Mapped Text Editor, *Byte Magazine*, Vol. 12, No. 1, January 1987, pp. 183-189.

[Joh88] Johnson RE, Foote B: Designing Reusable Classes, *The Journal Of Object-Oriented Programming*, Vol. 1, No. 2, 1988, pp. 22-35.

[Ker75] Kernighan BW, Cherry LL: A System for Typesetting Mathematics, *Comm. Assoc. Comp. Mach.*, Vol. 18, March 1975, pp. 151-157 (May 1974, revised April 1977).

[Kof89] Kofler T: Robust Iterators in ET++, *Structured Programming*, Vol. 14, No. 2, 1989.

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

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

[Mye87] Myers BA: Gaining General Acceptance for UIMSs, *ACM Computer Graphics*, Vol. 21, No. 2, April 1987, pp. 130-134.

[Ous94] Ousterhout JK: *Tcl and the Tk Toolkit*, Addison-Wesley Professional Programming Series, Addison-Wesley, Reading, MA., 1994.

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

[Sch86] Schmucker KJ: *Object Oriented Programming for the Macintosh*, Hayden, Hasbrouck Heights, New Jersey, 1986.

[Sch93] Schnorf P: Integrating Video into an Application Framework, In *Proceedings First ACM International Conference on Multimedia (Anaheim, CA, August 1-6)*, ACM Press, New York, NY, 1993, pp. 411-417.

[Str87] Stroustrup B: 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.

[Vli89] Vlissides JM, Linton MA: Unidraw: A Framework for Building Domain-Specific Graphical Editors, *Technical Report: CSL-TR-89-380*, Stanford University, Computer Systems Laboratory, July 1989.

[Wei92] Weinand A: Objektorientierte Architektur für grafische Benutzungsoberflächen, (in German) Springer-Verlag, Berlin 1992.