Integration of a Programming Environment into ET++ A Case Study

Erich Gamma
André Weinand
Rudolf Marty

Institut für Informatik, University of Zürich
Winterthurerstr. 190, CH-8057 Zurich, Switzerland
E-mail: gamma@ifi.unizh.ch, gamma@unizh.uucp

ABSTRACT:

ET++ is an object-oriented application framework designed to provide a foundation for interactive graphic applications with a consistent user interface. ET++ is implemented in C++ and runs under UNIX and SunWindows or the X11 window system. This paper presents a case study of the integration of a programming environment consisting of an *inspector*, a *class browser*, and a *hierarchy viewer* into ET++ based on an extended run-time support for C++. As an enhancement of the programming environment a prototype of a *hypertext cookbook* illustrating the application building process with ET++ is briefly discussed. The paper also includes a description of the basic functionality of ET++ and a discussion of the design and implementation of the programming environment. Emphasis is put on the benefits of using an object-oriented application framework.

Keywords: application framework, user interfaces, C++, run-time support, programming environment, teaching

1. INTRODUCTION

When working with an object-oriented programming language class libraries are an important foundation for the implementation of an object-oriented software system. In fact the libraries might even be considered more important than the language itself. ET++ is a class library implemented in C++ for a UNIX environment. It runs on SunWindows and the X11 window system. The architecture of the ET++ application framework is based on MacApp and integrates a rich collection of user interface building blocks as well as basic data structures to form a homogeneous and extensible system. The primary design goal of the ET++ class library was to combine the concepts found in user interface toolkits with those of general support libraries in order to form a seamless system [Wein88, Wein89].

We soon realized, however, that a class library with a rich functionality considerably increases the learning time of novice users to become proficient. This is particularly evident with deep class hierarchies where many programmers have trouble grasping all the inherited behavior of a subclass. This has led the designers of Interviews [Lint87] (a user

interface toolkit written in C++ for X windows) to implement only a very flat hierarchy (most classes are at level 2 or 3). Such a restriction was not acceptable for ET++ since we wanted to use inheritance to avoid code redundancies wherever possible. See reference [Wein88] for some statistics about the ET++ class hierarchy.

To reduce the problem of the high learning effort we initiated the design of a programming environment and other support tools for ET++. Implementing such an environment for ET++ with ET++ itself was another test to verify the functionality of the ET++ class library. In retrospect, we can say that this test was very successful. The first version of the system as described below was running after two person weeks and only required about 800 lines of code. The final version of the ET++ programming environment was finished after two person months.

2. ARCHITECTURAL OVERVIEW OF ET++

This chapter gives a brief overview of ET++ as a basis for understanding the ET++ programming environment. For a detailed description of ET++ we refer to [Wein88, Wein89].

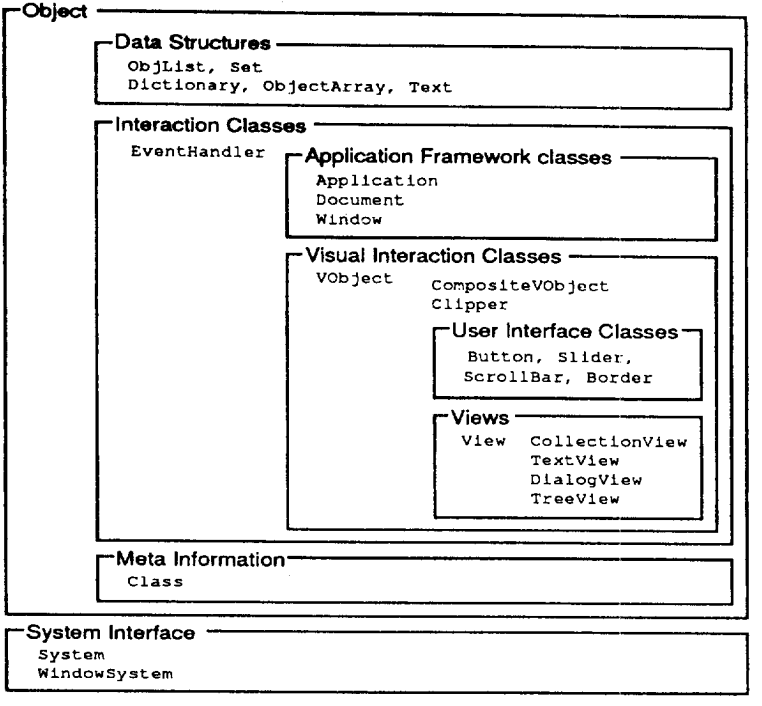

Figure 1: Overview of the Class Hierarchy

Fig. 1 shows an overview of the ET++ class hierarchy. Most of the ET++ classes are derived from class `Object` *which implements a change propagation mechanism modeled after Smalltalk-80's "changed-and-update-principle".* `Object` *also provides the mechanism to transfer arbitrarily complex objects between disk and memory.*

2.1. Data Structures

The foundation of ET++ is made up of generally useful data structures and includes a simplified version of the Smalltalk-80 collection classes.

2.2. 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 event 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. Doing so automatically results in such algorithms working on any kind of graphical object.

A very important 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.

Another important mechanism of the graphical foundation classes is the ability to combine several `VObjects` into a single, composite object which can be treated as a composite `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` lies in the responsibility of a subclass.

The class `Clipper` defines an independent coordinate system and clips the graphical output of a `VObject` to a rectangular area. The implementation of scrolling and zooming 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 hierarchies of independently scrollable `VObjects` nested to arbitrary depth.

2.3. Application Framework Classes

One of the major components of an application framework is implemented in the class `View`, which is also a subclass of `VObject` and represents an abstract and possibly arbitrarily large drawing surface. Its main purpose is to consolidate at a single point all control-flow necessary to manage printing, holding a current selection, and maintaining a clipboard.

The purpose of class `Window` is twofold: it implements the binding to the window system interface (see 2.5), and provides mechanisms to ease and optimize the screen update. The ET++ application framework employs an indirect drawing scheme which makes it completely unnecessary to call drawing methods directly. An application simply tells ET++ which objects must be redrawn by calling a method to add 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.

The non-graphic application classes `Document`, `Application`, and `Command` are basically derived from MacApp and therefore have a similar behavior and similar interfaces. They provide methods for the management of documents and undoable user commands. For a detailed description of their structure and functionalities we refer to [Schm86].

2.4. User Interface Building Blocks

The most important design idea for creating complex dialogs was to hierarchically and sequentially combine all sorts of user interface building blocks like scrollbars, buttons, and editable texts (subclasses of `VObject`) in a declarative way applying layout operators (subclasses of `CompositeVObjects`).

More complex user interface building blocks with the notion of a current selection and clipboard support are derived from the class `View`.

To design a text editing framework usable in many different contexts such as dialog boxes (dialog items), diagrams (annotations), and browsers (program text) we followed the *model-view-controller* (MVC) paradigm of Smalltalk-80 by strictly separating between classes to render and format text (`TextView` and subclasses) and classes for managing the data structures to store the text (`Text` and subclasses).

The protocol supported by visual objects makes it possible to treat any `VObject` as a single glyph which can be integrated into text and behaving as an ordinary character. This integration of `VObjects` into text is realized by the class `VObjectText` which extends the methods for cutting and pasting text intermixed with arbitrary visual objects.

Another specialized view is the `CollectionView` which displays any collection of `VObjects` as provided by the foundation classes in a tabular format and takes care of selecting and deselecting items. The `CollectionView` is a basic building block for all user interface objects which have to show a collection of selectable items (e.g. menus, menu bars, tools' palettes, or scrollable lists of dialog items).

A `DialogView` implements a standard behavior for modal or modeless dialog boxes. A single method must be overridden to create the dialog, another to react to all dialog interactions.

2.5. System Interface

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 between different UNIX implementations. In order to be independent from a particular environment, all system dependencies were encapsulated by introducing an abstract system interface defining a minimal set of low-level functionality necessary to run ET++. This system interface layer is structured into several abstract classes which are to be subclassed for a specific environment.

The two abstract classes `System` and `WindowSystem` define the entry point into the system interface layer by providing methods for the instantiation of new objects representing operating system resources like files and directories or window system resources like ports, fonts, and bitmaps.

As an example the class `Port` defines the graphical output primitives common to all output devices (see Fig. 2). Subclasses of `Port` override the abstract output primitives with a device dependent implementation or add device specific methods. The abstract class

`Port`	*an abstract output port*
`PicturePort`	*generates the ET++ exchange format for images*
`PrinterPort`	*abstract printing device*
`PicPort`	*generates pic-troff code*
`PostScriptPort`	*generates PostScript*
`WindowPort`	*abstract window device*
`SunWindowPort`	*an implementation for SunWindows*
`ServerPort`	*a server-based implementation for SunWindows*
`XWindowPort`	*a implementation for the X11 window system*

Figure 2: `Port`-hierarchy

`WindowPort`, as another example, extends the output interface of a `Port` with methods for input handling and window management. The underlying window system must actually only provide mechanisms for the management of overlapping rectangular areas on the screen. The subclass `SunWindowPort` is an implementation of `WindowPort` for Sun workstations, the `XWindowPort` for the X window system. Due to dynamic binding of the device dependent methods we are currently able to run applications on different window systems without any recompiling.

3. THE ET++ PROGRAMMING ENVIRONMENT

During the design of the ET++ programming environment (ET++PE) we noticed that a lot of its desired functionality is already included in the Smalltalk-80 programming environment [Gold84], and it was a challenge to integrate this functionality into a conventional environment based on compilation rather than code interpretation. 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 within the application's process. The browser or inspector can be invoked either at startup time or at any time while the application is running.

3.1. The Inspector

A central question that has to be addressed in an inspector is the identification of the object to be inspected. An interpreted environment like Smalltalk-80 allows evaluating any Smalltalk expression and using the resulting object for inspection. A compiled environment, without the possibility of interactively evaluating expressions, has to provide other access methods to the objects of a running application. One way of identification offered by the ET++ inspector is to start a session with a display of the global objects of an application. The desired object can then be found by interactively following pointers emanating from these objects. A problem with this approach is the cumbersome navigation through chains of pointers and the intimate knowledge of the run-time structure of an application required to do so.

Providing inspecting support for applications with a graphical user interface can profit from the fact that most objects of interest are visible on the screen. Consequently, the ET++ inspector supports a so called *inspect-click*. An inspect-click is just a mouse click together with some modifier keys. It allows to click on any visible object on the screen for inspection.

An alternative access path to an object is provided by the two top right panes of the inspector (Fig. 3). These two panes are used for browsing instances of a specific class. The left pane shows a list of all classes used in an application and, in parenthesis, the number of its instances. Clicking on a class lists its instances in the middle pane. The object to be inspected can then be selected from this list. To better identify an object in this list its address and some additional text is displayed. In Fig. 3, for example, the instances of `StyledText` a class managing a text data structure use the beginning of their text contents as an identification. This additional identification information cannot be provided automatically and requires the implementor of a class to override one method returning this identification string. The implementation of this method typically uses the value of an instance variable to return it as a string.

Once an object was selected for inspection the values of its instance variables are displayed in one of the two bottom panes. These two panes allow to view the state of two objects concurrently. The name of the classes the variables are inherited from are indicated together with the values of the instance 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.

An inspector for viewing object structures at run-time is expected to provide more functionality than just viewing the state of an individual object. It should also integrate some way of studying the relationship among different objects, e.g. the pointer references. To give some support in this direction, the ET++ inspector integrates queries for other objects having a reference to the currently inspected one. The result of this query fills the top right pane with a list of such objects (Fig. 3). Clicking on an object in this list displays the values of its instance variables in one of the bottom panes.

The inspector always provides for instant access to the source code. The classes' definition or implementation code is displayed by the source code browser (see below) after selecting a menu item. Instant access to the source code and the inspect-click 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 can learn about instance variables and has also instant access to the corresponding source code.

3.2. The Source Code Browser

Figure 4 shows a screen dump of the source code browser. Its left pane displays an alphabetically sorted list of the classes known in the current application. The right pane contains a full fledged text editor showing the pretty printed source code of the selected class using 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.

An alphabetically sorted list does not reveal the structure of a class hierarchy. Therefore, a tree representation showing the inheritance relationship between the classes is also available (Fig. 5). Clicking on a class name in the tree loads the corresponding source code into the editor.

The tree view allows collapsing and expanding a subtree. Notice that our work is based on a version of the C++ compiler not including multiple-inheritance, therefore a simple tree suffices. (The algorithms for the layout of graph structures to display the structure of a

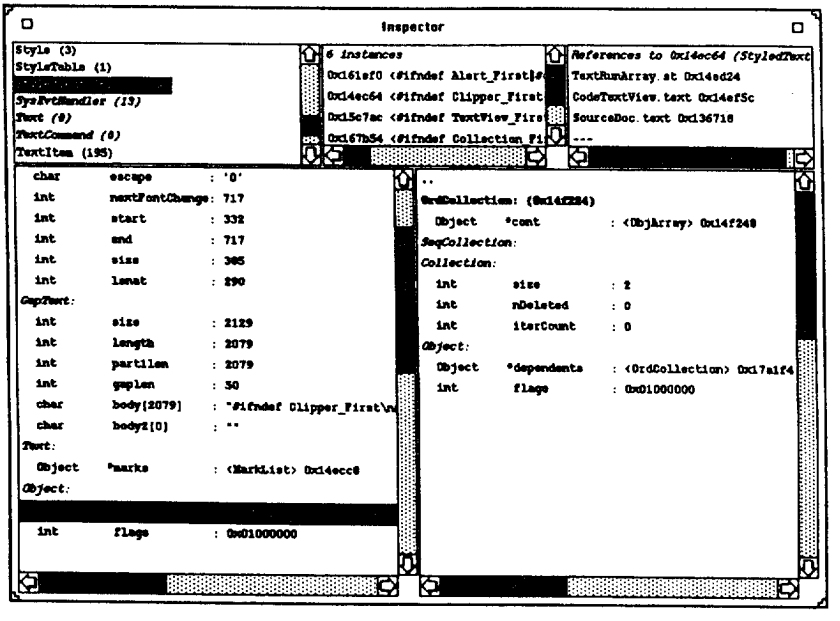

Figure 3: The Inspector

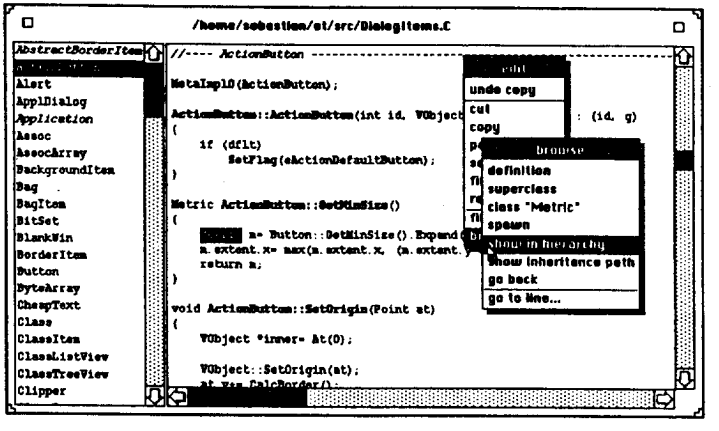

Figure 4: The Source Code Browser

multiple-inheritance class hierarchy have been explored in another ET++ based project) [Schm89]. 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 (Fig. 5). The flat inheritance view allows to quickly explore where some behavior is inherited from. In an application framework the abstract classes are particularly important, for this reason they are highlighted with an italic styled font in all the views of the class

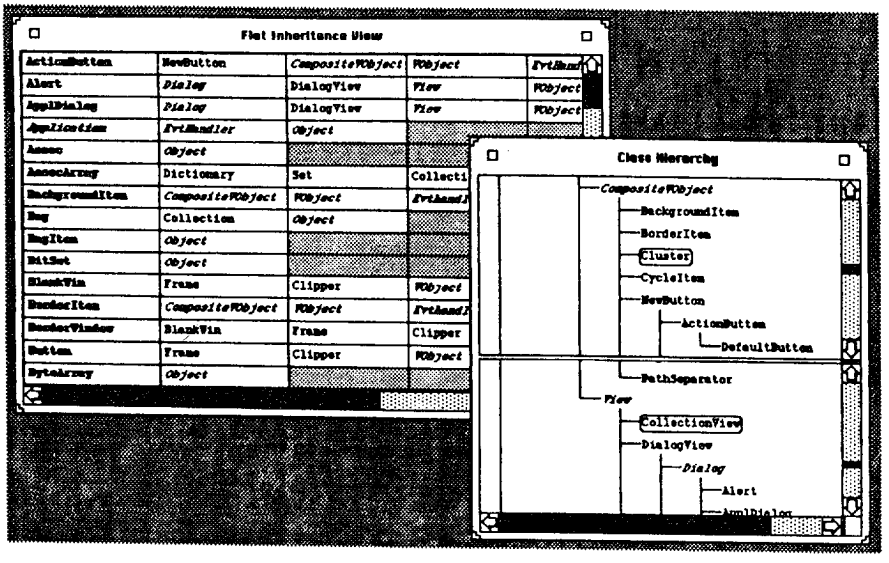

Figure 5: The Hierarchy Viewer and the Flat Inheritance View

hierarchy.

A typescript application for a UNIX shell completes the self supporting environment for ET++.

4. IMPLEMENTATION

This section discusses some aspects of implementing the user interface for ET++PE and its underlying data structures.

4.1. Implementing the User Interface

The implementation of ET++PE once more revealed the high reusability of the ET++ classes. All the user interface components of the inspector, of the source code browser, and of the hierarchy viewer had already been part of the ET++ class library or have been implemented as very simple subclasses therefrom. The views showing the instance variables, the sorted list of all classes in the system, and the flat inheritance view, for example, are just instances of `CollectionView`. The source code editor is a subclass of the `CodeTextView`, another subclass of `TextView` adding features to edit source code. The view for a tree display of the classes is derived from a simple tree editor application. Implementing the hierarchy viewer showed that a tree viewer is useful as a basic building block. Therefore, we migrated the code for rendering a tree into the ET++ library as a class `TreeView`.

4.2. Run-Time Support for C++

One problem of building programming support tools like an inspector for C++ is that C++ does not preserve any information about the class structure or the instance variables of an object until run-time. Consequently, an additional mechanism has to be introduced to

gather this information in order to give the necessary run-time and programming environment support.

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

As almost any other part of ET++, the current implementation of metaclasses is the result of an evolutionary process. In the first version we only stored information about the structure of the class hierarchy. This was sufficient to implement an `IsKindOf` method to check the type of an object at run-time (the run-time system of C++ does not provide an `IsKindOf` method). The ET++ class library includes in its foundation classes abstract data types often referred to as container classes. Their implementation relies on this `IsKindOf` method in order to compare objects or to perform type secure casts. C++ currently has no parameterized types but offers only a simulation of this concept based on macros. We preferred the approach relying on some kind of dynamic type checking.

Having access to the type of an object at run-time was also necessary for the object input/output facility of ET++ because not only the state of an object has to be transmitted but its corresponding class as well.

The next step was to add information about the instance variables of an object. This is kind of a "map" for the structure of an object and was the base for implementing the inspector.

To give access to the source code of a class we finally stored a reference to the definition and implementation part of a class in its metaclass.

The final version of metaclasses stores the following information about a class:
- the associated superclass
- the name of the class
- the size of an instance in bytes
- the types and names of the instance variables
- a source code reference to the definition and implementation part of the class

Since the C++ run-time system gives no access to the information described above it is mandatory that the programmer provides some of this information manually. During the design of metaclasses we always followed the principle "do not bother the programmer". For this reason, we try to extract as much as possible automatically based on mechanisms of C++ and the C preprocessor. The convention that should be followed by a programmer when introducing a new class is to call a macro in the definition and implementation part of a class. The code excerpt in Fig. 6 illustrates the call of these macros for the class `Set` a subclass of `Collection`. A call of the `MetaImpl` macro consists of the class name followed by the list of instance variables and their corresponding types. The type of an instance variable is specified with the predefined symbols starting with an '`I_`' prefix. For example, `I_O` indicates a pointer to an instance of `Object` (or a subclass thereof in the example `ObjectArray`) and `I_I` refers to an integer type. As can be seen in this example, only a list of the instance variables' types has to be provided manually, the rest is extracted automatically.

Expansion of these macros generates two methods for the class and declares the instance of the associated metaclass. One of the generated methods returns an object's metaclass. The other one enumerates all the instance variables of an object with their name, type, and

definition part:

```
class Set: public Collection {
    ObjectArray *contents;
    int size;
public:
    MetaDef(Set);
    Set();
    ~Set();
    //...
};
```

implementation part:

```
MetaImpl(Set, (I_O(contents), I_I(size)));

Set::Set()
{
    //...
}
```

Figure 6: Usage of the Metaclass Macros

offset. It is used, for example in the inspector, to get the necessary information about to the internal structure of an object.

When introducing additional conventions that are not enforced and completely checked by the compiler (albeit misspellings of the instance variable names in the macros are detected), it is important that failure to obey the conventions does not introduce spurious bugs. The only consequences of omitting the two macros invocations are:

- the instance variables of an instance cannot be inspected in the inspector
- the metaclass of an instance refers to a superclass
- the class cannot be used in an `IsKindOf` test, because there is no corresponding metaclass
- the source code is not accessible from the browser.

An alternative approach to get access to compile time information would be to partially parse the C++ source files and to base tools like a browser on this extracted information [Ragh87]. But this is hard to do without a full compiler, and we had no intentions of building yet another C++ compiler nor to modify an existing one. Moreover, relying on some previously collected information about a program can always result in inconsistencies with the current version of it.

OOPS introduced the idea of additional class descriptors in C++ similar to ET++. Considering the effort required by clients of the OOPS library to build a new subclass we are convinced that the scheme implemented in ET++ is much easier to use and still more powerful. The class objects of OOPS provide no support for a programming environment.

In Smalltalk-80 the concept of metaclasses was introduced to refer to the class of a class in order to make classes first class objects. The difference of Smalltalk-80 metaclasses compared with the ET++ approach is that Smalltalk-80 metaclasses are not only used to give

programming environment and run-time support, but also provide for defining flexible protocols for instance creation. C++ on the other hand has the notion of constructors to create instances, which makes the complex recursive structure of metaclasses as found in Smalltalk-80 unnecessary in our C++ environment.

In addition to metaclasses the run-time support system for ET++ maintains a hash table of pointers to all instantiated objects of an application. This table is used by the inspector to find all instances of a class. The constructor of the root class `Object` of the ET++ class hierarchy adds an entry to this table and its destructor removes it.

To determine which objects reference an inspected object, all pointer instance variables of the instances stored in this table are checked. The necessary typing information about the instance variables is retrieved from the corresponding metaclass. On first sight it seems that an exhaustive search is too expensive to find out the references in a reasonable amount of time. But due to the hybrid nature of C++ not every data item is an object. This results in a not too large number of objects in an application, say 1000 to 10000. This number of objects can always be searched through in a reasonable amount of time.

5. EXPERIENCES IN DEFINING AND IMPLEMENTING ET++PE

The implementation of the inspect-click feature is a good example for the power of an object-oriented application framework: 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.

Another mechanism of ET++ that proved to be useful for the implementation of the inspector is the change propagation mechanism provided by the root class of ET++ (`Object`). The inspector registers itself as one of the inspected object's dependents. If the inspected object follows the change propagation protocol and announces its changes, the display of the inspector will be updated automatically. There is also a menu command to update the display manually, but most of the foundation classes of ET++ follow the change propagation protocol.

Permanently updating the inspector window may slow down the interactive response of the system, for example while an object changes its state during a rubberbanding operation. To circumvent this problem the inspector uses the delayed update mechanism as provided by the ET++ window classes. The inspector window displaying the changed instance variables is just invalidated; the redraw is only performed when the application is idle.

Due to the single root structure of the ET++ class hierarchy it was very simple to add some support behavior to classes which is then automatically inherited by all their subclasses. Much of our design goal "don't bother the programmer" has been fulfilled as described in 4.2. Even if an application builder does not explicitly invoke the macro to announce the names of a new class and its instance variables, an object is inspectable. The limitation in this case is that only the name and the instance variables of the last conforming superclass are shown in the inspector.

Comparing ET++PE with an interpreted environment like Smalltalk-80 shows its major drawback. A Smalltalk programmer changes some behavior by simply stopping the running application, modifying the code, and restarting the system *without* any compiling or linking delays. In ET++ it is not possible to modify a class from within ET++PE, compile it

and link it incrementally to the running application *replacing* its old version†. But one goal of ET++ always was its integration into a conventional environment without any modifications to existing tools like compilers and linkers.

The advantage of a conventionally compiled language with strong static type checking like C++ is that it ensures type consistency. Compile time checks are particularly helpful during a major reorganization of a class hierarchy. In Smalltalk-80 inconsistencies are only detected at run-time. To overcome this limitation the Smalltalk browser offers rich cross referencing facilities that are currently missing in our environment.

Our experience with ET++ has shown that it is not only *convenient* to have some information about the classes and their structure available at run-time, but sometimes even *necessary*. The approach of building meta-objects by hand 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, e.g. as part of the *vtbls* with a standard interface. Eiffel follows this approach [Meye88]. The difficult part is to agree upon what information is really necessary. 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. Replacing the ET++ foundation classes with the OOPS classes, just to give an example, would result in a very heterogeneous system due to the different conventions of integrating run-time support.

6. OTHER ENVIRONMENTS FOR CONVENTIONALLY COMPILED OO-SYSTEMS

The latest release of MacApp [Bian88] includes an inspector too. Similar to ET++ they use some conventions to get a map of an object. Their approach consists of overridding a method to enumerate the instance variables of an object. In ET++ we use a more declarative approach based on a macro with type specifiers. The MacApp implementation is not based on metaclasses and the inspector does not integrate access to the source code of an inspected object.

ParcPlace system's Cynergy is a C++ environment implemented in Smalltalk-80 with the possibility to incrementally compile and link as suggested in [Stro87]. Cynergy does not run in a conventional window systems but runs in the Smalltalk environment.

The Eiffel environment [Meye88a] provides a tool called *Viewer* for the inspection of object structures. The Viewer uses a crude conventional teletype interface and allows the traversal of the object structure by following references. *GOOD* (Graphics for Object-Oriented Design) a tool for the graphical exploration of the static structure of a system runs under X11 windows and can display the client and inheritance relationship among classes for an existing program. GOOD does not integrate access to the source code of the displayed classes. In contrast to C++, Eiffel offers information about the structure and methods of an object at run-time and, for this reason, does not require conventions to provide it manually.

† *There is support in ET++ for dynamic loading and linking of new classes with the functionality sketched in [Stro87].*

7. BEYOND BROWSING

We believe that a "learning-by-example" approach is an efficient way to reduce the high learning effort when starting to work with an object-oriented application framework. Browsing in a hierarchy of existing classes is just one step in this direction. What is really needed is an online *cookbook* with hypertext facilities including a rich collection of existing applications. The cookbook should include a collection of recipes illustrating the usage of the framework.

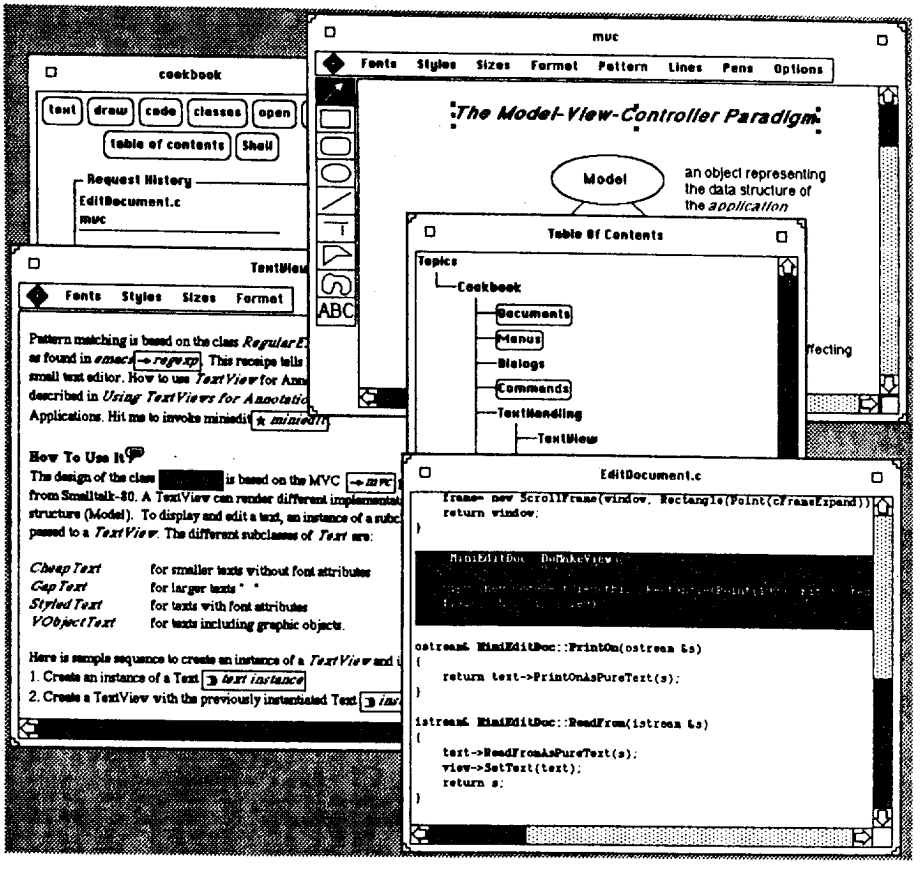

Figure 7: The ET++ Cookbook

To implement a prototype of an "active/interactive cookbook" for ET++ we added a thin hypertext layer similar to Intermedia [Meyr86] to the existing ET++ framework. 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 an example application referenced from a descriptive text, we introduced so called "action links". 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 previously.

A special hierarchical link type provides for a hierarchical organization of the contents of a cookbook. Based on these links a table of contents can be generated and displayed in an ET++ `TreeView` similar to the hierarchy viewer.

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

Fig. 7 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 Fig. 7 the user first followed the 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 selected it invokes the example application "miniedit". The application window titled "Cookbook" shows the list of documents already visited.

As mentioned earlier, `VObjectText` treats arbitrary graphical objects as glyphs in a text. This was very helpful to implement the link buttons. The sticky behavior of the buttons (the buttons flowing with the text when characters are inserted or deleted) came for free also. Even text within the graphic editor can include link buttons behaving as expected.

The `TreeView` has been reused once more for the table of contents of the cookbook. The support of ET++ for handling different document types formed the base for integrating the different kinds of editors into one single application. On the other hand, the implementation of the cookbook prototype revealed the need for some kind of data base support for the linking facility including concurrency control as found in Intermedia.

The idea to add hypertext facilities to an online manual can also be found, for example, in the Symbolics Document Examiner [Walk85].

8. CONCLUSION

The authors have shown how easy it was to build a powerful support environment for the object-oriented application framework ET++ based on ET++ itself. The paper illustrated that it is possible to implement Smalltalk-like tools based on some simple conventions even in a conventionally compiled C++ environment. Due to the high reusability of the ET++ classes this has been implemented in a minimum amount of time.

ET++ and its programming environment running under X11 and SunWindows are available in the public domain.

REFERENCES

[Bian88] Curt Bianchi and David Goldsmith, *MacApp 2.0 Display Specification*, Apple Computer, Inc., Cupertino, CA, May 1988.

[Gold84] Adele Goldberg, *Smalltalk-80, The Interactive Programming Environment*, Addison-Wesley, Reading, Mass., November 1984.

[Lint87] Mark A. Linton and Paul R. Calder, "The Design and Implementation of InterViews," in *USENIX Proceedings and Additional Papers C++ Workshop (Santa Fe, NM, 1987)*, pp. 256-268, USENIX Assoc., 1987.

[Meye88a] Bertrand Meyer, "The Eiffel Environment," *Unix Review*, August 1988.

[Meye88] Bertrand Meyer, *Object–Oriented Software Construction*, Prentice Hall, Englewood-Cliffs, New Jersey, 1988.

[Meyr86] Norman Meyrowitz, "Intermedia: The Architecture and Construction of an Object-Oriented Hypermedia System and Applications Framework," *OOPSLA'86, Special Issue of SIGPLAN Notices*, vol. 21, no. 11, pp. 186-201, Portland, Oregon, November 1986.

[Ragh87] Raghunath Raghavan, Niranjan Ramakrishnan, and Sue Strater, "A C++ Class Browser," in *USENIX Proceedings and Additional Papers C++ Workshop (Santa Fe, NM, 1987)*, pp. 274-280, USENIX Assoc., 1987.

[Schm89] Duri Schmidt, *Das TOPOS-Component Management System (to be published)*, Institut für Informatik der Universität Zürich, Zürich, 1989.

[Schm86] Kurt J. Schmucker, *Object Oriented Programming for the Macintosh*, Hayden, Hasbrouck Heights, New Jersey, 1986.

[Stro87] Bjarne Stroustrup, "Possible Directions for C++," in *USENIX Proceedings and Additional Papers C++ Workshop (Santa Fe, NM, 1987)*, pp. 399-416, USENIX Assoc., 1987.

[Walk85] Jan H. Walker, "The Document Examiner," in *SIGGRAPH Video Review*, 1985.

[Wein88] André Weinand, Erich Gamma, and Rudolf Marty, "ET++ – An Object Oriented Application Framework in C++," *OOPSLA'88, Special Issue of SIGPLAN Notices*, vol. 23, no. 11, pp. 168-182, San Diego, California, November 1988.

[Wein89] André Weinand, Erich Gamma, and Rudolf Marty, "Design and Implementation of ET++, a Seamless Object–Oriented Application Framework," *Structured Programming*, vol. 10, no. 2, June 1989.

TRADEMARKS