# ET++2.2 -Introduction and Installation

Erich Gamma
André Weinand
UBILAB
Union Bank of Switzerland

December 21, 1990

ET++ is a *homogeneous* object-oriented class library integrating user interface building blocks, basic data structures, and support for object input/output with high level application framework components. ET++ is implemented in C++ and runs under UNIX™ and either SunWindows™, NeWS™, or the X11 window system.

## 1. Introduction

This paper is intended for programmers who are familiar with the basics of object-oriented programming and the C++ programming language. It should provide enough information to start developing applications with ET++. The paper is not intended to replace studying the ET++ example

applications and their source code. Emphasis is put on the description of conventions and mechanisms which are not immediately understood by reading source.

ET++ papers/documentation:

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

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

1.3. A. Weinand, E. Gamma, and R. Marty, “Design and Implementation of ET++, a Seamless Object–Oriented Application Framework,” Structured Programming, Vol. 10, No. 2, June 1989.

This is the most recent paper about ET++ and describes the basic architecture, the imaging model and the built-in programming environment.(a copy of this paper in postscript is included in this distribution).

1.4. E. Gamma, A. Weinand, “ET++: A Portable C++ Class Library for UNIX(TM) Environment” OOPSLA’90 Tutorial notes.

1.5. Bryan Boreham, "ET++ Review", C++ Report, Vol. 2, No. 4, April 1990

In addition the following books may be of help:

1.6. K. J. Schmucker, *Object Oriented Programming for the Macintosh*, Hayden, Hasbrouck Heights, New Jersey, 1986.

This book describes the MacApp application framework. The basic architecture of ET++ is close to MacApp and the concepts of Views, Documents and Commands are similar enough to get a basic understanding of the functionality of these classes.

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

The ET++'s collection classes use the terminology of the Smalltalk-80 collection classes.

# 2. Installation

The distribution is organized hierarchically as follows:

| | |
|---|---|
| et | |
| .cshrc | example profile (mandatory environment variables) |
| CSHRC | same as above |
| src | source code of ET++ class library |
| MALLOC | extended malloc |
| SUNWINDOW | interface for SunView/SunWindows |
| SERVER | a window server running under SunView (not supported in this release) |
| XSERVER | interface for X11R4 |
| NEWS | interface for the NeWS window system (not supported in this release) |
| PIC | interface for producing pic output (experimental) |
| POSTSCRIPT | interface for producing postscript output |
| SUNOS | interface for SUNOS (or other BSD based Unix systems) |
| PROGENV | ET++'s programming environment |
| images | bitmap images |
| applications | example applications developed with ET++ |
| postscript | postscript libraries |
| IO | stream classes |
| doc | this installation guide<br>ET++-Introduction Paper<br>GNU's regular expressions syntax |
| fonts | fonts in bdf format |
| sunfonts | vfonts for SunWindow (initially empty) |
| xfonts | snf fonts for X11 (initially empty) |
| owfonts | fonts for OpenWindows |
| bin | utility programs for ET++ |
| dyn.sparc | *.o files for dynamic loading/linking with "app" (initially empty) |
| util | |
| makedepend | public domain makefile dependency generator |
| makemap | utility to extract informations about member functions in a source file |

## 2.1. Hardware and Software Requirements

Reading the ET++ tape requires about 10 MB of diskspace. A full ET++ installation requires about 40 MB of disk space. The following table gives some numbers for the sizes of the directories after all the fonts and example applications are compiled:

| | |
|---|---|
| src | 2.1.1. MB |
| sunfonts | 2.1.2. MB |
| xfonts | 2.1.3. MB |
| applications | 2.1.4. MB |

**Supported C++ Compilers:**

– AT&T C++ 2.0, SUN C++2.0:
ET++2.2 is 2.0 “friendly” but does neither use nor support multiple inheritance!

– g++1.37.1:
If g++ is used with ET++ do not use the libg++ version of malloc. An application using this malloc will immediately crash with "`malloc/free/realloc: clobbered space detected`". To avoid the inclusion of the libg++ version of malloc in libg++.a build it with:

```
XTRAFLAGS = -DNO_LIBGXX_MALLOC
```

**Supported Hardware and OS:**

– Sun OS 4.x (680xx, Sparc), SunWindows, X11R4, X11R3

– Sony News 1850, X11R3

The ET++ collection classes are window system independent and can be used on all platforms.

## 2.2. Installation Procedure

The ET++ software is location independent, i.e.. it can be installed at any place in the file system. Choose the directory where ET++ should be installed, for example /local. Extract the files:

```
tar -xvfb /dev/rst0 126
```

The standard location for ET++ is /local/et. If the standard location is not used the environment variable `ET_DIR` has to be set accordingly. For example, if ET++ is installed in /home/oolibs/et, `ET_DIR` has to be set to /home/oolibs/et. The installation directory is referred to as `ET_DIR` in the following explanations.

### 2.2.1. Configuring ET++ for your C++ Compiler

All compiler dependencies of ET++ are now located in the `etCC` script located in ET_DIR/bin. Adapt this script to your site specific C++ installation.

At the end of this script the utility `makemap` is called. Makemap extracts information about member-functions wich is used by the source browser. `Makemap` generates an output file with the same name as the source file and the suffix “`.map`”. The file is stored in a directory “.MAP”. If this directory does not exist no map-files will be generated. Instead of using map-files you can set the environment variable ET_NO_MAPFILES and the source browser will extract the member functions on the fly (this approach is slower than using map-files).

### 2.2.2. Configuring the ET++ library in /src/makefile

ET++ class library includes support for several different window systems and printing devices. The makefile in /C++/src includes under the section entitled “configuration” some macros to tailor ET++ to a specific environment.

#### 2.2.2.1. Selecting the Window System Interfaces

By default, all supported window system interfaces are included.

```
WS_OFILES    =  $(XSERVER) $(SUNWINDOW)
WS_IFDEFS    =  -DWS_X -DWS_SUNWINDOW
WS_DIRS      =  SUN XSERVER
```

To remove a window system interface, delete the corresponding entry from these lines. The macro names stand for:

| | |
|---|---|
| `SUNSERVER` | interface to a server for the sunwindow system |
| `NEWSSERVER` | interface for NeWS 1.1, this interface is only experimental |
| `XSERVER` | interface for X11.3(BETA) |
| `SUNWINDOW` | interface for sunwindow/sunview |

The following example shows the definition of the macros for a version of ET++ which includes only the X interface

```
WS_OFILES    =  $(XSERVER)
WS_IFDEFS    =  -DWS_X
WS_DIRS      =  XSERVER
```

#### 2.2.2.2. Selecting the Printers

ET++ supports generating either postscript or pic output. By default, both formats are included. To remove a printer interface, adapt the macros as described above.

```
PR_OFILES    =  $(POSTSCRIPT) $(PIC)
PR_IFDEFS    =  -DPR_POSTSCRIPT -DPR_PIC
PR_DIRS      =  POSTSCRIPT PIC
```

#### 2.2.2.3. Operating System Interface

Currently, only an interface for SunOS — or similar BSD systems — is supported.

#### 2.2.2.4. Programming Environment

By default the ET++ library and therefore all the ET++ applications linked with it include a programming environment (an inspector, and a browser). These tools add 40KB to an ET++ application but have no influence on the execution speed. In order to remove this inspecting and browsing code in a final version of an application modify the macros as shown below:

```
PE_OFILES    =  # $(ET_PROGENV)
PE_IFDEFS    =  # -DET_PROGENV
PE_DIRS      =  # PROGENV
```

Recompile the class library with `make config` and relink your application. `make config` should be called whenever the `PE_` or the `WS_` macros are changed.

### 2.2.3. Select the Fonts for your Environment

The ET++ distribution includes fonts for X11 and SunWindow in bdf format. By default, the ET++ installation will generate binary versions of these fonts for sunwindow, X11 and OpenWindows. The compiled fonts are stored in:

| | |
|---|---|
| `sunfonts` | fonts for sunwindow |
| xfonts | fonts for X11 |
| owfonts | fonts for OpenWindows |

For which window systems the binary versions of the fonts have to be generated can be defined in ET_DIR/makefile. To avoid the creation of binary fonts for a specific window system remove the corresponding directory name from the DIRS macro:

```
DIRS    = util src applications examples sunfonts xfonts owfonts.
```

If you are running ET++ applications under OpenWindows or X11 do not forget to set the font path with

```
xset +fp $ET_DIR/???fonts.
```

#### 2.2.3.1. Build the ET++ Library and the Example Application "hello"

Execute make in ET_DIR which will compile the ET++ library, the example application "hello" (ET_DIR/applications/hello), and the fonts. After this step you should be able to verify the installation by executing the hello application.

The ET++ Library is not generated as archive including the ET++ object files but as a relocatable object file with the name et.o. Experience has shown that this approach speeds up linking of an ET++ application. An ET++ application linked with an archive is not significantly smaller than an application linked with a relocatable object file containing all the object files.

#### 2.2.3.2. Compile the ET++ Example Applications

Now you are ready to build the ET++ example applications. They are compiled by calling make in the directory ET_DIR/applications. More information about these example applications can be found in "ET++ Introduction".

### 2.2.4. Compiling ET++ Support Applications

In order to support cut/copy/paste operations among different application processes under SunWindow, a special clipboard server is required. If the clipboard server is not running applications, can only

execute cut/copy/paste operations between their own windows. `Make clipboard` compiles the SunWindow clipboard server. The clipboard is started by calling `clipboard`.

The application `app` generated with `make app` starts up any ET++ application and dynamically links the missing classes for this application. `App` is called with the name of the application as argument, e.g. `app micky`, after dynamically linking all the classes required by this application `micky` will start up as usual. The search path to be used to find object files can be set in the environment variable `ET_DYN_PATH`. ET_DIR/dyn.sparc contains all the necessary .o files to dynamically load applications micky, vobedit, miniedit, and layout.

> ***Notice: `app` is (still) not completely finished yet. We are waiting for better support for dynamic C++ loading and linking from the operating system.***

The command `make install` moves these additional applications to /bin.

## 2.3. Streams

In order to be independent from the stream-classes provided with C++ compilers ET++ includes now in the directory ET_DIR/src/IO its own implementation of stream classes. This classes are modelled after the stream classes of AT&T cfront 1.2.

## 2.4. Hardcopy Documentation

The /doc directory includes this document and the paper “Design and Implementation of ET++, a Seamless Object–Oriented Application Framework” as compressed postscript files.

## 2.5. Environment Variables

The following environment variables control the behaviour of ET++:

| | |
|---|---|
| ET_DIR | The root directory for ET++ files. |
| ET_FONT_SIZE | The font size used by the Inspector. |
| ET_SRC_PATH | The directories to search for source code. |
| ET_DYN_PATH | The directories to search for object files. |
| ET_NO_STYLEDCODE | Set to prevent pretty-printing of source code. |
| ET_NO_MAPFILES | Do not use map-files |
| ET_DISPLAY | For the SunWindow server system. |

# 3. Generic Behaviour of ET++ Based Applications

This section describes the generic behaviour of ET++ based applications.

## 3.1. The User Interface of ET++ applications

The assignment of the mouse buttons is:

| | |
|---|---|
| LEFT | selects an entry from a permanent menu, a button, or a range of text |
| MIDDLE | application specific |
| RIGHT | displays a pop-up menu |

### 3.1.1. Manipulating windows

Commands to change the state of a window can be invoked while the cursor is in the title bar or in the border surrounding a window. To bring a window on top of all other windows or to change its position use the left mouse button. The commands displayed in the pop-up menu of a window are self explanatory. A window can be resized with the little stretch icons in every corner. The boxes on the left of the title bar can be used to collapse a window to an icon. An icon can be expanded either by a double-clicking with the left mouse button or by selecting expand form the menu. The right box will bring the window below of all other windows.

Most ET++ applications provide scrollbars to control the visible portion of a document. Moving the thumb in the scrollbar updates the visible portion of the window concurrently. Windows which can be subdivided into several panes showing disconnected portions of a view have fence markers in their lower right corner. They can be dragged with the left mouse button to adjust the size of the different panes. A double click on these markers moves them back to their initial position.

Every ET++ application has an associated application window. This window is used to create new, to open existing documents and to quit the application. There is a button for each of the above operations. Applications working with several documents types, typically add a button to create a new

document for each type. The menu item application window of the file submenu displayed in the interior of a window can be used to bring the application window on top whenever it is obscured.

### 3.1.2. The Main Menu

The standard pop-up menu displayed in the content area of a window contains the generic editing commands cut, copy, paste and in a pullright menu the file and printing commands. To display a pullright menu move the mouse a few pixels to the right while the item is selected. Menu items with a pull right menu are marked with an arrow. The commands of the file submenu are:

load — replace the contents of a window with another document. (the open button from the application window always creates a new window)

import — import the contents of another document or file (only applications which can import anything include this command)

save — save the document

save as — save the document under a different name

revert — revert the document to the latest version stored on disk

close — closes the document

print — command to choose a printer and define the print options.

Every application is free to add additional pullright menus after the file entry.

### 3.1.3. Dialog Boxes:

ET++ applications use several styles of dialog boxes:

Alerts
: for short confirmations and messages, alerts grab all input events and block other applications from receiving input

Process-Modal Dialogs
: block all windows of one application. Input in windows of other applications is still possible and the dialog window can be moved around. An example is the file dialog described below. The windows of Process-Modal Dialogs do not have close boxes.

Modeless Dialogs
: behave like ordinary windows.

All dialogs provide a pop-up menu with the standard editing commands (cut, copy, paste). The TAB key can be used to jump to each text item from top to bottom in sequence. Some dialogs have a default button marked with a double border; typing a carriage return has the same effect as selecting it with the mouse.

### 3.1.4. File Dialog

Whenever a filename has to be entered during a save, load, open, or import command a file dialog box is displayed. This dialog box displays a list of the files from the current directory. The filename can be selected either from this scrollable list or entered directly into the text field at the bottom.

The file name entered in the text field may include shell meta characters. A double click in the file list opens the document, if the name corresponds to a file; if it is a directory the working directory will be changed accordingly.

> ***Notice: All ET++ applications displaying a list of items support to scroll with the cursor and home, end, page up, page down keys or to type in a letter to jump to the first entry starting with this letter while the cursor is over the list.***

Clicking on the pop-up item on top of the dialog displays a horizontal menu with the names of the parent directories. To change the current working directory to a parent directory the corresponding item can be selected.

### 3.1.5. Editing Text

All ET++ applications provide the same user interface to manipulate text in dialog boxes or text editors. The LEFT mouse button is used to select a range of text. A double click selects words and a triple click lines or paragraphs. A selection can be extended by clicking the left mouse button while the SHIFT key is depressed. The cursor keys can be used to move the caret from the keyboard. Beside the common cut/copy/paste operations a quick paste and a CTRL click command are available. Quick paste can be used to copy some text and insert it at the current position in one step. For a quick paste the text has to be selected while the CTRL and SHIFT keys are depressed. A CTRL click (eg. a click while the CTRL key is depressed with the left mouse button) inserts the selected text at this position.

Some ET++ applications use a standard find/change dialog to search for a text pattern. This pattern can be specified with a regular expression with the same syntax as in GNU-Emacs. Refer to the file /doc/regex.doc for a description of their syntax.

### 3.1.6. Interrupting and Aborting Applications

If an ET++ application performs a complex, lengthy task, you can interrupt the calculation in SunWindows and X11 with the L1 function key on the SUN keyboard.

This brings up a modefull dialog giving you the name of the interrupted operation and the choice of aborting or continuing.

Whenever ET++ applications hang (e.g because of sending output to *stdout* or *stderr* while in fullscreen mode) press the L1 function key on a sun keyboard three times. This will exit your application with a core dump but without saving anything.

***Notice: Pressing the interrupt key in the startup shell window will bring up the inspector.***

### 3.1.7. Printing

Printing is initiated with the print-entry of the main menu or with the key combination META-CTRL-SHIFT-p. The former selects a document's view for printing while the latter selects the current window for a screen dump. In both cases a modefull dialog is opened for selecting a printing device, and setting various printing options. The printing device is selected by clicking on an item in the top left printer list. Currently PostScript™, Pic and Pict devices are supported. If the selected printer has options available the option button is enabled.

Apply
: dismisses the print dialog and applies the selected options (e.g. the paper format) to the current view or the current window without printing anything.

Print
: prints the view or window to the specified printing device. Printing can be aborted with the L1 function key.

Save as ...
: opens a file dialog where you can specify a file for collecting the printing output.

The check box show pages enables drawing page breaks. Currently page breaks are displayed for each and every view of an application. This is considered a bug not a feature.

#### 3.1.7.1. PostScript options

The options Resolution and Orientation are self-explanatory.

PS-prolog
: If set, the PS-prolog option includes the PostScript prolog /postscript/ET.ps in the output. Otherwise a non-standard PostScript structuring comment "`%%Include: ET`" is used.

smooth
: The smooth check box enables smoothing of bitmaps and bitmap fonts. This option is only applicable if you are using a LaserWriter and your spooler supports encrypted PostScript files (as our spooler does). Then the file /postscript/smooth.ps is downloaded to the printer to implement the non-standard smooth command.

printer name
: The editable text line Printer name specifies the name of the printer used with the lpr -P*name* command.

### 3.1.8. Command Line Arguments

All ET++ based applications can be invoked with the following command line arguments:

| | |
|---|---|
| –Eb | turn double buffering off |
| –Ei | start the inspector together with the application |
| –Ee | start the source code browser together with the application |
| –EE | show the palette of ET++PE |
| –Em | gather and display memory statistics |
| –Epx,y | specify the position of the top left corner of the application window (`micky -Ep400,500`) |
| –Ew | enable warning messages |
| –Ec | simulate a monochrom screen on a color display |
| –Ed | set debugging mode |

Trailing command line arguments without a "-" prefix are interpreted as the names of documents that should be loaded initially.

## 3.2. The Built-in Programming Environment (ET++PE)

If the ET++ library has been configured to include the programming environment all applications linked with this library have built-in an inspector, object structure browser, source code browser, and a class hierarchy browser.

Upon activating ET++PE a window with a palette of buttons for each of the above browsers is displayed.

- The inspector is used to view the state of the objects of a running application.
- The object structure browser displays the part-of and other relationships among the objects of an ET++ application.
- The source browser gives access to the source code and member functions of a class.
- The class hierarchy browser shows a graphical display of the class hierarchy.

The entry point to ET++PE is the inspector. There are several ways to invoke the ET++ inspector:

- Hit the interrupt key in the window (control terminal) the application was started from. The inspector initially shows the only instance of the application object (including the global variables).
- Invoke the application with the **–Ei** option.
- Execute a so called inspect-click over the object to be inspected. An inspect click consists of a click with the left mouse button while the META and SHIFT keys are depressed (in a previous

realease an inspect click was initiated by depressing META/SHIFT and the CTRL key). This method cannot only be used to start the inspector but also to select any visible object for inspection. An inspect-click is a convenient feature to explore the structure and objects of a running application.

The inspector window is subdivided into 4 panes. The top right pane displays an alphabetically sorted list of all classes of the running application (abstract classes are shown in italic). Together with the class name the number of instances of this class is displayed in parenthesis. The menu provides the commands:

update
: update the list of classes and the instance count

hide/show empty classes
: show or hide classes with no instances

Clicking on a class name shows all its instances in the pane to the right. On top of the top middle pane the current number of instances of this class is indicated. The instances are identified by their hexadecimal address. Some classes provide some additional identification for their objects, for example, instances of the class `Document` display the name of the document or collections display their size and the class name of their first entry. The menu of this subview consists of the entry all instances. This command expands the list of instances with the indirect members of the currently selected class, e.g. instances of its subclasses. This feature is convenient when the exact class of an instance to be inspected is not known. The desired instance can be found by clicking on one of its superclasses followed by the menu selection all instances. Clicking on a member of a class displays the values of its instance variables in the left of the bottom panes.

Together with the values of the instance variables the name of the classes where the variables are inherited from are shown. If an instance variable is a pointer to another object the hexadecimal value together with the static type of the pointer is displayed. If the dynamic type of the pointer differs from the static type then the dynamic type will be displayed enclosed "<>" brackets. To dereference the pointer click on the instance variable and the corresponding object is displayed in the bottom right pane of the inspector. A shift-click on a pointer instance variable spawns another instance of the inspector.

The buttons in the middle of the inspector support to navigate along the path of visited objects.

From the inspector there is instant access to the source code of the inspected object. The menu items edit definition or edit implementation from the menu displayed in the bottom pane pop-up the ET++ source code browser (see below) with the code of the definition or implementation of the corresponding class.

The menu item references fills the top right pane of the inspector with a list of all other objects pointing to the currently viewed object. Clicking on an item in this list shows the corresponding object in the bottom left pane. This feature is particularly helpful to explore the relationships among the objects of an application. In addition this function helps to find garbage objects, e.g. objects that are not referenced by any other objects.

The inspector provides special views for several data structures. ET++ uses trees of `VObjects` for the layout of dialog items or the window contents. If a `VObject` is inspected, which is part of such a `VObject` tree the menu item VObject Tree is enabled. After selecting this command the inspector displays the corresponding `VObject` tree graphically in a separate window. An object is selected for inspection in this representation by clicking on the corresponding node in the tree. If the inspected object is a `Collection` the menu includes the item Collection Table. This command shows the contents of a collection in a tabular form in a separate window and clicking on an object loads it into the inspector.

### 3.2.1. The Source Code Browser

The source browser consists of 4 panes. The top-left pane shows the list of the classes of the running application. This pane provides a pop-up menu with the following commands:

implementation/definition
: a toggle to choose whether the definition or implementation of a class should be edited

super class
: edit the super class

spawn
: spawn another instance of the source code browser

show in hierarchy
: selects the currently edited class in the class hierarchy browser.

show inheritance path
: displays in the so called *flat inheritance view* all classes in a sorted table together with a list of their superclasses.

previous class
: switch back to the previously edited class

The top middle pane display a flattened view of all methods of the class. Clicking on a method shows its implementation in the text pane below. This pane includes a menu with the commands:

inherited
: displays all the inherited methods with the same name as the currently selected method in the top right pane.

overrides
: displays a list of methods which override the selected method in the top right pane

implementors
: displays a list of the implementors of method in the top right pane.

references
: to be implemented.

filter
: shows a dialog box and provides for entering a regular expression. The regular expression filters all those methods from the middle pane which do not match.

The top left pane displays the results from a query of the menu described above. Methods to be displayed in the text pane below can also be selected from this list.

The bottom text pane shows the pretty printed source code and provides the usual text editing commands. The editor uses the convention to display comments in italic and the names of classes and the headers of methods or functions in a bold face. It is possible to turn the usage of different faces off by defining the environment variable `ET_NO_STYLEDCODE`. This reduces the loading time of large source files. To reformat the source code after some changes use the reformat menu item.

The source browser uses the following searching order to locate the source files:

3.2.1.1. the current directory

3.2.1.2. ET++'s source directory /src

3.2.1.3. any other directory as specified in the environment variable `ET_SRC_PATH`.

The editor gives some support for finding matching brackets, a double click near a delimiter `(`, `)`, `"`, `{`, `}` sets the selection up to the matching delimiter.

### 3.2.2. The Class Hierarchy Browser

The class hierarchy browser provides a graphical display of the class hierarchy. A class can be selected with the left mouse button. The class hierarchy browser's menu includes the commands:

clients
: displays a connection to the other classes which use the selected class in an instance variable.

members
: displays a connections to the classes which are used as members of the selected class

collapse/expand
: collapses/expand the subtree below the currently selected class..

show source
: shows the source code of the currently selected class in the source browser.

inspect some instance
: shows an instance of the selected class in the inspector.

At the bottom of the class hierarchy browser there is a pop-up item which provides for replacing all concrete classes with a small place holder. Clicking on this place holder displays the full class name.

### 3.2.3. The Object Structure Browser

The object structure browser shows the part-of hierarchy of an object as a graph. The layout of the graph can be changed with the middle mouse button. A node of the graph displays the identifier of the object together with its class name. A double click on a node shows the corresponding instance in the inspector. From a pop-up menu a display of other relationships between the selected and the other displayed objects can be requested.

# 4. Implementing an ET++ Application

## 4.1. Coding Conventions

This section describes the coding conventions or idioms used for developing applications with ET++.

### 4.1.1. Run-time Information about Classes

4.1.1.1.+ does not provide information about the class hierarchy and the instance variables of objects at run-time. ET++ requires this information for the object input/output facility and the programming environment. In order to preserve enough information until run-time a few conventions have to be followed. The basic idea is to call a macro in the definition and implementation part of a class. The following example shows a class conforming the ET++ coding conventions.

```
// file Example.h
class Example: public Object {
    class Collection *col;
    int size;
    char *name;
public:
    MetaDef(Example);
    Example();
    //...
};

// file Example.c
    MetaImpl0(Example);
    Example::Example()
          {
                //...
          }
```

In the simplest form both macros just take the name of the class as argument. Information about the instance variables can be specified with the `MetaImpl` macro. Hint: under C++2.0 if there is a MetaDef without a matching MetaImpl macro C++2.0 will not generate the vtbl. In the MetaImpl macro the instance variable's type and name are enumerated:

```
MetaImpl(example, (TP(col), T(size), TP(name), 0));
```

***Notice: the list of instance variables has to be 0 terminated!***

The symbols `T?` are used to specify the type of an instance variable. Their meaning is described in the following table:

| | |
|---|---|
| `T` | any simple predefined types (int, char, Point, Rectangle) or Object |
| `TP` | Pointer to ... (including strings) |
| `TA` | fixed sized array of ... , TA(contents, 20) |
| `TAP` | fixed sized array of Pointers to ... |
| `TV` | dynamically growing array of ..., TAV(contents,size) |
| `TVP` | dynamically growing array of Pointers to ... |

***Notice: The old ET++1.0 I_? Macros can still be used, but the list must now be 0-terminated!***

Instance variables that are structures have to be described by enumerating their individual fields:

```
struct sv {
    int iv1;
    int iv2;
};

class X: public Object {
    iv sv;
    //...
};
MetaImpl(X, (T(sv.iv1), T(sv.iv2)));
```

The order of the instance variable declarations in the `MetaImpl` macro is irrelevant. Misspelled instance variable names are detected by the compiler.

For abstract classes the macro `AbstractMetaImpl` should be used. This macro has the same calling conventions as `MetaImpl`. The information whether a class is abstract or not is used only in the programming environment to highlight them with an italic faced font.

These macros declare an instance of a so called meta class for a class. The term metaclass is used in ET++ to refer to a class that stores information about another class. Metaclasses are instances of the class `Class` which is itself a subclass of `Object`. Inspecting an instance of `Class` illustrates what kind of information is stored in a meta class. In order to get access from an instance of a class to its corresponding meta class the macros generate a method called `IsA` returning the meta class instance. Another generated method called `Members` enumerates all instance variables of an object and provides information about their types and offsets.

The consequences of not conforming these conventions and not calling these two macros are:

– the fields of an instance cannot be inspected in the inspector

– the type of an instance refers not to the class itself but to a superclass
– it is not possible to test the dynamic type of an `Object` with the `IsKindOf` method
– it is not possible to store a pointer to such an object with the ET++ object input/output facility
– the source code is not accessible from the browser

Because MetaDef introduces a constructor for a class, you must define at least one constructor for your own use, even if it does nothing.

### 4.1.2. Providing additional Information for ET++PE

The two top right panes in the inspector are used to browse the instances of a class. Instances of a class are displayed with their hexadecimal address. In order to give some additional information to identify an object the method `InspectorId` can be overridden. This method typically uses the value of an instance variable and fills a buffer given as argument with a string. For example, documents use their name as illustrated below:

```
void Document::InspectorId(char *buf, int size)
{
    strncpy(buf, docName, size);
}
```

### 4.1.3. Header Files

To protect header files against multiple includes ET++ uses `#ifndef`, `#endif` preprocessor directives:

```
--- File Object.h ---
#ifndef Object_First
#define Object_First

// class definition

#endif Object_First
```

The number of included files in a header file should always be minimized, otherwise cpp will soon reach its limits. For this reason only the superclass of a class should be included in a header file whenever possible. Consequently pointer instance variables are declared as shown below:

```
#include "Collection.h"

class ObjArray;

class OrdCollection: public Collection {
    ObjArray *contents;
    //...
};
```

The header file of `ObjArray` is only included in the implementation file of the class.

### 4.1.4. Naming Conventions

The implementation of ET++ uses the following naming conventions:

– Class names and methods start with a capital letter

– Instance and local variables start with a lower case letter

– The constants of an enumeration type start with an "`e`"

```
enum Direction { eHor, eVert };
```

– Global variables start with a "`g`" (`gApplication`)

– Constants start with a "`c`"

– Constants for command codes start with a "`c`" and the rest of the identifier is capitalized ("`cCUT`").

### 4.1.5. Deleting Objects

An object should never by deleted by calling the delete operator directly it is preferable to use the macro `SafeDelete`. This macro tests if the pointer is 0 (under cfront 1.2), and if not calls the delete operator and sets the pointer to 0.

This and some other useful macros, inline functions and basic types functions are defined in the file Types.h.

### 4.1.6. Static Constructors/Destructors

Implementations of C++ typically have some problems with correct calling order of static constructors. This results in some nasty bugs when there are dependencies among the different constructors.

The best way to circumvent this problem is to avoid static objects completely. Wherever possible try to use *delayed creation* of global objects as illustrated below:

```
PrintDialog *gPrintDialog= 0;

Class::AnyMethod()
{
    if (gPrintDialog == 0)
          gPrintDialog= new PrintDialog;
}
```

A global object created in this way can be freed with the help of the `ONEXIT` macro. The statements defined with this macro will be executed when the application terminates. The only argument of `ONEXIT` is a unique name, typically the name of class implemented in this file.

```
ONEXIT(PrintDialog)
{
    SafeDelete(gPrintDialog);
          // any other statement that should be executed upon exiting
}
```

The counter part to `ONEXIT` is `ONENTRY` providing for the execution of code when the application is started.

### 4.1.7. Secure Casts

The Guard construct provides for explicit secure type conversions between subclasses of `Object`. Example:

```
int TextItem::Compare(Object *op)
{
    return strcmp(text, Guard(op, TextItem)->text);
}
```

`op` is casted to a `TextItem` if its dynamic type is `TextItem` otherwise an error will be signaled.

### 4.1.8. Using the +e2 Compiler Option

If you are not using cfront 1.2.1 with the patches provided in this distribution skip this section. The ET++ distribution includes a patch to AT&T's cfront which adds a **+e2** option. The intend of this option is to reduce the binary size of an application by minimizing the number of generated *vtbls*. This new option is comparable with cfront's standard options **+e0**, +**e1** but is easier to use. When cfront is called with the **+e2** option a vtbl for a class is only generated in those files including a special comment. This comment consists of a list of class names indicating that a vtbl for such a class has to be included in the generated object file. The convention for using this comment is to add it in the file implementing the corresponding class. For the file Set.c which implements the classes `SetIter` and `Set` the **+e2**-comment is shown below:

```
//$Set, SetIter$
```

The error message of the loader:

```
4.1.8.1. __Set_vtbl undefined
```

indicates that the **+e2** option is used but a corresponding **+e2**-comment is missing.

### 4.1.9. ET++ Makefiles

When compiling a new application start with copying an existing makefile and use it as a template. /applications/micky/makefile is a good example.

The only macro that has to be set for ET++ is “CC” (ET_DIR/bin has to be included in the search path).

**CC =** *etCC CCFlags*

The makefile dependencies are maintained with a public domain makefile generator. The source for this tool is distributed with ET++ in the /util/makedepend directory. The dependencies are automatically stored in the makefile by calling `make depend`. The original makefile becomes makefile.bak. Because makedepend generates absolute filenames for standard include files this release contains no dependencies.

## 4.2. Other built-in Utilities

ET++ applications have built-in some utilities which can be invoked from the keyboard. In order to use them depress the SHIFT, CTRL, META keys together with one of the following characters:

- **s** display on the standard output the actual number of instances (sorted by class name)
- **i** display the actual number of instances (sorted by the number of instances)
- **p** produce hardcopy output of the window contents
- **a** abort with a core dump
- **q** close this window (if the window blocks and you can't find a menu)
- **w** enable warning messages

# 5. An Overview of the ET++ Class Library

## 5.1. ET++'s Foundation

The ET++ foundation are the root class of the class library the class `Object` and the class `Class`, together with an object input/output facility.

### 5.1.1. The Class Object

All objects that should be managed by the collection classes described below have to be derived from the class `Object`. The most important methods of `Object` are:

**`Object`**`(int f= eObjDefault)`
The standard `Object` constructor sets up the flags to those passed as an argument, plus `cObjNonDeleted`, and adds the object to the global object table.

`virtual` **`~Object()`**
The destructor removes the object from the global object table, and informs any observers that it has died.

virtual void **FreeAll**()
"Deep free", i.e. delete recursively all the objects refered to by instance variables.

#### 5.1.1.1. Inquiries about an Object

`virtual Class *IsA()`
Returns the instance of class `Class` that contains the information about this class. This method is generated by the `MetaDef`-macro. See `Class`, for more information.

`virtual void Members()`
This method calls a function for each of the object's instance variables. This is used by the Inspector to display all the values of the instance variables.

`char *ClassName()`
Returns the class's name, as given to the `MetaDef` macro.

`bool IsKindOf(className)`
`IsKindOf` is a macro returning `TRUE` if the object is an instance of the same or of a subclass of `className`. Example:

```
Object *op;
if (op->IsKindOf(Collection))
    //...
```

#### 5.1.1.2. Comparing Objects

To compare two objects for equality the method `IsEqual` should be used. The collection classes described below use this method to perform comparisons.

`bool IsEqual(Object *anotherObject)`
The class `TextItem` implements the `IsEqual` test as shown below.

```
bool TextItem::IsEqual(Object *op)
{
    return (op->IsKindOf(TextItem) &&
          strcmp(text, ((String*)op)->text) == 0);
}
```

This method takes a pointer to any object as argument, for this reason the dynamic type of the object has to be checked before performing the equality test.

`int Compare(Object *op)`
Compare `op` with `this`, this method should return a negative result if `this < op`, zero if `this == op`, and a positive result if `this > op`.

`int Hash()`
Return a hash value of an object used as probe in the ET++ collection classes based on hashing (`Set`, `Dictionary`). `Hash` should always be overridden when a class overrides `IsEqual`.

#### 5.1.1.3. Object Input/Output

Input/Output of an object structure of arbitrary complexity is based on the two methods:

`ostream& **PrintOn**(ostream &s)`

`istream& **ReadFrom**(istream &s)`
These methods have to be overridden to write or read the object's instance variables to or from a C++ stream. Example:

```
class Foo: public Object {
    int i;
    ObjList *ol;
public:
    ostream& PrintOn(ostream &s)
    {     Object::PrintOn(s);
          return s << i SP << ol SP;
    }
    istream& ReadFrom(istream &s)
    {     Object::ReadFrom(s);
          return s >> i >> ol;
    }
};
```

Instance variables that are pointers to other objects, `ol` in the example above, can be treated as all the other instance variables.

> ***Notice: In order to correctly store a pointer to an object it is necessary that the corresponding class conforms the ET++ coding conventions, e.g. it defines `MetaDef` and `MetaImpl` macro.***

ET++ takes care of linearizing (even circular) pointer structures. Both methods should always start with a call to the inherited method to store or read the instance variables of the superclass. Instance variables written to a stream have to be separated with some white space characters. As a shorthand notation for `<< " "` the macro `SP` can be used (`NL` can be used for `<< "\n"`). In addition to reading the instance variables of an object the `ReadFrom` method should initialize variables that have not been written out in the `PrintOn` method. If an instance variable is of one of the types enum, `bool` or `char*` it has to be treated specially. How to handle those instance variables is illustrated in the following example:

```
enum Flags {
    eflag1,
    eflag2
};

class Foo2: public Object {
    char *str;
    bool b;
    Flags f;
public:
    //...
    ostream& PrintOn(ostream &s)
    {     Object::PrintOn(s);
          PrintString(s, str);
          return s << b SP << f SP;
    }
    istream& ReadFrom(istream &s)
    {     Object::ReadFrom(s);
          ReadString(s, &str);
          return s >> Bool(b) >> Enum(f);
    }
};
```

The functions `PrintString/ReadString` take care of quoting characters which would conflict with the ET++ file format for objects.

#### 5.1.1.4. Copying Objects

There are two different methods to produce a copy of an object.

`Object *Clone()`
Returns a copy of this object without copying the references to other objects. All pointers point to the same objects as before.

`Object *DeepClone()`
`DeepClone` returns a copy of an object and recursively copies the references to other objects, e.g. a deep clone object does not share any pointer instance variables with the original object. The implementation of `DeepClone` in the class `Object` is based on the ET++ object input/output facility described in the next section. This method currently can't be overridden.

`Object *New()`
Returns a new (uninitialized) instance of the same class as `this`.

`virtual void InitNew()`
This is called when an object is created from the meta-class `New` method. It is overridden in subclasses to initialize variables as appropriate.

#### 5.1.1.5. Change Propagation

The class Object defines the framework to synchronize the state of different objects.

```
void AddObserver(Object*);
Object* RemoveObserver(Object*);
void Send(int id, int part, void *data);
void DoObserve(int id, int part, void *data, Object *);
```

This so called *change propagation mechanism* is modelled after the Smalltalk-80 changed and update principle. An object can announce its changes by calling the `Send` method. `Send` triggers a call to `DoObserve` for all objects that have been registered as dependent of `this` with `AddObserver`.

`Send` can be called with arguments to specify the reason for the change which will be passed to the `DoObserve` method. Refer to the file CmdNo.h for some predifined constansts which are used by ET++. `DoObserve` is implemented as an empty method in the class `Object`. In order to react on a change `DoObserve` has to be overridden.

`bool` **PrintOnWhenDependent**(Object *anObject)
This method is a hook that allows to control whether a dependency relationship of an object should be stored during a call to PrintOn. By default this method returns TRUE indicating that the dependency relationship between this and anObject should be stored.

#### 5.1.1.6. Programming Environment Support

The class `Object` defines several methods to be used in conjunction with the ET++ programming environment.

`void` **`Inspect`**`()`
Inspect starts the Inspector and displays the state of this.

`void` **`EditSource`**`(bool definition)`
Brings up an editor with the source code of this object; either the definition or the implementation, according to the parameter. It uses the information from the MetaDef and MetaImpl macros to find out where the file is.

`virtual void` **`InspectorId`**`(char *buf, int bufSize)`
This method can be overridden to give the Inspector more information about the object. A short description (e.g. the title of a window) should copied into buf, subject to the maximum size bufSize.

#### 5.1.1.7. Flags

The class `Object` provides some space to associate with each object a set of flags. These flags can be used instead of boolean valued instance variables.

The flags used by the `Object` class are:

| | |
|---|---|
| `eObjDefault` | Default state; no flags set. |
| `eObjIsDeleted` | Indicates that this object is one that has been deleted from a collection, but is still ``there'' in the collection for the benefit of iterators, etc. |
| `cObjNonDeleted` | The object has not been deleted from memory. |
| `cObjDelayChanges` | Changes to this object (indicated by calls to Send()) will be delayed until FlushChanges() is called. |
| `cObjVisited` | Used by the ObjectTable to indicate that this object has been visited on a traversal of the table. |
| `cObjIsProto` | Indicates that the object is one used by the metaclass system of ET++, and has been created by the MetaImpl macro. |

Apart from `eObjIsDeleted`, these flags are private to `Object`, and can be tested but not set by sub-classes. Many sub-classes add their own flags, which are stored in the same member variable.

Various methods to test and set flags.

```
void SetFlag(int f)
void ResetFlag(int f)
bool TestFlag(int f)
void InvertFlag(int f)
void MarkAsDeleted()
bool IsDeleted()
void SetVisited()
void ClearVisited()
```

`void` **`SetFlag`**`(int f, bool b)`
Sets the flag if b is TRUE; otherwise resets it.

ET++ uses the convention as shown below to avoid a conflict with a superclass that uses the same flag. For example the flags for the class `Class` a subclass of `Object` are:

```
enum ObjFlags {
    eObjIsDeleted = BIT(1),
    eObjLast = 1
};
    // class Class a subclass of Object
enum ClassFlags {
    eClassAbstract = BIT(eObjLast + 1),
    eClassLast = eObjLast + 1
};
```

`BIT(`*n*`)` is a macro defined in Types.h which returns a mask with bit *n* set.

### 5.1.1.8. Error Handling

ET++ allows a client to register a procedure to be called whenever a fatal or non-fatal error occurs. This facility is intended for error reporting and logging but not for error correction or recovery. An error handler is of the type:

```
void (*ErrorHandlerFunc)(int level, bool abort, char *location, char *msg);
```

`Level` indicates the severity of the error. Errors are categorized into:

| | |
|---|---|
| warning | 5.1.1.8.1. |
| error | 5.1.1.8.2. |
| system errors | 5.1.1.8.3. |
| fatal errors | > 3000 |

If `abort` is set to `TRUE` the application aborts after reporting the error. `location` indicates the method or function where the error occurred and `msg` stands for the corresponding error message. If only the collection classes are used without the application framework classes the function to be called on error conditions is set with `SetErrorHandler`. Otherwise the method `DoOnError` of `Application` can be overridden to report the error. The default implementation of this method shows an alert box with the options to ignore, abort, or to invoke the ET++ inspector. To call the installed error handler, use either:

```
void Warning(char *location, char *msgfmt, ...)
void Error(char *location, char *msgfmt, ...)
void SysError(char *location, char *msgfmt, ...)
void Fatal(char *location, char *msgfmt, ...)
```

All these functions provide a `printf` like interface to format the message. The class `Object` overloads these functions and prepends the class name to `location`.

The `location` should be the name of the method or function where the error was detected, and the following parameters provide a printf-like interface to format messages. For example:

```
Error("Open", "Cannot find %s.", fileName)
```

The class-name will be added to the location, so in this case the error message might read `File::Open: Cannot find xyz.c`.

`virtual void DoError(int level, char *location, va_list va)`
This is used by the above error functions.

`void AbstractMethod(char* method)`
This is used in abstract classes, where an interface to a method is defined, but the method should be implemented by a subclass. The abstract method just calls this function, with the name of the method, so if it is not overridden, an error will be generated.

`void MayNotUse(char* method)`
This generates a warning message that this object is not allowed to use `method`.

The functions below allow to set at which error level the application should be aborted (`SetAbortLevel`) and which messages should be ignored (`SetIgnoreLevel`). Both methods return the old level.

```
int SetAbortLevel(int newlevel); int SetIgnoreLevel(int newlevel);
```

### 5.1.2. The Collection Classes

The ET++ collection classes are modelled after the Smalltalk-80 collection classes. The inheritance relationship among the different classes is shown in the following table.

| | |
|---|---|
| `Object` | |
| **`Collection`** | abstract superclass for all collection classes |
| `Set` | hash table |
| `IdSet` | hash table based on the address of an object |
| `Dictionary` | data structure for storing key value pairs |
| `IdDictionary` | same as above, but the address of an object will be used as key |
| `ObjArray` | array of object pointers, with range checking and the possibility to grow or shrink |
| `SeqCollection` | abstract superclass for collections preserving the the order in which the objects were added |
| `ObjList` | doubly linked lists |
| `SortedObjList` | sorted list based on `Object::Compare()` |
| `OrdCollection` | Ordered Collection - array based implementation of a list |

The collection classes are written to work on instances of class `Object`. Some of the methods supported by all collections come in pairs, e.g. `Remove`, `RemovePtr`. The difference between these methods is that `Remove` uses `IsEqual` in comparisons and `RemovePtr` uses pointer identity. This distinction is important for collections with duplicate entries like `ObjLists` or `OrdCollections`, because `IsEqual` does not uniquely identify an object.

`Object* **Add**(Object*)`
Adds an `Object` to a collection and returns a pointer to an object actually in the collection.

`Object* **Remove**(Object*)`

`Object* **RemovePtr**(Object*)`
Remove an `Object`.

`Object* **Find**(Object*)`

`Object* **FindPtr**(Object*)`
Find an `Object` in a collection, returns 0 when the `Object` is not found

`bool **Contains**(Object*)`

`bool **ContainsPtr**(Object*)`
Test whether an `Object` is contained in a collection.

The elements of a sequenced collection are accessed by the `At()` method, and run `foo->At(0)`, `foo->At(1)`, ..., so the last entry is numbered ***one less than*** the value given by `Size()`.

#### 5.1.2.1. Memory Management

The collection classes are implemented by storing pointers to the objects and not the objects itself. Due to the possibility that an object is shared among several collections, the collection itself cannot decide when an object should be freed. For this reason all collections use the policy to manage only the storage for the collection data structures but not for the objects stored in the collection. The decision when to delete an object is up to the client of a collection. The destructor of a collection does not free the objects stored in the collection. The method `FreeAll` can be used to free the contents of a collection.

#### 5.1.2.2. Iterating over a `Collection`

The ET++ collection classes use the notion of an *iterator* to visit all the objects of a collection. An iterator is responsible to store the state of the traversal and to provide the method `operator()()` to skip to the next object of the collection. When all the objects of a collection have been visited `operator()()` returns 0. Every collection class provides an implementation of an `Iterator`. The abstract interface of all these iterators is defined in the class `Iterator`.

The class hierarchy for iterators:

| | |
|---|---|
| **`Iterator`** | abstract superclass for iterators |
| `SetIter` | iterator for `Set`s |
| `DictIter` | iterator for `Dictionaries` |
| `ObjListIter` | iterator for `ObjList`s |
| `RevObjListIter` | reverse iterator for `ObjList`s |
| `OrdCollectionIter` | iterator for `OrdCollection`s |
| `RevOrdCollectionIter` | reverse iterator for `OrdCollection`s |

Iterators are used as shown below:

```
ObjList shapes;
shapes.Add(new BoxShape);
//...
ObjListIter next(Shapes);
Shape *s;
Rectangle r;

while (s= (Shape*)next())
    s->Draw(r);
```

The type of the object returned from `operator()()` is `Object*`. For this reason it typically has to be casted to a more specific type before an operation can be executed.

An algorithm implemented to work for all kinds of collections which needs an iterator can request an instance of an iterator with the method `MakeIterator` from a collection. The returned instance is dynamically allocated and has to be freed by the programmer (to reduce the memory allocation overhead ET++ uses its own memory management for iterators). An example for such an algorithm is

`Collection::Contains` which is implemented only in terms of the abstract interfaces of `Collection` and `Iterator` and is therefore applicable to any collection.

```
bool Collection::Contains(Object *anObject)
{
    Iterator *next= MakeIterator ();
    Object *op;
    bool found= FALSE;

    while (op= (*next)())
          if (op->IsEqual(anObject)) {
                Found= TRUE;
                break;
          }
    delete next;
    return found;
}
```

It is error prone and results in ugly code to delete dynamically allocated iterators manually. For this reason ET++ provides as a kind of syntactic sugar an additional class `Iter` which takes care of freeing a dynamically allocated iterator object instance in its destructor.

```
bool Collection::Contains(Object *anObject)
{
    Iter next(this); // or: Iter next(GetIterator());
    Object *op;

    while (op= next())
          if (op->IsEqual(anObject))
                return TRUE;
    return FALSE;
}
```

A method can be applied to all objects of a collection with the macro:

`void ForEach(itemType,method)(arguments,...)`

Example:

```
Collection *col;
bool on, redraw;

col->ForEach(VObject,Enable)(on, redraw);
```

In this case `col` is a collection of `VObjects` and the method `Enable` is called for all its members with the arguments `on` and `redraw`.

> ***`ForEach` is a macro which expands into several statements without scope delimiters `{ }`. Enclose `ForEach` with `{ }` where it becomes necessary.***
>
> ***In addition `ForEach` constructs two variables by concatenating the `itemType` and `method` parameters. So don't use any whitespace in `ForEach`'s parameter list.***

In order to assert that all items of a collection are of a certain type `AssertClass(ClassName)` can be used.

### 5.1.3. Built-in Types and Collections

The collection classes can only manage instances of classes derived from `Object`. In order to store the built-in types float or int, ET++ provides the classes `ObjInt` and `ObjFloat`. These classes have as instance variable a number of the corresponding type and implement the abstract methods defined in `Object`.

### 5.1.4. Strings

The current version of ET++ does not include a class `String` for string manipulation and only offers some utility functions (/src/String.h).

### 5.1.5. Point and Rectangles

The classes `Point` and `Rectangle` should be used whenever coordinates have to be manipulated. Predefined `Points` and `Rectangles` are:

```
const Point  gPoint0(0,0),
             gPoint1(1,1),
             gPoint_1(-1,-1),
             gPoint2(2,2),
             gPoint4(4,4),

const Rectangle gRect0(gPoint0, gPoint0)
```

Using this constants reduces the binary size of an application.

# 6. Interaction Classes

## 6.1. EvtHandler

This is the abstract superclass for all objects that handle events such as input from the user.

Each `EvtHandler` may have a pointer to another `EvtHandler` which is to handle all the events that this one is not interested in. So, a shape on the screen might be dragged around with the left mouse button, but pass the right mouse button on to the Document object, which would pop up the application menu.

This table describes the *meaning* of each of the member functions of `EvtHandler`; most of the implementations simply pass the call on to the next handler, if there is one.

`virtual EvtHandler *`**`GetNextHandler`**`()`
Return the next `EvtHandler`, which will handle any events this one does not handle.

`virtual Menu *`**`GetMenu`**`()`
Return the menu for this `EvtHandler`. This call will get passed out to the next enclosing `VObject` that has a menu to pop up.

`virtual void` **`DoCreateMenu`**`(Menu*)`
Called once, when the menu is first popped-up. This method is to allow menu items to be inserted by successive `EvtHandlers` in a chain;for example most application `Views` will first call the superclass method, then add a few items of their own.

`virtual void` **`DoSetupMenu`**`(Menu*)`
Called each time the menu is popped up, so that menu items can be enabled or disabled according to the circumstances when the mouse is pressed.

`virtual Command *`**`DoMenuCommand`**`(int)`
Called when an item on the menu is selected by the user. The parameter is the number assigned to the menu item when it was created.

`virtual void` **`PerformCommand`**`(Command*)`
This is the method that applies undo-able Commands. Usually, it is passed along the `nextHandler` chain to the `Document`, which does the work.

`EvtHandlers` that want to react to particular mouse buttons should override these, returning a `Command` object of the appropriate type. For example, a shape in a drawing program might return a `VObjectMover` when the left button is pressed. See `Command` for more information.

```
virtual Command *DoKeyCommand(int, Point, Token)
virtual Command *DoCursorKeyCommand(EvtCursorDir, Point, Token)
virtual Command *DoFunctionKeyCommand(int, Point, Token)
```

Similar to the mouse-button functions, but for keyboard events.

`virtual Command *`**`DoIdleCommand`**`()`
This is called when the system is idle, so that deferred updates can be flushed. For example, `TextViews` don't re-format until the system is idle.

`virtual Command *`**`DoOtherEventCommand`**`(Point, Token)`
Anything else not covered above.

`virtual void` **`Control`**`(int id, int part, void *val)`
This is used by things such as buttons and check-boxes to tell the parent (most often a `Dialog`) that it has been changed.

`virtual Command *`**`DispatchEvents`**`(Point, Token, Clipper*)`
This examines the `Token` that comes from the system, and calls one of the `Do...Command` routines.

It can be overridden in order to catch particular events, such as pressing the `Return` key, so that a subclass of EvtHandler can do something special.

## 6.2. VObject

`VObject` is an abstract superclass for all objects that are displayed on the screen.

### 6.2.1. Alignment

Many `VObject` subclasses use the `VObjAlign` enumeration to specify how objects should be aligned with respect to each other. Possible alignment specifications are:

| | |
|---|---|
| `eVObjHLeft` | flush left |
| `eVObjHCenter` | centered horizontally |
| `eVObjHRight` | flush right |
| `eVObjHExpand` | expand horizontally as required |
| `eVObjVTop` | at the top |
| `eVObjVBase` | align the bases of the objects |
| `eVObjVCenter` | centered vertically |
| `eVObjVBottom` | at the bottom |
| `eVObjVExpand` | expand vertically as required |

### 6.2.2. Instance Variables

`short` **`frameId;`**
This is a unique identifier, used to identify each `VObject` to other parts of the system. For instance, a Dialog will give each of its component parts (push-buttons, text, etc) a number, so that when the user does something, it can find out which element was involved.

If a `VObject` is not given a specific frameId, it is set to `cIdNone = -1`.

`VObject *`**`container;`**
The containing `VObject`, such as the `Window`, or `Cluster` that this `VObject` is part of.

`Rectangle` **`contentRect;`**
The enclosing rectangle of this `VObject`.

### 6.2.3. Public Methods

```
VObject(EvtHandler *next, Rectangle r, int id= cIdNone)
VObject(Rectangle r, int id= cIdNone)
VObject(int id= cIdNone)
```

Constructors to set up various of the `nextHandler` (which is in the `EvtHandler` superclass), `contentRect` and `frameId` member variables.

`BlankWin *GetWindow()`
This recurses up the chain of containers, until it finds a `VObject` that has no container, which must be the containing window.

`Point GetPortPoint(Point p)`
This transforms a point into the coordinate system of the port, by recursively calling `ContainerPoint` until there are no more containers.

`virtual Rectangle GetViewedRect()`
Return the rectangle which this `VObject` occupies. This is the `contentRect`, except for `Clipper`s.

`EvtHandler *GetNextHandler()`
This overrides EvtHandler's `nextHandler` variable, and returns the `container`.

`virtual void SetContainer(VObject*)`
Set the `container` member variable.

`virtual VObject *GetContainer()`
The `GetContainer` method returns the container.

`virtual void AddToClipper(Clipper*)`
This is called when a `VObject` is placed inside a `Clipper`, to set the view and container, and to set the `VObject`'s origin to zero.

`virtual void RemoveFromClipper(Clipper*)`
This is called when a `VObject` is removed from a `Clipper`.

```
int GetId()
void SetId(int id)
```

Get and set the `frameId` member variable.

```
virtual void Enable(bool b= TRUE, bool redraw= TRUE)
void Disable(bool redraw= TRUE)
```

Enable or disable the `VObject`, and re-draw it if `redraw` is `TRUE`. `Disable` calls Enable with `b` set to `FALSE`. `Enable` can be overridden to do things like greying-out selections when disactivated.

```
bool Enabled()
bool IsOpen()
```

Test if the `VObject` is enabled or open.

`virtual Metric GetMinSize()`
Return the minimum size of the `VObject`. See `Metric`, for more information.

```
Rectangle ContentRect()
Point GetExtent()
Point GetOrigin()
int Width()
int Height()
void SetContentRect(Rectangle, bool)
virtual void SetExtent(Point)
virtual void SetOrigin(Point)
void SetWidth(int w)
void SetHeight(int h)
```

Methods to access the `contentRect` member variable.

`virtual int Base()`
Return the base of the `VObject`, for alignment purposes. For simple `VObjects`, this is the y-coordinate of the `contentRect`, whilst for text, this is the baseline, below which letters like '`y`' descend.

`void Move(Point delta, bool redraw= TRUE)`
Move the `VObject` a distance of `delta`, by adding this to the `contentRect`. Force re-drawing of both the old and new positions if `redraw` is `TRUE`.

`void CalcExtent()`
In `VObject`, this just sets the extent to the minimum size, but it can be over-ridden to do more interesting things if necessary. It is called by composite `VObjects` such as `Dialog` and `CollectionView`.

`virtual void Open(bool mode= TRUE)`
Open or close the `VObject`, depending on the value of `mode`. In `VObject`, this just changes the setting of the `eVobjOpen` flag.

`void Close()`
This just calls `Open(FALSE)`.

`void Align(Point at, Metric m, VObjAlign a)`
Move the origin of the `VObject` to align with the given point, according to the alignment specification `a` and metric `m`. See `Metric`, for more information.

`virtual bool ContainsPoint(Point p)`
Returns `TRUE` if the rectangle of the `VObject` contains the point, otherwise return `FALSE`

`virtual void SetFocus(Rectangle r, Point p)`
Set the drawing port to the one for this `VObject`, set its origin to `p` and clip drawing commands to the `r`. This gets passed out along the container chain to the enclosing `Window`, which does the work.

`void Focus(Rectangle r)`
This calls `SetFocus` with the Point(0,0).

`void Focus()`
This calls `SetFocus(GetViewedRect(), gPoint0)`.

`void` **`Print`**`()`
This puts up a print dialog which will let the user print out this `VObject` to disk or to the printer, or cancel the print operation.

```
virtual void DrawAll(Rectangle)
virtual void DrawHighlight(Rectangle)
virtual void Draw(Rectangle)
```

Various methods which can be overridden to provide the drawing behaviour of `VObject` subclasses.

`virtual void` **`Outline2`**`(Point p1, Point p2)`
Draws the outline of a rectangle between `p1`} and `p2`}.

`void` **`Outline`**`(Point delta)`
Draws the outline of the `contentRect` offset by `delta`.

`void` **`OutlineRect`**`(Rectangle r)`
Draws the outline of `r`.

`void` **`Feedback`**`(Rectangle r, bool on, bool b)`
This is called continuously as a `VObject` is dragged around the screen or re-sized. It moves the `VObject` to the position of the rectangle `r`, and updates the screen by erasing the old position and calling `DrawAll` in the new position.

`virtual void` **`Highlight`**`(HighlightState)`
Highlight the `VObject`. The default action is to invert the whole of the content rectangle. `HighlightState` can be either Off or On, but it is ignored in the default implementation.

`virtual void` **`InvalidateRect`**`(Rectangle r)`
State that the given rectangle is invalid, and that it will have to be re-drawn. This is passed out to the enclosing window, which passes it to the `WindowPort`. Invalid rectangles are accumulated until the window gets an Update event.

`void` **`ForceRedraw`**`()`
This invalidates the content rectangle, forcing the contents to be re-drawn at the next update.

`virtual void` **`UpdateEvent`**`(bool batch= gBatch)`
This is passed out to the window, which will do any updating that is pending. Update events are generated after each input token is processed by a window, or they can be called directly.

`virtual GrCursor` **`GetCursor`**`(Point p)`
This returns the appropriate cursor for this `VObject`. The default is an arrow pointing north-west; things like scrollbars override this to change the cursor to other shapes as it enters their content rectangle.

`virtual Command *`**`Input`**`(Point lp, Token t, Clipper *vf)`
This passes an input Token to the `DispatchEvents` method if the `VObject` is enabled.

`Command *`**`DispatchEvents`**`(Point, Token, Clipper*)`
This extends the `EvtHandler` behaviour by changing the mouse cursor shape on entry and exit events, and calling up the Inspector when the control, left and shift buttons are held down during a mouse click.

`virtual Point `**`ContainerPoint`**`(Point)`
This transforms a point into the co-ordinate system of the container. For most `EvtHandler`s, this just returns the point itself, but for a `Clipper`, it returns the point plus the `Clipper`'s offset.

```
virtual Command *DoLeftButtonDownCommand(Point, Token, int)
virtual Command *DoMiddleButtonDownCommand(Point, Token, int)
```

`Command *`**`DoRightButtonDownCommand`**`(Point, Token, int, Clipper*)`
Puts up the menu returned by `GetMenu`, if any.

`virtual Command *`**`TrackInContent`**`(Point, Token, Command*)`
`EvtHandler` just returns the `Command` passed to it; `Clipper` does more complex things.

`virtual VObject *`**`Detect`**`(BoolFun f, void *arg)`
Return the first entry where `f` returns `TRUE`. For single `VObjects`, this is either the object itself, or `NULL`, but for composite `VObjects` it iterates over the collection.

`VObject *`**`FindItem`**`(int id)`
Find a `VObject` with a frame Id of `id`, using `Detect`.

`VObject *`**`FindItem`**`(Point p)`
Find a `VObject` which contains the point `p`.

`VObject *`**`FindItem`**`(VObject* g)`
Find a `VObject` which `IsEqual` to `g`.

`VObject *`**`FindItemPtr`**`(VObject* g)`
Find a `VObject` based on pointer identity.

`bool `**`WantsInputFocus`**`()`
Tests if the `VObject` wants the input focus.

`virtual Command *`**`GetMover`**`()`
Returns a `VObjectMover` command that will allow the user to drag the `VObject` around with the mouse. If the `VObject` is in a `View`, the dragging is constrained to the `View`'s rectangle.

`virtual Command *`**`GetStretcher`**`()`
Returns a `VObjectStretcher` command that lets the user stretch the shape of the `VObject` with the mouse, constrained to the rectangle of the `View`, if there is one.

`char *`**`AsString`**`()`
Returns a tilde sign “~” (tilde).

`int` **`Compare`** `(Object*)`
Calls `strcmp` on the `AsString` result of the two objects.

## 6.3. CompositeVObject

`CompositeVObject` is a `VObject` that contains a `Collection` of other `VObjects`. Mostly, the methods of `VObject` are applied in turn to each of the contained `VObjects`.

`CompositeVObject` also implements a number of `Collection` methods, to make it easier to access the contained `VObjects`.

### 6.3.1. Instance Variables

`bool` **`modified`**
Set to `TRUE` whenever the list of `VObjects` changes. This is used by Cluster to tell if it should re-calculate its dimensions.

`Collection *`**`list`**
The list of contained `VObjects`.

### 6.3.2. Public member functions

```
CompositeVObject(int id= cIdNone, Collection *cp= 0)
CompositeVObject(int id, VObject*, ...)
CompositeVObject(int id, va_list ap)
```

Constructors that set the `frameId`, and may pass a a collection or a zero-terminated argument list of `VObject`s to put in the `list`.

`void` **`SetModified()`**
Set the `modified` member variable to `TRUE`.

`Collection *`**`GetList()`**
Return a pointer to the collection of contained `VObjects`.

`Iterator *`**`MakeIterator()`**
Return an Iterator for the collection of contained `VObjects`.

```
int Size()
virtual void Add(VObject*)
VObject *Detect(BoolFun, void *arg)
VObject *At(int n)
```

Various Collection methods that are passed straight through to the `list`. See `Collection`, for more information.

`void` **`FreeAll()`**
This deletes all the contained `VObjects` in the list, then deletes the list.

`Command *DispatchEvents(Point, Token, Clipper*)`
Tries each `VObject` in the list in reverse order to see if it is interested in this Token; otherwise calls `VObject::DispatchEvents`.

`void SetItems(va_list ap)`
Add the arguments passed (which should be `VObjects`) to the list of items contained by this `CompositeVObject`.

```
void Open(bool mode= TRUE)
void Enable(bool b= TRUE, bool redraw= TRUE)
void Draw(Rectangle)
void SetContainer(VObject*)
void SetOrigin(Point)
void SetExtent(Point e)
```

These are all methods of `VObject`, implemented here by calling the `VObject` version and then calling the method on all of the contained `VObjects` in turn.

`Metric GetMinSize()`
Merges together the minimum sizes of all the contained `VObjects`.

`void InspectorId(char *buf, int sz)`
Calls `InspectorId` on the first `VObject` in the list, if there is one; otherwise calls `VObject::InspectorId`.

## 6.4. Cluster

The constructors all take an identifying number, and an alignment specification. The alignment specifications are explained in `VObject`. In the next release the `Cluster` will be merged with the `Expander`.

## 6.5. Scroller

`ScrollDir` is an enumeration that controls the way scrollbars appear on the sides of `Scrollers`.

| | |
|---|---|
| eScrollNone | no scrollbars |
| eScrollRight | left-right |
| eScrollDown | up-down |
| eScrollLeft | left-right |
| eScrollUp | up-down |
| eScrollHideScrolls | hide the scrollbars |
| eScrollDefault | left-right and up-down |

## 6.6. Menus

Menus are maintained with the following methods of class `EvtHandler`:

```
virtual Menu *GetMenu();
virtual void DoCreateMenu(Menu*);
virtual void DoSetupMenu(Menu*);
virtual Command *DoMenuCommand(int);
```

`Menu *GetMenu()`
is called whenever the right mouse button is pressed in order to bring up a menu. When overridden `GetMenu` should return a menu pointer held in an instance variable of `EvtHandler` or a subclass thereof. The default implementation calls `GetNextHandler()->GetMenu()`.

`void DoCreateMenu(Menu*)`
is called once for a new menu returned from `GetMenu`. When overridden it should call the inherited `DoCreateMenu` and then install its own new menu entries in the menu given as parameter. The following code fragment is a typical example:

```
void aSubClass::DoCreateMenu(Menu *menu)
{
    aBaseClass::DoCreateMenu(menu);
    menu->Append(new MenuLineItem);
    menu->Append(new TextItem(cMENUCMD1, "command 1"));
    menu->Append(new TextItem(cMENUCMD2, "command 2"));
}
```

In this example a `MenuLineItem` and two `TextItem`s are appended to the menu (it is possible to append any `VObject` to a menu). The following code shows a shorter notation for this common case:

```
void aSubClass::DoCreateMenu(Menu *menu)
{
    aBaseClass::DoCreateMenu(menu);
    menu->AppendItems("-",
                      "command 1", cMENUCMD1,
                      "command 2", cMENUCMD2,
                      0);
}
```

`Command *DoMenuCommand(int id)`
Every menu item should have a unique id which is given to `DoMenuCommand` when a menu command is selected. Standard ids can be found in CmdNo.h. Application ids should start with id `cUSERCMD`.

Example:

```
Command aSubClass::DoMenuCommand(int id)
{
    switch (id) {
    case cMENUCMD1:
          return new MenuCmd1;
    case cMENUCMD2:
          return new MenuCmd2;
    default:
          return aBaseClass::DoMenuCommand(id):
    }
}
```

`void DoSetupMenu(Menu *m)`
Initially all menu items are disabled (greyed out). You must override `DoSetupMenu` in order to enable all currently selectable menu items. `DoSetupMenu` is called whenever a menu is going to be opened.

```
void aSubClass::DoSetupMenu(Menu *m) {
    aBaseClass::DoSetupMenu(m);
    if ( possible to choose cmd 1 )
          m->EnableItem(cMENUCMD1);
    if ( possible to choose cmd 2 )
          m->EnableItem(cMENUCMD2);
}
```

### 6.6.1. DialogViews and Dialogs

A `DialogView` implements a standard behavior for a `View` dealing with different dialog items. It maintains an active text and allows to cycle through all editable text items with the TAB key. In addition it identifies a default item (a special `ActionButton`) which can be activated with the RETURN key.

`VObject *DoCreateDialog()`
This method is called once to create and return the tree of dialog items (`VObjects`, `CompositeVObjects`) making up the dialog. Output only dialog items like `TextItems` and `ImageItems` can be found in VObject.h. Grouping of several items can be achieved with `Clusters` or `Expanders`. Various input sensitive items like `ToggleButtons`, `ImageButtons`, `ActionButtons`, and `RadioButtons` are defined in DialogItems.h. In addition DialogItems.h contains the specialized clusters `OneOfCluster` and `ManyOfCluster` to implement a *one of* and a *many of* behavior of several buttons.

More complex dialog items can be build by installing a `View` in a `Clipper`, in a `Scroller`, or even in a `Splitter`. For example to implement a scrollable list of arbitrary `VObjects` it is only necessary

to build a Collection of VObjects, install the Collection in a CollectionView and put the View in a Scroller.

Another example is a TextView inside a small Clipper to implement an editable text line. The predefined class EditTextItem implements this often used abstraction and can be found in DialogItems.h.

Every dialog item must have a unique identifier (id) in order to be able to identify any actions performed on it. Items with a standard behavior (e.g. ok-, cancel-, and default-buttons) should have one of the predefined ids found in CmdNo.h. User defined items should use ids above cIdFirstUser. Output only items like Clusters or Expanders should have id cIdNone.

`void` **`Control`**`(int id, int part, void *data)`
Whenever an action is performed on a dialog item Control is called giving the items's id (id), a so called part code (part) and any additional information (data). The part code identifies the action performed on the item. For example when single clicking in a CollectionView Control is called with parameter part == cPartCollSelect. In this case data is a pointer to a Rectangle, which gives the selected item or a range of selected items. Other predefined part codes can be found in CmdNo.h. The inherited Control method should be called whenever an action dismisses the Dialog, e.g. for ids cIdOk, cIdYes, cIdNo, cIdCancel, cIdDefault.

A subclass of DialogView, Dialog automatically creates and installs itself in a window, thereby implementing modal and modeless dialog boxes. In addition it implements the behavior to close the window whenever an ActionButton is pressed and it factors out the behavior for undoable dialog boxes by overriding Control.

`void` **`DoSetDefaults`**`()`
is called after the dialog tree is created the first time or an ActionButton returns the id cIdDefault. In the latter case the dialog remains opened even if the inherited Control is called.

`void` **`DoSetup`**`()`
must be overridden to enable and disable dialog items according to other selected options. It is called whenever the dialog is opened.

`void` **`DoSave`**`()`
must be overridden to save the state of all dialog items in order to be able to undo all settings when the Cancel button is pressed or an item returns id cIdCancel.

`void` **`DoRestore`**`()`
is called whenever the Cancel button is pressed to restore the dialog items to their previous state.

`void` **`DoStore`**`()`
is called whenever the OK button is pressed.

You find examples for implementing dialogs in FileDialog.C, PrintDialog.C, FindDialog.C, GotoDialog.C, and POSTSCRIPT/PostScript.C.

## 6.7. EditTextItem

This is a DialogItem that allows some text to be edited. It uses a `TextView` to do the displaying.

### 6.7.1. Public member functions

**`EditTextItem`**`(int id, char* initText, int width, int lines)`
This constructs an `EditTextItem` with `initText` as the initial contents, of size width by lines.

**`EditTextItem`**`(int id, TextView *tv, int w, int l)`
This is the constructor for when you have a `TextView` already.

```
void Init(TextView *tv, int width, int lines, char *it)
Metric GetMinSize()
int Base()
void SetNoSelection()
void SetSelection(int from= 0, int to= cMaxInt, bool redraw = TRUE)
```

`Text *`**`GetText()`**
Return the current text displayed. In order to get the string, call `GetText()->AsString()`.

```
int GetTextSize()
virtual bool Validate()
```

Prepare to take input from the keyboard, by re-setting the selection.

# 7. Window System Interface

The graphic primitives and the corresponding constants can be found in the following files in /src

Drawing/Cursors (Port.h):
This file exports constants for patterns, rasterops, linecaps, polygon types and cursors together with all the drawing primitives.

Fonts (Font.h)
This file exports the definitions and constants related to font management. The meaning of some standard fonts defined in this file are:

| | |
|---|---|
| `gSysFont` | font to be used in menus, titlebars etc. |
| `gApplFont` | default font of an application |
| `gFixedFont` | a fixedwidth font |

The size of these fonts is can be set in the environment variable `ET_FONT_SIZE`. 12 point fonts will be used by default. A font is created with function `new_Font(fontid, size, face)`.

Bitmaps (Bitmap.h)

The example below illustrates how to create a static bitmap:

```
static short UpArrowBits[]= {
#   include "images/UpArrow.image"
};
static Bitmap ArrowUp(Point(16,16), UpArrowBits);
```

The file images/UpArrow.image contains a list shorts and is typically generated on SUNs with iconedit(1). Iconedit allows only to create either 16x16 or 64x64 bitmaps, in order to extract a subregion of such a bitmap the utility bmcut (/util/bmcut) can be used.

Events (Token.h):

This file contains the definition of a class `Token`, which used to describe window system events.

## 7.1. Drawing on the screen

The methods used to actually draw things on the screen are declared in Port.h. The kinds of things that can be drawn are: lines, rectangles, rounded rectangles, ovals, wedges, bitmaps, polygons, text and pictures.

Each kind of object can be filled and stroked with a so called *Ink*. Ink is the common superclass of Colors (RGBColor), Pattern (Bitmaps). Four static instances of Ink represent the special “colors” None, Xor, Black and White. All Inks can be used for all graphic primitives.

The ET++ color model is very new and not very well implemented. But it works and its interface will not change dramatically in coming versions.

To use colors in an application build a palette like the following example:

```
OrdCollection *pal= new OrdCollection;
pal->Add(gInkNone);                    // transparent
pal->Add(gInkWhite);                   // white
                              // (more efficient than RGBColor(1.0)
pal->Add(gInkBlack);                   // black
                              // (more efficient than RGBColor(0.0)
pal->Add(new RGBColor(0.25));          // dark grey
pal->Add(new RGBColor(0.50));          // medium grey
pal->Add(new RGBColor(0.75));          // light grey
pal->Add(new RGBColor(255, 0, 0));     // red
pal->Add(new RGBColor(0, 255, 0));     // blue
pal->Add(new RGBColor(0, 0, 255));     // green
```

Using a color:

```
GrPaintRect(r, pal->At(8));
```

The fourth optional parameter of RGBColor specifies the “precision” of the color. 0 gives you the nearest matching color without allocating new color cells. 255 gives you an exact match (if possible).

These values are a hack! In the next version the precision factor will specify the tolerated distance between the requested and returned color. So the meaning of 0 and 255 will swap!

Exact colors can be changed dynamically with the method SetRGB(r,g,b). If the method returns TRUE the change failed and you must redraw the part of your image (possible on screen without color map or monochrome screens). On monochrome screens colors are approximated by halftoning (ordered dither). Look at the example application “color”. Try the application on a monochrome screen!

For compatibility reasons the following symbols are still supported in this release (but they are Ink, RGBColor and Bitmap pointers, not enums!):

```
ePatNone
ePatWhite
ePatBlack

ePatGrey87 ← percentage of blackness in the grey.
ePatGrey75
ePatGrey60
ePatGrey50
ePatGrey40
ePatGrey25
ePatGrey12
```

and `ePat00` to `ePat15`, which are 16 useful patterns.

The current pattern is set with:

```
GrSetPattern(GrPattern p)
```

The current text attributes are set with these functions:

```
GrSetFont(Font *fp)
GrSetFamily(GrFont f)
GrSetSize(int s)
GrSetFace(GrFace f)
GrSetTextPattern(Ink *)
```

Text is drawn with these functions:

```
GrDrawChar(byte c)
GrDrawString(byte *s, int l= -1)
GrShowString(FontPtr fd, GrPattern pat, GrMode mode,
          Point pos, byte *text, int l= -1)
```

`GrDrawChar` and `GrDrawString` draw from the current text position, which is accessed with these functions:

```
GrTextMoveto(Point p)
GrTextAdvance(int h)
Point GrGetTextPos()
```

There is also a drawing current position, which can be changed with `GrMoveto(Point p)`. Lines can be drawn from the current position with `GrLineto(Point p)`, or lines can be drawn between two points with `GrLine(Point p1, Point p2)`.

Other drawing commands:

```
Rectangles        GrFillRect(Rectangle r)
Round rectangles  GrFillRoundRect(Rectangle r, Point dia)
Ovals             GrFillOval(Rectangle r)
Wedges            GrFillWedge(Rectangle r, int s, int e)
Bitmaps           GrShowBitMap(Rectangle r, Bitmap *bmp)
Polygons          GrFillPolygon(Point at, Point *pts, int npts, GrPolyType t)
Pictures          GrShowPicture(Rectangle *r, Picture *pic)
```

# 8. Application Framework Classes

The following discussion of the basic application framework classes `Application`, `Document`, `View`, `Window`, and `Command` is intended as a conceptual introduction to the most important aspects of ET++ and of an application framework (e.g. MacApp) in general. Some of these classes originate from MacApp and therefore have a similar behavior and interface.

Upon starting an ET++ application the main program creates an `Application` object and calls the `Run` method for that object in order to hand control to ET++. When the user wants to create a new document or open an existing one ET++ puts up the appropriate dialogs and calls a method of the application object to create a new `Document` object.

A `Document` object contains an applications's data and provides methods to save the data on disk and read it back into memory. In addition it creates `Windows`, and defines the window layout by creating `Expanders`, `Clippers`, or `Scrollers` and installs the corresponding `Views`. `View` objects provide drawing surfaces on which the document's data or other information is displayed. It also forwards input events to interaction objects within the `View` and knows how to print its surface onto a printer.

A `Clipper` is a rectangular area of the window that shows part of a `VObject` (a `View` is a subclass of `VObject`) and knows how to scroll that part of a `VObject`.

A `Window` object controls the document's window on the screen and manages all operations pertaining to windows including opening, closing, resizing, moving, and redrawing. All the classes mentioned above are subclasses of the class `EvtHandler` which defines abstract methods to react on input events. The default implementation forwards them to another `EvtHandler`.

The `Command` class provides a convenient framework for implementing undoable commands. It is subclassed for every command of an application to add fields that will maintain the state necessary to do, undo and redo the command, and to implement the corresponding methods `DoIt`, `UndoIt` and `RedoIt`. Calling these methods is done completely under the control of ET++.

The following sections describe some mechanisms related to the application framework classes.

## 8.1. Storing/Loading Documents

To store or load the contents of a document in a file the methods `DoRead` and `DoWrite` have to be overridden in a subclass of `Document`.

```
void DoRead(istream &, FileType *ft); void DoWrite(ostream &, int option);
```

The implementation of these methods typically uses the object input/output facility to transfer the contents of a document to a C++ stream (FileType is explained below). If the standard ET++ format is used to store a document both methods should first call the inherited `DoRead`/`DoWrite` methods. These inherited methods take care of writing and reading an additional header line stored together with the document. The header line consists of the external document type (e.g. `DRAW, VOBTEXT`) and the name of the application which generated the document. If a document has to be stored as a pure ASCII document without the header line the calls of the inherited method have to be omitted. The external document type is associated with a document in the class `Document`'s constructor:

**`Document(const char *extDocType)`**
This external document type is used whenever ET++ has to decide whether an application can handle the contents of a file. A set of predefined external document or file types can be found in FileType.h.

An application handling only a single type of documents hands their external type name over to ET++ in the constructor of the class `Application`.

**`Application`**`(int argc, char *argv, const char *mainDocType, char *opts)`
Every application that handles several document types must override the method:

`bool` **`CanOpenDocument`**`(FileType *ft)`
ET++ calls this method to test whether an application can handle the contents of a file. `Filetype` is a class giving access to status information about a file. Its member functions are:

`const char *`**`Type`**`()`
returns the external document type associated with the file

`char *`**`Creator`**`()`
returns the name of the ET++ application which generated the file, 0 indicates that the creator is unknown.

`long` **`SizeHint`**`()`
returns the size of the file in bytes (-1 indicates that no information about the size available)

`bool` **`IsAscii`**`()`
returns whether the file is an ASCII file or not.

The following example shows how `CanOpenDocument` is implemented for an application handling ASCII files and files with the external type `DRAW`.

```
bool myApplication::CanOpenDocument(FileType *ft)
{
    return strismember(ft->Type(), "DRAW", cDocTypeAscii, 0);
}
```

`Strismember` is a utility function which checks whether the first argument is in the set of strings specified in the following zero terminated argument list.

The method `Document::CanLoadDocument` can be overridden to specify which subset of the external file types specified in `CanOpenDocument` an already open document can handle, e.g. can replace its current contents.

A `Document` class which can read its contents from a file stored in different formats uses the additional `FileType` parameter passed by ET++ to the `DoRead` method in order to perform the necessary conversions. Example:

```
void myDocument::DoRead(istream &s, FileType *ft)
{
    if (strcmp(ft->Type(), "DRAW") == 0)
          // read document stored in DRAW format
    else if (ft->IsAscii())
          // read the contents from an ordinary ascii file
    else  Error("Cannot Read files of type %s", ft->Type());
}
```

The mechanism to import the contents of another document or a file is based on the two methods:

```
bool CanImportDocument(FileType *ft);
Command *DoImport(istream &s, FileType *ft);
```

ET++ calls `CanImportDocument` for the first time with `ft` set to zero, only if this call returns `TRUE` the import command will be inserted into the pop-up menu. `Import` returns a command object which can be used to provide undoable import commands.

### 8.1.1. Creating the Initial Window Layout

In addition to storing the data of an application on disk, the class Document is responsible to create an initial window layout for a document.

`Window *DoMakeWindows()`
This method has to be overridden in subclasses of `Document` to create the `Window` and to install a document's views therein.

A `View` is typically not installed directly into a `Window` but combined with one of the following classes:

`Clipper`
: instances of `Clipper` show a rectangular cutout of a `View` and establish a clipping boundary

`Scroller`
: in addition to the functionality provided by the class `Clipper` instances of `Scroller` surround a `View` with scrollbars.

`Splitter`
: instances of `Splitters` allow to split a `View` in up to four panes, showing disconnected portions of the `View`.

In the simplest case a `Window` shows only one `View` through a `Scroller` as illustrated in the following example:

```
Window *myDoc::DoMakeWindows()
{
    myView *view= new myView(this, /* ... */);
    return new Window(this, Point(560, 400), eBWinDefault,
          new Scroller(view)
    );
}
```

The first argument of the class `Window`'s constructor is the object which will receive events not handled by the window itself, eg. the next event handler. The `Point` parameter specifies the window's extent. The third argument is used to specify the desired behaviour of the window. The set of flags that can be specified are:

| | |
|---|---|
| eBWinOverlay | used in menus or alerts to give a hint to the underlying window system to optimize the damage handling of a window. |
| eBWinBlock | used for modefull dialogs, setting this flags preempts any input in another window of this process |
| eBWinFixed | the extent of the window cannot be changed by the user |
| eWinDestroy | clicking on the close box will destroy the window |
| eWinCanClose | clicking on the close box will hide the window; this option is typically for windows of modeless dialogs |
| eBWinDefault | used for application windows |

The last argument is the `VObject` representing the window's contents. In this case a new `Scroller` is created and the `View` is passed as first argument to the `Scroller`. To exchange the `Scroller` with a `Clipper` or a `Splitter` only the call `new Scroller` has to be replaced with the corresponding class name.

The class `Expander` is used to describe more complex window layouts. `Expanders` distribute the available space evenly among their contained `VObjects`. The following example illustrates how an `Expander` is used to show two `Scrollers` side by side in one window. When this window is resized the layout is recalculated and each `Scroller` will be assigned the same extent.

```
Window *TwoViewDoc::DoMakeWindows()
{
    myView *view1= new TextView(/*...*/);
    myView *view2= new TextView(/*...*/);

    return new Window(this, Point(700, 400), eBWinDefault,
          new Expander(eHor, gPoint2,
                new Scroller(view1),
                new Scroller(view2),
          0)
    );
}
```

The first argument of the `Expander`'s constructor specifies whether the available space should be subdivided either in the horizontal (`eHor`) or vertical (`eVert`) direction. The `Point` argument defines the gap to be used between the `VObject`s following in the zero terminated variable argument list.

A `VObject` can further control the layout mechanism of `Expander`s by setting the flags `eVObjHFixed` or `eVObjVFixed`. If one of these flags are set an `Expander` will not change the size of the `VObject` in the corresponding direction. `Expander`s can be nested to arbitrary depth.

A document using several windows can create them in `DoMakeWindows()`. Each of them is added to the list of a document's windows with a call to `AddWindow()`. The window returned from `DoMakeWindows` is considered as the base window of the document. ET++ automatically updates the base window's title bar with the document name and the initial position of the icon is set to its top left corner.

### 8.1.2. Commands

Every `Command` objects returned from an event handler method is assigned a name and an numerical identifier. The name is used in the undo menu entry to show which command can be undone. The name, identifier and some flags are associated with a command in its constructor.

> ***Never assign the id 0 to an undoable command.***

The following flags can be set in the constructor of commands:

| | |
|---|---|
| `eCmdCanUndo` | command is undoable (default) |
| `eCmdCausesChange` | command modifies the contents of a document (default) |
| `eCmdMoveEvents` | provides for mouse up drawing, e.g. TrackMouse !will be called until this flag is reset |
| `eCmdIdleEvents` | call TrackMouse even when there is no mouse activity |
| `eCmdFullScreen` | allows tracking and drawing on the full screen |
| `eCmdDoDelete` | the command object will be deleted by ET++ (default) |

The predefined command object `gNoChanges` is returned when an `EvtHandler` method (e.g. `DoKeyCommand, DoMenuCommand`) does not modify the contents of a `Document`.

### 8.1.3. Text handling

The ET++ classes to handle text can be subdivided in classes storing the text and classes which render the text on the screen. The classes storing the text contents are all derived from `Text`, the corresponding class hierarchy is shown in the following table:

| | |
|---|---|
| **`Text`** | abstract superclass for text storage |
| `CheapText` | for smaller texts without font attributes |
| `GapText` | for larger texts |
| `StyledText` | for texts with font attributes |
| `VObjectText` | for texts including graphic objects |

The text classes have constructors to set either their initial size or their initial contents. The class `StyledText` has an additional constructor to specify a string using different font faces.

**`StyledText(Font fd, char* format, ...);`**
`format` contains a format specification a la `printf`. In addition the face of the font `fd` can be changed with the notation `@P` (plain), `@B` (**bold**), `@I` (*italic*), `@O` (outline), `@S` (shadow), @U (underline). Example:

```
StyledText message("File @B%s@B does not exist", filename);
```

The same notation is used in `Alerts` to specify the format of the message string.

The contents of the text is modified with methods modelled after the cut/copy/paste operations:

```
void Cut(int from,int to);
void Paste(TextPtr t,int from,int to);
void Copy(Text* save,int from, int to);
void Insert(byte c, int from,int to);
void Append(byte c);
void Copy(Text* save,int from, int to);
```

The arguments `from` and `to` specify the range characters affected by an operation.

The text classes give support to attach a so called *mark* to a range of text. Marks are instances of the class `Mark` and can be considered as a robust pointer to a text range which remains valid in the face of insertions and deletions in the text.

In addition the text classes include the pattern matching method `Search` based on regular expressions. The class `RegularExpr` uses the syntax described in doc/regex.doc. In order to create an editable text on the screen instance of a text storage class is created and handed over to any text rendering class. `Text` classes and Views can be freely combined. An exception are `VObjectTexts`, they should always be used together with a `VObjectTextView`. A `VObjectTextView` takes care of dispatching events among the contained VObjects and allows to resize them interactively. The class hierarchy for the text rendering classes is shown below:

| | |
|---|---|
| **`StaticTextView`** | readonly textview without a selection |
| `TextView` | textview with a selection |
| `CodeTextView` | adds features to edit source code |
| `ShellTextView` | a textview connected with a UNIX shell, providing a fancier interface to the shell |
| `VObjectTextView` | a textview adding some support to manipulate graphical objects stored in a `VObjectText` |
| `RestrTextView` | restricts the input from the user, input is only accepted if the resulting text matches a client specified regular expression |

The properties of a `TextView` can be set in its constructor:

```
TextView(EvtHandler *eh, Rectangle r, Text *cont, eTextJust just,
         eSpacing sp, bool wrap, TextViewFlags tf, Point border)
```

| | |
|---|---|
| `next` | TextView next event handler |
| `r` | dimensions of the view. If `r.extent.width` or `r.extent.height` is set to `cFit`, the corresponding dimension is always adapted to the size of the text. |
| `cont` | install the given text in the view |
| `just` | text justification (`eLeft`, `eRight`, `eCenter`, `eJustified`) |
| `sp` | line spacing (`eOne`, `eOneHalf`, `eTwo`) |
| `wrap` | lines wrap around at right margin? |
| `tf` | flags of a `TextView` see below |
| `border` | border around the text |

Except for the first three arguments default values are provided in all constructors of `TextViews`. Further characteristics are specified in `tf` with the following flags:

| | |
|---|---|
| `eTextViewReadOnly` | text is read only |
| `eTextFormPreempt` | preempt formatting when there is user input and resume when the application is idle |
| `eTextNoFind` | do not add the find/change menu entry in TextViews |

`TextViews` batch incoming input from the keyboard and insert the new text in one batch. A side effect of this optimization is that, if `DoKeyCommand` is overridden in order to handle some input characters especially, it will not always be called for these characters. This case can be handled properly by calling `SetStopChars` with a string specifying the characters which may not be batched.

Whenever a `TextView`'s dimension or contents changes it notifies its observers by calling `Send` with some predefined part codes as definded in CmdNo.h. These change notifications are typically used to adapt the size of graphical boxes keeping text to the text's extent.

# 9. Clipboard

Class `ClipBoard` is the basic abstraction for the ET++ clipboard. The basic idea is to hide all details of maintaining a local and global clipboard completely from the application. But currently this abstraction exists only in a very rudimentary implementation. The local clipboard works under all window systems; the global clipboard only under SunWindows and X11.

In order to make clipboard support available between different ET++ applications under SunWindow it is necessary to start a separate process clipboard, a prototypic implementation of a simple minded clipboard server process. At start up time every ET++ application automatically connects to this server process, thereby making the clipboard data available to other ET++ applications. Currently it is not possible to cut and paste between the SunView clipboard and ET++ applications.

The `ClipBoard` class is not directly accessed in an application, but used through four methods of class `View`. In order to make a selection available for the clipboard, the following methods of class `View` must be overridden.

```
Command *PasteData(char *type, istream &is)
```
`PasteData` is called after selecting the paste menu command. The clipboards data is accessible via the `stream` parameter.

```
bool CanPaste(char *type)
```
This method is called whenever the `View`'s popup menu is displayed. The `View` must check if it is possible to paste the clipboard's data into it's data structure. The paste menu entry is only enabled if `CanPaste` returns `TRUE`.

```
void SelectionToClipboard(char *type, ostream &os)
```
Whenever an application want to paste the clipboard, this method is called for the current selection

holder. The selection holder must write the selection in the ET++ standard input/output format to the stream `os`.

But remember: the ET++ clipboard mechanism is not yet finished! But it works between most applications dealing with text. It is for example possible to copy a text containing different `VObjects` from vobedit into the draw application even if the draw application does not contain the code to support the selection's data. Dynamic loading “automagically” loads the necessary “.o”-files into the running application (try to copy an annotation from vobedit into a text object in draw).

# 10. Learning by Example

We currently do not provide manual pages or a lot of documentation about ET++ classes.

> ***A full fledged documentation for ET++ is in preparation***

The best way to learn ET++ is to study the example applications included in this distribution. The following table lists in order of increasing complexity the features illustrated by the example applications.

Hello:
: The famous “hello world” example

Micky:
: The almost empty application

TwoShapes:
: An application for moving and stretching two simple shapes and includes a menu to modify the fill pattern of one shape.
  - Undoable commands
  - Handling menus, including graphical menu entries
  - Generic stretchers and movers
  - Handling the type of a document
  - Alert boxes

MultiUndo:
: The same as TwoShapes but with n-level undo. The only implementation difference is the use of `CmdHistDocument` instead of `Document`.

ThreeShapes:
: An extension of two shapes with a boxed editable text shape. A shape's fill pattern can be changed with a menu while the cursor is over the shape.

- TextViews and their change propagation protocol
- Redirecting input events to an arbitrary object

FileBrowser:
A simple filebrowser. Presented at the OOPSLA ´90 Tutorial.

FileBrowserII:
A full-fledged filebrowser with integrated typescript. Clicking on an icon above the file list boxes brings up a shell window in the current directory. Later clicking synchronizes shell and viewed directory. The small icon above the right scrollbar brings up a menu showing all functions of the current C/C++ file. Select an item to quickly select (highlight) a method. Start a make by selecting the make-entry in the editors utilities menu. In case of C++ error messages place the caret into the error line and select the menu command “Find Error”. The browser will load the corresponding file and selected the line containing the error.

Miniedit:
A simple text editor for ascii files.

- Handling pure ASCII files
- Using `CodeTextViews`

TwoViews:
An application displaying two `TextViews` in the same window. Both `TextViews` show the same text and are updated automatically due to the delayed update mechanism used in the text classes.

- Two views in one window

Typescript:
An interface to a unix shell providing the text editing functions for the current command line as found in all ET++ applications. Output from the shell is read only and is not editable a reverted caret marks the fence between output from the shell and the current command line. Input of the user is shown in a bold styled font. Typescript emulates only a dumb terminal, e.g. more can be used but more demanding full screen applications a la vi are not supported.

- Using ShellTextViews

PolyDocApp:
A merge of miniedit and twoshapes, illustrating how different document types can be handled within one application. Miniedit documents have an additional entry import in the file submenu. Import allows to include another text file into the document.

- Extending the application window
- Handling different document types in one application
- Importing files

Dialog:

The intend of this application is not to show how to create dialogs but to present all the different dialog items of ET++ in a window.

Calculator:

A simple infix calculator.

- Advanced usage of event distribution mechanism
- Layout of a two dimensional cluster

Tree:

A tree browser. Trees can be stored either in the standard ET++ or pure ASCII format. The ASCII format uses indenting by two blanks to define the levels of the hierarchy. Clicking on a node invokes the source code editor if the name of the node corresponds a class name. An example of an ASCII file is import. This file was generated with makehier provided in the /bin directory. makehier is a awk script that analyzes all *.[cCh] files and determines the inheritance relationship among the different classes.

- Extending the save dialog with additional options
- Storing a document in different formats
- Using `TreeViews`
- Building up a tree for a `TreeView`
- Loading documents of different types with automatic conversion
- Using a `DialogView` as a palette

Vobedit:

A text editor with the ability to integrate arbitrary graphical objects (instances of `VObject`). These objects behave as ordinary characters and flow with the text when characters are inserted. Some graphical objects are provided in the submenu vobjects. Selecting an entry from this menu inserts the corresponding object into the text. Examples are `Button`s, `TextView`s, a *window* with a running shell, or recursively another instance of a `VObjectTextView` that can be scrolled independently of the rest of the text. The inserted objects still behave as expected, e.g. they respond to mouse clicks and keyboard input. The currently active `VObject` receiving the keyboard input is highlighted with a grey border. Once inserted the size of a graphical object can be changed by stretching the bottom left corner with the left mouse button.

- Using the classes `VObjectText`/`VObjectTextView`

Layout:

This application allows to study and to verify the layout management of the ET++ dialog classes.

- Generic stretchers, movers
- Layout management of clusters

Trofftool:

A previewer for ditroff output. Open document *demo.f*. You can flip pages by menu commands or the page up and down keys. The applications shows how to convert ditroff output to a compact Picture object. In addition an abstract Parser class with several simple subclasses is used for scanning, finding pages, searching, and converting to ASCII text (by saving the document). I think this is the shortest troff-previewer on the market :-)

Color:

The document's window shows 16 grey rectangles. Clicking on a color starts a color picker dialog for changing the color. The application illustrates the usage of animated colors and double buffering of complicated view backgrounds. Drag the small circle on the colorwheel! It's implemented without Xor but with invalidation. It's fast because the colorwheel is updated from a so called "Form".

Draw:

A full fledged drawing editor with the possibility to dynamically link new shape types while the application is running. Examples are the shapes defined in DynShape.c. The file dynshapedoc is a draw document including a `DynShape` shape. When this document is loaded the corresponding classes are dynamically linked and the palette is updated accordingly.

Draw can maintain connections among different shapes. A connection between two shapes is established by selecting the shapes (a selection can be extended with a shift click) and then choosing the menu entry connect. Text entered while a shape is selected is attached to this shape, e.g. when the shape is moved the text will be moved accordingly. SUN rasterfiles can be imported with import from the file submenu (works only under SunView). Shapes can not only be moved with the mouse but with the cursor keys as well.

- Change propagation to maintain the connectivity among shapes (Connection.[ch])
- `TextViews` as shapes (TextShape.[ch])
- Dynamic linking of classes
- Using a collection class to manage the shapes
- Using pull down menus
- Adding submenus to the main menu
- Using the clipboard

Sources of other code examples can be found in the ET++ class library itself.

FindDialog/ChangeDialog.[Ch]:

A modeless dialog.

- Aborting a lengthy operation (change all command)

PrintDialog.[Ch]:

A modefull dialog

### ShellTextView.C

Dispatching input from other sources than the window system, e.g. a pseudo tty.

- Using StyledText
- Using robust pointers to text positions with Marks

### PROGENV/Inspector.C

The inspector window itself is an example of a more complex layout of a window with several panes.