XVT Development Solutions

C++ Guide

xvt

GUIDE

CONTENTS

GUIDE

PREFACE

This *Guide* presents a basic yet thorough treatment of portable GUI programming with XVT's Development Solution for C++ (DSC++) and the XVT-Power++ application framework.

Note: XVT offers a Development Solution for C (DSC) and a Development Solution for C++ (DSC++). The XVT Portability Toolkits are the portable API for both DSC and DSC++.

XVT-Power++ Documentation

The XVT-Power++ documentation consists of:

- *Guide to XVT Development Solution to C++* (this manual)
- *XVT-Power++ Reference* (online)

The documentation also includes the Rogue Wave *Tools.h++ Guide* and *Reference Manual*© because XVT-Power++ uses the Rogue Wave class library to implement many of its classes.

When to Use the Guide

The purpose of this *Guide* is to describe XVT-Power++'s structure, survey the overall functionality that is available in related groups of classes, and explain how things work in XVT-Power++. In short, it gives you the total picture. When you want details on a particular class, consult its description in the alphabetically sorted, online *XVT-Power++ Reference*.

This *Guide* constitutes a good overview of DSC++. On the other hand, you'll want to repeatedly refer to the *Reference* throughout your development efforts, because of the details you can find there about specific classes and their methods.

What You Already Need to Know

Throughout the XVT-Power++ documentation, we assume that you have some basic knowledge of GUI features and programming, a working knowledge of C++, and access to the XVT Portability Toolkit documentation, which is available online.

Reference Information Available Online

Reference information for all XVT products is available online in the Adobe® Portable Document Format (PDF). PDF was chosen because it is the open de facto standard for electronic document distribution worldwide. Adobe PDF files are compact and can be shared, viewed, navigated, and printed exactly as intended by anyone with free Adobe Acrobat® Reader® software. The Adobe Acrobat® Reader® software may be downloaded at http://www.adobe.com/products/acrobat/readstep2.html.

How to Read This Manual

If you are a first-time DSC++ user, we recommend:

- Read the first two chapters of this manual to get an overview of how to use XVT-Power++ and the visual application builder for DSC++, XVT-Architect.
- Work through the Tutorial chapter of this *Guide*. The tutorial introduces you to using XVT-Architect in conjunction with the XVT-Power++ application framework. The tutorial introduces you to the GUI application structure and to laying out GUI objects, setting their attributes, and constructing menubars.
- Read Chapters 12 and 13, which explain the philosophy and advantages of XVT-Power++'s application framework, and present information that every XVT-Power++ user should know, including XVT-Power++ coding conventions and style guidelines.
- Read Chapter 14, which explains the structure of XVT-Power++'s application framework and the important relationship between application, document, and view.
- Read the *Guide*'s other chapters to the depth required. The modules are organized so that deeper levels of subject matter detail are treated in later modules.

Note: Keep in mind that you can always consult the online *XVT-Power++ Reference* for a detailed discussion of any class in the XVT-Power++ hierarchy.

Troubleshooting

Before calling Customer Support, try using the following resources:

XVT Documentation

XVT provides several kinds of documentation:

- Technical manuals
- **readme** files
- Technical Notes
- Frequently asked questions (FAQs)
- Errata

The documentation is designed to give you the information you need to use XVT-Power++ at all levels, from introductory to advanced. To suggest documentation changes, contact Customer Support.

XVT Sample Set

Using the provided XVT-Power++ Sample Set, you can:

- Verify that your XVT-Power++ installation is correct by building and running the examples in the **...samples** directory (includes makefiles or project files where applicable)
- See a working example while learning XVT-Power++
- Compare sample functions to functions in your code, to verify correct behavior

XVT Customer Support may ask you to use the Sample Set to verify a problem when you report it.

Error Messages File

The XVT-Power++ library does some internal error checking. If you use an XVT-Power++ function incorrectly, you may see an "XVT-Power++ Error" dialog box, which displays the error number followed by the timestamp of the product release version (for example, "Internal PWR Error: 37007-408323").

If you contact XVT Customer Support about the problem, please mention the error number and the description of the error.

Other XVT Documentation

XVT provides many different pieces of documentation online:

Release Notes and Installation Instructions
Platform-specific information about how to install the DSC++ is placed in the installation instructions. The release notes tell you what is new or changed with this release. This information, which includes environment configuration for your platform, also appears in the **install.txt** file.

Guide to XVT Development Solution for C++
This manual presents an introduction to portable GUI programming using XVT-Architect and the XVT-Power++ framework. It also introduces you to the application–document–view paradigm that is the cornerstone of XVT-based object-oriented programming.

XVT Development Solution for C++ Quick Reference
This booklet encapsulates the online *XVT Power++ Reference*.

XVT Platform-Specific Books
Each book contains information you need to use the PTK on a particular XVT-supported platform. The information about non-portable attributes and escape codes is especially useful.

XVT Portability Toolkit Guide
This manual presents a basic yet thorough treatment of portable GUI programming with the XVT Portability Toolkit (PTK).

XVT Portability Toolkit Quick Reference
This booklet encapsulates the online *XVT Portability Toolkit Reference*, along with details about the URL language.

XVT-Design Manual
This manual is both guide and reference for XVT-Design, the visual programming tool that is included with XVT Development Solution for C (DSC). It introduces you to portable GUI programming using XVT-Design.

XVT-PowerObjects GUI Components Pak 1

This document is both guide and reference for the GUI components, XVT-PowerObjects, included with the XVT Development Solution for C.

XVT Technical Notes

XVT provides Technical Notes in the **...doc** directory. Additionally, as new Technical Notes become available, XVT posts them on the anonymous FTP site.

XVT also provides the following as online HTML documentation:

XVT Portability Toolkit Reference

This online documentation contains reference information for all API elements of the XVT PTK: portable attributes, events, data types, constants, and functions.

XVT-Power++ Reference

This online reference manual describes all the classes of the DSC++ framework.

XVT DSC Online Help

This online help, accessed from XVT-Design, contains extensive instructions about using XVT-Design, a copy of the *XVT Portability Toolkit Quick Reference*, and some other useful information for C programmers.

XVT-Architect Quick Help

This help file provides context-sensitive descriptions of the classes that you are using and the data members that you can set.

The documentation is designed to give you the information you need to use XVT products at all levels, from introductory (the *Guides*, particularly the “Fundamental” and “Basic” modules) to advanced (the *References* and *Quick References*).

About This Manual

XVT takes pride in its documentation, and continually seeks to improve it. If you find a documentation error, please contact XVT Customer Support. They will forward your suggestion to XVT's documentation team.

Conventions Used in This Manual

In this manual, the following typographic and code conventions indicate different types of information.

General Conventions

code
: This typestyle is used for code and code elements (names of functions, data types and values, attributes, options, flags, events, and so on). It also is used for environment variables and commands.

code bold
: This typestyle is used for elements that you see in the user interface of applications, such as compilers and debuggers. An arrow separates each successive level of selection that you need to make through a series of menus, e.g., **Edit=>Font=>Size**.

bold
: Bold type is used for filenames, directory names, and program names (utilities, compilers, and other executables).

italics
: Italics are used for emphasis, for the names of other documents, and in cross-references to chapters inside the same document.

t This triangle symbol marks the beginning of a procedure having one or more steps. These symbols can help you quickly locate "how-to" information.

Note: An italic heading like this marks a standard kind of information: a Note, Caution, Example, Tip, or See Also (cross-reference).

In XVT documents, this symbol designates the new functionality provided by the XVT Portability Toolkit Release 5.

Code Conventions

<non-literal element> or non_literal_element
: Angle brackets or italics indicate a non-literal element, for which you would type a substitute.

[optional element]
: Square brackets indicate an optional element.

...
: Ellipses in data values and data types indicate that these values and types are opaque. You should *not* depend upon the actual values and data types that may be defined.

XVT CUSTOMER SUPPORT

When you buy an XVT product or an XVT maintenance agreement, you gain access to some of the most advanced application development assistance in the industry.

If you have problems or questions while using XVT products, you can talk to an XVT Customer Support Engineer. XVT Customer Support helps you make more effective use of XVT products, enabling you to get your application up and running as quickly as possible. Customer Support is available to customers who have purchased an XVT product and have a current maintenance contract.

Please note that only one individual per purchased copy of an XVT product may request support. Questions will be taken only from the individual named on the software registration form.

Feel free to contact XVT Customer Support if you have a question, or would like to suggest a software enhancement or a change to any document. Your call is always welcome.

How Customer Support Works

XVT's Customer Support goal is to respond to all requests within twenty-four hours. As soon as we log your call into our system, you will receive a service request number.

If we have questions about your request, we ask that you respond to them within five working days. Please let us know if you need more time; otherwise, if we receive no response from you after five days, your service request will be closed.

Electronic Communication with Customers

XVT has a T-1 connection to the Internet, and we provide anonymous and secured FTP and WWW sites. These services allow XVT to:

- Provide FTP (File Transfer Protocol) access to XVT products and services
- Deliver product patches via the FTP site
- Update products via the FTP site
- Disseminate marketing information via a WWW (World-Wide Web) home page

Tip: The address of XVT's WWW site is http://www.xvt.com; XVT's FTP site address is ftp.xvt.com.

XVT Developers' Forum

The XVT Developers' Forum (xvtportabilitytools) is a community created to allow XVT developers to exchange XVT problems and solutions with one another. The forum is open to all XVT developers and other interested parties. The forum can be accessed at:

http://groups.yahoo.com/group/xvtportabilitytools/

What XVT Customer Support Provides

XVT Customer Support can serve you better if you understand what services are available.

This is what XVT's Customer Support can do:

- Provide "tips" to help you effectively use XVT functionality
- Explain XVT functionality and specifications
- Diagnose and analyze XVT-related application problems
- Suggest workarounds
- Suggest how to access native window system development tools
- Collect feedback for future product development

Keep in mind that XVT Customer Support *cannot* do the following:

- Design customer applications
- Debug user code

- Explain how to use operating systems, window systems, or compilers (except with regard to XVT application resources)
- Explain how to use native window system development tools
- Extensively teach XVT programming

If you need more help than Customer Service can provide, consider contacting XVT's Professional Services Group. More information about this group can be found on the last page of this section of this *Guide*.

Customer Support Services

XVT's Customer Support engineers can answer questions that arise from the use of a native GUI platform, or the operating system itself (see the following subsection, "Standard Customer Support Services"). When questions require investigation, you are given a follow-up reference number that identifies your inquiry in our Customer Support database. These pending requests for information are reviewed several times each day, to ensure a timely reply.

Standard Customer Support Services

XVT Customer Support personnel are experienced software developers that specialize in the use of XVT products on supported MS-Windows 3.1, Windows NT, Windows 95,Windows 98, Windows 2000, Windows ME, Windows XP, Motif, OS/2, Macintosh and Power Macintosh computer systems. For registered named users, XVT offers the following standard services:

- Easy access by phone, fax, and electronic mail
- One day (maximum 24 hour) call backs after your initial call is received
- An assigned Service Request number when your questions are logged, so they are tracked and responded to efficiently
- Tips that speed the use of XVT functionality
- Explanations of XVT functionality and specifications to reinforce product manual descriptions
- Suggested workarounds for common development obstacles
- Suggestions for complementary products or native window system environment tools that might enhance your application or make you more productive

- Product enhancement requests are tracked and analyzed so that customers significantly influence future product development decisions

Online FTP Site

XVT's online FTP site is available to all currently registered customers, and offers valuable information for XVT application developers. Technical papers and notes, programming examples, product updates, and programming utilities are available from the FTP site.

Support for XVT Software Purchased from Distributors

XVT products are sold around the world, often through an independent distributor licensed by XVT Software. If you purchased your XVT product through an international distributor, your customer support requests must be routed through that distributor. If, on the other hand, you purchased your XVT product from an international office of XVT, you may contact XVT directly for support. Instructions for contacting XVT Customer Support are listed on the last page of this section of this *Guide*.

Information We Need to Help You

When you contact XVT Customer Support, please supply the following information:

- The name and version number of the product (for example, XVT/Win32 5.6)
- The serial number of the product (found on your distribution media)
- Your platform type (for example, Sun 4, IBM RS/6000)
- The operating system and version number (for example, HP-UX 11.0, Solaris 2.9)
- The compiler and version number (for example, Microsoft Visual C++ .NET 2003)
- The window manager and version number (for example, MS-Windows XP)
- A detailed description of the problem, including information displayed with any internal error message—such as the called function, filename, and code line

Product Updates

XVT actively updates its products, issuing both minor releases (averaging 2–3 per year) and major releases (averaging 12–18 months apart). For most minor releases, and for all major releases, XVT supplies additions to or complete replacements for product documentation. As a service to XVT customers, all product updates are made available from XVT's FTP site.

Tip: Customers who are current on their XVT maintenance agreements can download product updates from XVT's FTP site.

How to Contact Customer Support

You can contact XVT Software Customer Support in several different ways:

- Telephone us at 919/854-1800 (hours 9 AM-12 PM and 1:30 PM-5 PM, Eastern Standard Time, Monday–Friday)
- Send us electronic mail via the Internet at xvt-support@xvt.com
- Use the FTP site at ftp.xvt.com
- Write us at Providence Software Solutions, 1143_H Executive Circle, Cary, NC 27511 USA

Note: Some portions of the FTP site can be visited by anyone. However, before you can access *all* areas of the XVT FTP site, you must contact XVT Customer Support to get a valid password.

XVT's Consulting and Training Services

The XVT Professional Services Group offers extensive fee-based services to help customers use XVT products. Experienced XVT professionals can help you learn GUI programming, or help you prototype, design, code, debug, and maintain your XVT applications.

In addition to consulting, XVT personnel also conduct on-site and public training classes in XVT and GUI programming techniques.

See Also: For more information about the XVT Professional Services Group, call 919/854-1800.

1

INTRODUCTION TO XVT-POWER++

1.1. What's in the XVT-Power++ Package?

The classes in XVT-Power++ fall into four major categories: application framework, utilities, data structures, and pass-through functionality. This chapter surveys the functionality that XVT-Power++ makes available to a GUI application developer within these categories—in terms of specific development tasks. It also discusses the advantages of using XVT-Power++'s object-oriented approach to design an application.

1.1.1. Introducing XVT-Power++

XVT-Power++ is a C++ application framework that allows application programmers to produce extensive graphical user interfaces with relatively few lines of code. XVT-Power++ features an object-oriented design that incorporates an application framework and encapsulates the complex work associated with GUI programming into several hierarchies of object classes. Application designers can use these objects as building blocks to assemble into programs. Thus, XVT-Power++ dramatically reduces application development time and effort.

XVT-Power++ works with most major GUI platforms, including Motif®, MS-Windows™, Windows XP ™, Windows 2000 ™, and Macintosh®. XVT-Power++ is "portable" in that it offers one body of code for all targeted platforms. In most cases, once you develop a program on one platform, you can take the program to another platform, recompile it, and run it without further changes.

Because XVT-Power++ predefines a number of GUI components, program development is easier and faster. Whether an application needs scrollers, list boxes, validating text fields, grids, user-sizeable

objects, buttons, or other similar features, XVT-Power++ provides classes that can simply be assembled. On the other hand, XVT-Power++ does not confine the GUI programmer to predefined XVT-Power++ objects. Using C++ and the XVT Portability Toolkit, you can extend and modify the entire class library by overriding methods and creating new classes.

1.1.2. XVT Portability Toolkits

XVT-Power++ achieves portability through the use of the XVT Portability Toolkit. Consisting of a set of C functions that communicate with the underlying graphical toolkits of several machines, the XVT Portability Toolkit provides a portable layer between the application program and the underlying graphical system.

Each Portability Toolkit implements the XVT interface over native GUI functionality. This ensures native look-and-feel, low overhead, interoperability with other applications, and access to native toolkits when required.

Figure 1.1. XVT Portability Toolkits — the foundation of a well-written, versatile, and maintainable application

1.1.3. The XVT-Power++ Application Framework

The definition of an application framework is derived primarily from the word “framework.” A framework can be defined as a tree or a structure. Each must have a strong foundation to support the higher-level elements. These higher-level elements are connected to lower elements. Elements at different levels perform different functions, just as the roots of the tree perform a different function than the leaves.

C++ lets you define all the different application elements or functions into a hierarchy of classes. You can then use these classes repeatedly to create applications. Some vendors refer to these class libraries as application frameworks. But an application framework is more than a class hierarchy of C++ classes with interdependencies. A true application framework provides an infrastructure that encapsulates an application’s functionality. This infrastructure exists both in the non-visual application support classes and in the visual (or view) classes.

The XVT-Power++ class hierarchy is shown in Figure 1.2. Notice that XVT-Power++ includes many classes that are not part of the hierarchy “chain.”

The advantage of using an application framework is that it imposes a clear, rigorous separation of responsibility within the application. For example, documents deal only with data, while views passively display data. Because the division of responsibility is so well-defined, the application lives up to the “object-oriented manifesto”—in other words, it fulfills the expectation that different modules of the application are truly reusable, maintainable, and hopefully, scalable.

See Also: For more discussion of the advantages of using class hierarchies, see section 1.2.2 on page 1-7.

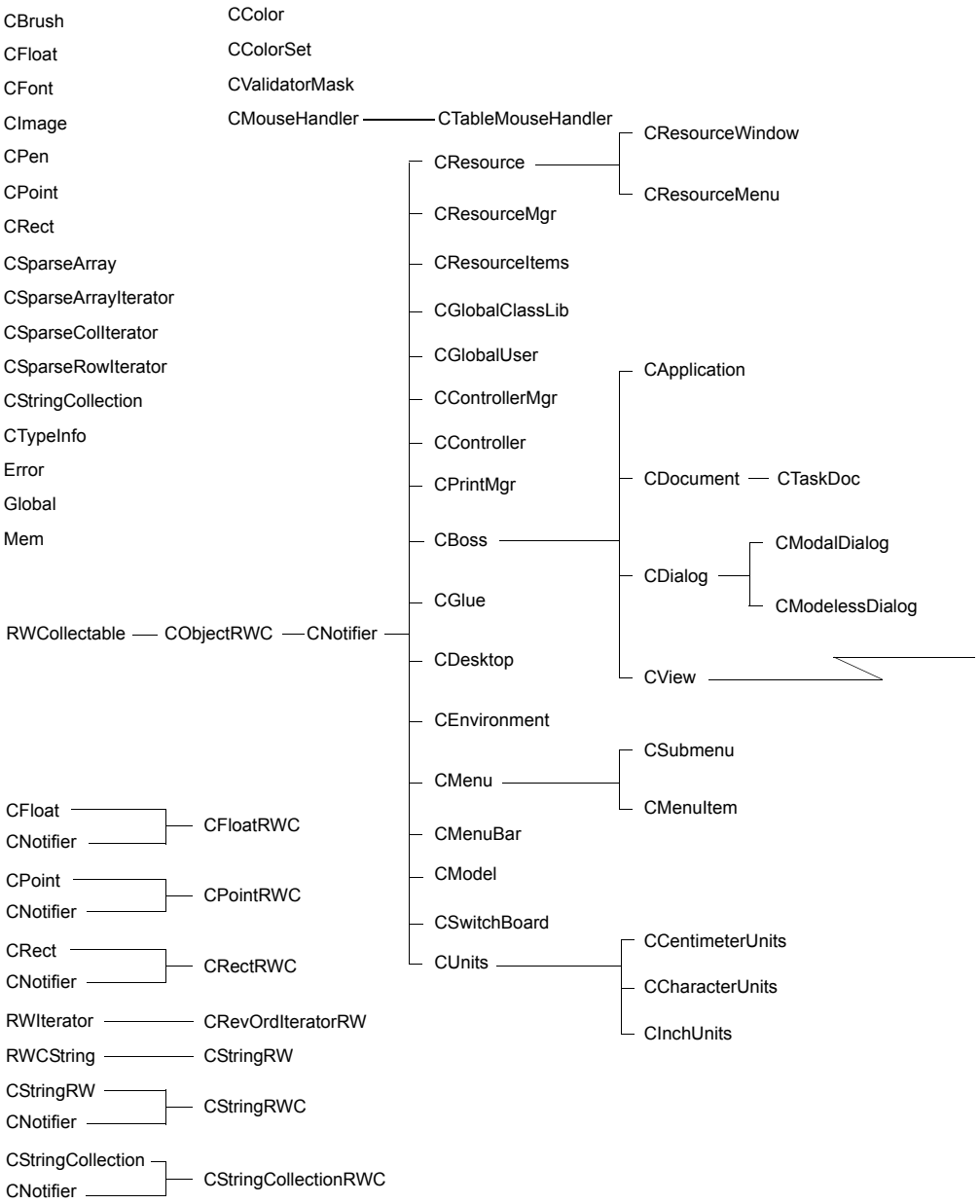
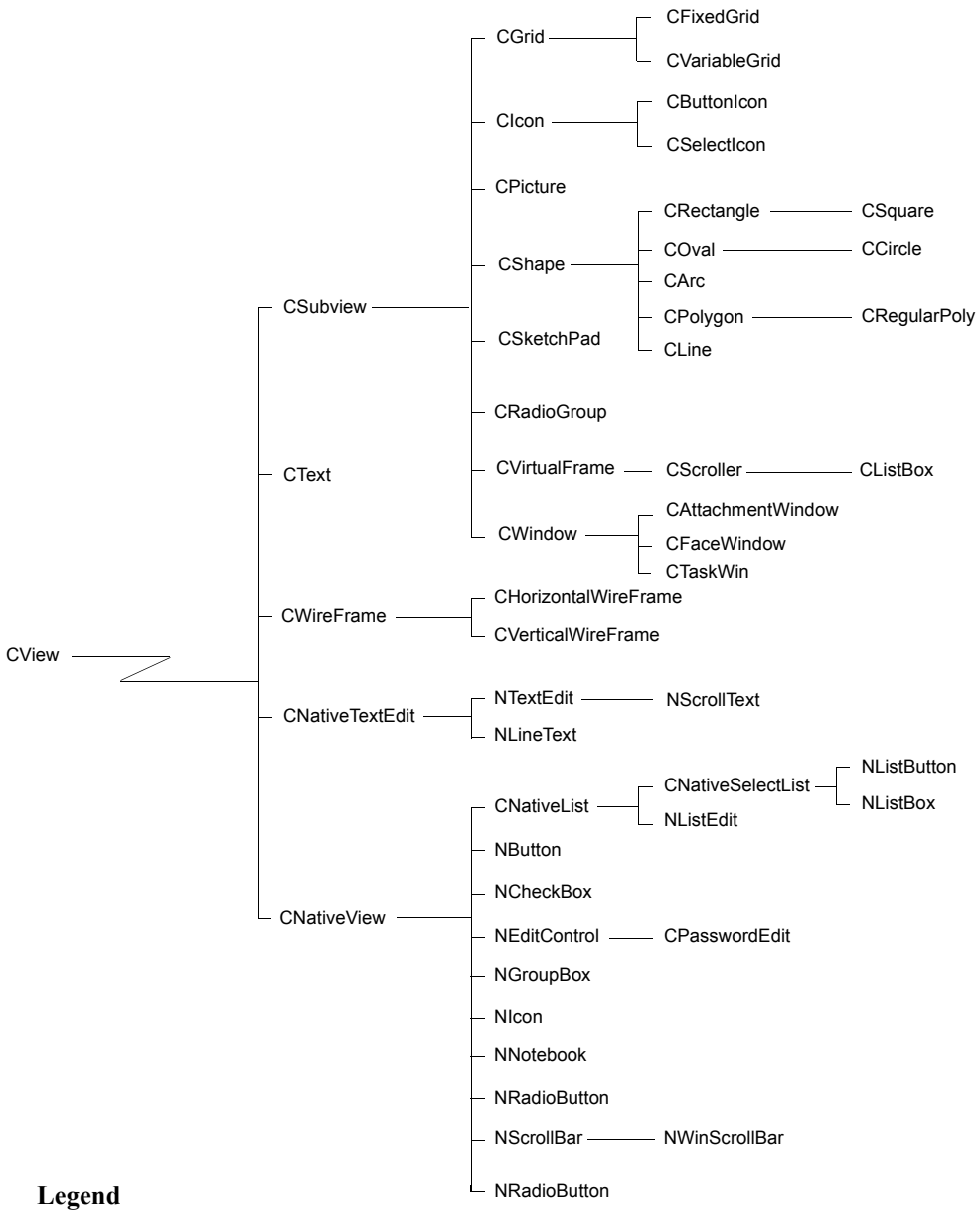


Figure 1.2. The XVT-Power++ class hierarchy



Legend

——— indicates a derivative relationship, which proceeds from left to right

Note: The Rogue Wave data structures are not included in this tree. For a listing of these data structures, see the *Tools.h++* manual.

1.2. Designing an XVT-Power++ Application

When you begin to design an application, you must make several basic decisions: what it will look like, what will happen when the user executes the program, what will come up first, and the order of the steps that the user must perform to interact with the application. XVT-Power++'s application framework helps you to make these decisions. When you design an application, follow these basic steps:

1. At the very least, write a new application, a new document, and a new window class, deriving them from their corresponding XVT-Power++ classes.
2. Determine what XVT-Power++ classes you want to use.
3. Derive/write other classes that reflect the things you want your application to do.
4. Design the structure of your own class and object hierarchies and decide how to assign the *supervisor* relationships among these classes. Who is going to own whom? Who is going to instantiate what?

Your application will have the same skeleton, the same underlying object hierarchy, as XVT-Power++, but it will also have its own specific flavor.

1.2.1. Development Platform

Naturally, the platform on which you develop an application has an impact on your design decisions—imposing constraints, for example, on what is going to be displayed first and how it will look. On the Macintosh, when the user executes a program, a menubar appears, and the user is expected to do a "New" or "Open" operation. This is also true on Motif. On MS-Windows, an application always starts up with one window on the screen, the task window. Whatever the platform, when an application starts up, something appears on the screen for the user to see. It acts as a clear visual cue about what to do next in order to gain entry to the program's functionality.

XVT-Power++ automatically adapts to the native look-and-feel of the platform for which it is compiled. Thus, it creates the appropriate windows and menubars upon application startup.

1.2.2. Advantages of Object Hierarchies

Class hierarchies are useful structures because a class can inherit many features from a parent class, allowing developers to reuse some of the code inside a parent class. Developers can reuse this code because classes inherit down the line.

This is similarly true of object hierarchies, though for a different reason. We illustrate this point with the list box, which is a composite of several objects. If you look at a list box on the screen, you will notice that it has horizontal and vertical scrollbars, which are instantiations of XVT-Power++'s CScroller class. Less obvious to you is the fact that the list box contains a grid so that all the objects in the list box can easily be aligned. Inserted inside the grid are the actual text or string items, CText objects.

Although CListBox, as a composite, is a fairly complex class, it was easy to implement because the objects composing it already existed in XVT-Power++—scrollers, grids for formatting, and text objects. Thus, all we have to do is put it all together and add a few specifics that list boxes require in order to manage the list: insertion and removal of items, selection and deselection, reporting which items are selected, and so forth.

Likewise, if you want to display a hierarchical picture of the directory structure on your computer—a *tree view*, as it is typically called—your application draws graphical objects such as icons, lines, and different colored shapes on the screen to construct a picture of the tree hierarchy. The overall arrangement of the objects is controlled by the CTreeView class (if you are using this class to organize your hierarchical display).

1.2.2.1. Advantages for XVT-Power++

As demonstrated by the CListBox and the CTreeView classes, XVT-Power++ reaps one of the great advantages of object-oriented programming—the ability to put together new objects out of objects that have already been written. Through the reuse of code and the reuse of ideas, XVT-Power++ easily implemented these new composite objects.

1.2.2.2. Advantages for XVT-Power++ Users

An XVT-Power++ user does not need to know that a CListBox is a composite of the NScrollBar, RWOrdered, and CGrid classes. When the user instantiates a list box, its behaviors are already built into it. To get complete list box functionality, a user simply creates a new list box (new CListBox) and passes it some parameters that give it a location, strings to display, and so on. Then the list box appears inside the view, fully functioning.

However, if you want to take full advantage of XVT-Power++ and derive your own classes, then you need to know how such composite objects as the list box are put together so that you can create your own.

1.2.2.3. Advantages for Designers of XVT-Power++ Applications

XVT-Power++ contains a wide range of classes that provide most of the functionality needed in a typical GUI application. However, when designing an XVT-Power++ application, you may find it necessary to write a completely new class from scratch. Most of the time, when XVT-Power++ does not supply a specific class or behavior that you need, you can build a composite object from classes that XVT-Power++ already provides. This is the essence of object hierarchies, and it is what makes XVT-Power++ extensible and very easy to use.

There may be times when you also want to alter the behavior of an XVT-Power++ class. To create a new class, derive it from the class you want to change and then override the appropriate method(s). For example, if you want a list box that contains graphical objects instead of text objects, you would derive it from XVT-Power++'s CListBox class and modify one of the methods after studying the class to find out what can be overridden and what cannot. You discover that you can override the method InsertItem. In another case, you might have had to *add* a method that takes graphical objects and inserts them into the grid.

In overriding the method on CListBox, you are creating a completely new class—say, MyListbox—leaving the original CListBox class intact. You should never change the actual code
of an XVT-Power++ class. The class library is designed so that you do not change the code if you want to modify behavior. Instead, you derive a new class and override one or more of the methods on the XVT-Power++ class.

1.3. Application Framework

The XVT-Power++ application framework consists of three different levels:

- An *application level* that controls a program and is analogous to main
- A *document level* that gets access to data and stores and manages data
- A *view level* that provides windows and other specialized structures in which to display data and graphical objects

All communication layers are contained within this hierarchy. XVT refers to these components as Application-Document-View, which is based on the Model-View-Controller (MVC) paradigm. Model-View-Controller is a well known concept for organizing and maintaining information in a dynamic system.

1.4. Utility Classes

XVT-Power++'s *utility* classes provide links between different parts of the XVT-Power++ system, such as:

- Events and the application
- The XVT-Power++ library and your specific application
- Global objects and the rest of the objects in the hierarchy
- The screen window layout in your application (the desktop)
- Drawing tools and your application (the environment)
- Resources and your application

Each of XVT-Power++'s main objects has several helping and utility objects attached to it. For example, a native view can have a glue object, an environment object, a title object, and a list of commands. Every view has a CRect object that tells it where to draw and a CPoint object that tells it which origin to draw from and what its coordinates are relative to.

1.4.1. Storing Program Resources

Almost all resources used by XVT-Power++ applications can be coded using XVT Portability Toolkit's URL and the **curl** compiler. A resource can be a bitmap, an icon, the resource format of dialog boxes, an internationalized string, and so on.

Anything that your application uses can be stored, managed, and created by the resource manager. For example, a stop sign icon for a STOP button would be recorded in the resource manager file as a resource for the application. The resource manager then knows that the icon resource exists, and when there is a call for it, it can draw the icon.

1.4.2. Defining Colors, Font Types, Drawing Modes, Line Colors and Widths

The CEnvironment class provides a data structure containing information on such environment attributes as color, font, pen, brush, and drawing mode. Environment information can be propagated down the object hierarchy. By default, a global environment object is shared by every displayable object in an XVT-Power++ application.

1.4.3. Reporting Errors

XVT-Power++'s error reporting facility is provided in the Error class.

1.4.4. Memory Management

XVT-Power++ does not override the global new and delete methods. It is up to you to decide the appropriate local or global overriding for heap management. Note that in some of its classes, Rogue Wave overrides these methods locally.

See Also: For more details on Rogue Wave, see the *Tools.h++* manual.

1.4.5. Using Portable Images

The CImage class provides a portable image facility. A subview-derived class, CPicture, maps a CImage to a region on the screen. Using CClipboard, images can be transferred to and retrieved from the clipboard.

See Also: For more information on these classes, see their separate entries in the online *XVT-Power++ Reference*.

1.4.6. Accessing the Clipboard

Any persistent, streamable object (text, XVT PICTURE, or binary data) can be streamed to and from the clipboard using the CClipboard class. Several derived classes automatically assist with the I/O associated with a clipboard operation.

See Also: For more information on the clipboard, see the description of CClipboard in the online *XVT-Power++ Reference*.

1.4.7. Translating XVT Portability Toolkit Events to XVT-Power++ Calls

CSwitchBoard serves as a liaison between the XVT Portability Toolkit and XVT-Power++. The switchboard is in charge of channeling whatever events are occurring in the system to the appropriate object. For example:

- A startup or termination event goes to the application object
- A resize event goes to the appropriate window object
- A keyboard event is channeled by the switchboard to the appropriate view

See Also: For more details about how keyboard events are routed by CSwitchBoard, see section 15.3 on page 15-6.

1.4.8. Printing

XVT-Power++'s interface to the XVT Portability Toolkit's printing facilities is called CPrintMgr. This class is in charge of queuing up data and printing it. Normally, when you want to print a view, you can call DoPrint inside the view. The actual implementation of printing is handled inside CPrintMgr.

1.5. Data Structures

XVT-Power++ uses the Rogue Wave class library to implement its data structures. Rogue Wave provides a rich set of collections, data structures, and utility classes that you can take advantage of while using XVT-Power++.

Most XVT-Power++ classes make use of these structures, which are generally *container* classes that are used to store objects. Users often need to store objects within data structures. For example, XVT-Power++ itself uses lists extensively.

See Also: For more information on Rogue Wave, see the *Tools.h++* manual.

1.5.1. Specifying Locations on the Screen

Every XVT-Power++ object has a CPoint object that tells it which origin to draw from and what its coordinates are relative to. Each CPoint is an *x,y* coordinate.

1.5.2. Placing Views on the Screen

Every XVT-Power++ object has a CRect object that tells it where to draw. A CRect is a rectangular region (set of coordinates) on the screen that defines an area in which another object will be located. CRect has methods for translation, coordinate system conversion, intersection, union, inflation, height, and width.

1.5.3. Converting Global to Local Coordinates and Vice Versa

The CRect class has several methods for coordinate system conversion. These methods allow you to go back and forth between global and local coordinates without having to do any calculation.

1.5.4. Specifying Logical Units

By default, all XVT-Power++ applications use a one-to-one pixel mapping for drawing or printing. You can set a different mapping—in centimeters, inches, characters, or a user-defined unit—by instantiating a CUnits object and then calling a certain object in the application framework hierarchy and setting its units through its SetUnits method.

The use of logical units makes applications more portable because different machines have different screen widths, heights, and

resolutions. The logical units are mapped out to the physical device on which you are displaying your information, that is, either the screen or a printer.

1.5.5. Representing and Comparing Character Strings

CStringRW is a class representation of character strings that encapsulates the data structures and utilities of character strings.

The CStringRW class contains several methods for concatenating and appending character strings. Plus, several of its methods have been overloaded so that CStringRW can handle multibyte characters in a manner that is consistent with the underlying PTK functionality.

The CStringRW class also contains several methods for comparing character strings. It enables you to do equality, inequality, greater-than, less-than, greater-than-or-equal, and less-than-or-equal comparisons.

1.5.6. Storing Items in Lists

The RWOrdered class stores pointers to objects that inherit from the RWCollectable class. Items are stored as a linear array. You can add the same pointer more than once.

1.5.7. Iterating Over Lists

RWOrderedIterator is a class that iterates over RWOrdered objects.

1.5.8. Storing Two-dimensional Arrays and Conserving Memory

When the data stored is sparse, CSparseArray is useful because the storage requirement of the array is proportional to the number of non-empty locations, rather than to the size of the array.

1.6. Pass-through Functionality

There are some features that it does not make sense to encapsulate in C++, or that XVT has not yet encapsulated in C++. These features include the following:

- Color palettes and color look-up tables
- Cursors
- Diagnostics and debugging
- Files
- Hypertext online help
- Native functionality
- Predefined dialogs
- Resources
- User-defined font mappers
- Pre-translated resources (five languages)

In order to use these features, you need to use the C API of the XVT Portability Toolkit. Many of these API functions require using XVT Portability Toolkit data types. However, XVT-Power++ provides methods on all CView objects to get these data types. GetXVTWindow returns an XVT Portability Toolkit window type. In addition, conversion operators are provided to the appropriate XVT Portability Toolkit types in these classes: CPoint, CRect, CBrush, CPen, CEnvironment, and CFont.

See Also: For more information on these classes, see their separate entries in the online *XVT-Power++ Reference*.

1.6.1. Color Palettes and Color Look-Up Tables

While XVT-Power++ provides an encapsulation of portable images through our CImage and CPicture classes, additional functionality is provided by the XVT Portability Toolkit for color palettes and color look-up tables.

See Also: For more information on CImage and CPicture, see the online *XVT-Power++ Reference*.
For more information on color palettes and color look-up tables, see the “Portable Images” chapter in the *XVT Portability Toolkit Guide*.

1.6.2. Cursors

A cursor is a pointer or other shape that indicates the current mouse position. Each CWindow object can have a cursor, which is set to one of four standard shapes, or to a shape that is defined as a resource.

See Also: For more information on cursors, see the "Cursors and Carets" chapter in the *XVT Portability Toolkit Guide*.

1.6.3. Diagnostics and Debugging

In addition to XVT-Power++ Error and debugging functionality, you can use the XVT Portability Toolkit error handling functions. The XVT Portability Toolkit provides error signaling with error codes, error handler functions, error definition and message files, and error dialogs for reporting errors and warnings.

See Also: For more information on diagnostics and debugging, see the "Diagnostics and Debugging" chapter in the *XVT Portability Toolkit Guide*.

1.6.4. Files

The XVT Portability Toolkit provides a portable data type for referring to filenames, directories, and file types. This feature allows you to set and get file attributes, use standard functions for file input and output, and use standard file dialogs.

See Also: For more information on files, see the "Files" chapter in the *XVT Portability Toolkit Guide*.

1.6.5. Hypertext Online Help

XVT Portability Toolkit's online help feature provides a powerful, flexible, hypertext-based help system for your applications. The online help feature includes the following key elements: a hypertext viewer, complete text formatting with multiple fonts and styles, association between CNativeView objects and specific help topics, embedded bitmap images, and support for native help display facilities.

See Also: For more information on hypertext online help, see the "Hypertext Online Help" chapter in the *XVT Portability Toolkit Guide*.

1.6.6. Native Functionality

Each XVT Portability Toolkit has a set of platform-specific, non-portable attributes. These attributes let you fine tune your application, or let you add functionality not provided by the XVT Portability Toolkit interface.

See Also: For more information on non-portable attributes, see the various *XVT Platform-Specific Books*.

1.6.7. Predefined Dialogs

The XVT Portability Toolkit supports several common dialogs on all platforms. These dialogs allow you to perform the following:

- Ask a yes or no question
- Display a note or an error alert
- Get a string typed by the user
- Display an About Box
- Prompt the user for a filename for input or output

See Also: For more information on predefined dialogs, see the "Dialogs" chapter of the *XVT Portability Toolkit Guide*.

1.6.8. Resources

XVT-Power++ uses the Universal Resource Language (URL) at the XVT Portability Toolkit level to specify resources for menus, dialogs, windows, strings, images, and fonts.

With every Portability Toolkit, XVT supplies a compiler for URL, called **curl**. You can port your URL code to any supported XVT platform and compile it to the native resource format using the XVT compiler, **curl**. The **curl** compiler reads specifications in the Universal Resource Language and generates specifications in the format appropriate to the native platform.

See Also: For details on how XVT-Power++ supports URL, see Chapter 28, *Resources and URL*.
For more information on URL and resources in general, see the "Resources and URL" chapter in the *XVT Portability Toolkit Guide*.

1.6.9. User-defined Font Mappers

XVT-Power++ features an encapsulated font model through the CFont class. The CFont class provides full coverage of the XVT Portability Toolkit font functionality, including user-defined font mapping. However, CFont does not include an available interface for setting your own application specific font mapper. To access this functionality, you assign a value to the ATTR_FONT_MAPPER attribute at the PTK level.

See Also: For more information on user-defined font mappers and the ATTR_FONT_MAPPER attribute, see the "Fonts and Text" chapter in the *XVT Portability Toolkit Guide*.

1.7. Where To Go Next

At this point, we have covered the functionality that XVT-Power++ makes available in the following:

- The XVT-Power++ class hierarchy, which is the main set of classes in the XVT-Power++ *application framework*. This hierarchy provides the view classes.
- The *utility* classes or files that provide links between different parts of the XVT-Power++ system.
- The *data structures*, which provide the means to create the data structures such as lists and arrays.
- The *pass-through functionality*, which maps directly into the C API of the XVT Portability Toolkit.

Now that you have surveyed the XVT-Power++ classes and are aware of the overall functionality available to the XVT-Power++ application developer, XVT recommends that you read about style guidelines and coding conventions in Chapter 13.

See Also: For more information on the Rogue Wave class library, see the *Tools.h++* documentation. For detailed information on any of the classes mentioned in this chapter, see the online *XVT-Power++ Reference*.

2

INTRODUCTION TO XVT-ARCHITECT

This chapter introduces XVT-Architect and its main components, and it describes the interface elements of these components.

In addition, the chapter provides an overview of the process of designing and building an application with the tool, as well as a description of the process of saving projects and generating files with XVT-Architect.

2.1. What is XVT-Architect?

XVT-Architect is a visual, object-oriented GUI development tool and application generator. It is built using and is part of the XVT Development Solution for C++ (DSC++), which includes the following components: XVT-Power++, the XVT Portability Toolkits, the **curl** resource compiler, and the **helpc** help text compiler.

XVT-Architect is built with the XVT-Power++ application framework for use with the framework, and it is designed specifically to leverage XVT-Power++ development. While introducing you to the XVT-Power++ application framework, this tool also simplifies the design and implementation of XVT-Power++ applications.

Like all XVT-Power++ applications, applications developed using XVT-Architect are portable across all XVT-supported platforms. XVT-Architect's project files are also portable to all supported platforms. In addition, to simplify your porting tasks, XVT-Architect allows you to create platform monitor and language-specific attribute settings from any platform.

See Also: For more information on the DSC++ components, see the "Preface" of this manual.

2.2. Designing and Building Applications with XVT-Architect

When you design and build an application with XVT-Architect, you follow basic steps. This section briefly describes these steps. However, you can change the sequence of the steps, if you wish. As you get acquainted with XVT-Architect, you will find the approach that works best for your individual projects.

t To build an application with XVT-Architect, follow these basic steps:

1. Design and lay out your application using XVT-Architect's Blueprint, Drafting Board, and Strata modules, as well as XVT-Architect's editors.
2. Save the project. You must save the project before you can generate the Shell files (you can save the project as early and as often as you like).

   When you Save a project for the first time (or "Save As" a project), XVT-Architect prompts you for a name. The name that you indicate is used for the new project directory as well as for the name of the project. Each XVT-Architect project *must* be maintained in a unique directory.
3. Generate the Shell files, which include a file for each object, a startup file, a URL file, and a makefile. Typically, you generate the Shell files once. However, if you modify your project, rename a file, or delete a file, you should generate the Shell files again.

   By default, XVT-Architect scans the files and generates only the files that it cannot locate; XVT-Architect does *not* overwrite a file that already exists unless you indicate otherwise. You can choose to overwrite Shell files in the File Generation dialog.
4. Generate the project's object Factory, which includes your project files and header files. You use the Factory to create the objects in your application. Therefore, if you modify your project with XVT-Architect, you should generate the Factory again to update the information. However, when you regenerate the Factory, all of the project and header files are *overwritten*.
5. Establish a project file or makefile for your compiler, and add all necessary files. If you are using IDEs, you need to link any source files that are in the Factory directory.

6. Run **curl** to compile XVT's Universal Resource Language (URL), and add the generated ***.rc** file (or ***.r** file on the Macintosh) to your project. Every time you generate files, or regenerate files, it is a good idea to compile with **curl**. (For instructions on compiling resources with **curl**, see the sheet, "Installing XVT Development Solution for C++" for your particular platform.)
7. Compile, link, and execute your application. When you execute the application, the windows that you laid out in XVT-Architect will be displayed on the screen, but the application will not be interactive. You must write the code to implement the application.
8. Modify any files that you need to, and link in anything that you added as part of your application. If you change the names of any objects, you must (at least) edit the makefile.
9. Interact with the PAFactory class to create the user interface objects for your application.
10. Compile, link, and execute your application.

2.3. Visual Components

XVT-Architect's three main visual components, the Blueprint, Drafting Board, and Strata, have consistent interface elements. When appropriate, the menubars and toolbars contain the same items. In addition, both the Blueprint and the Drafting Board have attachable and detachable palettes to assist you in creating and laying out the objects of your application. (For more information, see section 2.3.1 through section 2.3.3.)

This section briefly describes these three main visual components of XVT-Architect. They work together to help you design your GUI applications. XVT-Architect allows you to navigate easily between these modules.

Blueprint Module
: A browser that allows you to visually lay out the basic internal architecture of your applications. In this module, you graphically create the application, documents, and windows that are the basis of your XVT-Power++ application. You establish the primary object hierarchy for your application.

Drafting Board Module
: A GUI builder with all of the facilities necessary to quickly produce advanced user interfaces. In this module, you can lay

out and manipulate the interface objects of your application; you can lay out XVT-Power++ visual objects, and you can manipulate them with a robust set of tools.

Strata Module

An object-attribute editor that provides quick access to both view and modify object attributes, letting you refine your application and user interface objects. In this module, you can set all of XVT-Power++'s data members, and you can use the class browser to understand and effectively use XVT-Power++'s class hierarchy and object inheritance. In addition, from the Strata, you can open all of XVT-Architect's editors.

See Also: For specific information on a component, see the corresponding chapter. For more information on editors, see Chapter 7.

2.3.1. Blueprint Interface

In the Blueprint, you have basic interface elements, including a menubar, a toolbar, and a status bar. In addition it has a Tools and an Alignment palette (see Figure 2.1).

Figure 2.1. Blueprint interface

2.3.2. Drafting Board Interface

The Drafting Board, much like the Blueprint, has a standard interface including a menubar, a toolbar, a status bar, and Alignment palette. In addition, the Drafting Board has a main View palette, which has several subpalettes. You will use these palettes for most of your work in the Drafting Board (see Figure 2.2).

Figure 2.2. Drafting Board interface

See Also: For a full description of the View palettes, see section 5.2.

2.3.3. Using XVT-Architect's Palettes

When you open the Blueprint or Drafting Board, the main tool palette is open. In the Blueprint, that is the Tools palette, and in the Drafting Board, that is the View palette. If you close a palette, you can reopen it by selecting the appropriate item from the Palettes menu.

All of the palettes of XVT-Architect are draggable, attachable, and detachable. You can position them in the window, or you can attach them to or detach them from the side, top, or bottom of the window, by dragging them to and from the edges.

t To create objects using XVT-Architect's Tools palette or View Palette and its torn-off subpalettes, use one of the following methods (all objects that you create in the Blueprint are a default size):

Drag-and-drop method
Press down a button on a palette, and drag the cursor to the

sketch region of the window and release the mouse button.When you release the button, an object of default size is created at the cursor location, and the cursor reverts to the previous tool.

Sketch method
Click a button on a palette, and click in the sketch region of the window. You can create multiple objects of default size by clicking multiple times. When you are done, click on another object or on the pointer tool of the palette.

Sketch method
Click a button on a palette and drag out an area in the sketch region of the window. With this method, you can create multiple objects of any valid size. When you are done, you can click on another object or on the pointer tool of the View palette.

See Also: For more information on the Tools palette, see section 4.3.1.1. For more information on the View palettes, see section 5.1.3.

2.3.4. Strata Interface

To view and modify an object's attributes, you go to its Strata.

t To open the Strata for an object, in either the Blueprint or Drafting Board:

Double click on the object.

Double clicking on an object always opens the Strata for that object. The Strata is divided into two main components: the class browser and the notebook control (see Figure 2.3). The class browser shows the class hierarchy for the object, and the notebook control contains a "page" for each class from which an object inherits. The pages are populated with controls for each attribute that you can set. Some classes have more than one page, which you can access from their main page (e.g., CWindow).

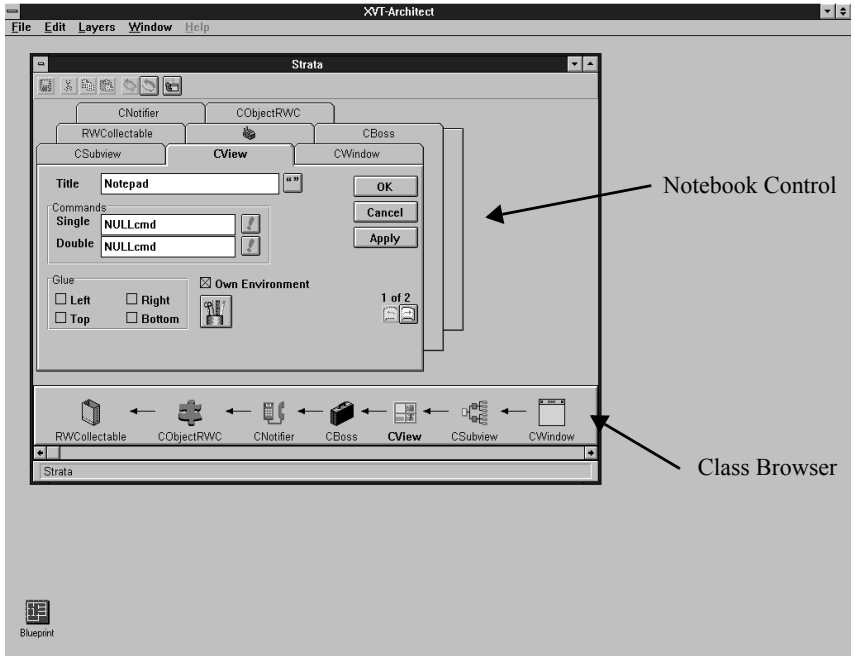


Figure 2.3. Strata interface

2.4. Saving Projects and Generating Files

From the Blueprint and the Drafting Board, you can generate the Shell files and the Factory project files. However, before you do so, you must save your project. If you attempt to generate files before saving the project, XVT-Architect brings up the Save dialog.

t To save your project:

Choose Save from the File menu, which brings up the Save dialog.

When you Save a new project (or "Save As" a new project), XVT-Architect asks for a new directory name. This name also serves as the name of the project. An XVT-Architect project *must* always be maintained in its own individual directory. The generated Factory and Shell files are placed in this directory.

Note: If you choose a file from the Save dialog's list box, and then choose OK, XVT-Architect issues a warning and requires that you enter a new directory.

Generating Files

t To generate files:

1. Choose Generate Files from the File menu, which brings up the XVT-Architect File Generation dialog.
2. Indicate which files you want generated.
3. Click the Generate button.

When indicating which files you want generated, you have several choices. You can choose to generate the Factory and/or the Shell files by checking or unchecking the appropriate check box. If you choose to generate the Shell files, you have several other choices, which are described in section 2.4.2.1.

2.4.1. Factory Files

If the Factory check box is checked, XVT-Architect generates the Factory files. The generated Factory files store information regarding the objects and object relationships of your XVT-Architect project. You interact with the Factory files when writing the necessary code to implement your application.

Your application uses the object factory at runtime to instantiate objects. The object Factory is separate from the user code. This clean separation of code makes it safe and easy to change and maintain your application.

If you modify your XVT-Architect project, you should generate, or regenerate, the Factory files, so that the project information is updated. When you regenerate the Factory, all of the files are overwritten.

Note: If after you generate the Shell files, you change the Factory name of an object, you must search and replace the old name in your application code.

XVT recommends that you do *not* modify the Factory files.

See Also: For more information on generated Factory files, see Chapter 8. For more information on Factory names, see section 4.4.1.2.

2.4.2. Shell Files

XVT-Architect uses templates to generate the Shell files for your application. To generate Shell files, XVT-Architect reads the template, processes it, and writes an output file. When writing these

files, XVT-Architect uses the project name to name some files, and class names that you have indicated in the Blueprint to name other files.

The File Generation dialog, has a list box that contains a list of all of the generated Shell files. When you bring up the dialog the Shell check box is checked and all of the files are selected. Therefore, all of the files will be generated. However, when generating Shell files, you can choose to generate only selected files.

Note: On MS-Windows and OS/2, class names that are over eight letters will be truncated during Shell-file generation.

See Also: For more information on class names, see section 4.4.1.1.

2.4.2.1. Generating Shell Files

You can uncheck the Shell check box, and XVT-Architect will not generate the Shell files. However, if you check the Shell check box, you can choose to generate all or part of the Shell files.

t To generate part of the files:

Deselect the files that you do *not* want generated.
-OR-
Click the Select None button, and then select the files that you want generated.

After you generate the Shell files, you can compile, link, and execute your application. The windows of the application will come up, but they will not be interactive. You can modify the generated files to add the needed code for your application. Unless you specify otherwise, these files are *not* overwritten during subsequent generations.

Generally, you generate the Shell files once. However, if you modify your project, rename a file, or delete a file, you should generate the Shell files again. If you generate them a second time, XVT-Architect scans the files and generates only the missing files; XVT-Architect does *not* automatically overwrite a file that already exists.

Note: You can select to overwrite existing files. If you do so, you will also overwrite and lose any modifications that you have made to these files.

2.4.2.2. Generated Files

This section describes the Shell files, using the convention whereby the complete name of a project is represented by **<ProjectName>**,

and the truncated, "8.3," name by **<TruncatedProject>**. Likewise, the complete name of a class is represented by **<AppClass>** (for example), and the truncated name by **<TruncatedAppClass>**.

The generated Shell files consist of the following:

Startup file
The startup file contains the application's main function. On MS-Windows, MS-Windows NT, and OS/2, it is **cstartup.cpp**. On the Macintosh, it is **CStartup.cp**. On UNIX it is **CStartup.cxx**.

Main application header file
The main application header file references the header file that the factory generates, so you do not have to add the reference. Every generated source file references this file; whatever you put in this file is global information. On MS-Windows, MS-Windows NT, and OS/2, it is **<TruncatedProject>.h**. On UNIX and the Macintosh, it is **<ProjectName>.h**.

Main application Universal Resource Language (URL) file
The URL file references the **url.h**, **PwrURL.h**, and **factory.url** files. This file references the factory generated **.url** file; you do not have to add this reference. If you want to reference or include any code (e.g., preexisting files or native resources), you should add it to this file. On MS-Windows, MS-Windows NT, and OS/2, it is **<TruncatedProject>.url**. On UNIX and the Macintosh, it is **<ProjectName>.url**.

Application class source and header files
The source and header files for the application class. On MS-Windows, MS-Windows NT, and OS/2, they are **<TruncatedAppClass>.h** and **<TruncatedAppClass>.cpp**. On the Macintosh, they are **<AppClass>.h** and **<AppClass>.cp**. On UNIX, they are **<AppClass>.h** and **<AppClass>.cxx**.

Document and window class source and header files
The source and header file for each document and window that you created and linked into your application. On MS-Windows, MS-Windows NT, and OS/2, the files are **<TruncatedClass>.h** and **<TruncatedClass>.cpp**. On the Macintosh, the files are **<Class>.h** and **<Class>.cp**. On UNIX, the files are **<Class>.h** and **<Class>.cxx**.

Application icon
The application icon. On MS-Windows and MS-Windows NT, it is **<TruncatedProject>.ico**. On OS/2, there are two files, **<TruncatedProject>.ico** and

<TruncatedProject>.bmp. On UNIX and the Macintosh, the icon definition is located in the main URL file.

Default icon

The default icon is used for icons placed inside windows. On MS-Windows and MS-Windows NT, it is **null.ico**. On OS/2, there are two files, **null.ico** and **null.bmp**. On UNIX and the Macintosh, the icon definition located in the main URL file.

Makefile or project file

The projects makefile or project file. The following is a list of a platforms, with compilers when necessary, and the name of the generated makefile or project file:

MS-Windows (Borland): **<TruncatedProject>.ide**
MS-Windows (MSVC): **<TruncatedProject>.mak**
MS-Windows NT: **<TruncatedProject>.mak**
OS/2 (Borland): **<TruncatedProject>.prj**
OS/2 (IBM C-Set++): **makefile**
Macintosh (Think C): **<ProjectName>.pi**
Macintosh (Symantec): **<ProjectName>.pi**
Macintosh (Metrowerks): **<ProjectName>.mu**
Macintosh (MPW): **makefile**
UNIX: **makefile**

AppDef.h

Contains the #defines for the objects in your application. This file is placed in the Factory directory, and you should not modify it.

3

XVT-ARCHITECT TUTORIAL

This Tutorial chapter demonstrates how to use XVT-Architect to build a simple application. While teaching you how to build the application, this Tutorial also illustrates the use of many of XVT-Power++ features.

Each section of this Tutorial walks you through parts of the layout process and teaches you to use different features of XVT-Architect. The section then describes portions of the application code that you need to write to implement the component.

Note: This Tutorial contains information about many key concepts that you need understand to successfully use the XVT Development Solution for C++ (DSC++).

3.1. The Notepad Application

This tutorial guides you through the creation of the Notepad application. The Notepad is a simple text file editor that allows you to open, edit, and save files.

3.1.1. Learning XVT-Architect

By building this application, you use and learn XVT-Architect's three modules: Blueprint, Drafting Board, and Strata. In addition, you learn about several of XVT-Architect's editors.

You also learn XVT-Architect's Shell and Factory file generation, as well as how to use the Shell files and the Factory interface to write your application code.

3.1.2. Learning XVT-Power++

The Notepad application demonstrates the use of the following features of the XVT-Power++ application framework:

- Object hierarchies based on the Application-Document-View paradigm, visualized in the Blueprint
- XVT-Power++ view classes and Rogue Wave Tools.h++ classes
- Persistence
- Handling key input

You will learn the use of several XVT-Power++ view classes, which are the visual objects that make up the user interface of your applications. You will lay them out in XVT-Architect's Drafting Board module, set their initial properties in XVT-Architect's Strata, and implement them in your application code.

In addition, XVT-Power++ has integrated the Rogue Wave Tools.h++ class library, which provides a set of collections, data structures, and utility classes that you can take advantage of. You will start using the Tools.h++ class library when you write the Notepad application code.

The Application-Document-View paradigm is the basis for almost all well-designed XVT-Power++ applications. This object hierarchy defines the infrastructure of XVT-Power++ applications. By laying out the Application-Document-Window relationship in XVT-Architect's Blueprint, you rapidly learn this paradigm.

3.2. Getting Started

t To begin, run XVT-Architect.

The Blueprint module is opened.

By default, the Blueprint has an instance of a CApplication-derived class, an instance of CTaskDoc, and an instance of CTaskWin. These objects are linked together to form the rudimentary application. You cannot delete these objects.

CTaskWin is a class that XVT-Power++ uses internally to represent the logical or physical window that carries the application menubar. On some platforms, this window is a container for all of the application windows.

CTaskDoc owns and manages CTaskWin. Both classes are private classes that only XVT-Power++ can instantiate. Each application has instances of a CTaskDoc and a CTaskWin.

Note: You can only have one project open in XVT-Architect at a time. However, you can copy and paste between project files, which allows you to structure your development for team-oriented development.

See Also: For more information on CTaskDoc and CTaskWin, see their individual descriptions in the online *XVT-Power++ Reference*.

3.3. Designing the Notepad Application

XVT-Architect's Blueprint is used to design the object hierarchy of an XVT-Power++ application, based on the Application-Document-View paradigm. In this paradigm, the application creates and manages the documents. Each document stores and provides access to data; it also and creates and manages windows to display this data.

This object hierarchy defines the message paths for inter-object communication and assigns categories of tasks to be performed at each level.

The Notepad application is organized according to this Application-Document-View paradigm. In the design process of this application, therefore, consider the roles of the application, the documents, and the windows.

The Application Object's Roles

The application object, or instance of a CApplication-derived class, manages the flow of the entire application. Its responsibilities include the following:

- Initializing the startup environment
- Creating the documents that must be created initially
- Responding to application events, such as requests to create new documents or to open saved documents
- Cleaning up and closing documents before exiting the application

The Notepad, like all XVT-Power++ applications, must have a single application object.

The Document Object's Roles

Document objects, or instances of CDocument-derived classes, create and manage the windows that display its data. Each document class is responsible for many tasks, including the following:

- Managing data, which can include creating data objects, saving and restoring data from disk or a data base
- Changing its own data as requested by the views that it manages
- Creating and destroying windows to display data

The Notepad application contains several types of document objects.

The View Object's Roles

Finally, applications contain views that display the data and allow the user to interact with the data. In XVT-Power++, window objects, or CWindow-derived classes, act as the top-level enclosures for all other views, and therefore are the view classes that you lay out in the Blueprint.

In the Notepad application, as in many applications, some windows are managed by the same document. In this example application, there are five windows and four documents.

Note: In the Blueprint, you define derived classes; you create CApplication-derived, CDocument-derived, and CWindow-derived classes. When you name an object in the Blueprint, you indicate that XVT-Architect should generate a new class by that name.

3.3.1. Defining the Application Object

Since all XVT-Power++ applications must have *one* application object, XVT-Architect automatically creates an application class for each new project. When you start XVT-Architect or open a new project, the application class is laid out in the Blueprint.

Name the application appropriately, however. This and all of the names that you give objects in the Blueprint are used as class names in the generated Shell files.

3.3.2. Defining the Documents and Windows

To the initial layout in the Blueprint, you add your own document and windows. For the Notepad application, you need to lay out only one document and one window.

The document, which will be named TNoteDoc, must manage the data associated with a single file being edited. The document must also manage the storage and retrieval of the file. The window, which will be named TNoteWin, must contain the appropriate views to display the contents of the file; it must also allow users to edit the file.

3.4. Building the Notepad

This section describes the process of building the Notepad. The Notepad allows the user to do the following:

- Open a file for editing
- Edit the file within a window
- Save the file back to disk
- Save and close the file before exiting the program

This section describes how to lay out and implement the Notepad window. You will learn the following:

- Using the Blueprint to visualize the object hierarchy of the application and to name classes
- Using the Drafting Board to lay out objects
- Using the Strata to set document, window, and view attributes, including environment settings, and to set Factory names
- Using the Menu Editor to define a menubar for the window
- Generating Shell and Factory files

- Building, compiling, linking, and running your application
- Writing code to respond to menu selections and keyboard input
- Writing code to implement persistence
- Writing code to notify the document of keyboard input

t To begin this section:

If it is not already running, start XVT-Architect.

3.4.1. Defining the Notepad's Application Classes

Starting in the Blueprint, lay out document and window classes necessary for the Notepad, name the classes, and link them into the existing application.

t In the Palette menu, choose Tools.

The Tools palette contains the following buttons:

- Pointer tool
- Linker tool
- CDocument tool
- CWindow tool

When you move the Pointer over the buttons on the palettes and toolbar, a short help statement for the button under the Pointer appears in the status bar.

Notice that you can drag the palette to any location in the window, or you can attach it to the side, top, or bottom of the window.

Name the application class for this project.

t To name the application for this project:

1. Select the name below the icon CApplication(), and type "TNoteApp".
2. Click in the sketch region of the window to complete the editing process.

Note: XVT recommends that you give your derived classes a prefix that is unique within the application. XVT uses "C" and "N" prefixes for its classes, and Rogue Wave uses the "RW" prefix. In this Tutorial, you will use "T" as a prefix to help recognize your derived classes. To assure that your generated files match the code presented in this Tutorial, use all of the class and Factory names that are given.

3.4.1.1. Defining the Documents and Windows

The next step is to create and name the document class, TNoteDoc, and the window class, TNoteWin (see Figure 3.1). When you create a document or window in the Blueprint, the name of the object is selected and ready for editing.

t To create documents or windows:

Press on the CDocument or CWindow tool of the Tools palette, and then drag the Pointer onto the sketch region. When you release the mouse button, a document or window object is created, and the cursor returns to the previously selected tool.
-*OR*-
To create multiple objects, click on the CDocument or CWindow tool of the Tools palette, and then, click in the sketch region of the Blueprint window. The tool remains selected, and you can create as many documents or windows as needed. To return to the Pointer tool, click on the pointer of the Tools palette.

t To name documents or windows:

1. With Pointer tool, click on CDocument and name it TNoteDoc.
2. Then click on CWindow and name it TNoteWin.

After defining documents and the windows, click on the Pointer tool. You can move the documents and window in the sketch region by dragging them.

3.4.1.2. Linking Documents and Windows into the Application

When you have laid out the documents and windows in the Blueprint, link them together to visualize the object hierarchy of your application.

In XVT-Power++, all documents are managed by the *single* application object. In addition, documents manage the windows that display their data. This forms the object hierarchy, which is at the center of the Application-Document-View paradigm.

First, link the document to the application.

t To link the document to the application:

Click on the Linker tool on the Tools palette, and then drag out a connection from the center of the TNoteDoc icon to the center of the TNoteApp icon.

Then, link the window to the document that manages it.

t To link the window to the document:
With the Linker tool, drag out a connection from TNoteWin to TNoteDoc.

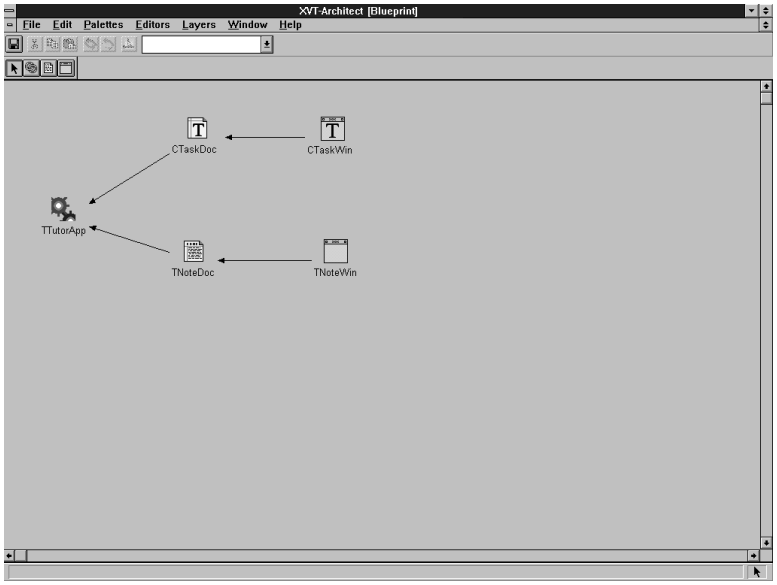


Figure 3.1. Blueprint with the Notepad's document and window

Note: Links must be made from the document to the application and from the window to the document. XVT-Architect does not allow invalid links.

See Also: For more information on linking rules, object hierarchies, and the Application-Document-View paradigm, see Chapter 4.

3.4.1.3. Setting the Document's Attributes

To view and modify the attributes of all objects (instances of classes) in XVT-Architect, go to the Strata of an object.

t To open the Strata of an object:

Using the Pointer tool, double click on the object.

To see the default document attributes, you can open the Strata for TNoteDoc now. The main components of the Strata are the class browser and the notebook control.

The class browser is at the bottom of the window, and it illustrates the class hierarchy of the selected object. To get an idea of the functionality an object is inheriting, you can look scroll through the class browser (using the scrollbar at the bottom of the window) or click on each item in the class hierarchy of the object.

Each class from which an object inherits also has a tabbed page in the notebook control. Each page contains the attributes that can be set for that class; in the Strata, view and set attributes of an object at each inheritance level of the class hierarchy. The Strata will help you learn the XVT-Power++ class hierarchy.

t To see the class hierarchy of the object:

Scroll through the class hierarchy in the browser.

t To bring a page of the notebook control to the front:

Click on the specific object icon in the class browser at the bottom of the window.
-*OR*-
Click on the tab of the specific class.

See Also: For more information on the Strata, see Chapter 6.

Setting TNoteDoc's Attributes

In your application code, you will need a way to refer to the document. When you interact with any object in your application, you use the Factory name.

XVT-Architect generates a Factory name for each object you lay out. However, you can change the Factory name so that it makes sense in your application. To set the name and other Factory information, use the Factory settings page in the Strata (see Figure 3.2).

t To set the Factory name for the document:

1. Double-click TNoteDoc's icon in the Blueprint.
2. Bring the Factory page to the front by clicking on the tab with the Factory bitmap.
3. Then, select the text in the Factory ID Name field, and type "NoteDoc", and click the Apply button, which applies the change.

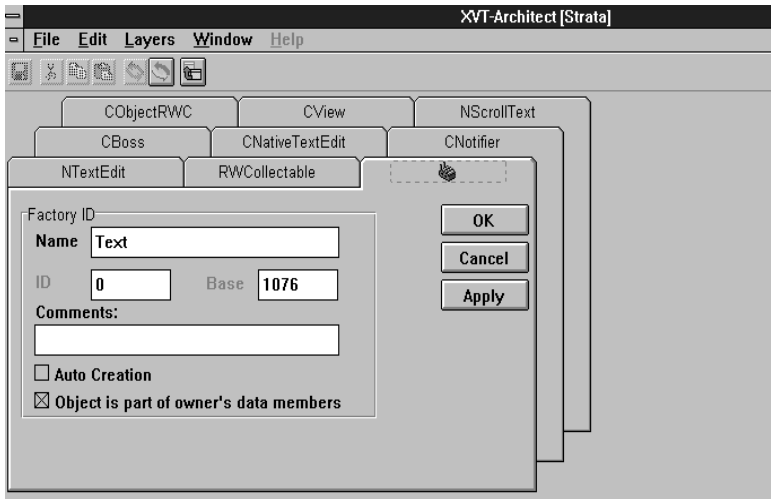


Figure 3.2. Factory settings page of Strata

While you have the Factory page at the front of the Strata, note the Auto Creation check box on the page. By default, this box is checked, and thus all documents, all windows for documents, and all objects contained in the windows are automatically created at runtime. To indicate otherwise, you must uncheck this box. However, for this Tutorial, leave it checked.

When you are done in the Strata, click OK, which closes the window.

Note: When choosing a Factory name, do *not* use the class name that you gave the object in the Blueprint.

A Note on Giving Factory Names to Objects

When you write your application code, you must access some of the objects that you have laid out in XVT-Architect. In general, you must access those objects whose attributes you will be getting or setting dynamically at runtime.

XVT-Architect automatically generates data member classes with pointers to the objects that you have indicated. That is, by default all objects are part of their enclosure's data member class, where the application encloses the documents, the documents enclose their windows, and the windows enclose views. You can change the setting in the Strata on the Factory settings page so that certain objects are not generated as part of the enclosure's data member class.

In addition, when you interact with the Factory, through the PAFactory public interface, you can pass in an instance of a data member class. If you do so, the methods return pointers to the nested objects.

XVT-Architect names the returned pointers by taking the object's Factory name and adding the prefix "its". For example, the returned pointer for the document will be "itsNoteDoc".

Generally, it is a good idea to change the Factory names of objects that you will be accessing in your code. Of course, you can change the Factory names of all the objects in your application, but that is not really necessary.

Later in the tutorial when you generate code with XVT-Architect, you will see how the attributes set in the Factory Page of the strata affect the code that is generated.

3.4.1.4. Setting the Window's Attributes

For the Notepad, you will lay out a text editing field that uses the window's scrollbars. To set this attribute and other window attributes, go to the Strata for the TNoteWin.

t To open the Strata for the Notepad window:

1. Double click on the TNoteWin object in the Blueprint, which opens the window's Strata.

To open the Strata for any object, either in the Blueprint or the Drafting Board, double click on the object.

In the Strata for TNoteWin, do the following:

2. Set the Title on the CView page to "Notepad"
3. Set the Factory ID Name to in the Factory page to "NoteWin"
4. Then set the CWindow attributes in the CWindow page. For this window, you need to check the following Decorations attributes:
   - Sizable (checked by default)
   - Closable (checked by default)
   - Iconizable (checked by default)
   - Draws Background
   - Scrollbar—Horiz
   - Scrollbar—Vert
5. When you have checked these attributes, click OK.

3.4.2. Laying Out the Notepad's Interface

When you have set the attributes for the application objects, you can lay out the window's text editing region in the Drafting Board (see Figure 3.3).

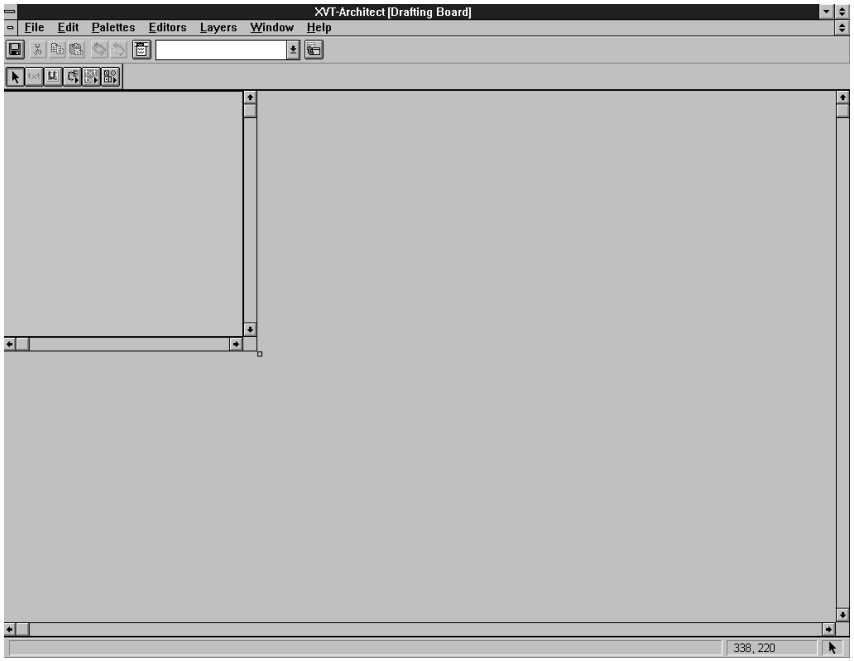


Figure 3.3. Final Notepad window—Drafting Board

3.4.2.1. Laying Out the Notepad Window

To lay out the interface of Notepad, you will use XVT-Architect's Drafting Board. The Drafting Board is a GUI builder with the facilities to lay out advanced user interfaces. To modify the attributes of the object that you lay out in the Drafting Board, you will use the Strata.

t To open the Drafting Board for the TNoteWin window:

Select TNoteWin, and click the Drafting Board button in the Blueprint's toolbar.

Using the Drafting Board, you can lay out all XVT-Power++ visual objects (i.e., you can lay out instances of classes that are derived from CView). To layout these objects, use the Views palette and its

subpalettes. When you open the Drafting Board, the Views palette is open.

The Views palette is similar to the Tools palette: it is draggable, attachable, and detachable. However, unlike the Tools palette, it has subpalettes. Buttons that have a subpalette have a small arrow in the bottom-right corner. You can tear off these subpalettes, and they will behave like the main Views palette.

To tear off a subpalette, click on any button with the Pointer and drag the palette into the sketch area. The subpalette will be detached. Drag the subpalette right up against the border to reattach it.

See Also: For more information on the Drafting Board and the Views palette, see Chapter 5.

Laying Out the NScrollText for the Notepad

In the Notepad, you lay out an instance of the NScrollText class (an NScrollText object), and specify that it should use the scrollbars that you just set for the window.

t To create and lay out an NScrollText object:

1. Click the CNativeTextEdit button on the main Views palette, and tear off the Text Edit subpalette. (Note that to locate any button in the View palettes, you can move the cursor across the palettes and a class name and short description appears in the status bar for the button beneath the cursor.)

2. On the Text Edit palette, click and drag the NScrollText button off of the palette.When you are in the position that you want the text object, release the mouse button.

Note: This method creates one NScrollText object of default size. When you release the mouse button, the cursor will revert back the previously selected tool, in this case the Pointer. To determine which tool is selected, look at the icon at the right side of the status bar.

3. Using the Pointer tool, select the NScrollText object, and use the wire frame to size the object so that it fills TNoteWin's *entire window* region. The "window region" is indicated in the Drafting Board's sketch area by a sizing frame with a handle. The sizing frame also has a representation of the window's scrollbars.

When you have laid out the NScrollText object, you need to modify its attributes using the Strata module.

See Also: As you work through this Tutorial, you will learn the Views palette and its subpalettes. For an overview of the Views palettes, see section 5.2.
For information on how to manipulate view objects, see Chapter 5.
For more information on any class mentioned in this Tutorial, see its entry in the online *XVT-Power++ Reference*.

3.4.2.2. Modifying the NScrollText Object's Attributes

To modify an object's attributes, you open its Strata. In the Strata, you need to modify the NScrollText's scrollbar assignment, color (environment setting), glue setting, and Factory ID name.

t To open the Strata for the NScrollText:

1. Using the Pointer tool, double click on the object.

Assigning the NScrollText Object Scrollbars

First, you need to indicate that the NScrollText object should use the window's scrollbars.

t To set the NScrollText to use the window's scrollbars:

2. Bring the NScrollText page to the front by clicking on its tab.
3. Check "Uses Window Scrollbars"
4. Click Apply.

Setting the NScrollText Object's Environment

Next, you need to set the environment for the NScrollText object. To set the environment of objects in your application, use the Environment Attributes dialog. An environment object encapsulates the following information:

- Text attributes, which include font family, style, and size and background and foreground colors
- Brush attributes
- Drawing mode
- Pen attributes

By default, there is an environment object defined at the application level of the object, which is shared by every visible object in an XVT-Power++ application. Because an environment propagates down the object hierarchy, many objects can share the same environment.

You can also attach a separate environment object at any level of the object hierarchy, anywhere from the application object to any view object in the application. In XVT-Architect, you use the Environment Attributes dialog to give an object its own environment.

t To open the Environment Attributes dialog:

1. Bring the CView page to the front by clicking on its tab.

2. Check the Own Environment box, which enables the Environment button.
3. Click the Environment button that becomes enabled just below the Own Environment field.

t To set the background color for the NScrollText object:

1. Select Blue from the Background list button in the Text Attributes field.
2. Click on the Color button to the right of the list button, which brings up the Standard Color Palette, and select a lighter shade of Blue from the palette. Click OK.
3. Click OK in the Environment Attributes dialog, and click Apply in the Strata.

Tip: You do not have to click Apply in every page of the Strata to apply changes. You can simply click OK when you have finished modifying the attributes of an object. However, if you would like the changes to be applied to the objects in the Drafting Board (so that you can see these changes without closing the Strata), click Apply.

See Also: For more information on the Environment Attributes dialog, see section 6.4.1.
For more information on environment settings, see section 16.2.2 or see the description of CEnvironment in the online *XVT-Power++ Reference*.

Setting the NScrollText Object's Glue and Active State

Since the window is sizable, you need to specify that the NScrollText should resize when the window size changes. You accomplish this by setting the glue properties of the object, which determine how an object behaves when its enclosure is sized.

t To set the glue properties and active state:

On CView's first page, check all of the Glue settings. (That is, check Left, Right, Top, and Bottom).

See Also: For more information on glue properties, see section 16.2.1 on page 16-6.

Setting the NScrollText Object's Factory ID Name

In your application code, you will need a way to refer to this object, so give it a Factory name.

t To set the Factory name for the NScrollText object:

1. Bring the Factory page to the front by clicking on the tab with the Factory bitmap.
2. Select the text in the Name edit field on the Factory page,
3. Type "Text".
4. To commit all of the changes that you have made in the Strata, click OK, which closes the window.

Sizing the Window

To resize the window, first locate the sizing frame of the window by scrolling the Drafting Board's sketch region. The sizing frame will also have a representation of the scrollbars that you specified for the window in its Strata.

t To size the window:

"Grab" the frame and drag.

To adjust the height of the window, grab the frame on the bottom. To adjust the width, grab the frame on the right side. To adjust the height and width, grab the handle on the corner of the frame.

See Also: For information on Factory names, see section 6.7. For more information on the Factory interface, see section 8.2.

3.4.2.3. Setting the Menubar for the Notepad Window

For the Notepad, you will create a menubar using XVT-Architect's Menu Editor.

t To open the Menu Editor:

Click the Menu Editor button on the Drafting Board's toolbar.

When you open the Menu Editor, it is populated with standard menus. (You may have to move the Menu Editor window.)

For the Notepad window, you will delete the Font menu.

t To delete the Font menu:

1. Select the Font menu by clicking on it.
2. Click on the Delete button in the toolbar (it is the second button from the right).

When the notepad is running, several menu items should already be enabled including "New", "Open" and "Save As". To enable these items:

1. Select the File menu by clicking on it.
2. Select the "New" menu item. (It is grayed out, but you can still click on it.)
3. Check the "Enabled" box in the lower half of the window.
4. Repeat the steps for "Open" and "Save As".
5. Click OK, which closes the window and saves the menubar as part of the window.

The changes made above affect the menubar used by the NotePad window. You may also define a default menubar that is used when no windows are open. To do so:

1. Close the Drafting Board.
2. Go to the Blueprint Window.
3. Open the Drafting Board for the CTaskWin class.
4. Edit the menu of this window in the same way you edited the NScrollText object's menu.
5. Also remove the Font Menu and enable the "New" and "Open" menu items.

See Also: For more information on using the Menu Editor, see section 7.1.

3.4.3. Generating the Application

When you have finished laying out the Notepad component, you need to save the project and generate the application.

Saving the Project

When you Save a new project (or "Save As" any project), XVT-Architect prompts you for a name. This is the name of the new project directory as well as the name of the project. An XVT-Architect project should always be maintained in its own individual directory, because every project has its own Factory subdirectory.

t To save your new project:

Select Save from the File menu in the Blueprint or Drafting Board, and type **notepad**, which is the name of the new project directory. This name is also used to name many of the generated files.

It is a good idea to save your project regularly. When you save a project for the first time, you create an **.amf** project file, which is a portable file (**amf** stands for *architect meta file*). All of the XVT-Architect project information is stored in this file.

Note: You must save your project before you generate the Factory and Shell files. If you attempt to generate files before initially saving the project, however, XVT-Architect brings up the Save As dialog.

Generating Factory and Shell Files

Next, generate the Factory and Shell files for your application.

The Factory is a repository that stores information about the objects in the application. However, unlike the project file, you interact with the Factory to create the objects of your application. You do *not* modify the Factory files.

The Shell files are an initial set of user files, which you can add to and modify. The initial set of Shell files include a startup file, source and header files for the application class, each document class, and each window class, Universal Resource Language (URL) files, and a makefile. All of your application code goes in the Shell files or files that you add to this initial set.

This clean separation of user and Factory code makes modifying and maintaining your applications safe and easy. You can change your

project in XVT-Architect, and regenerate the Factory files without affecting your user files.

t To generate the Factory and Shell files:

1. Select Generate Files from the File menu, which opens the XVT-Architect File Generation dialog.
2. Click the Generate button. By default, both Factory and Shell are checked to be generated.

This generates the necessary files. In later generations, use this same process. Using the default generation process, XVT-Architect generates all Factory files, overwriting the existing ones, and it generates only missing Shell files.

See Also: For more information on saving and generating files and the generated Shell files, see section 2.4.
For information on the object Factory and its generated files, see Chapter 8, *Object Factory*.

A Look at Generated Code

Take a look at the files that have been generated by XVT-Architect. There should be directory named "notepad". Inside you will find the following files (the file extensions may vary depending on platform or options settings):

cstartup.cpp	Contains the main() of the application
factory/	Subdirectory containing factory files
notepad.amf	XVT-Architect meta file (the notepad project)
notepad.h	Header file for application global symbols
notepad.ico	Customized icon for the application
notepad.mak	Makefile (name/type may vary per platform)
notepad.url	Resource file
null.ico	Default icon for the application
tnoteapp.h/.cpp	Templates for TNoteApp class
tnotedoc.h/.cpp	Templates for TNoteDoc class
tnotewin.h/.cpp	Templates for TNoteWin class

Of particular interest are the template files generated for the classes in the Blueprint. For example, open the **tnotedoc.h** file to see the following class definition:

```
#ifndef TNoteDoc_h
#define TNoteDoc_h

#include "XVTPwr.h"
#include "AppDef.h"

#include NOTEPAD_i
#include CDocument_i

#ifdef DSP_RELEASE
#include TNoteDoc_f
#endif

//#include CTypeInfo_i

class TNoteDoc : public CDocument
{
public:
        TNoteDoc( CApplication * theApplication = G -> GetApplication( ),
        PWRID theDocumentId = G -> GetId( ) );
        virtual ~TNoteDoc( void );

        virtual void BuildWindow( void );
        virtual void DoCommand( long theCommand, void * theData = NULL );
        virtual void DoMenuCommand( MENU_TAG theMenuItem,
        BOOLEAN isShiftKey, BOOLEAN isControlKey );

        virtual BOOLEAN DoSave( void );
        virtual BOOLEAN DoOpen( void );

        virtual BOOLEAN Save( void );
        virtual BOOLEAN Open( void );

protected:
        // By default, copy and assignment are disallowed

        TNoteDoc( const TNoteDoc& theDocument ) : CDocument( theDocument ) { }
        TNoteDoc& operator=( const TNoteDoc& theDocument ) { return( *this ); }

private:
        NoteDocData itsData

       //      PWRClassInfo
};

#endif //TNotedoc_h
```

The two lines shown in boldface type above reveal interesting information. The first line includes a special header file generated in the factory directory:

```
#include TNoteDoc_f
```

For each class in the Blueprint, XVT-Architect generates a special header file in the factory directory where it defines special support symbols and classes. You will look at that file soon to find out about its contents.

The second line declares a data member of type NoteDocData:

```
NoteDocData itsData;
```

This NoteDocData type is declared inside TNoteDoc_f in the factory directory. This is a class derived from CDataMembers which contains pointers to all windows managed by the document. If you look at the template files for TNoteApp you will see a similar data member class which contains pointers to all documents in the application.

Similarly, TNoteWin contains a data member with pointers to all views. As mentioned earlier, you can control which pointers are actually added to these data member classes by editing the factory options page inside the Object Strata for each class.

Open the TNoteDoc_f file in the factory directory. The file's name is actually **_tnotedoc.h**; below is shown most of its contents:

```
// Factory IDs:

#ifndef NoteDoc
#define NoteDoc 1009
#endif

// Object Data Members:

class NoteDocData : public CDataMembers
{
      PWRClassInfo
      virtual void Initialize(PAFactory* theFactory);

      public:

      TNoteWin* itsNoteWin;
};
```

Among other things, you can see that this file defines factory IDs for the document and then defines the NoteDocData class with a pointer to the window managed by this document.

Finally, open the **tnotedoc.cpp** file to see the implementation of this class. Take a look at the BuildWindow() method:

```
void TNoteDoc::BuildWindow( void )
{
// Create all window objects associated with
// this document that have the AutoCreate option
// specified in XVT-Architect.

    NOTEPADFactory.DoCreateWindows( this,
                    NoteDoc, FALSE, &itsData );
}
```

Each document class must define a BuildWindow() method to create actual window objects that view the data managed by the document. The code in this generated method takes care of this action by using the XVT-Architect factory to instantiate any window which is linked to this document inside the XVT-Architect Blueprint. The factory object, NOTEPADFactory, is created internally by code in the factory directory. You are encouraged to treat this code as a black box, and therefore it will not be explored further in this tutorial.

See Also: For more information on factory object creation, refer to the description of the PAFactory class in the online *XVT-Power++ Reference*.

3.4.4. Building and Running the Basic Application

When you have generated your application with XVT-Architect, you can build and run the basic application. When you run the application, the window appears, but it does not have the ability to read and write the files. You must write the application code to implement this functionality.

t To build your application:

1. Set up your generated makefile or project file for your compiler, and add all necessary files. (Enter make -f notepad.mak.)
2. Run **curl** to compile XVT's Universal Resource Language (URL), and if necessary, add the generated file to your project. Every time you generate files, it is a good idea to compile with **curl**. On most platforms, the execution of **curl** is automatically embedded into the generated makefile, and it is executed automatically when you "make" the application.

   See the *XVT Platform-Specific Book* for your platform for information about **curl** that is specific to your platform.
3. Compile and link your application.

4. Run your application, and the Notepad window will come up.

The Notepad program may be in a subdirectory.

To implement the edit and save functionality of the Notepad, you must close the application and modify the generated Shell files.

Note: If you want to change the layout of the Notepad after you see it running, go back into XVT-Architect, and make your changes and regenerate the Factory files. Then, you can compile, link, and run the application again.

See Also: For instructions on compiling resources, see the *XVT Platform-Specific Book* for your specific platform.

3.4.5. Writing the Notepad Code

XVT-Architect generates Shell files for each document and window classes indicated in the Blueprint. XVT-Architect generates a header and a source file for each class. This section describes the modifications that you must make to the Shell files to implement the Notepad.

The Notepad is a component that reads and writes text files. The document's role as data manager is to provide the interface with the stored files. The window's role is to display the files and allow the user to edit their contents.

3.4.5.1. Modifying the TNoteDoc Class

To implement the reading and writing features of the Notepad, start by modifying the TNoteDoc header file, which is the document class that XVT-Architect generated.

To add the save and load mechanisms to the Notepad, you must modify the generated TNoteDoc class. The following is the modified **tnotedoc.h** header file, with the necessary changes indicated with bold type:

```
#ifndef TNoteDoc_H
#define TNoteDoc_H
#include "XVTPwr.h"
#include "factory/AppDef.h"
#include Notepad_i
#include CDocument_i
#ifndef DSP_Release
#include TNoteDoc_F
#endif
#include CStringRW_i
//#include CTypeInfo_i
class TNoteDoc : public CDocument
{

public:

        TNoteDoc(CApplication* theApplication =
                G->GetApplication(),
                PWRID theDocumentId = G->GetId());
        virtual ~TNoteDoc(void);
        virtual void BuildWindow(void);
        virtual void DoCommand(long theCommand,
                void* theData=NULL);
        virtual void DoMenuCommand(
                MENU_TAG theMenuItem,
                BOOLEAN isShiftKey,
                BOOLEAN isControlKey);
        virtual BOOLEAN DoSave(void);
        virtual BOOLEAN DoSaveAs(void);
        virtual BOOLEAN DoOpen(void);
        virtual BOOLEAN Save(void);
        virtual BOOLEAN Open(void);

protected:

        // By default, copy and assignment are
        // disallowed
        TNoteDoc(const TNoteDoc& theDocument) :
                CDocument( theDocument ) {}
        TNoteDoc& operator=(
          const TNoteDoc& theDocument){ return*this; }
private:

        CStringRW itsText;
        NoteDocData itsData;
//      PWRClassInfo
};
#endif // TNoteDoc_H
```

Modifying TNoteDoc's Source File

The following sections describe the methods that you need to add the **tnotedoc** source file to implement the Notepad. Note that the Shell file has method "stubs" for some of your document class methods. In these cases, you add code only to the necessary methods. The code that you need to add or modify is indicated in bold.

Add the following statements to the top of **tnotedoc**'s source file, after the TNoteDoc_i include statement:

```
#include "XVTPwr.h"
#include "AppDef.h"
#include NotePad_i
#include TNoteDoc_i
#include CMenuBar_i
        // Used to update the menu bar
#include <fstream.h>
        // Used to read and write file streams
#include TNoteWin_i
```

You do not need to do anything to the constructor because the initialization of the document's data is delayed until the object receives a DoNew or DoOpen message. By default, these CDocument methods invoke the appropriate file dialog box to allow users to select a file.

The following is the TNoteDoc constructor:

```
TNoteDoc::TNoteDoc(
        CApplication* theApplication,
        PWRID theDocumentId)
                : CDocument(
                        theApplication,
                        theDocumentId)
{
}
```

Now, locate the Open method of TNoteDoc. The Open method is called when DoOpen has successfully chosen a file to be opened.
To this method, add code to read the file from an input file stream, like this:

```
BOOLEAN TNoteDoc::Open(void)
{
        // Add code here to set internal data members
        // of the class based on information read from
        // some data source
        itsText = "";
        ifstream aFileStream(
                itsXVTFilePointer->name);
        itsText.readFile(aFileStream);
        SetSave( FALSE );
        return ( !NeedsSaving());
}
```

Locate the BuildWindow method. Once the file to be opened is selected, this virtual method is sent a message to create the views that display the new or opened data.

Modify the BuildWindow method as follows:

```
void TNoteDoc::BuildWindow( void )
{
        // Create all window objects associated with
        // this document that have the AutoCreate
        // option specified in XVT-Architect.
        NOTEPADFactory.DoCreateWindows(this, NoteDoc,
                FALSE, &itsData );
        // Initialize window with text
        itsData.itsNoteWin->SetText(itsText);
        if (itsXVTFilePointer)
                itsData.itsNoteWin->SetTitle(
                itsXVTFilePointer->name);
}
```

In the generated BuildWindow method, the document creates its views. XVT-Architect places a call to the Factory that creates all of the windows linked into (enclosed by) the document.

The itsXVTFilePointer data member is automatically initialized for you when a DoOpen message is sent to the document. This data member is a FILE_SPEC structure that contains the name of the file along with other information about the file, such as directory information. Use this information to set the title of the newly created window, which you now have access to through the itsData data member.

Now, modify the Save method to save the contents of the Notepad, like this:

```
BOOLEAN TNoteDoc::Save(void)
{
        // Save the contents of the notepad to a file
        itsText = itsData.itsNoteWin->GetText();
        ofstream aFileStream(
                itsXVTFilePointer->name);
        aFileStream << itsText;
        SetSave( FALSE );
        return ( !NeedsSaving());
}
```

Next, override the DoSaveAs method by inserting the lines shown below at the end of the file:

```
////////////////////////////////////////
// DoSaveAs
//
// This method is automatically called
// when the File Save As menu option is
// selected.
//
////////////////////////////////////////
BOOLEAN TNoteDoc::DoSaveAs(void)
{
      // Extend base class behavior to set
      // the window's title.
      if (CDocument::DoSaveAs())
      {
            itsData.itsNoteWin->SetTitle(
                  itsXVTFilePointer->name);
      return TRUE;
      }
      else
            return FALSE;
}
```

Note that the inherited CDocument::DoSaveAs method is called first. This inherited method is called automatically when the user chooses Save As from the File menu. Here, you extend this method so that the title of the window reflects the name of the file currently open.

3.4.5.2. Modifying the TNoteWin Class

Once the document is prepared to open and save files, you must modify the generated TNoteWin class. The TNoteWin class needs to display the contents of a file and notify its document of changes to the file, so that the document can update the file when necessary.

The following is the modified **tnotewin.h** header file, with changes indicated with bold type:

```
#ifndef TNoteWin_H
#define TNoteWin_H

#include "XVTPwr.h"
#include "factory/AppDef.h"

#include NotePad_i
#include CWindow_i

#ifdef DSP_RELEASE
#include TNoteWin_f
#endif

//#include CTypeInfo_i

class TNoteWin : public CWindow
{
public:
        TNoteWin(CDocument *theDocument,
                const CRect& theRegion,
                const CStringRW& theTitle  = NULLString,
                long theWindowAttributes = WSF_NONE,
                WIN_TYPE theWindowType   = W_DOC,
                int theMenuBarId         = MENU_BAR_RID,
                WINDOW theParentWindow   = TASK_WIN);
        virtual ~TNoteWin( void );

        BOOLEAN INoteWin( void );

        virtual void DoCommand(long theCommand,
                void* theData=NULL);
        virtual void DoMenuCommand(MENU_TAG theMenuItem,
                BOOLEAN isShiftKey,
                BOOLEAN isControlKey);
        virtual void UpdateMenus(CMenuBar* theMenuBar);
        virtual void DoUpdateModel(long theControllerId,
                long theCommand,
                const CModel* theModel);

        virtual void Key(const CKey& theKey);
        void SetText(const CStringRW& theText );
        CStringRW GetText(void);

protected:

        // By default, copy and assignment are
        // disallowed
         TNoteWin(const TNoteWin& theWindow) :
                CWindow( theWindow ) {}
         TNoteWin& operator = (
                const TNoteWin& theWindow){ return *this; }

private:

        NoteWinData itsData;

//      PWRClassInfo
};

#endif // TNoteWin_H
```

Modifying TNoteWin's Source File

The following sections describe the methods that you need to add to the **tnotewin** source file to implement the Notepad. Note that the Shell file provides method "stubs" for some of your window class methods. In these cases, you add code to only the necessary methods. The code that you add or modify is indicated by bold.

The following indicates the statements that you should add or uncomment at the top of **tnotewin**'s source file, after the TNoteWin_i include statement:

```
#include "XVTPwr.h"
#include "AppDef.h"

#include TNoteWin_i
#include CDocument_i
#include NScrollText_i
#include CNavigator_i
#include CMenuBar_i
```

Override the Key method so that the document is informed of a change in the Notepad by adding the following lines of code to the end of the file:

```
//////////////////////////////////////
// Key
//
// Automatically called when the user
// types on the keyboard and this window
// has the input focus.
//
//////////////////////////////////////
void TNoteWin::Key(const CKey& )
{
        // Tell the document the data has
        // changed when the user types in the
        // notepad
        itsDocument->SetSave(TRUE);
}
```

The call to SetSave() tells the document that its data has been modified. CDocument reacts by default to such a change by calling all of its window's UpdateMenus() virtual methods. Inside UpdateMenus() a window can set its menubar according to its state. The generated TNoteWin class already has a UpdateMenus() method with some commented-out common code. Remove the comments for the line of code, which in turn enables the M_FILE_SAVE menu item according to the document's need for saving:

```
void TNoteWin::UpdateMenus( CMenuBar * theMenuBar )
{
    // Enable the File=>Save menu option only
    // when the data needs to be saved
    // Enable the Edit=>Paste menu option if this
    // window supports pasting and there is something
    // paste-able on the clipboard

    //theMenuBar->SetEnabled( M_EDIT_PASTE, ...);
    theMenuBar->SetEnabled( M_FILE_SAVE,
        itsDocument -> NeedsSaving( ) );
}
```

Next, add the SetText method, which is a modifier used in setting the current state of the Notepad window. Note that you use the Suspend and Resume methods to avoid flashing. This approach suspends updates until the text has been set. Add the following SetText method:

```
////////////////////////////////////////
// SetText
//
// Allow the document object to set the
// contents of the notepad.
//
////////////////////////////////////////
void TNoteWin::SetText( const CStringRW& theText )
{
    itsData.itsText->Suspend();
    itsData.itsText->SetText( theText );
    itsData.itsText->Resume();
}
```

Finally, add the GetText method to the TNoteWin class, so that you can query the current state of the Notepad window. Add the following GetText method:

```
////////////////////////////////////////
// GetText
//
// Allow the document object to retrieve
// the contents of the notepad.
//
////////////////////////////////////////
CStringRW TNoteWin::GetText( void )
{
    return itsData.itsText->GetText();
}
```

You are done modifying the TNoteDoc and TNoteWin classes.

3.4.6. Compiling and Running the Application

When you are done modifying the Notepad files, you can compile, link, and run the application. You can then use the Notepad to open, edit, and save files.

4

BLUEPRINT

This chapter describes the interface and usage of the Blueprint module of XVT-Architect. The Blueprint allows you to visually design, using XVT-Power++'s Application-Document-View paradigm, the internal architecture of your applications. In the Blueprint, you can do the following:

- Layout the application, documents, and windows (top-level views) that form the basic architecture of your application
- Establish inter-object communication by connecting the created objects
- Use XVT-Architect's Editors to view and modify global information
- Define the "object layers," which allows you to create variations of windows and their enclosed views

This chapter describes how to use the features of the Blueprint module. However, it first describes the features of XVT-Power++'s application framework that the Blueprint module assists you in learning and using: XVT-Power++ object hierarchies and the Application-Document-View paradigm, which is the basis for all XVT-Power++ applications.

See Also: For more information on object layering, see Chapter 9, *Object Layering*.

4.1. Understanding Object Hierarchies and the Application-Document-View Paradigm

The XVT-Power++ application framework contains two types of hierarchies: a class hierarchy and an object hierarchy. The class hierarchy shows the XVT-Power++ class library and its inheritance structure. The Strata module of XVT-Architect illustrates and helps you use, to its fullest extent, the XVT-Power++ class hierarchy.

The Blueprint module of XVT-Architect illustrates and helps you use the XVT-Power++ object hierarchy. The object hierarchy defines the relationship between objects in an application. It sets up the structure that determines how messages are propagated and controls what tasks are performed at each level.

XVT-Power++'s object hierarchy is based on the model-view-controller (MVC) paradigm, a well know mechanism used for organizing and maintaining information in a dynamic system. However, XVT-Power++'s object hierarchy consists of the following three levels: application, document, and view.

Application
: Controls the program and is analogous to main. Applications consist of a set of documents.

Document
: Accesses, stores, and manages data. Documents manage windows and can be displayed in any number of views.

View
: Displays data for user interaction.

XVT calls these three levels the Application-Document-View paradigm. It is the basis for almost all well-designed XVT-Power++ applications. The Blueprint module of XVT-Architect illustrates and helps you use the Application-Document-View paradigm.

See Also: For more information on the application-document-view paradigm and the XVT-Power++ application framework, see the "XVT-Power++ Overview" and the "Application Framework" chapters in the *Guide to XVT Development Solution for C++*.

4.2. Application, Documents, and Views

For every application you build with XVT-Architect, you begin in the Blueprint module. In the Blueprint, you design the object hierarchy of your application.

By default, the Blueprint has an instance of a CApplication-derived class, an instance of a CTaskDoc class, and an instance of a CTaskWin class. Each application has a CTaskDoc and a CTaskWin. Both classes are private classes that only XVT-Power++ can instantiate.

CTaskWin is a class that XVT-Power++ uses internally to represent the logical window that carries the application menubar. On some platforms, this window is a container for all of the application windows. The CTaskDoc owns and manages the CTaskWin.

This layout creates the rudimentary architecture of your application based on XVT-Power++'s Application-Document-View paradigm. To this, you can add and link in the documents and windows (views) for your application, which sets up the basic object hierarchy.

4.2.1. Application Object

The application object, an instance of a CApplication-derived class, is the highest object in the object hierarchy. In each application, the application object is the first to be instantiated, the first to receive control, and the last to be destroyed.

The application object performs many application management functions, and it controls the program from start to finish.The application object is responsible for the following functions:

- Initializing any objects that the program needs upon application startup
- Initializing any connections the application needs
- Setting up global objects and global data, which are provided to all XVT-Power++ objects through CObjectRWC
- Creating and managing an application's documents
- Cleaning up after the program upon application shutdown

Note: CApplication, CDocument, and CWindow are all abstract classes. In the Blueprint, you actually create instances of derived classes (i.e., instances of both CDocument-derived classes and CWindow-derived classes).

See Also: For more information on application objects, see the "Applications" chapter in the *Guide to XVT Development Solution for C++*, and see CApplication in the *XVT-Power++ Reference*.

4.2.2. Document Objects

The document object, an instance of a CDocument-derived class, is the second layer of the object hierarchy. The document object links the application object and the window objects and their contained views; basically, the document object links the "back-end" of the application to the "front-end."

Each document object creates and manages a window or set of windows. Windows act as the top-level view for all XVT-Power++ applications. The document object can display its data in one or more windows, and it functions as a central means of communication for the changes and updates in its windows.

XVT-Power++'s paradigm supports document-centric application development, which helps you efficiently build applications that are easy to extend, modify, and maintain.

4.2.2.1. Document-Centric Development

The document-centric approach to application development focuses on data and servicing data. In XVT-Power++, the view is separate from and secondary to the application's data.

The separation of data from views is a logical separation of resources and expertise. That is, you can easily divide projects into data-based teams and user-interface teams. Because these teams have distinct and unique goals, large portions of a projects can be worked on concurrently, which promotes productivity.

This separation of data from views also allows you to easily extend and modify either your application's data model or user interface (without having to modify the other). In addition, you can create generic view interfaces, which can be used by several applications.

See Also: For more information on document objects, see the "Documents" chapter in the *Guide to XVT Development Solution for C++*, and see CDocument in the *XVT-Power++ Reference*.

4.2.3. View Objects

Window objects, instances of a CWindow-derived classes, act as the top-most enclosures for any other type of XVT-Power++ view, or CView classes. Windows are the top-level view in the nesting of views, and they are the third layer of the object hierarchy. Windows connect their contained views to the document.

Window objects have a predefined physical enclosure, which is platform-specific. A window's physical enclosure is either the screen itself, or the task window (on XVT/Win16, XVT/PM, and XVT/Win32).

Thus, the windows of your application contain the views and subviews (the visible objects) that will display the data and allow the user to interact with the application. XVT-Power++ supports a powerful set of view classes designed to support multiple target domains.

In the Drafting Board module of XVT-Architect, you can lay out the views and subviews that make up the GUI interface of your application.

See Also: For information on laying out views and subviews in windows, see Chapter 5, *Drafting Board*.
For additional information on views and subviews, and windows, see Chapter 16, *Manipulating Views and Subviews*.
For additional information on windows, see Chapter 18, *Windows*.

4.2.4. Inter-Object Communication and Message Propagation

As described above, the XVT-Power++ object hierarchy assigns categories of tasks to be performed at each level and defines the message paths for inter-object communication.

The application's communication paths are based on these relationships as suggested by the Application-Document-View paradigm and as defined in the Blueprint. You can think of the application as the enclosure of the documents, the documents as the enclosure of the windows, and the windows as the enclosures of other views.

All XVT-Power++ applications can propagate messages from an enclosure to its enclosed objects, or from an enclosed object to the enclosure. These messages are propagated in three different ways: bidirectional, upward, or downward.

This process is called "message propagation." With message propagation, you don't have to specifically define callbacks and determine where messages will go next. This simplifies your tasks and adds consistency throughout all XVT-Power++ applications.

See Also: For more information on message propagation, see the "Propagating Messages" section of the "Application Framework" chapter in the *Guide to XVT Development Solution for C++*.

4.3. Blueprint Interface

In the Blueprint, you have basic interface elements, including a menubar, a toolbar, and a status bar (see Figure 4.1).

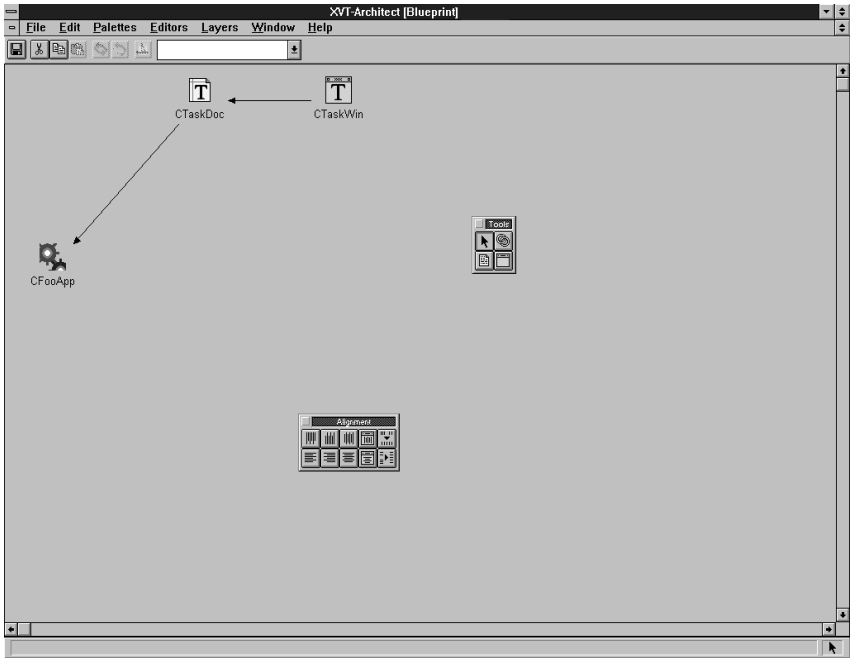


Figure 4.1. Blueprint interface

4.3.1. Menubar

The menubar of the Blueprint module includes the following menus:

- File
- Edit
- Palettes (the Tools and Alignment palettes are described below)
- Editors
- Layers
- Help

The File and Edit menus contain standard items. However, the File menu also contains Generate, Import, and Export.

Select the Generate item from the File menu to generate either Factory or Shell files. Select Import or Export from the File menu, to import or export files to and from XVT-Architect.

The Palettes menu contains Tools and Alignment items (which are described below).

The Editors menu gives you access to XVT-Architect's global editors. Using these Editors, you can edit the definitions of the following elements of your application: Menus, Accelerators, Commands, Strings, and String Lists.

Use the Layers menu to open the Layers Editor, which you can use to create and view layers, and to change the parent layer of layers in your project.

See Also: For information on generating files, see section 2.4 on page 2-8.
For more information on XVT-Architect's editors, see Chapter 7, *Editors*.
For more information on layering, see Chapter 9, *Object Layering*.
For information on importing and exporting files, see Chapter 11, *Importing and Exporting Strings*.

4.3.1.1. Tools Palette

When you first open the Blueprint, the Tools palette is open. However, if you close the Tools palette, you can reopen it by selecting Tools from the Palettes menu.

Like all of the palettes of XVT-Architect, the Tools palette is attachable and detachable—you can attach it to or detach it from the sides of the window simply by dragging it to and from the edges.

You use the Tools Palette when you lay out and link in documents and windows. It contains the following tools:

Pointer
Use as a basic pointer tool.

Linker
Links documents and windows into the application.

CDocument
Creates documents.With this tool, you actually create an instance of a CDocument-derived class.

CWindow
Creates windows. With this tool, you create an instance of a CWindow-derived class.

See Also: For more information on using these tools, see section 4.4.

4.3.1.2. Alignment Palette

With the Alignment palette, you can align and space the documents and windows that you have laid out.

t To open the Alignment palette:

Select Alignment from the Palettes menu.

Use this palette to align and space the documents and windows in the following ways:

- Align multiple objects along a vertical line by specifying left, center, or right alignment
- Align multiple objects along a horizontal line by specifying top, middle, or bottom alignment
- Align objects within the page by specifying horizontal center or vertical center of the page
- Space objects evenly by specifying vertical or horizontal spacing

t To select multiple objects:

Press Shift, and click on the objects.
-OR-
Using the Pointer tool, drag out a selection rectangle around the objects.

4.3.2. Toolbar

The toolbar contains a series of menubar accelerators, including the following:

- Save
- Cut
- Copy
- Paste
- Undo
- Redo

In addition, on the toolbar there is a Drafting Board icon, which takes you to the Drafting Board for the selected window. Next to this icon is a Child Window list button, which lists all of the “child” windows that are open.

The Blueprint can have Drafting Board, Strata, and Menu Editor windows as children. You can open a Drafting Board for each

window and a Strata for each class laid out in the Blueprint. You can also open the global Menu Editor.

4.3.2.1. Undo and Redo

All instances of Undo and Redo in XVT-Architect provide the ability for an unlimited number of Undo or Redo actions. The following are the only times that Undo and Redo are disabled:

- The first time you open a window
- When you have undone or redone all of the operations

See Also: Undo/Redo information is not saved between XVT-Architect sessions.

4.3.3. Status Bar

The status bar at the bottom of the Blueprint window, contains a short cursor-status description on the left side, and it contains the icon of the selected tool on the left side.

4.4. Laying Out the Application, Documents, and Views

The Blueprint module of XVT-Architect helps you design and manage the Application-Document-View structure of your application.

There is only one application class per project, and, when you open a new project, it is already laid out for you (along with the task document and task window). However, to customize each application, you lay out additional documents and windows (see Figure 4.2), and you link them into the initial application class. Use the Tools palette to lay out and link classes.

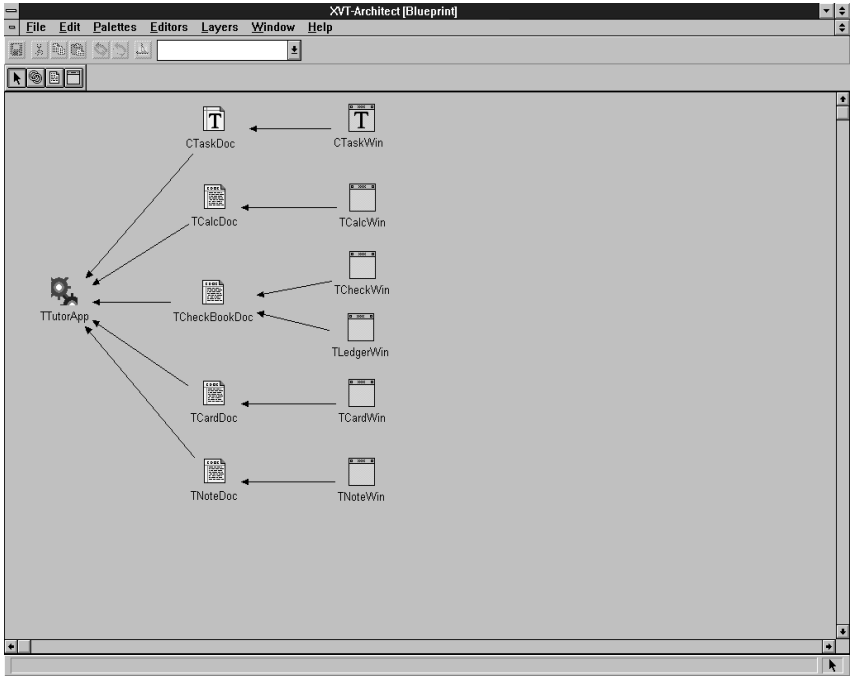


Figure 4.2. Blueprint with windows and documents laid out and linked

See Also: For a description of the Tools Palette, see section 4.3.1.1.

4.4.1. Laying Out Documents and Windows

To customize your application, you can lay out the documents and windows that it requires.

t To lay out a document or a window:

Press the appropriate button on the Tools palette, and then drag off of the palette; when you release, a document or window icon appears. This method allows for single object creation, and, on mouse up, the cursor returns to the previous tool.
-OR-

Click the appropriate button on the Tools palette, and then click in the sketch region of the Blueprint window for each document or window you want to create. This method allows for multiple object creation. When you have created the desired documents or windows, click on another tool.

After creating the documents and windows, you can use the Alignment palette to manipulate them.

See Also: For information on the Alignment palette, see section 4.3.1.2 on page 4-8.
For more information on naming classes, see section 4.4.1.1 on page 4-11.

4.4.1.1. Naming Classes

When you create a document or a window, the name of the class is selected and ready for editing. The names that you give the objects in the Blueprint are the class names that XVT-Architect uses when generating Shell files.

t To name the application, documents, or windows:

Using the pointer tool, select the title, and type in the desired name. To commit the name, click in the sketch region of the Blueprint window.

When you generate the Shell files, these names are used as class names in the files. These names must be valid identifiers. A valid identifier is a *unique* string that begins with a letter. The string can be any combination of letters, numbers, and underscores, but it *cannot* contain spaces.

Note: On MS-Windows and OS/2, any class names that are over eight letters will be truncated during Shell-file generation.

See Also: For more information on Shell file generation, see section 2.4 on page 2-8.

4.4.1.2. Factory Names

In addition to naming classes, you can indicate a Factory ID Name for the “objects” in the Blueprint, using the Factory Settings Face in the Strata. For the Factory name, you must indicate a name that is different from the class name in the Blueprint.

The Factory ID Name that you indicate in the Factory Settings Face is the name that XVT-Architect uses in the generated Factory files, and the name you will use when interacting with the Factory. In addition, you can indicate other Factory information in the Strata.

Note: If after you generate the Shell files, you change the Factory name of any object, you must search and replace the old name in your application code.

See Also: For more information on setting Factory data for an object, see section 6.7 on page 6-10.

4.5. Linking Applications, Documents, and Views

In the Blueprint module, you also need to link into the application any documents and windows that you created. Linking these classes determines, and helps you visualize, the object hierarchy. That is, the linked path defines the communication paths and the delegation of tasks to be performed.

When linking the objects in the Blueprint, you must follow these rules:

- Link documents to applications; link from the document to the application
- Applications can have multiple documents linked to them
- Link windows to documents; link from the window to the document
- Documents can have multiple windows linked to them
- Each window can be linked to only one document

Note: If you do *not* follow these rules when linking, XVT-Architect does not allow you to make the connection.

See Also: For more information on the dynamics of the object hierarchy, see the "Application Framework" chapter in the *Guide to XVT Development Solution for C++*.

4.5.1. Editing Links

You can use all of the Blueprint's editing tools to edit links. However, when you cut, copy, or paste a link, you must select the link, and you *must* select both its source and destination objects. If you do not do this, the link does not appear when you paste.

In addition, you can move a link between a window and a document to a different document. To do so, drag the arrow end of the link to another document.

4.5.2. Linking Documents to the Application

Linking documents to the application establishes the documents as part of the application.

t To link a document to the application:

Click on the linking icon in the Tools palette, and then drag a connection from the document to the application.

You can create as many links as you need. When you are finished, click the pointer button on the Tools palette.

See Also: For information on the relationship between the application and the documents, see the "Applications" and the "Documents" chapters in the *Guide to XVT Development Solution for C++*.

4.5.3. Linking Windows to a Document

Windows are at the third-level of the object hierarchy. At the second-level of the object hierarchy are the documents. Documents make the connection between the application-level and the view-level of your project. Documents create and manage windows, and store and provides access to the data that is displayed in the windows (and in the views and subviews that the windows contain).

You can have several different windows associated with one document, and these windows can display the same set of data in different formats.

t To link a window to a document:

Click on the linking icon in the Tools palette, and then drag a connection from the window to the document.

You can create as many links as you need. When you are finished, click the pointer button on the Tools palette.

See Also: For more information on the relationship between documents and windows, see the "Documents" and the "Windows" chapters in the *Guide to XVT Development Solution for C++*.

4.6. Navigating Between Modules

When you have laid out, named, and linked your application, you will want to open other modules of XVT-Architect. You can navigate between the modules at any time in your design process.

4.6.1. Getting to and from the Drafting Board

You can go to the Drafting Board module to lay out and manipulate the interface of each window that you created in the Blueprint module.

- To go to the Drafting Board module:

  Select a window with the pointer tool, and click the Drafting Board icon on the toolbar.

  If the Drafting Board for a specific window is open, you can also use the Child Window list button on the toolbar to navigate to that Drafting Board. The list button contains a list of all the open child windows for the project, including Drafting Board, Strata, and global Menu Editor windows.

- To return to the Blueprint module:

  Click on the Back to Parent Window button on the Drafting Board toolbar.

See Also: For more information on the Drafting Board, see Chapter 5, *Drafting Board*.

4.6.2. Getting to and from the Strata

In the Strata module, you can view and modify the attributes of any objects (instances of classes) that are laid out in the Blueprint. Thus, moving from the Blueprint, you can view and set the attributes of the application, documents, and windows of your application. You can set the attributes of an object at each level of the class hierarchy. For example, to set the attributes that determine the window type, you can go to the Strata for a window.

- To go to the Strata:

  Double click on the object for which you want to view and set the attributes.

- To return to the Blueprint module:

  Click OK (or close the window).
  -OR-
  Click the Back to Parent Window button on the Strata toolbar.

  If you leave the object's Strata open, you can also use the Child Window list button on the Blueprint's toolbar to navigate to the Strata for a specific object.

See Also: For more information on the Strata, see Chapter 6, *Strata*.

5

DRAFTING BOARD

This chapter describes the interface and usage of the Drafting Board module—a GUI builder with the facilities necessary to lay out advanced user interfaces. In this module, you can lay out and manipulate the XVT-Power++ visual objects of your application.

5.1. Drafting Board Interface

For each window, you go to the Drafting Board to lay out the window's user interface.

t To go to the Drafting Board:

Select a window in the Blueprint, and click the Drafting Board button on the toolbar.

The Drafting Board module has a standard interface including a menubar, a toolbar, and a status bar. In addition, the Drafting Board contains a main View palette, which has several subpalettes (see Figure 5.1).

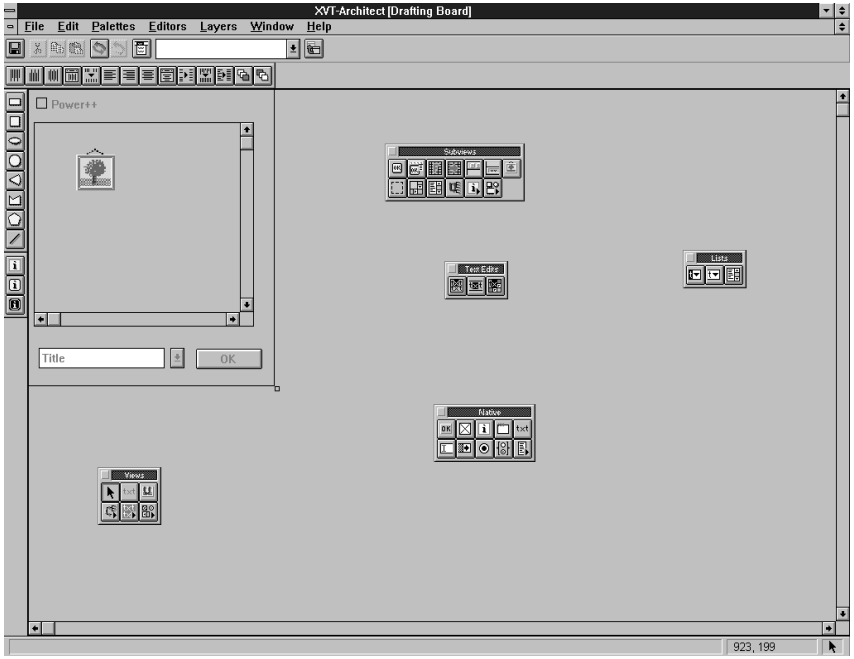


Figure 5.1. Drafting Board

5.1.1. General Overview

In its menubar, toolbar, and Alignment palette, the Drafting Board supplies you with a robust set of tools to manipulate the objects that you lay out using the View palette. The tools allow you to do the following tasks:

- Undo and Redo actions
- Cut, Copy, and Paste
- Align and center objects
- Distribute objects evenly, vertically and/or horizontally
- Size objects
- Change the stacking order of objects

After you have created a CView object using the View palette, you can use these tools on the menubar, toolbar, and Alignment palette to manipulate the objects.

Most of these tools behave as you would expect. You simply use the pointer tool (located on the View palette) to select the object or objects, and then select the action that you would like to perform.

However, you should be aware of the special features of Undo and Redo. All instances of Undo and Redo in XVT-Architect provide the ability for an unlimited number of Undo or Redo actions. The following are the only times that Undo and Redo are disabled:

- When you open a window for the first time
- When you have undone or redone all of the operations

5.1.2. Menubar

The menubar of the Drafting Board module includes the following menus:

- File
- Edit
- Palettes
- Editors
- Layers
- Help

The File and Edit menus contain standard items. However, the File menu also contains the Generate, Import, and Export items.

Select the Generate item from the File menu to generate Shell and Factory files.

To import projects into XVT-Architect, or to export XVT-Architect projects to a human-readable format, use the Import and Export items on the File menu.

The Palettes menu contains the main View palette and its subpalettes and the Alignment palette. (These palettes are described below.)

Using the Editors menu, you can open XVT-Architect's global editors.

Use the Layers menu to open the Layers Editor, which you can use to create and view layers, and to change the parent layer of layers in your project. In addition, you can use the Layers menu to view layered objects and revert objects to their parent-defined state.

See Also: For information on generating files, see section 2.4 on page 2-8. For information on importing and exporting files, see Chapter 11,

Importing and Exporting Strings.
For more information on layering, see Chapter 9, *Object Layering*.

5.1.3. View Palettes

Along with the pointer tool, the View palette and its subpalettes contain the tools that you use to create and lay out XVT-Power++ CView objects (see Figure 5.3).

When you open a Drafting Board, the View palette is open. The View palette has subpalettes, which are all tear-off palettes. You will use the View palette and its subpalettes for much of the work you do in the Drafting Board. You can close it by clicking its close box, and reopen it by selecting View from the Palettes menu. You can open all of the subpalettes either from the main View palette (and its subpalettes) or from the Palettes menu.

Like all palettes in XVT-Architect, the View palette and its torn-off subpalettes are attachable and detachable. To customize your workspace, you can position palettes in the window, or you can attach them to the side, top, or bottom of the window.

By dragging it to the edge of the window, you can attach a palette to that edge. To detach the palette, drag the attached palette away from the edge of the window.

See Also: For more information on the main View palette and its subpalettes, see sections 5.2 and 5.3.

5.1.4. Alignment Palette

With the Alignment palette, you can align, space, and size objects, and change their stacking order.

t To open the Alignment palette:

Select Alignment from the Palettes menu.

Use this palette to manipulate selected objects in the following ways:

- Align multiple objects along a vertical line by specifying left, center, or right alignment
- Align multiple objects along a horizontal line by specifying top, middle, or bottom alignment
- Align objects within the page by specifying horizontal center or vertical center of the page

- Space objects evenly by specifying vertical or horizontal spacing
- Size objects by specifying same width or same height
- Bring an object to the front
- Send an object to the back

t To select multiple objects:

Press Shift, and click on the objects.
-*OR*-
Using the Pointer tool, drag out a section rectangle around the objects.

5.1.5. Toolbar

The toolbar contains a series of menubar short cuts, including the following:

- Save
- Cut
- Copy
- Paste
- Undo
- Redo
- Menu Editor

Using the Menu Editor, you can design and lay out the menubar for each window in your application.

To the right of the Menu Editor button, there is a Child Window list button and a Back to Parent Window button, which assist you in navigating child and parent windows (described below).

See Also: For more information on the Menu Editor, see section 7.1 on page 7-1.
For more information on Undo and Redo, and on Cut, Copy, and Paste, see section 5.1.1 on page 5-2.

5.1.5.1. Navigating to Child and Parent Windows

On the toolbar, there is a Child Window list button that contains a list of the open "child" windows for the specific Drafting Board. Drafting Boards can have both Menu Editor and Strata windows as child windows.

To the right of the Child Window list button, there is a Back to Parent Window button. If you click this button, you return to the Blueprint window for that Drafting Board.

See Also: For more information, see the "Navigating Between Modules" section below.

5.1.6. Status Bar

On the left side of the status bar, there is a short "help" statement. As you move the cursor around in the Drafting Board, short help statements appear in this status bar field. The next field contains the coordinates of the cursor relative to the window you are laying out. To the right of the coordinates field is the icon of the selected tool.

5.2. Understanding the View Palette

There are several subpalettes contained in the main View palette, but once you understand the organization of the palettes, you should find them easy to use.

Along with the pointer tool, the main View palette contains the tools to create XVT-Power++ visual objects, or CView objects. There are the following five tools on the main View palette: pointer, CText, CSubview, CNativeTextEdit, CNativeView, and CUserView. Use the pointer tool to select, deselect, move, and size view objects. Use the other tools on the palette for laying out the CView objects.

The organization of the View palettes follows closely the XVT-Power++ CView class hierarchy (see Figure 5.2).

The main View palette's buttons represent an object or a group of related objects. The buttons that represent a group have subpalettes that contain the related objects. Several of the buttons on the subpalettes in turn have their own subpalettes. The buttons that represent a group of objects, or a subpalette, have a small arrow in the lower-right corner (see Figure 5.3).

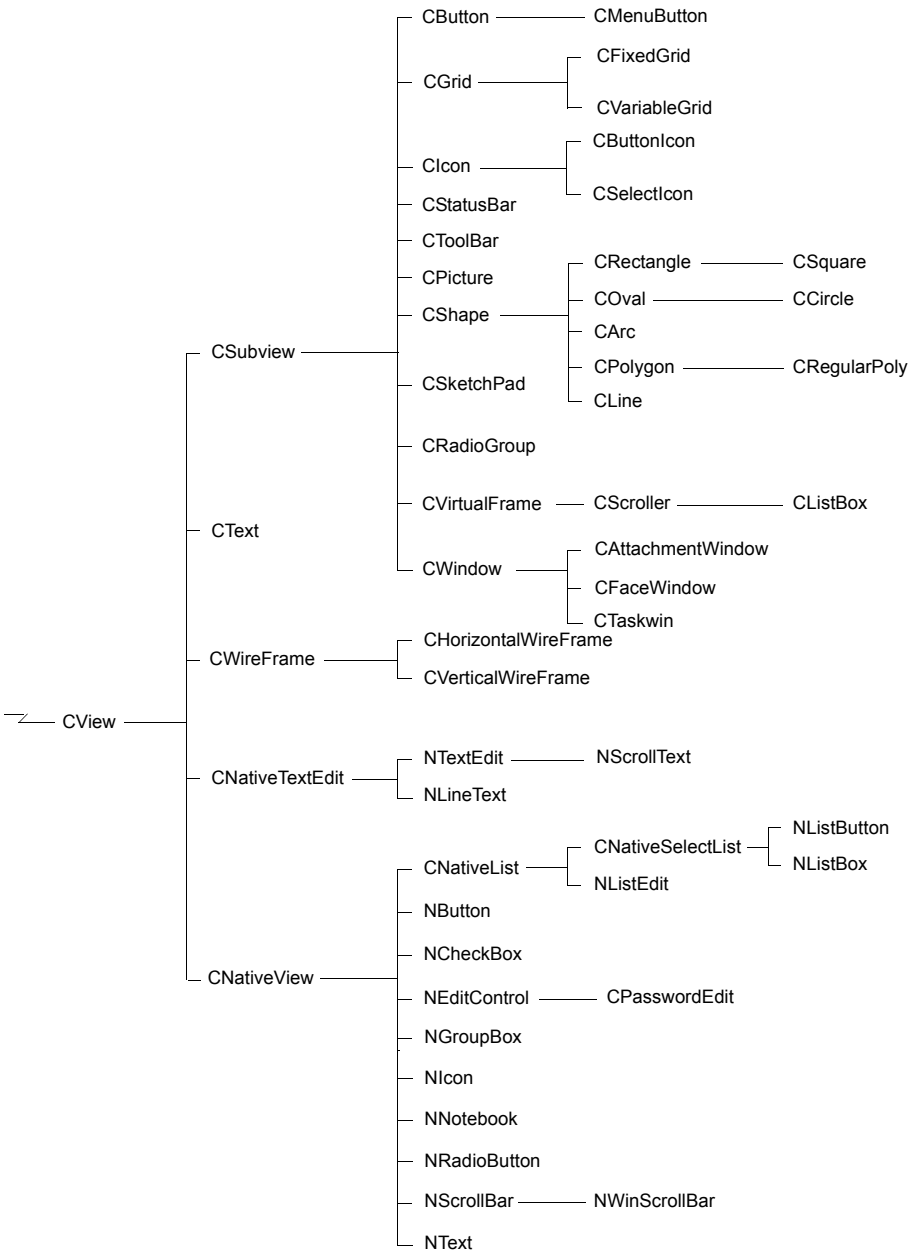


Figure 5.2. XVT-Power++ CView hierarchy

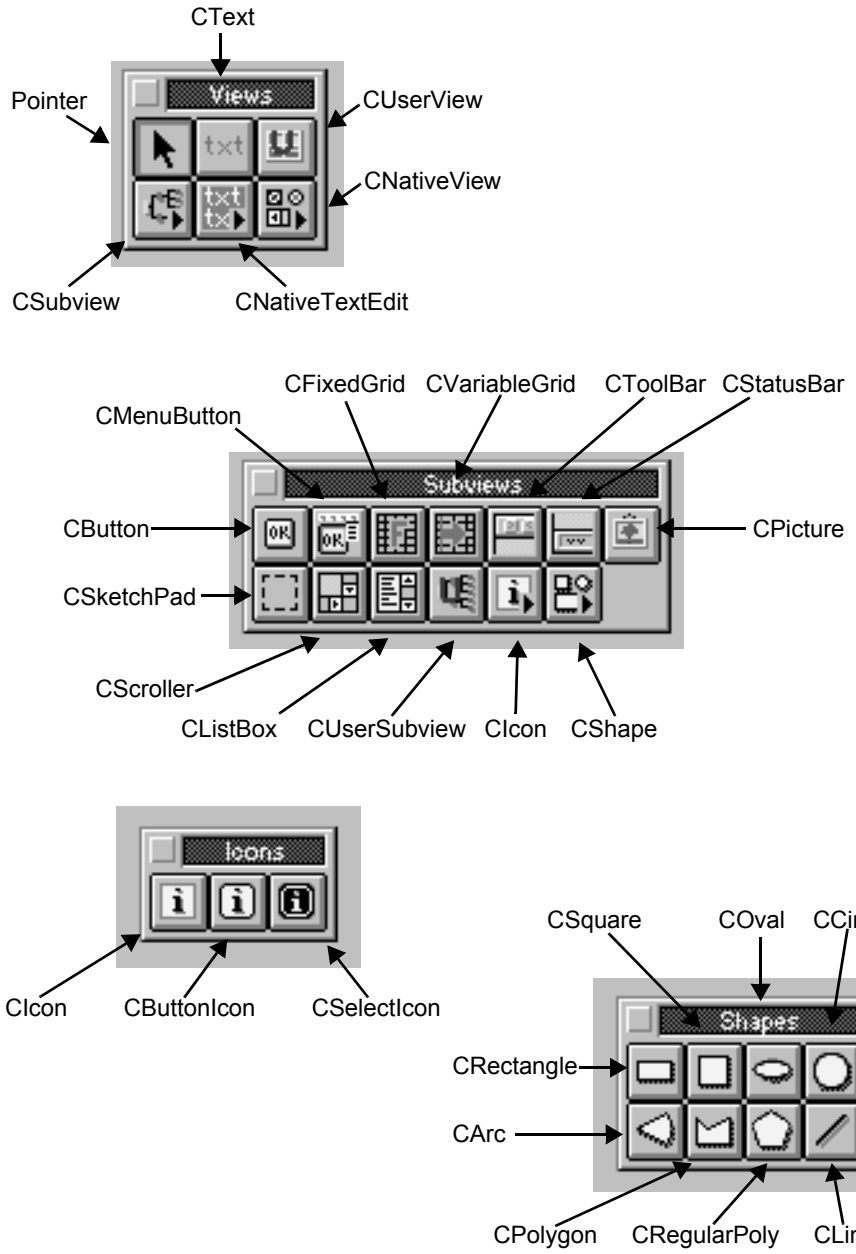


Figure 5.3. View palette and subpalettes (continued on next page)

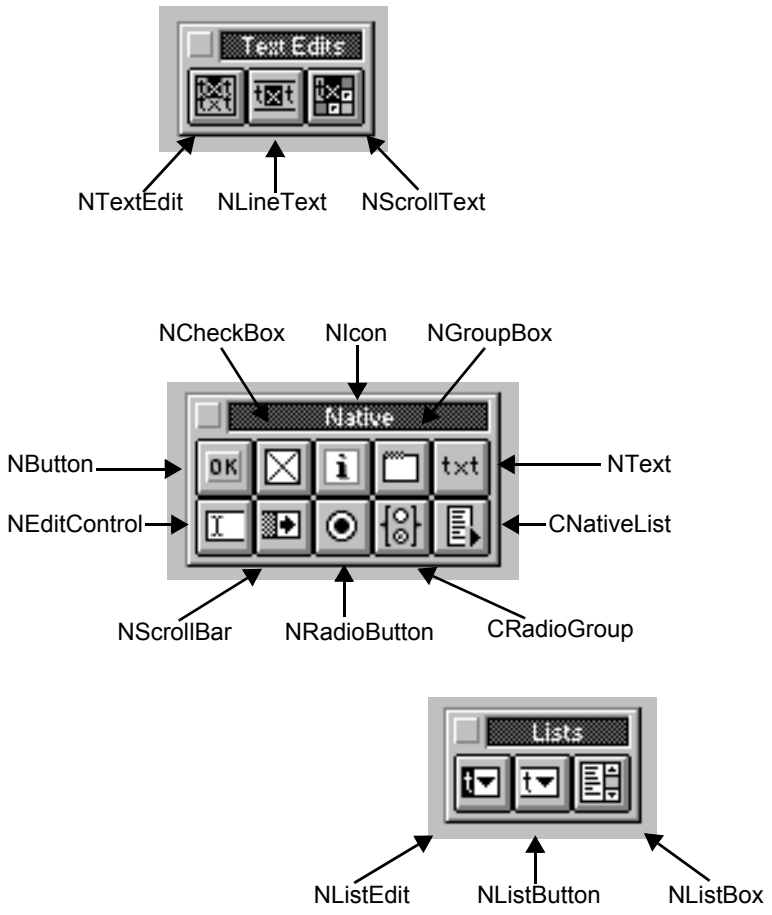


Figure 5.3. View subpalettes (continued from previous page)

The above figures illustrate the organization of the View palettes. You can open the palettes from the main View palette and its subpalettes, or from the Palettes menu.

For usability, there are classes on the View palette that have been positioned at a "level" that differs from the class hierarchy. For example, instead of appearing on their own palette, CScroller and CListBox are on CSubview's subpalette.

Note: There are several CView classes that you cannot lay out in the Drafting Board, so they are not represented on the View palette. For example, you cannot lay out a CVirtualFrame or a CNativeSelectList, so these classes do not appear on the View palette.

See Also: For a description of each CView-derived class, see the online *XVT-Power++ Reference*.

5.3. Using the View Palette to Lay Out Objects

Using the View palette, you can create and lay out objects using the following creation methods:

Drag-and-drop method
Press down a button on a palette, and drag the cursor to the sketch region of the window and release the mouse button. As you drag the object, the enclosure of the new view will highlight. The enclosure for the new view is the deepest subview that encloses the created object. When you release the button, an object of default size is created at the cursor location. When you release the button, an object of default size is created at the cursor location, and the cursor reverts to the previous tool.

Sketch method
Click a button on a palette, and click in the sketch region of the window. The enclosure for the new view is the deepest subview that contains the view. You can create multiple objects of default size by clicking multiple times. When you are done, click on another object or on the pointer tool of the palette.

Sketch method
Click a button on a palette and drag out an area in the sketch region of the window. The enclosure for the new view is the deepest subview that contains the sketched region. With this method, you can create multiple objects of any valid size. When you are done, you can click on another object or on the pointer tool of the View palette.

Once you have laid out objects, you can drag and size them, and you can manipulate them with the editing tools supplied in the Drafting Board module.

Note: To lay out a radio button, you must first lay out a radio group; a radio button must be enclosed by a radio group.

5.3.1. Dragging and Sizing the Objects

All CView objects are draggable and sizeable. Thus, when you select an object using the pointer tool, a wire frame with sizing handles appears. You can use the handles to size the object. You can also move the object within its enclosure simply by dragging it, or you can drag it out of its enclosure.

t To drag an object from one enclosure to another enclosure:

Press Control and drag the object.

As you drag the object, the current enclosure of the object is highlighted.

5.4. Sizing the Window

In addition to sizing the view objects of your application, you can size any window object. To do so, you must first locate the sizing frame of the window by scrolling the Drafting Board's sketch region. If, in the Strata, you indicated that the window should have a scrollbar, it will be represented by lines inside the sizing frame.

t To size the window:

"Grab" the frame and drag.

To adjust the height of the window, grab the frame on the bottom. To adjust the width, grab the frame on the right side. To adjust the height and width, grab the handle on the corner of the frame.

5.5. Navigating Between Modules

For each object, or instance of a class, that you lay out in XVT-Architect, there is a Strata. The Strata is an attribute editor. If you want to view and modify the attributes of an object, go to its Strata.

t To open the Strata of an object:

Double click on the object.

t To return to the Drafting Board:

Click OK, which closes the Strata window.
-*OR*-
Click Cancel.
-*OR*-
Click the Back to Parent Window button on the Strata toolbar.

If you leave an object's Strata open, you can also use the Child Window list button on the toolbar of the Drafting Board to navigate to the Strata for a specific object. This list button contains a list of the Drafting Board's open child windows. The Drafting Board can have either Strata windows or a Menu Editor window as child windows.

t To return to the Blueprint:

Press the Back to Parent Window button on the Drafting Board toolbar.

See Also: For more information on the Strata, see Chapter 6, *Strata*.

6

STRATA

This chapter describes the interface and usage of the Strata module of XVT-Architect—an object-attribute editor that provides quick access for viewing and modifying the attributes of both application and interface objects attributes. Every object in your application has a corresponding Strata.

6.1. Strata Interface

t To open the Strata for an object:

Double click on the object.

Double clicking on an object, either in the Blueprint or the Drafting Board, opens the Strata for the object.

The Strata window contains a menubar, a toolbar, a status bar, but its main interface elements are a class browser and a notebook control (see Figure 6.1). The class browser, along the bottom of the window, contains a class hierarchy for the object.

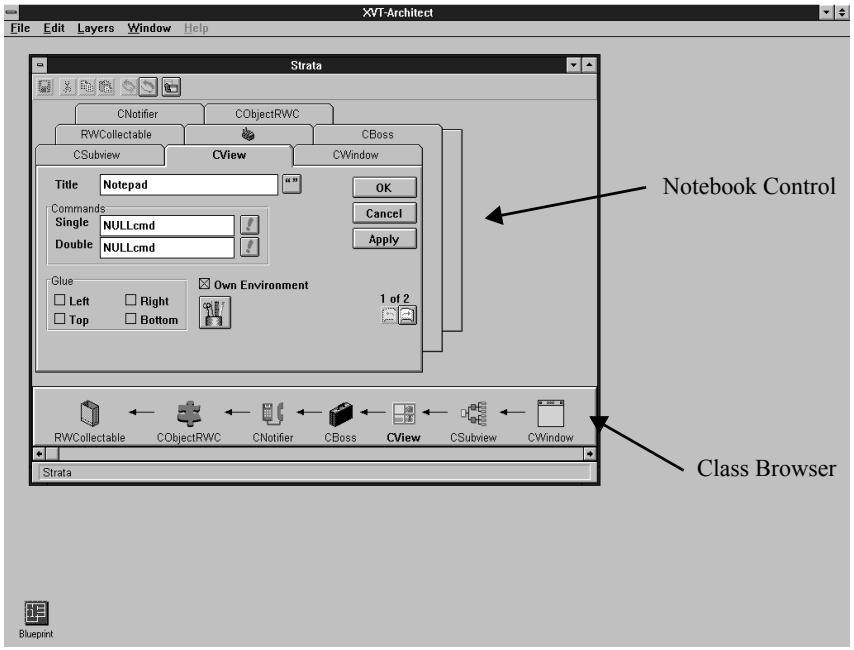


Figure 6.1. Strata Interface

The Strata's notebook control partitions an object based on its class hierarchy. Each named tab in the notebook control contains a page of the attributes declared at that inheritance level (i.e., the data members of that class).

See Also: For information on the class browser and the notebook control, see their individual sections below.

6.1.1. Closing the Strata

- To close the Strata without committing your changes:

  Click Cancel.

- To commit all of the changes that you have made in the Strata, and close the window:

  Click OK.

- To commit the changes without closing the Strata window:

  Click Apply.

When you click Cancel or OK, the Strata window closes, and you return to the window from which you opened the Strata.

While the Strata is open, it is listed in the Child Window list button on the toolbar of the window from which it was opened. In this case, you can bring that object's Strata to the front either by double clicking on the object, or by selecting its Strata in the Child Window list button.

6.2. Class Browser

The class browser (the icons along the bottom of the screen) illustrates the class hierarchy of the object. You can scroll across to see a full class hierarchy of the object (to get an idea of the classes from which an object is inheriting functionality). If you want to see which attributes are declared for a specific class, click on its icon. This action brings that class page in the notebook control to the front.

See Also: For a picture of the XVT-Power++ class hierarchy and inherited functionality, see the "XVT-Power++ 5 Hierarchy" poster, and see the *XVT-Power++ Reference*.

6.3. Notebook Control

Using the Strata, you can learn and fully utilize each XVT-Power++ class's inherited attributes.There is a Strata for each object of your application. In the Strata, each class from which that object inherits (every base class) is represented by a tabbed "page" in the notebook control, and each page contains a control for each of the attributes that you can view and modify for that class.

For example, the CWindow page has a group of decoration attributes. These decorations are a part of the itsAttributes data member of CWindow. Similarly, since itsGlue is one of CView's data members, the first CView page has controls defining an object's glue properties.

When the you first open the Strata for an object, the attribute fields all contain default values. Certain values, such as itsEnvironment, may be inherited from the object's enclosure. However, in the Strata, you can attach an environment to any object in your application.

See Also: For more information on setting an object's environment, see section 6.4.1.

6.3.1. Using the Notebook Control

Using the notebook control, you can view and set attributes of an object at each inheritance level of the class hierarchy. The data members, or attributes, represented on each page of the notebook control are the initial data of the specific object instance.

Once you are in the Strata, you can "page through" the notebook control to view and modify the attributes at each inheritance level.

t To bring a page of the notebook control to the front:

Click on the tab of the specific class (tabs are in alphabetical order).
-OR-
Click on the specific class icon in the class browser at the bottom of the window (icons are ordered by class hierarchy).

When you first open the Strata for an object, the attribute fields are populated with standard default values. You can change any value by interacting with the controls that populate the page. You simply tab to navigate through the controls on a page. Note that if there are "page turning" buttons at the bottom-right corner of a page, there are other pages containing attribute fields for that class.

The values that you specify can either be XVT-Power++-defined values, or they can be values that you have defined for your project. When you make a change and click OK or Apply, the object is updated so that you can see the results in the Drafting Board.

Note: For all icon objects, there is a resource ID field on the page, which contains the default ID for NULLicon, but you can type in any ID as long as you define it in the **<ProjectName>.url** file.

In addition, you can reference **.bmp** files from the CPicture page, using a directory path.

See Also: For more information on the attributes of a particular class, see its entry in the *XVT-Power++ Reference*, and see XVT-Architect's online documentation for information on the following classes: CToolBar, CStatusBar, CButton, CMenuButton, CDrawingContext, CStackable, CUserView, and CUserSubview. For more information on specifying resource IDs for icons, see the *Guide to XVT Development Solution for C*, and see your XVT platform-specific book.

6.4. CView Pages

CView supplies a lot of functionality to its derived classes. It is important to acquaint yourself with the pages of CView, and the attributes that you can set at this level of the class hierarchy (to see CView's first page, see Figure 6.1). The attribute fields on the CView page are:

Title
Sets the title of the object. For any object that has a title, you set the title on the CView page.

Commands
Sets single-click and double-click commands. From these fields, you can also get to the Command Editor.

Glue
Allows objects to have stickiness properties and helps in geometry management. When a view's enclosure is sized, the view's glue maintains a constant distance between the glued object and its enclosure's borders. For example, if you set RIGHTSTICKY and BOTTOMSTICKY, the object will remain the same distance from its enclosure's right and left borders.

Environment
Determines if the object inherits an environment or has its own. If you check the Own Environment box, the object will have its own environment. You can press the Environment button and use the Environment Attributes dialog to set the environment attributes, which is described below (see Figure 6.2).

Decorations
Sets draggable, sizable, visible, enabled, and active attributes.

Frame
Sets the location and size of the object relative to its enclosure.

See Also: For more information on CView's data members, see CView in the *XVT-Power++ Reference*.

6.4.1. Environment Attributes dialog

Using the Environment Attributes dialog, you can set the environment for the objects of your application. An environment of an object contains information regarding text attributes, drawing modes, brush and pen attributes, and colors. It is encapsulated by an instance of the CEnvironment class.

By default, there is an environment object that is shared by every visible object in an XVT-Power++ application. Because an environment propagates down the object hierarchy, many objects can share the same environment.

However, you can also attach a separate environment object at any level of the object hierarchy, anywhere from the application object to any view object in the application. That is, objects can share environments, or they can have their own environment.

Using the Strata, you can attach an environment object to any object in your application. Every CApplication, CDocument, and CView page has an environment field. An environment field consists of an "Own Environment" check box and an Environment button.

t To attach an environment object to a view object:

Check the Own Environment box.

When you check the box, the Environment button is enabled.

t To open the Environment Attributes dialog so that you can view and modify environment attributes:

Click the Environment button.

Then, using the Environment Attributes dialog, you can set the following environment information (see Figure 6.2):

- Text attributes, which include font family, style, and size and background and foreground colors
- Brush attributes, which indicate the color and pattern used to fill many closed shapes
- Drawing mode
- Pen attributes, which indicate the color, pattern, width, and style

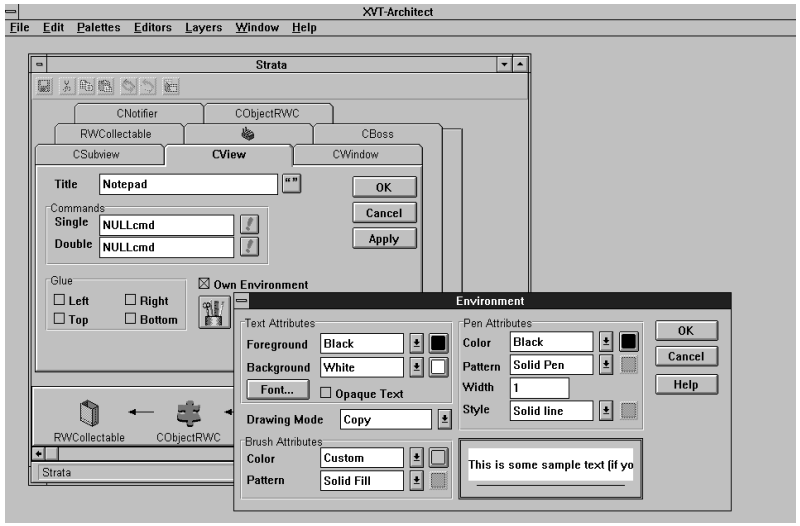


Figure 6.2. Strata, Environment Attributes dialog

See Also: For more information on environment settings, see the "Application Framework" chapter in the *Guide to XVT Development Solution for C++*, and see CEnvironment in the *XVT-Power++ Reference*.

6.4.1.1. Using the Environment Attributes dialog

The Environment Attributes dialog is a window containing edit fields and list buttons that represent the environment variables. The color list buttons have color buttons on the right, and clicking one of the color buttons brings up a color palette. Using these controls, you can modify the environment information.

t To commit the changes that you have made in the Environment Attributes dialog:

Click OK, which closes the editor and returns you to the Strata.

6.5. CWindow Pages

You can use the CWindow pages for setting many window attributes. Use the CWindow pages to set the Border Type, Size, Modality, Decorations, and Initial Conditions of the window. In addition, use the Menus list box to assign an existing menubar to the window. Finally, use the fields on CWindow's third page to specify a screen height and width.

See Also: For more information on assigning menubars to windows, see section 7.1.3. For information on CWindow's attributes, see its entry in the online *XVT-Power++ Reference*.

6.5.1. Sizing and Placing Windows

When you first create a window in the Blueprint, it is given an initial default position and size, or frame rectangle. For top-level windows, the frame indicates the initial location and size of the window relative to the screen or task window, and relative to other top-level windows.

XVT-Architect allows you to change a top-level frame rectangle in the following three places:

- In the window's Drafting Board, you can modify the size of a window by adjusting the window's sizing frame (see section 5.4)
- In the window's Strata, you can adjust both the position and size by entering coordinates on CView's second page
- In the window's Strata, you can specify screen size and relative placement by using the editor on CWindow's third page

On CWindow's third page there is a rectangular region called the Window Placement Region, which represents the size of the task window or screen. You can modify these dimensions using the controls to the right of the Window Placement Region (see Figure 6.3).

Within the Window Placement Region are rectangles representing the relative size and position of all top-level windows that you have created the Blueprint module. The current window (the window for which you have the Strata open) is shown as a solid rectangle. You can move and size this rectangle.

Other top-level windows are visible within this region as dotted-line rectangles, and cannot be moved. Since all top-level windows are

given the same initial default frame, all rectangles within the Window Placement Region are likely on top of one another initially.

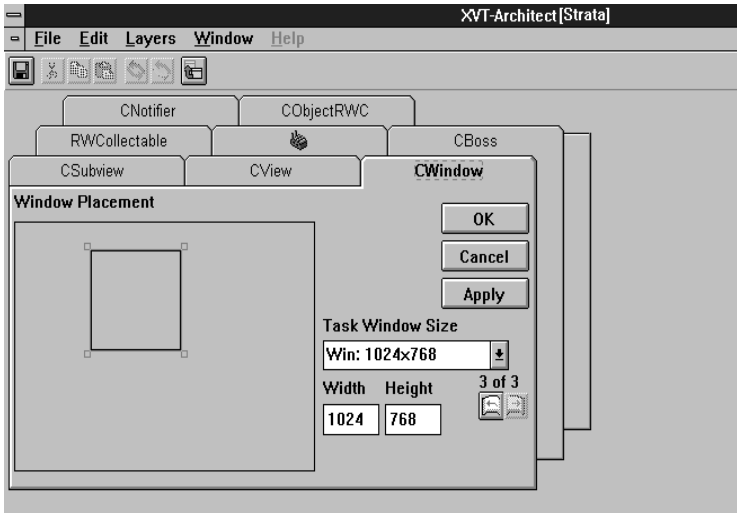


Figure 6.3. CWindow's third page, window placement

6.5.2. Creating a Modal Window

The XVT-Power++ CWindow class supports the creation of modal windows.

t To create a modal window in DSC++:

1. Create the window using XVT-Architect.
2. Using the CWindow Strata tab, set the window's type to "Modal".
3. In your code, invoke the CWindow::DoModal() method for the modal window defined in XVT-Architect.

The following code fragment shows how a modal window defined in XVT-Architect is created and made modal:

```
CMyWin *aWindow = MYFactory.CreateWindow(aDoc, MYWin);
    aWindow->DoModal();
```

The call to DoModal does not return until the window is closed.

See Also: Refer to a sample application in **...samples/arch/about** for more information about creating modal windows in DSC++.

6.6. CUserView and CUserSubview Strata Pages

The CUserView and CUserSubview objects have Strata faces where you can register information for your customized classes. On these Strata faces, you must specify the name of the class, which must be a valid identifier.

You can also provide #include file names in the Text Edit field provided, but this is optional. Note that when typing in your #include file names, you should type what would normally follow the #include directive. That is, you must put quotes or brackets around those file names that require them.

6.7. Factory Settings Page

In addition to setting the attributes of an object, you can also set its Factory information in the Strata. Use the Factory Settings page, indicated by the "Factory" bitmap on the tab, to specify the information that XVT-Architect uses when generating Factory files. The Factory object information includes the following:

- Name
- ID (read only)
- Base (read only)
- Comments
- Whether it is automatically created at application startup
- Whether it is included in its enclosure's data member class

6.7.1. Using the Factory Settings Page

t To set Factory information:

Bring the Factory page to the front, and enter the information (see Figure 6.4).

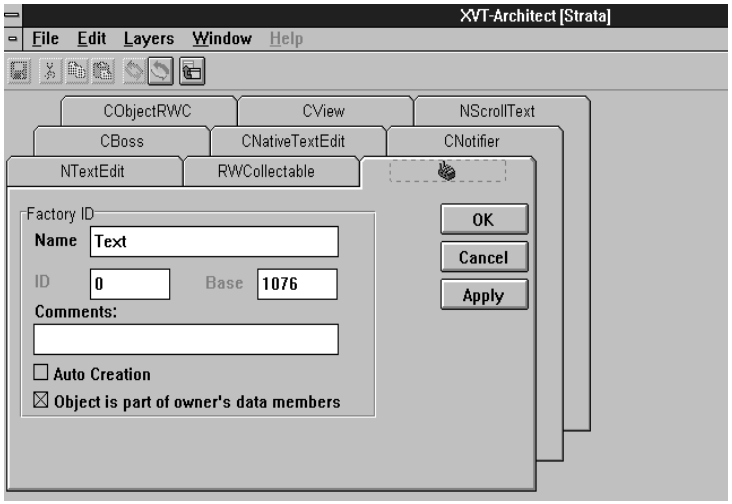


Figure 6.4. Strata, Factory Settings page

Name

The string identifier that you will use when accessing and creating objects using the Factory interface. Object Names are part of the #defines in the generated Factory files.

When you write your application code, you must access some of the objects that you lay out in XVT-Architect. In general, you must access those objects whose attributes you will be getting or setting dynamically at runtime. To do this, you will use the Factory ID Name.

Generally, it is a good idea to change the Factory ID Names of objects that you know that you will be accessing in your code; you should change the name to something that makes sense in your application. Of course, you can change the Factory names of all the objects in your application, but this is not really necessary.

ID and Base

Read only values that are generated by XVT-Architect. These variables are also included as part of the #defines in the generated Factory files. However, you will only need to use the Name when interacting with the Factory.

Comment

Field in which you can make any comments that you would like to be in the generated Factory files.

Auto Creation

Determines if a window is visible on application startup. By default, this box is checked.

Object is part of owner's data members

Determines if a specific object is in the generated Factory data member class of its enclosure. By default, this box is checked. In this case, XVT-Architect generates a data member class with a pointer to each object that you have indicated. Therefore, when you call PAFactory::DoCreate* on nested objects, you will be returned a pointer to these indicated objects.

XVT-Architect names the returned pointers by taking the object's Factory name and adding the prefix "its". For example, if you give an object the Factory name of "Text", the returned pointer will be "itsText".

Note: If after you generate the Shell files, you change the Factory name of any object, you must search for the old name in your application code, and replace it with the new name.

See Also: For more information on the generated Factory files and data member classes, see Chapter 8, *Object Factory*.

6.7.2. Using XVT-Architect's Editors

XVT-Architect supplies many editors to refine your application and manage global data. You can open the global editors from Editors menu (in the Blueprint and Drafting Board). In addition, you can open these editors from the Strata. When you open them from the Strata, they are primarily local editors, but, in this case, some editors can also function as global editors.

From the pages of the CView, CButton, NCheckBox, NEditControl, and NListEdit classes, you can set commands. In addition, from the pages for these classes you can access the Command Editor. Use the Command Editor to define command variables, such as a command base, base value, command name, command value.

From the CView page, you can also open the String Editor. Each object has a "Title" string associated with it, and you can use the String Editor to apply an existing string to an object, modify strings and string information, and indicate strings that you want to be generated for later use.

From the pages of the CListBox and CNativeList classes, you can open the String List Editor. The String List Editor allows you to set the string lists that appear in the list controls of your application.

7

EDITORS

XVT-Architect has several editors that you can use to refine your application and manage your application's global information. The following is a list of XVT-Architect's editors, which are described in this chapter:

- Menu Editor
- Accelerator Editor
- Command Editor
- String Editor
- String List Editor

When opened from the Blueprint or Drafting Board, the Menu Editor, Accelerator, Command, String, and String List Editors can be used as global editors. Each of these editors give you the ability to view, modify, and manage your application's global information.

See Also: For information on the Layer Editor, see Chapter 9.

7.1. Menu Editor

In XVT-Architect's Menu Editor, you can design and lay out menubars. In XVT-Power++, a menubar, or an instance of the CMenuBar class, is a collection of submenus. A submenu is a collection of menu items and other submenus.

t To open the Menu Editor:

Click the Menu Editor button on the Drafting Board toolbar.
-*OR*-
Select Menu Editor from the File menu in the Blueprint or Drafting Board.

When you open the Menu Editor from the Drafting Board, the menubar that you create is associated with that window. If you commit the changes in the Menu Editor, XVT-Architect generates the menus in the Factory files.

When you open the Menu Editor from the Blueprint or Drafting Board, you can edit the existing menubars in your application.

In the Menu Editor, you can do the following:

- Create a menubar that contains standard menus
- Create new menubars, by modifying and creating menu items and submenus
- Arrange and rearrange menu items and submenus
- Access the Accelerator Editor to specify keyboard accelerators for menu items

In addition, once you have created a menubar, XVT-Architect allows you to associate it with other windows in your application using the Strata.

Note: You do not have to associate a menubar with every window; some window types are by definition not allowed to have a menubar.

See Also: For more information on associating an existing menubar with a window, see section 7.1.3.

7.1.1. Using the Menu Editor

The Menu Editor window is divided into two areas. The top of the window is a scrollable area that you use to lay out the menubar. The bottom area of the window is an area populated with controls that you use set the menu-item data (see Figure 7.1).

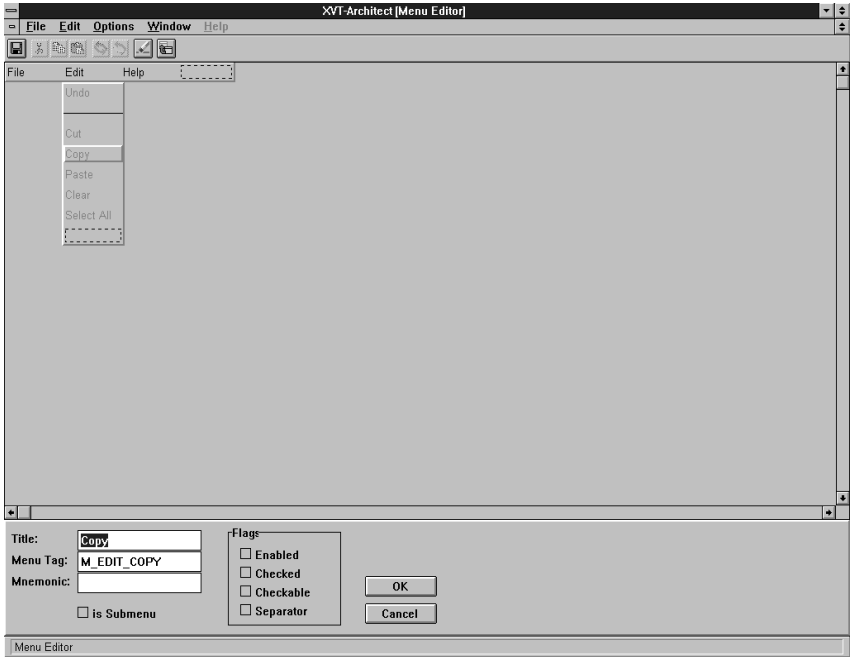


Figure 7.1. Menu Editor

t To close the Menu Editor without saving the menubar:

  Click Cancel.
  -OR-
  Close the Menu Editor window.

t To commit the changes that you have made to the submenus and menu items:

  Click OK.

  To further edit the menubar, you can reopen the Menu Editor at any time.

7.1.1.1. Using the Standard Submenus

When you open the Menu Editor, the top, scrollable area contains the standard File, Edit, Font, and Help submenus for your development platform. These are the DEFAULT_*_MENU values as defined by the XVT Portability Toolkit. You can click on each standard submenu to see its items.

The items that appear on the standard menus are platform-specific. When you port your application, the standard menus will be appropriate for the new platform. If you modify the standard submenus, they are no longer "standard"—in other words, they are no longer platform-specific.

In addition, the Menu Editor's Options menu contains the Standard File, Edit, Font, and Help submenus as checkable items. When you first open the Menu Editor, all of the standard submenus on the Options menu are checked. You can remove a standard submenu menu by unchecking it.

t To create a standard menubar:

In the Menu Editor, click OK.

This action closes the editor window and saves the Standard submenus menus. If you do not open the Menu Editor and click OK, XVT-Architect does not generate the standard menubar with that window. In that case, the standard task window menubar is generated.

t To close the Menu Editor without associating the standard submenus with a window:

Click Cancel.
-OR-
Close the Menu Editor window.

See Also: For more information on menubars, see the "Menus" chapter in the *XVT Portability Toolkit Guide*.
For more information on standard menus, see the "DEFAULT_*_MENU Values" section in the online *XVT Portability Toolkit Reference*.

7.1.1.2. Moving Menu Items

Once a menu item or submenu exists, you can move it in any direction; you can drag and drop any submenu or menu item.

t To move a menu or menu item:

Drag it to its new location.

When you drop the item, XVT-Architect places it in that position. The item in that position is moved behind the dropped item. If you drag a menu item to the top of the menubar, it becomes a submenu.

However, if you drag a top-level submenu down into the menu, it will remain a submenu, and it will retain its menu items. If you move a top-level submenu into a menu, you can uncheck the "is Submenu" state at the bottom of the window. This action deletes the menu items that the submenu contains.

Note: In the Menu Editor, dragging is always enabled; if you drag across the menubar in the Menu Editor, you will move an item.

7.1.1.3. Setting Menu-Item Data

Once you have named or renamed a submenu or menu item, you may need to specify or modify the data that is associated with the menu item. To do so, use the edit fields located at the bottom of the Menu Editor window.

Title
The name of the submenu or menu item.

Menu Tag
The unique tag that is associated with the menu item. In the Factory **defines.h** file, XVT-Architect generates the #defines for the menu tags.

For the Menu Tag, you can indicate one of the M_* Menu Tags defined by the XVT Portability Toolkit, you can use the tag that XVT-Architect generates for you, or you can define your own.

If you modify the Menu Tag, you must use a valid tag, or XVT-Architect issues an error and requests a valid tag. A valid Menu Tag is a *unique* string that begins with a letter. The string can be any combination of letters, numbers, and underscores, but it *cannot* contain spaces.

Mnemonic
The mnemonic key associated with the menu item. This setting is optional.

is Submenu
Indicates whether the item is a submenu or not. To make an item a submenu, which has menu items and other submenus but is not a menu item itself, check the "is Submenu" box. Note that

top-level items are always submenus. You can also have nested submenus (i.e., a submenu can contain other submenus).

Flags
The Flags set for the item at application startup. You can set a menu item to be Enabled (Disabled), Checked, or Checkable, or you can make the item a Separator.

You can modify any of the data that you have set, or that XVT-Architect has set for you. To modify the data, select the text in the edit fields and change information, or check and uncheck items. You can use the Tab key to move through all of these controls at the bottom of the Menu Editor window.

t To make the item a separator:

Select the item, and check Separator.

When you define an item as a separator, you do not have to fill in any menu-item data.

See Also: For more information on menubar variables, see the CMenu* and the CSubmenu descriptions in the online *XVT-Power++ Reference*. For more information on predefined menu tags, see the "M_EDIT_*, M_FILE_*, and M_HELP_* Menu Tags" section in the online *XVT Portability Toolkit Reference*.

7.1.1.4. Using the Accelerator Editor within the Menu Editor

You can set keyboard accelerators for the menu items that you have defined.

t To open the Accelerator Editor:

Choose Accelerators from the Menu Editor's Options menu.

See Also: For information on using the Accelerator Editor, see section 7.2.

7.1.1.5. Factory Name and Information

In the Factory files, XVT-Architect generates the menubar that you have laid out in the Menu Editor. The files contain the name and Factory ID of the menubar and its menu items. When you create a window stored in the Factory, you also create its associated menubar.

When you first open the Menu Editor from the Drafting Board or Strata of a window, XVT-Architect gives the menubar a Factory name. The Factory name is the window's Factory name with a "MB" suffix. In the Menu Editor, you can change the menubar's Factory

name. This is the only way that a menubar's Factory name is ever changed.

For example, if after opening the Menu Editor for a window, you change the Factory name of the window, XVT-Architect does *not* change the menubar's Factory name.

t To change the Factory name for the menubar:

1. Choose Factory Options from Menu Editor's Options menu, which opens a Factory Options dialog box.
2. Change the Menubar Name.
3. Click OK to commit the change.

7.1.2. Customizing Menus

In the Menu Editor, you can customize the menubar for each window in your application. You can either modify the existing menus, or you can delete these menus and define new menus.

Note, however, that when you modify a standard submenu in any way, it is no longer a standard, platform-specific submenu; it is the same on every platform.

7.1.2.1. Pop-up Menus

You can customize a menu to act like a pop-up menu.

t To change a menu from its normal status to a pop-up status:

1. Define a menubar with the Menu Editor. (Specify a menubar with only one menu Title.)
2. Outside of XVT-Architect, in one of your user files, write code that creates a CMenu using the menubar ID of the menu defined earlier in XVT-Architect.
3. Call the special DoPopup method.

A pop-up menu is displayed with its left edge oriented along the point where the mouse button was pressed. Depending on the way you call the parameters of the DoPopup method, the top edge of the menu can be oriented along the same point, or the menu can be displayed with a particular menu command title centered vertically over the point.

7.1.2.2. Modifying Standard Menus

If you modify a standard submenu or any of its items, XVT-Architect changes the Menu Tag so that it is no longer a DEFAULT_*_MENU value. The tool also unchecks the specific "Standard" submenu on the Menu Editor's Options menu.

However, since your code may reference existing menu tags, XVT-Architect does not automatically change the Menu Tag when you change a menu Title. You can change the Menu Tag, and any of the menu-item data located at the bottom of the window (see section 7.1.1.3).

- To change a menu item:

  Select the item, and modify the menu-item properties using the controls at the bottom of the window.

- To delete a submenu:

  Select the submenu, and click the Delete button in the toolbar.
  -OR-
  If it is a "Standard" submenu, choose the submenu from the Menu Editor's Options menu (i.e., "uncheck" the submenu).

- To delete a a submenu or a menu item:

  Select the item, and click the Delete button in the toolbar.
  -OR-
  Select the item, and choose Delete Menu from the Edit menu.

- To create a new submenu or menu item:

  Click on an empty box on the menubar, and type in the menu-item data.

XVT-Architect generates a valid Menu Tag for you, but you are free to modify it. Once you have created a new submenu or menu item, you can drag it to a new position.

Note: You can drag the empty menu field to a new location, but you must give it a Title and valid Menu Tag before you leave the field. If you try to select another field after moving an empty field, XVT-Architect issues an error and requests a valid Menu Tag.

See Also: For more information on valid Menu Tags, see section 7.1.1.3.

7.1.2.3. Translating Exported Menu Strings

The menu strings that need to be translated to the language used by a particular locale are contained in the PAUserString object files discussed in section 11.2.1 on page 11-3. These files contain a series of strings with the following syntax:

```
string Num
“String”
```

Each string enclosed in quotes needs to be translated for every target locale you are supporting (see following example).

For example, here is a portion of an **.aeo** file, generated for the “Spanish” layer of an XVT-Architect project:

```
string 258
"Nuevo"
string 261
"Abre..."
string 265
"Cierra"
string 268
"Asegura"
...
```

These four strings are found in almost all File menus, and in English, correspond to the familiar File menu commands “New,” “Open,” “Close,” and “Save.”

7.1.2.4. Five Languages Already Translated

Both at the Portability Toolkit level and at the XVT-Power++ level, XVT provides localized versions of its standard menus for U.S. English, German, French, Italian, and Japanese. These localizations are encapsulated in include files referenced by XVT URL and help source text files. You may control the inclusion of these files by defining a LANG_* constant on the command line for **curl** or **helpc**, or by defining the constant in your source files.

If you are localizing for one of these five languages, you will not need to translate the titles of standard menus, e.g., M_FILE, M_EDIT, M_FONT, and M_HELP. To ensure these localized versions are used by the application, make sure to run **curl** with the correct language definition for the locale.

Example: For example, German is one of the languages for which pre-translated resources are provided at the XVT Portability Toolkit level.

t To run the resource compiler and include German default XVT resources for an XVT/Win16 application, use a command line similar to the following:

```
curl -r rcwin -I..\..\include -DLANG_GER_W52
    -DLIBDIR=.\..\..\lib sample.url
```

Using this command line will cause the file **ugerw52.h** (which is the XVT-supplied German translation of the default resource file) to be included in your resources. As discussed in section 7.1.1.1, the standard menus contain different entries on the different platforms that XVT supports.

See Also: For a complete list of URL resource compiler options, refer to the online *XVT Portability Toolkit Reference*.
For a list of LANG_* constants supported by XVT, refer to the "Multibyte Character Sets and Localization" chapter in the *XVT Portability Toolkit Guide*.

7.1.3. Associating Existing Menubars with Windows

You can associate an existing menubar to the application, on the CApplication Strata page, and you can assign an existing menubar to each top-level window, on CWindow's second page.

These pages have list buttons of predefined-menu selections. These list buttons contain all menubars in this project that you have created from the Menu Editor. They also contain the "no menu" and "default menu" selections.

t To associate an existing menubar with an application or window:

1. Open the Strata for the window with which you want to associate the menubar.
2. On the CWindow page, select the menubar from the list of menubars, and click Apply (or OK).

Note: If you want to modify an existing menubar for a window, you should make a new menubar and associate it with the window. You can use Copy and Paste to move an existing menus to other windows, and then make the desired modifications.

7.2. Accelerator Editor

You can open XVT-Architect's Accelerator Editor from the Blueprint, Drafting Board, and Menu Editor. In the Accelerator Editor, you can set the key sequence, or keyboard shortcut, to send a DoMenuCommand for any menu items. When you generate the files for your application, XVT-Architect generates the proper accelerator statements in the Factory files.

The Accelerator Editor allows you to specify keyboard accelerators (shortcuts) for the menu items in your application. Once an accelerator is set for a particular menu tag, the accelerator applies to every instance of that menu tag (and associated menu item) in your application.

For example, setting "Control+S" as the accelerator for the standard menu tag M_FILE_SAVE ensures that pressing Control+S is equivalent to selecting Save from the File menu. Since accelerators apply to your entire application, pressing Control+S works in each of your windows that has M_FILE_SAVE in its menubar.

t To open the Accelerator Editor:

From the Blueprint and Drafting Board, choose Accelerators from the Editors menu.
-OR-
From the Menu Editor, choose Local Accelerators from the Options menu.

In the Accelerator Editor, you can do the following:

- Set and modify keyboard accelerators for menu items
- Create, modify, and delete accelerators for non-menu items, "ghost" items (only when opened, as a global editor, from the Blueprint's or Drafting Board's Editors menu)

7.2.1. Using the Accelerator Editor

The Accelerator Editor contains an "Accelerators" list box, which contains a list of Menu Tags and corresponding accelerators. The editor also contains a set of controls, which allow you to set the accelerators for these Menu Tags.

When opened from a Menu Editor, the Accelerator Editor only allows you to set accelerators for menu tags specified in that Menu Editor.

When opened from the Blueprint or Drafting Board, the Accelerator Editor allows you to set accelerators for all menu items in your application. It also allows you to specify accelerators for "ghost" menu items, which do not appear on any of the menus in your application.

In the Accelerator Editor, you can specify accelerators for all or part of the Menu Tags.

t To set an accelerator:

1. Select a Menu Tag from the Accelerators list box.
2. Check Shift, Control, and/or Alt.
3. In the Key edit field, type the accelerator key.
   -OR-
   Choose a Key from the drop-down list.

By using this method, you can also modify any of the accelerators that you have already indicated.

In addition, when you open the Accelerator Editor from the Menu Editor, you can delete accelerators.

t To delete an accelerator:

Delete the contents of the Key field.

From the Accelerator Editor, you *cannot* create or delete Menu Tags that are associated with menubars in your application. In this case, you must use the Menu Editor to create and delete menu tags (and menu items). However, if you open the Accelerator Editor from the Blueprint or Drafting Board, you can create new tags that are "ghost" items and specify accelerators for them.

7.2.1.1. Creating Accelerators for "Ghost" Menu Items

When you open the Accelerator Editor from the Blueprint or Drafting Board, you can create new menu tags and assign them accelerators. The tags are "ghost" tags, which are not associated with menubars in your application.

Creating a "ghost" menu item with an accelerator is an easy way to create a keyboard shortcut for any action in your application. Simply, create a ghost menu item for the action, set the desired accelerator key sequence, and all of your applications windows will receive a DoMenuCommand call with the ghost menu item when the user presses the appropriate key sequence.

Note: In this release of the DSC++, "ghost" menu items with accelerators will not work on the Macintosh. However, when porting, having ghost menu items does not affect your application code.

t To create a new ghost menu tag:

Click the Create New button at the bottom-right corner of the Accelerator list box.

When you create a "New Tag," it is selected in the Menu Tag field and ready to edit. You can specify any valid tag. A valid Menu Tag is a unique string that begins with a letter. The string can be any combination of letters, numbers, and underscores, but it *cannot* contain spaces. You can modify the accelerator and delete by modifying or deleting the Key indicated. In addition, you can delete the ghost menu tag and its accelerator.

t To delete a ghost menu tag and its accelerator:

Select the tag, and click the Delete button.

The Delete button is enabled only when you select a ghost menu tag that you have created in the Accelerator Editor (i.e., the menu tag is not associated with a menubar in your application). When you click the enabled Delete button, the tag and the accelerator are both deleted.

7.3. Command Editor

When end users manipulate views in an XVT-Power++ application, they cause commands to be generated, which are trapped programmatically (via the DoCommand method).

Using the Command Editor, you can assign project-specific command information. These assignments are reflected in the generated files. The Command Editor gives you a global picture of the commands in your project.

Certain classes of the class hierarchy define commands, and their subclasses inherit these commands. The following classes have command attributes, and thus have the command fields on their page in the Strata:

- CView has single and double command fields (itsCommand and itsDoubleCommand)
- CButton has in command, out command, up command, down command fields; if they are not NULLcmd, these commands

are sent when the pointer goes in or out of the button, or when the mouse button is pressed up or down

- CMenuButton has a menu tag (itsMenuTag) attribute, which, when the menu button is clicked, is sent to the window's DoMenuCommand method
- NCheckBox has select and deselect command fields (itsSelectCommand and itsDeselectCommand)
- NEditControl has key focus, key focus lost, and text command fields (itsKeyFocus, itsKeyFocusLost, and itsTextCommand)
- NListEdit has key focus, key focus lost, and text command fields (itsKeyFocusCmd, itsKeyLostCmd, and itsTextCmd)

t To open the Command Editor from one of these pages in the Strata:

Click a Command Editor button next to the Command field.

In addition, you can open the Command Editor, as a global editor, from the Blueprint or Drafting Board.

t To open the Command Editor as a global editor:

Choose Commands from the Editors menu.

In XVT-Architect's Command Editor, you can view and specify any of the following attributes for commands and command bases (see Figure 7.2):

- Name
- Value
- Comments

The Command Editor allows you to organize commands by specifying a command base for each command. The numeric value for a command is the sum of its value and its command base value.

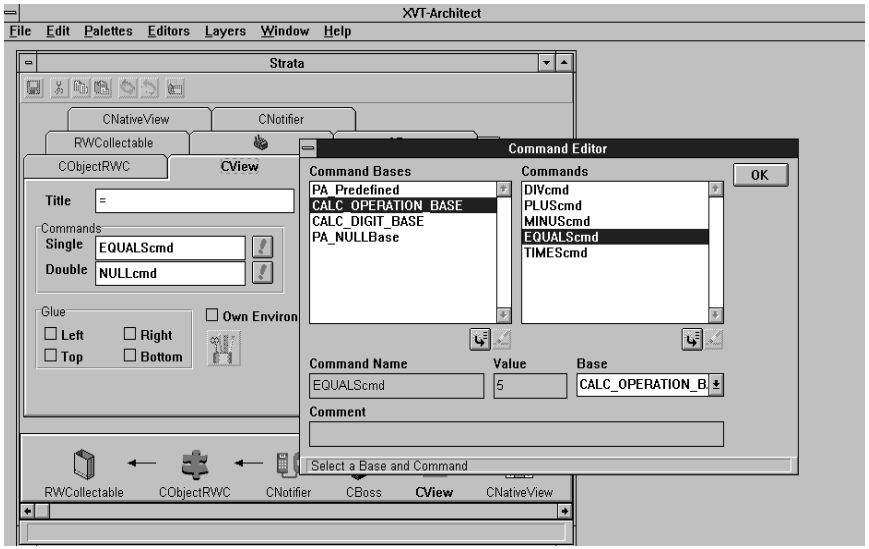


Figure 7.2. Strata with Command Editor

Example: You specify a "TOOL_COMMANDS" command base with a base value of 1000. Then, you specify several commands within that base with "TOOL1cmd", "TOOL2cmd", and "TOOL3cmd" as names and with command values 1, 2, and 3, respectively. Your generated Factory files will contain #defines for the commands as follows:

```
#define TOOL_COMMANDS 1000
#define TOOL1cmd (TOOL_COMMANDS + 1)
#define TOOL2cmd (TOOL_COMMANDS + 2)
#define TOOL3cmd (TOOL_COMMANDS + 3)
```

Note: XVT-Architect generates any numeric values that you do not explicitly specify.

See Also: For information on generated Factory files, see Chapter 8, *Object Factory*.
For information on specific XVT-Power++ commands, see the descriptions of CView, CButton, CMenuButton, NCheckBox, NEditControl, and NListEdit in the online *XVT-Power++ Reference*.

7.3.1. Using the Command Editor

The Command Editor has list boxes at the top, which list the Command Bases and the Commands that are already defined for your application.

t To get the list of commands that belong to a base:

Select the command base. The commands that belong to the base appear in the Commands list box.

7.3.1.1. Creating New Commands and Setting Command Data

In the Command Editor, you can create new commands, indicate command values, and specify comments.

t To create a new Command Base or a new Command:

Select a Base or a Command, click the Create button below the Command Base or Command list box, and type in the information.

t To edit a Command Base or a Command:

Select a Base or a Command, and type in the information.

The following are definitions of the variables that you can indicate using the Command Editor:

Base Name
Each command is attached to a base (e.g., CmdBase). You define the command bases for your project. Groups of commands can belong to a single base. If you type in a new command base, it is added to the list of command bases for the project. If you change a base, it is changed globally.

Command Name
Each command has a name that is a valid string identifier (e.g., NULLcmd). You can define the command names for your project. If you type in a new command, it is added to the list of existing commands belonging to the selected base.

Value
Each command base and command has a unique value associated with it. You can set the value. However, a specific base value is defined once per project, and if you change the base value, you change the value for that base globally. You can set this value, or you can use a value assigned by XVT-Architect.

Comments
You can indicate comments for the Base and the Command. XVT-Architect generates these comments in the header file that contains the #defines for the commands.

Base
Used to select the Base to which the Command belongs.

Note: The Base and Command Names must be valid identifiers. A valid identifier is a unique string that begins with a letter. The string can be any combination of letters, numbers, and underscores, but it *cannot* contain spaces.

7.4. String Editor

Using the String Editor, you can specify application-specific resource strings. The strings that you indicate for your application are placed within the Factory files.

The String Editor gives you the ability to view and modify the strings in your application. Using the String Editor, you have control over this global information.

You can open the String Editor from the Blueprint or Drafting Board by choosing Strings from the Editors menu. In addition, when indicating a Title on the CView page, you can access the String Editor.

Every view object has a Title string associated with it. If you indicate a Title on the CView page, the string is changed only for that instance. However, you can also open the String Editor to view and apply existing strings and to change global information.

t To open the String Editor:

In the Blueprint or Drafting Board, choose Strings from the Editors menu.
-OR-
In the Strata, click the Sting Editor button next to the Title field on the CView page.

In the String Editor, you can do the following:

- View the existing strings for the project
- Modify existing strings
- Create new strings, for use within XVT-Architect or directly in your application
- Delete strings that are not referenced within XVT-Architect

- Select a string to be used as the Title of an object (only when opened from the Strata)

See Also: For more information on generated Factory files, see Chapter 8, *Object Factory*.

7.4.1. Using the String Editor

The String Editor has a list box that contains an alphabetical list of the strings defined for your application. In the String Editor, you can also modify the existing strings and create new strings.

The editor has these two edit fields: String Name and String Value. When modifying an existing string or creating a new string, you can indicate both its String Name and its String Value. XVT-Architect gives all of the objects that you lay out a unique String Name.

String Name
The symbolic name of the string, which is the name you use to reference the string from within your application, like this:

```
new CText(CStringRW(String5));
```

XVT-Architect assigns a String Name to each string, but you can change the names. All String Names must be valid identifiers. A valid identifier is a unique string that begins with a letter. The string can be any combination of letters, numbers, and underscores, but it *cannot* contain spaces.

String Value
The literal string.

t To modify the String Name or String Value:

Click in the appropriate edit field, and type the new information.

t To create a new string:

Press the Create New String button, and type the new information.

When you create a new string, the String Value, “A New String,” is selected and ready for editing.

t To delete a string:

Select the string, and click the Delete button.

If a string is not referenced, you can delete it. If you select a string from the list box and it is not referenced, the Delete button is enabled.

7.4.1.1. Using the String Editor Opened from the Strata

When you open the String Editor from the Strata, you can also select an existing string from the list box and apply it to the object whose attributes you are editing; objects can "share" strings.

t To apply an existing string to an object:

Select the string from the list box, and click OK.

Once you give an object an existing string, its old string is no longer referenced (unless you have created another reference), and you can delete it.

Note: To apply an existing string to an object, you *must* click OK.

7.5. String List Editor

Using XVT-Architect's String List Editor, you can set string lists that appear in the list controls of your application at startup. You can access the String List Editor from the Blueprint, Drafting Board, and Strata.

The string lists that you indicate for your application are generated in the Factory files.

t To open the String Editor:

From the Blueprint or Drafting Board, choose String Lists from the Editors menu.
-OR-
From the Strata, click the Sting List Editor button next to the list button on the CListBox or CNativeList page.

In the String List Editor, you can do the following:

- Create or modify string lists
- Delete string lists that are not referenced in your XVT-Architect project
- Create or modify the strings of a string list
- Order and delete the strings of a string list
- Select a string list to use with an object (only when opened from the Strata)

See Also: For more information on generated Factory files, see Chapter 8, *Object Factory*.

7.5.1. Using the String List Editor

On the top-left the String List Editor, a list box contains the String Lists that are defined for your application. On the top-right of the editor, a list box contains the Strings of the selected string list (see Figure 7.3).

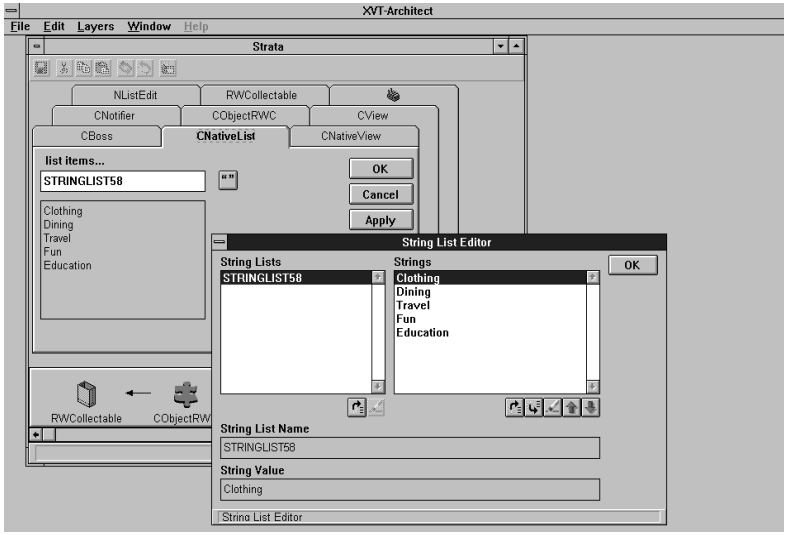


Figure 7.3. Strata with String List Editor

7.5.1.1. Setting String List Names

You can change the name of the string list or add new string list names. The String List Name is the symbolic name of the string list, which is the name you use to reference the string list from within your application, like this:

```
aNListBox->IListBox(CStringCollection(
      MY_STRING_LIST,
      MY_STRING_LIST_END));
```

XVT-Architect assigns a String List Name to each string list, but you can change the names.

Keep in mind that all String List Names must be valid identifiers. A valid identifier is a unique string that begins with a letter. The string can be any combination of letters, numbers, and underscores, but it *cannot* contain spaces.

t To change a string list's name:

1. Select the string list from the String Lists box.
2. Select the name in the String List Name edit field.
3. Type the new name.

t To create a new string list:

Click the Create New String List button (at the bottom-right corner of the String Lists list box), and type in the new name.

When you create a new string list, the new String List Name is selected and ready for editing, if you choose to do so.

t To delete an unreferenced string list:

Select the string list, and click the Delete button.

If a string list is not referenced in your XVT-Architect project, you can delete it. If you select a String List and it is not referenced, the Delete button is enabled.

7.5.1.2. Setting the Strings Values in String Lists

For each string list, you can indicate the Strings that it will contain.

t To add a string to a string list:

Click the Insert String button, which inserts a new string above the selected string.
-OR-
Click the Append String button, which inserts a string below the selected string.

When you create a new string, the String Value is selected and ready for editing. You can simply type in the new string.

In the Sting List Editor, you can also edit, move, and delete existing strings.

t To modify a string:

1. Select the string from the Strings list box.
2. Select the string in the String Value edit field.
3. Type the changes to the string.

t To move a string:

Select the string, click the Up or Down button.

t To delete a string:

Select the string, and click delete.

You can always delete strings in a string list, because the strings are not directly referenced.

7.5.1.3. Using the String List Editor Opened from the Strata

When you open the String List Editor from the Strata, you can also select an existing String List from the list box and apply it to the object whose attributes you are editing; objects can "share" string lists.

t To apply an existing string list to an object:

Select the String List from the list box, and click OK.

Once you give an object an existing string list, its old string list is no longer referenced (unless you have created the reference), and you can delete it.

Note: To apply an existing string list to an object, you *must* click OK.

8

OBJECT FACTORY

This chapter describes XVT-Architect's object Factory, including the Factory interface that you use to instantiate objects that you designed with XVT-Architect.

8.1. Object Factory

The Factory is a repository for application information, just like an XVT-Architect project file. However, they are different in that you use the Factory to instantiate the XVT-Power++ objects that you have laid out using XVT-Architect.

After you have designed your application using the Blueprint, Drafting Board, and Strata of XVT-Architect, generate the Factory. Then, while writing your application code, use the public interface of the PAFactory class to instantiate those objects. Using the PAFactory public methods, you can create single objects or multiple nested objects.

Note: You can generate the Factory at any point in your development process. However, since generating the Factory updates the application information, you should regenerate it if you modify your project.

If you change the Factory ID Name of an object after you have generated user Shell files and written code, you must search the old name and replace it with the new name in your application code.

You should *not* modify the generated Factory files. Every time you generate the Factory, the files are regenerated, and any modifications that you have made are overwritten. Additionally, if you modify the files, XVT cannot ensure compatibility with future releases of XVT-Architect.

See Also: For more information on generating files, see section 2.4 on page 2-8.

8.1.1. Factory Interface

The Factory has an interface that allows you to create objects. When you ask the Factory to create objects, you use an interface defined by the PAFactory class.

XVT-Architect also supports "object layering," which is the process of creating variations of a window in which you can change the objects that it contains to support different platforms or languages. When you generate the Factory files, you can generate the code for one or more of the layers. Then you can interact with the Factory interface to create the different layers.

See Also: For more information on the PAFactory class, see section 8.2 on page 8-4.
For more information on object layering, see Chapter 9, *Object Layering*.

8.1.2. Factory-generated Header Files

XVT-Architect generates Factory files, which include your project-information files and several header files. The generated Shell files reference these Factory files; they are automatically included in your application. They contain the following information:

- #defines for all object IDs
- #defines for all command IDs
- #defines for all string IDs and string list IDs
- Definitions of the data member classes

8.1.2.1. Object IDs

XVT-Architect's Factory generates a #define for each of the objects in your application. The Factory generates this internal ID for each object, but, when interacting with the Factory, you can use the Factory ID Name, or string identifier, which you gave the object in the Strata.

Both the PAFactory::Create* and the PAFactory::DoCreate* methods take a Factory ID Name. The Factory "knows" which object to create because XVT-Architect generates the necessary #defines.

See Also: For more information on indicating Factory names, see section 6.7.1.

8.1.2.2. Command IDs

XVT-Architect's Factory also generates a #define for each of the command IDs in your application. You indicate the command IDs in the Command Editor, which you can access from the Strata. In the Command Editor, you can indicate the command's name, value, base, and base value, and the XVT-Architect uses this data when generating the Factory files.

In addition, #defines are generated for the command IDs of menu tags, which you can specify using the Command Editor.

See Also: For more information on using the Command Editor, see section 7.3.1 on page 7-16.
For more information on the Menu Editor, see section 7.1.1 on page 7-3.

8.1.2.3. String and String List IDs

XVT-Architect's Factory also generates #defines for each string ID and string list ID in your project. In addition, XVT-Architect generates a string resource file, **strings.url**.

For your application, you specify strings and string lists in the String and String List Editors.

See Also: For more information specifying strings and string lists, see section 7.4.1 and section 7.5.1.

8.1.2.4. Data Member Classes

Finally, in the Factory files, XVT-Architect also generates a data member class for each object. A data member class contains pointers to the nested views of an object.

However, a data member class contains pointers to only those objects that you specified to be a part of the class on the Factory Settings page of the Strata. You can include only the objects that you will want access.

You use these classes when interacting with the PAFactory class. The PAFactory::DoCreate* methods have an optional parameter, the*DataMembers. If you want to be able to access an object's nested objects after you have created them, specify the*DataMembers parameter.

When you call DoCreate*, the nested views of an object are created. To access the nested views, pass in an instance of a data member class. The DoCreate* method then returns pointers to the nested objects specified in the data member class.

See Also: For more information on specifying whether an enclosed object is included in the data members class, see section 6.7.1.

Example: If you have MyWindow that contains both Oval and Rectangle, the generated data member class would look like this:

```
class MyWindowDataMembers : public CDataMembers
    {
        public:
            COval* itsOval;
            CRectangle* itsRectangle;
        ...
    };
```

8.1.3. Generating Factory Files at the Command Line

An alternate way to generate XVT-Architect factory files is to enter the following command at the command prompt:

```
arch -generate foobar.amf
```

Using this approach, you can generate factory files without starting the GUI version of XVT-Architect.

Tip: You can use this technique to automate your build process.

8.2. Using the PAFactory Class

XVT-Architect generates the PAFactory class, which is an abstract class. Through the PAFactory public interface, you can create documents, windows, and views. In addition, you have a choice of either creating a specific object, by calling one of the Create* methods, or creating a specific object and all of its enclosed objects, by first calling one of the Create* methods and then calling the corresponding DoCreate* method.

Use the Factory names (the string identifiers) that you gave the objects in the Strata to specify which objects should be created.

Note: XVT-Architect's object Factory is a PAFactory-derived class, and it can create the objects that you laid out in XVT-Architect. When using the Factory, make sure that your application is linked against the PAFactory library.

See Also: For information on naming objects, see section 6.7.1.

8.2.1. PAFactory Public Methods

This section lists PAFactory's public methods.

See Also: For more information on the optional the*DataMembers parameter for the DoCreate* methods, see section 8.1.2.4.

CDocument* ***CreateDocument***(long theDocID);

Creates a single document based on theDocID. theDocID is the string identifier of a document that you created and named with XVT-Architect. To create all of the documents of an application, call the DoCreateDocuments method. To create the document and its windows, first call this method, and then call the DoCreateWindows method.

CDocument* ***DoCreateDocuments***(
CApplication* theApp, long theAppID, BOOLEAN createAll = FALSE,
CDataMembers* theAppDataMembers = NULL);

Creates all of the documents of an application. theApp is a pointer to the application. theAppID corresponds to the string identifier of that document (the name you gave it in the Strata). If createAll is FALSE, only the documents that you checked to be automatically created (in the Factory Settings page of the Strata) are created. If createAll is TRUE, all of the documents are created. In addition, to get pointers to the documents, you can pass in a pointer to an object of type CDataMembers.

CWindow* ***CreateWindow***(
CDocument* theDocument,
long theWinID);

Creates a single window based on theDocument and theWinID. theDocument is a pointer to the document that manages the window. theWinID corresponds to the string identifier of a window that you laid out and named in XVT-Architect. To create the window and its enclosed views, first call this method, and then call the DoCreateViews method.

void ***DoCreateWindows***(
CDocument* theDocument,
long theDocID, BOOLEAN createAll = FALSE,
CDataMembers* theDocDataMembers = 0);

Creates all of theDocument's windows. To create a document and all of its windows, call CreateDocument, and then call this method.

theDocument is a pointer to the document that manages the windows. theDocID corresponds to the string identifier of that document (the name you gave it in the Strata). If createAll is FALSE, only the windows that you checked to be automatically created (in the Factory Settings page of the Strata) are created. If createAll is TRUE, all of the windows are created. Finally, to get pointers to the objects that the window encloses, you can pass in a pointer to an object of type CDataMembers.

CView* ***CreateView***(
CSubview* theEnclosure,
long theViewID);

Creates a single view based on theEnclosure and theViewID. theEnclosure specifies the enclosure of the view that you are creating. theViewID is the string identifier of a view that you laid out and named in XVT-Architect. To create the view and its enclosed views, first call this method, and then call the DoCreateViews method

void ***DoCreateViews***(
CSubview* theEnclosure,
long theEnclosureID,
BOOLEAN createAll = FALSE,
CDataMembers* theEnclosureDataMembers = 0);

Creates all of theEnclosure's views. If you want to create a window or subview and any or all of its enclosed objects, call CreateWindow or CreateView, and then call this method.

theEnclosure is a pointer to the enclosure. theEncloureID corresponds to the string identifier of the view (the name you gave it in the Strata). If createAll is FALSE, only the views that you checked to be automatically created (in the Factory Settings page of the Strata) are created. If createAll is TRUE, all of the views are created. Finally, to get pointers to the objects that the view encloses, you can pass in a pointer to an object of type CDataMembers.

9

OBJECT LAYERING

XVT-Architect provides you with the ability to "layer objects." This feature allows you to create variations of objects in a project. Each layer can be a modification of the layer that is defined as its parent.

You can adjust the windows of your project to achieve the appropriate look-and-feel on each platform to which you are porting. In addition, if you are localizing an application, you can create layers of objects that accommodate different languages.

This chapter describes the usage of XVT-Architect's layering feature.

9.1. Default and Parent Layers

When you create a new XVT-Architect project, you are working in the "default" layer. As you develop your application, you can add layers, but the first layer remains the default layer of the project. Based on this layer as the root, you can define your own layer hierarchy.

The default layer is the root that defines the common features of your application. Additional layers are "derived" from the default layer, and therefore they inherit all the attributes of that parent layer. If you make a change to the parent layer, all derived layers are also changed. However, derived layers can override certain aspects of their parent layer.

Example: In Figure 9.1, layers A and B inherit the objects of the parent layer, but some of the object attributes (position and size) have been modified.

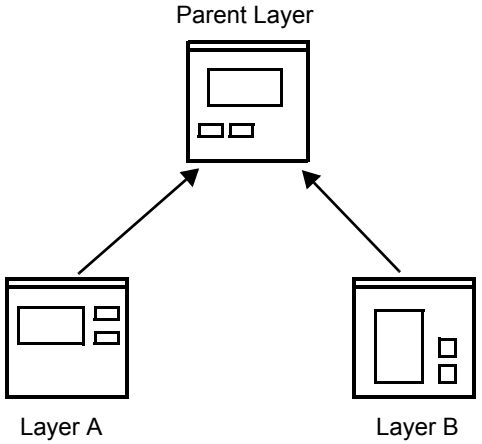


Figure 9.1. Object layering

9.2. Layering Objects

Using the Layer Editor, you can create new layers, indicate that XVT-Architect should generate the information for this layer in the Factory files, assign parent layers, and view a layer. You can open the Layer Editor from both the Blueprint and the Drafting Board.

t To open the Layer Editor:

Choose Edit Layers from the Layers menu in the I
the Drafting Board (see Figure 9.2).

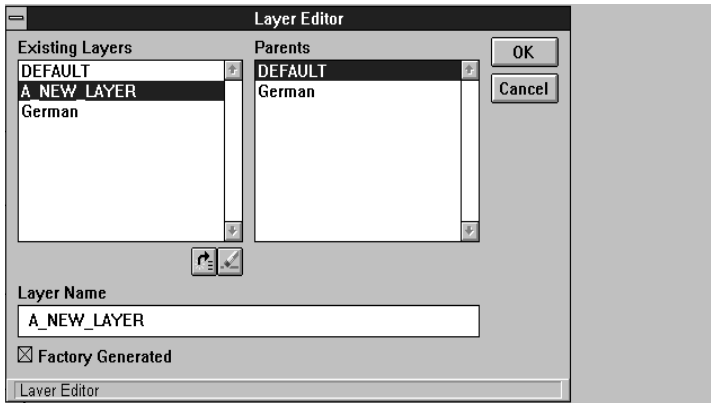


Figure 9.2. Layer Editor

9.2.1. Creating Layers

t To create a new layer:

1. In the Layer Editor, click the Add New button.
2. Name the layer and indicate whether you want XVT-Architect to include it in subsequent generations of the Factory. You can do so by checking the Factory Generated box.
3. Select the new layer's parent by selecting a parent from the Parents list box. The new layer "inherits" the objects and their attributes from the Parent that you indicate.
4. Click OK.

When you click OK, the Layers Editor closes and you return to the module from which you opened the editor. Note, however, that you are now in the new layer, which you just created.

Note: If you indicate that the layer should be generated, you can create the entire application based on that layer, using the Factory interface.

See Also: For more information on creating a layer from the Factory, see section 9.4.
For more information on the Factory, see Chapter 8, *Object Layering*.

9.2.2. Viewing and Modifying Layers

t To open a layer for viewing or editing:

Select the layer from the Existing Layers list box, and click OK.

See Also: For information on indicating attributes in a layer, see section 9.3.

9.2.3. Using the Layers Menu

The Layers menu provides access to the functionality related to layering. To open the Layer Editor, you can use the Edit Layers item from both the Blueprint and the Drafting Board. However, the other item on the Layers menu are only enabled in the Drafting Board. Using the Layers menu in the Drafting Board, you can view the layered objects and revert objects to be defined by their parent. First, choose a layer, and then use the Layers menu.

Viewing Layered Objects

t To view the layered, or overridden, objects:

Choose Show Layered objects from the Layer menu, which selects the objects that have overridden one or more attributes of the inherited layer.

Reverting an Object to be Defined by the Parent Layer

t To revert a layered object:

Select the object, or objects, and choose Revert Selected Objects from the Layers menu.

This action everts the selected layered objects to be defined by the parent, or base layer.

Selecting Multiple Objects

When using the Layer Selected Objects and Revert Selected Objects menu items, you can select multiple objects.

t To select multiple objects:

Press Shift, and click on the objects.
-OR-
Using the Pointer tool, drag out a section rectangle around the objects.

9.3. Indicating Variations in Layers

Once you enable an object to be layered, you can modify any layer just as you would the default layer, by using XVT-Architect's various features. Most of your modifications will only change the attributes of objects in the current layer, and the layers that inherit objects from that layer. These actions includes the following:

- Sizing or moving objects in the Drafting Board
- Editing most object properties in the Strata
- Editing menus
- Editing strings and string lists

Note however that some actions are layer-independent and will affect all layers. These actions include the following:

- Editing in the Blueprint
- Creation or deletion of objects
- Creation and deletion of commands and accelerators (available in all layers as global information)
- Changing the enclosure of objects in the Drafting Board

9.4. Factory Code

You can generate Factory code for one, some, or all defined layers. Using the Layer Editor, you can indicate which layers you want XVT-Architect to generate in the Factory files.

XVT-Architect always uses the default layer when generating the Shell files. If you have generated the Factory code for a layer, you can modify and add code to create that layer. You can create the entire layer, or you can create a portion of that layer.

Example: To create all objects from a layer, set the default Factory creation to that layer. For example, suppose your project has a layer for a French-speaking locale named "FRENCH". You can ensure that the application uses this layer at runtime by setting the default factory creation in your application class constructor like this:

```
factory.SetDefaultLayer(CFooFactory::FRENCH);
```

To create a portion of the layer, you would use the specific constructor. For example, if you want to create the German window, you would create that window using that constructor, like this:

```
CWindow* aWindow =
    factory[CFooFactory::GERMAN].CreateWindow(
    FavoriteWindow);
```

See Also: For more information on the Factory interface, see Chapter 8, *Object Factory*.

9.5. Creating Localized Projects Using Object Layers

XVT-Architect's layering capabilities allow you to create special versions of the same application specialized for different needs, such as language. In order to localize an application, you create a specific layer to represent each locale. In general, an XVT-Architect application is localized by going through the following steps:

1. Identify and define the locales that need support
2. Create a layer for each locale
3. Localize the attributes of the objects in each layer
4. Generate a localized factory

Each of these steps is described in detail below.

See Also: XVT differentiates between internationalization and localization—the characteristics and needs of your end users will determine whether you fully implement both in your application. For more information about internationalization and localization, including a comparison of the two processes, refer to the chapter on *Multibyte Character Sets and Localization* in the *XVT Portability Toolkit Guide*.

9.5.1. Choosing and Defining Locales

The first step in localizing an XVT-Architect application is to define the attributes of the locale.

A *locale* usually corresponds to a particular geographic area or district that shares the same language and the same conventions for representing culturally meaningful information.

9.5.1.1. Defining the Attributes of the Locale

To localize an XVT-Architect application, use the Global Options dialog, shown in Figure XXX. To invoke the Locale Options Editor, select **Options=>Global**, and within the dialog select the **Locales** tab. The contents of the **Locales** tab is shown in Figure XXX.

[Place snapshot of dialog]

Using this dialog, you create new locale definitions which are then displayed in various Locale list boxes, such as the one in the Layer Editor. Once created, you can reselect any of the locales to edit its settings. The settings you can modify are:

Name

Enter the string you wish to use to identify the locale. This name is used throughout XVT-Architect to refer to this locale.

Codeset

The codeset used to represent the characters of the language used with this particular locale. Before adapting your application strings and help files, you must select a character codeset that supports the target language. In making this decision, evaluate the language characters that must be represented, the fonts that support these characters, and the relative availability of these character codesets and fonts on your target windowing and operating systems.

Advanced

Reserved for future use. (Future attribute settings may include date formatting, monetary formatting, color preferences, etc.)

XVT-Architect pulls its codeset definitions from files created with XVT's codeset mapping tool, and uses these codeset definitions to "codeset map" each textual object. The conversion itself occurs inside XVT-Architect.

To locate these codeset files, XVT-Architect searches the **...bin/codemaps** directory in the **$XVT_DSP_DIR** path. Each time you invoke the Locale Options Editor, the directory is searched and available codeset files are presented as choices for each locale.

See Also: Refer to your native platform documentation for more information on setting up a system to support a particular locale.
For more information on XVT's codeset mapping tool, refer to the *XVT Platform-Specific Book* for your platform.
Different codesets used on the various platforms that XVT supports are listed in section A.2 in Appendix A.

9.5.1.2. Scope of Locale Definitions

Locales are defined via the Global Options dialog, using the Locale Options Editor. This means that the definitions of locales are stored in XVT-Architect's global options file. A global options file is associated with each version of XVT-Architect on each platform. Since all projects have access to these files, the locales themselves are not part of any project. Consequently, your locales must be defined on each platform where you are running XVT-Architect.

Implementation Note: Different platforms use different codesets for representing the same language. For example, the codeset used to represent Swedish on a Macintosh is not the same as the codeset used on a Windows platform. The Locale Options Editor allows you to correctly assign codesets for each locale for each supported platform.

9.5.2. Creating Layers

The second step in localizing an XVT-Architect application is to create a layer for each locale. To create or edit a layer, open the Layer Editor, shown in Figure XXX.

[Place snapshot of Layer Editor]

Once you have created a layer, you can assign it its own locale by choosing from a list of locales known to XVT-Architect. This list is displayed in the list button near the bottom of the Layer Editor. The locales listed there correspond to those created earlier using the Locale Options Editor.

After you assign a locale to a particular layer, XVT-Architect automatically transforms certain object attributes to match the settings chosen for an application. For example, XVT-Architect ensures that the strings displayed by the objects of each layer are processed using the correct codeset defined in the locale of the layer.

See Also: For more information on the fundamental concepts of layering, refer to section 9.1.
For more information on how to use XVT-Architect's Layer Editor, refer to section 9.2.

9.5.3. Localizing Each Layer's Objects

After creating a layer and assigning it a locale, you may need to adjust specific objects for the locale. This frequently includes changing the position or size of some objects. You may also need to choose different colors or images for certain contexts.

9.5.3.1. Replacing Colors and Graphics

End users will be more successful using your application if it conforms to their expectations and does not contain culturally offensive symbols or colors. Be sure to replace any locale-sensitive colors and graphics (drawn bitmaps, icons, or images) with ones appropriate to your target locale.

See Also: For more information on externalizing colors and graphics (to simplify the process you use when localizing your XVT applications), refer to the chapter on *Multibyte Character Sets and Localization* in the *XVT Portability Toolkit Guide*.
To see hundreds of examples of international symbols used in various fields of endeavor, refer to *Symbol Sourcebook: An Authoritative Guide to International Graphic Symbols*, by Henry Dreyfuss, published by Van Nostrand Reinhold, New York, N.Y., 1984.

9.5.3.2. Translating Strings

In almost all cases, the localization process involves translating the text displayed in the application to the correct language. Any strings that are displayed to your users are candidates for language translation. These include, but are not limited to, menu item titles, keyboard accelerators and mnemonics, window titles, dialog titles, control titles, text and mnemonics, error messages, and help topics and text.

The following subsections describe methods you may wish to consider to translate the various textual components of your user interface.

Note: When translating strings directly in XVT-Architect, you must be working at a system that has been set up to support the language(s) and any special codesets that are required to represent those language(s).

Object Titles

The most direct method for translating text displayed in the views of the application is to edit the titles or text contents of those objects directly. To do this, you select a specific view at a time and bring up its Object Strata. Of course, this method of translation is not very convenient and would be inefficient for a large project. Nevertheless, this method is sufficient when you only need to change a few view titles at a time for different locales.

String Editor

A more robust method for translating most strings is to open the String and String List Editors for the project. The function of these editors is to simultaneously display all the strings that the application will ever display at runtime inside its various views. The editors provide a "one-stop" place for translating all strings in a convenient and efficient manner. After making the changes, the views in XVT-Architect will display their newly translated titles and text contents.

See Also: For more information about the String and String List Editors, refer to Chapter 7, *Editors*.

Menu Titles

Each menu in an application can be translated directly by editing the titles of each menu item. Menu item titles are not displayed by the String or String List Editors, so they must be edited within the Menu Editor.

You may not need to translate the titles of standard menus, e.g., M_FILE, M_EDIT, M_FONT, and M_HELP, since the XVT Portability Toolkit already provides localized versions of them for five languages.

See Also: For more information on localizing menus with XVT-Architect, refer to section 7.1.2 on page 7-7.

Import/Export Files

Another convenient way to translate all strings in an XVT-Architect application is to export the project. The exported project contains a set of text files that can be manipulated with a text editor. This is especially convenient when you have decided to use a third party to help you with the translation, since the text files can be translated using any word processor running on virtually any platform the "third party" wishes to use.

See Also: For more information about the content of the exported text files, refer to section 11.1.1 on page 11-2.

Help Topics and Help Text

XVT provides pre-translated help topic text for several of the reserved topic symbols, including XVT_TPC_HELPONHELP, XVT_TPC_KEYBOARD, and others. You have access to this pre-translated help topic text at the XVT Portability Toolkit level.

A basic subset of help topics have been translated into the following five languages:

- Japanese
- Italian
- French
- German
- English

Help source text files are compiled, using **helpc**, into binary resource files. The resultant binary file can be associated with your XVT application at start-up time or runtime. This means that a program executable can be the same in all environments and only the binary resource files (including help) have to be customized for a target locale.

Help source text files and the binary help files generated by **helpc** are portable between XVT-supported platforms if the target character codesets are compatible. Specifically, characters used in the help source files must come from the invariant character codeset. U.S. English and Japanese Shift-JIS are generally the only two languages and character codesets for which this is true.

See Also: For more information on providing help with your XVT applications, refer to the chapter on *Hypertext Online Help* in the *XVT Portability Toolkit Guide*.
For more information on the invariant character codeset, refer to the chapter on *Multibyte Character Sets and Localization* in the *XVT Portability Toolkit Guide*.

9.5.4. Generating a Localized Factory

The final step of the localization process is to generate an XVT-Architect factory that represents the locale for the executable application. The two options when doing this are:

Generate all layers
Generate a factory file that contains all layers of the application. When you do this, your application end users will be able to

choose a layer, or locale, at runtime. However, generating all project layers results in a larger factory file.

Generate just a few layers

Limit the generation to just the one or more layers that are needed by the executable that you are currently building for a particular group of end users. This results in a smaller factory file, but the application is restricted at compile time to support only the generated layers, or locales.

For either approach, the Layer Editor allows you to specify which layers should be generated in the factory. To find out how to select the layer to use within the generated application, see section 11.2 on page 11-2.

10

CUSTOMIZING XVT-ARCHITECT

10.1. Where to Save Shell and Makefile Templates

To change where XVT-Architect searches for its shell and makefile templates, select **Options=>Global** after starting XVT-Architect—this opens an Options dialog. Click the **Generation** tab of the Options dialog, then enter a new path in the **Shell Path** field.

Note: You can make a similar change by selecting **Options=>Project**, but then the new path is used for the current project only. (In previous releases of XVT-Architect, the path to the shell and makefile templates was set using the environment variable XA_SHELL_PATH.)

10.2. Number of Files Used to Store Generated Factory Code

To change the number of files used to store generated factory code, select **Options=>Global** after starting XVT-Architect—this opens an Options dialog. Click the **Generation** tab of the Options dialog, then enter a new number in the **Factory File Count** field.

Note: You can make a similar change by selecting **Options=>Project**, but then the change affects the current project only. (In previous releases of XVT-Architect, the number of files used to store generated factory code was set using the environment variable XA_FACTORY_FILE_COUNT.)

10.3. File Extension Used When Generating Shell or Factory Files

To change the file extension used when generating shell or factory files, select **Options=>Global** after starting XVT-Architect—this opens an Options dialog. Click the **Generation** tab of the Options dialog, then enter a new string in the **Source File Extension** field. The extension string you specify must include the period. Also, the string must constitute a valid extension for your platform, specifically, no more than three characters on platforms that require eight-dot-three filenames.

For example, you can set the filename extension to "**.cxx**", forcing XVT-Architect to append this extension rather than the default "**.cpp**" extension. Many platforms do not limit the length of filename extensions, and XVT-Architect does not, either. In other words, you can define the filename extension to be as many characters as you wish, as long as that string is valid for the platform on which you are compiling.

Note: You can make a similar change by selecting **Options=>Project**, but then the change affects the current project only. (In previous releases of XVT-Architect, the file extension of generated files was set using the environment variable XA_FILE_EXTENSION.)

11

IMPORTING AND EXPORTING STRINGS

XVT-Architect provides you with the ability to import and export strings and menu titles of a project. Using the import and export feature, you can do the following:

- Import strings into an XVT-Architect project
- Export XVT-Architect project strings to a human-readable format
- Export the layers of a project, so that they can be localized and then imported back into XVT-Architect

When you export a file from XVT-Architect, you create an "externalized project."

This chapter describes the format of the "externalized project," and it describes how to use the importing and exporting feature.

Note: Before you import or export the strings of a project, you should have an understanding of using XVT-Architect. In particular, you should understand object layering.

See Also: For more information on object layering, see Chapter 9, *Object Layering*.

11.1. Externalized Projects

An externalized project consists of several files containing the project information. Each file is referenced in the master file.

A master file contains an "overview" of an externalized project's information. It also indicates the file in which specific information is stored. The master file contains the following:

- Name of the project from which it originated
- The layers in the project (that have been exported)
- The classes of objects that are in the project, separated by layer
- The locations of these instances of data

See Also: For more information about exporting strings for the purpose of localizing an XVT-Architect application, refer to section 11.2.1.

11.1.1. Export File Types

Two types of export files are generated. The first type is a file with the extension **.ame**. This file is an index that tells you where to find externalized data specific to a specific object type in XVT-Architect. The other type of export file has the extension **.aeo**. These files contain the actual object definitions for the contents of the project.

Each layer has its own set of object files. An object file contains all of the instances of a specific class of object in that layer. An object file first enumerates the attributes of the class, and then lists the values for each instance of that class.

11.2. Exporting Project Strings

You can export XVT-Architect project strings to the externalized project format.

t To export XVT-Architect project strings:

1. Choose **File=>Export Strings**; the Export Strings dialog is displayed.
2. In the Save File dialog that is displayed, indicate the location and name of the master file. The default filename extension of a master file is **.ame**.

The Export dialog contains a list of all the layers in the project.When you click OK, all layers in the list are exported. The entire externalized project is saved in the same location.

Once you export the strings, it is best not to modify the project (**.amf**) with XVT-Architect until the strings are imported back into XVT-Architect. The externalized strings are identified by IDs that are referenced by the objects in the XVT-Architect project. Modifying the project could invalidate some of the ID references.

11.2.1. Exporting Strings for Localization

One obvious reason to export a project is to localize one or more of its layers. XVT-Architect generates numerous files, but you only need to localize a small subset of them. Follow the standard steps for exporting, as described in section 11.2. When you export a layer, all strings in it are located in a single object file, and it is easy to translate the strings in that file.

XVT-Architect generates all strings for each layer in a separate file. This includes strings used inside views as well as the titles of menu items. The first step you must take after exporting the project is to open the project's **.ame** file and use it to identify the files for each layer that contain the text of the application. For example, the following is a portion of the **.ame** file generated for the sample project in **...samples/arch/i18n**:

```
project I18N
layer Spanish DEFAULT
layer French DEFAULT
objfile 0 CStringRWC DEFAULT 0 00004e90.aeo
objfile 0 CStringRWC Spanish 1040 41004e90.aeo
objfile 0 CStringRWC French 1041 41104e90.aeo
objfile 0 PAUserString DEFAULT 0 00005617.aeo
objfile 0 PAUserString Spanish 1040 41005617.aeo
objfile 0 PAUserString French 1041 41105617.aeo
...
```

user strings for each layer of project

This project contains three layers: Default, Spanish, and French. The three lines of the file highlighted with the callout represent the three files that define all the displayable strings (or "UserStrings") in each layer. Once the strings in each file are translated to the desired language, the project can be re-imported into XVT-Architect, and will appear to be localized.

Tip: To translate these files into another language, or to have them translated by a third party, you must: 1) have access to an editor that supports the target language, 2) have fonts for that language, and 3) have access to the character codeset that matches the language and platform from which XVT-Architect will import the file.

11.2.2. Additional Files that Can Be Localized

Both at the Portability Toolkit level and at the XVT-Power++ level, XVT provides localized versions of its standard resource text and help source text for U.S. English, German, French, Italian, and Japanese. These localizations are encapsulated in include files referenced by XVT URL and help source text files. You may control the inclusion of these files by defining a LANG_* constant on the command line for **curl** or **helpc**, or by defining the constant in your source files.

If your target locale is not one of the encapsulated locales listed above, you create your own external resource and help source text files. Files you will need to translate (and rename) include **uengasc.h**, **pengasc.h**, **hengasc.h**, and **ERRCODES.TXT**. You also need to modify **url.h** to add references to the names of the newly translated files.

See Also: For more information about this alternate approach to internationalization and localization, refer to the chapter on *Multibyte Character Sets and Localization* in the *XVT Portability Toolkit Guide*.

11.3. Importing Project Strings

You can import externalized project strings into XVT-Architect.

t To import externalized strings:

1. Open a project (**.amf**) file.
2. Choose **File=>Import Strings**, and specify a master (**.ame**) file that contains the strings for the project.

XVT-Architect begins to read the strings and inserts them into the project's string manager and menubars. The objects in the project that use string resources will then start using the newly imported strings.

11.3.1. Detecting Problems

When XVT-Architect encounters a problem during the import process, it stops the process, reports the problem, and allows you to choose how to handle the problem.

For example, if there is an object ID conflict between the existing XVT-Architect project and the project that you are importing, XVT-Architect stops importing and opens a dialog that allows you

to handle the problem in different ways. You could choose to give the imported object a unique ID, skip the imported object in the import process, or override the existing object.

11.3.2. Detecting Errors

If XVT-Architect encounters an error during the import process, it stops the process and opens a dialog that describes the problem. For example, if XVT-Architect detects a syntax error in the external project files, or if the master file references a file that does not exist, you are alerted with the dialog. You can then fix the problem and try to import the project again.

12

APPLICATION PROGRAMMING WITH XVT-POWER++

This chapter gives you an overview of the roles assigned to different parts of the XVT-Power++ system so that you will know how the various components interact with one another. The main parts that are discussed are: Application, Document, and View. The chapter is organized in terms of the specific development tasks that you are likely to perform.

12.1. Application Level

The application level of an XVT-Power++ application controls various aspects of the program: starting it and shutting it down and initializing the network connections, database connections, and any other connections needed by the application.

The following development tasks are handled at the application level:

- Controlling the program
- Handling application startup and cleanup
- Providing global objects and global data
- Getting access to global objects and global data
- Finding out about global definitions
- Creating documents

Each one of these tasks is now discussed in more detail.

12.1.1. Controlling the Program

Each XVT-Power++ application has one CApplication object that takes the thread of control when the program is started. This object is the first one to be instantiated and the last one to be destroyed. CApplication is an abstract class from which you derive the specific application object for your program.

Once the CApplication object is instantiated, it sets up any default menus and may put up a menubar, display a splash screen, or display an About window. It has basic functions for managing and creating documents and for communicating with different objects within the application through the use of XVT-Power++'s commands and messages. It also provides some basic means of communicating with the user, such as bringing up dialog boxes to open files, set up printing, create a document, add a document, notify all documents to close, and indicate which document has a particular piece of data.

12.1.2. Handling Application Startup

Your program transfers control to XVT-Power++ inside of main. To start your program, giving control to XVT-Power++, create an instance of your user-derived CApplication class and invoke the Go method. An example of this procedure, found in the **startap.cxx** file, is shown here:

```
void main(int argc, char* argv[])
{
    CMyApp theApplication;
    theApplication.Go(argc, argv, MY_MENU_BAR_RID,
        ABOUTBOX, "BaseName", "Application Name",
        "Task Title");
}
```

Since XVT-Power++ never returns control to your program once the Go message is processed, no code should be placed after the call to Go.

12.1.3. Handling Application Cleanup

The CApplication class has a virtual Shutdown method that is invoked during the termination of the program. You can override this method to handle any special cleanup required by your application.

12.1.4. Providing Global Objects and Global Data

XVT-Power++ provides a class, CGlobalUser, which you optionally can derive. Use the derived class to define your own references to global objects, flags, or attributes. If you derive off this class, keep in mind that it is instantiated by your user-defined CApplication object and is processed through the CApplication initializer.

Another class, CGlobalClassLib, contains the global variables for the class library. This class is privately defined, and you should not add any application-specific information to it.

One example of a global object is the XVT-Power++ desktop. It is a class in charge of managing the windows on the screen (e.g., their placement and stacking order). Each application has one desktop object that is global to everything and that is activated through CGlobalClassLib.

12.1.5. Getting Access to Global Objects and Global Data

Your application gets access to all global data through CObjectRWC, which is initialized through the CApplication class. The CApplication initializer allows you to set up the global user data for your user-derived application object.

CObjectRWC contains two static functions: GetGU() for user-supplied globals and GetG() for XVT-Power++ globals. Anything that inherits through CObjectRWC has access to the global data through one of these functions. You must set each of these functions to an actual object. When a CGlobalUser object is created, the CBoss initializer informs CBoss of the existence of the global utilities for the user application.

12.1.6. Finding Out About Global Definitions in XVT-Power++

The class that contains global definitions for XVT-Power++ is **Global.h**, which is actually a file. You may need to refer to it occasionally to find out how something is defined; do not modify this file.

Consult **Global.h** when you need to know about glue types, default parameters, internal XVT-Power++ commands, etc. You may need to know the resources that XVT-Power++ defines internally or the XVT-Power++ ID number base.

See Also: For more information about **Global.h**, see the description of Global in the online *XVT-Power++ Reference*.

12.1.7. Creating Documents

Your user-defined CApplication object is responsible for instantiating one or more user-defined and -derived CDocument objects.

12.1.8. Propagating Messages

CBoss, which is never itself instantiated, supplies the basis for the event and message-passing structure. It has three methods for event hooks that are located inside objects throughout the application framework hierarchy: DoCommand, DoMenuCommand, and ChangeFont.

DoCommands can be passed up the entire hierarchy, from the deepest subview on up to the window, from the window to the document, and finally from the document up to the application.

It is very important when you overload a DoCommand that it call the inherited DoCommand by default, which in turn calls the CApplication object's DoCommand. Calling the inherited DoCommand as a default permits the propagation of data.

12.1.9. Creating a Desktop to Manage Screen Window Layout

The user-derived CApplication object creates one CDesktop object per application. All core XVT-Power++ classes have access to the desktop through the global references stored by the following:

```
CObjectRWC::G->GetDesktop();
```

12.1.10. Setting Up Menus and Handling Menu Commands

The setting up of menus is typically done in the following two places:

- For a task window, the menubar is set up in CApplication's SetUpMenus method
- For any other window, the menubar is set up in the respective window's constructor

Menu commands are propagated via the CBoss virtual DoMenuCommand method. This method allows the handling of

menu commands at a variety of levels—either at the window, the document, or the application levels.

12.2. Document Level

The document level of XVT-Power++'s application framework is responsible for accessing and managing data. The CDocument object manipulates files or internal pieces of data and acts as the link between the application and the views of the data. A document cannot itself display data, so it instantiates a window in which the data can be viewed.

The following development tasks are handled at the document level:

- Getting access to data
- Creating windows
- Creating variations on the basic window
- Creating dialog windows
- Managing data

12.2.1. Getting Access to Data

Objects of the CDocument class are responsible for providing access to the model data to be displayed inside views, in the form of files, records, hooks to a database, and so on.

How a document obtains its data is up to the individual application. If a document is to be in charge of a file, it must be able to open a file, read and write from it, and close it. If a document needs information from a database, it must be able to make the connections to the database, select from it, commit changes to the database, and so on. You may write your own data access methods, or you can use XVT-Power++'s TDI adapter classes. You may choose to put some of the code for obtaining and setting up data in the CDocument constructor—if there is not much initial setup to do or if the document is going to use the data upon creation.

12.2.2. Managing Data

A CDocument object serves as a central means of communication for changes and updates in different windows. It has hooks for saving data, printing it, closing it, and so on.

Even after the document is created, you can delay the obtaining of data until a method of the document is called. This depends on how

you design your documents. Basically, keep in mind that you must add methods that obtain data, and these methods should be called at different points by the application. A CDocument-derived class could override the CDocument DoOpen method so that it is called by the application after it creates the document or when a user selects Open from the menubar.

Automatic Data Propagation (ADP) and Transparent Data Integration (TDI) both provide a way for you to build complex models and have associated views updated automatically. Using the Model View Controller approach, ADP automatically propagates a change of data from objects to other objects. TDI is a good alternative to ADP when working with complex data sources through an abstract interface. TDI can serve as a bridge between objects developed independently in separate products/projects.

See Also: For more information on ADP, see Chapter 29.
For a comparison of ADP and TDI, see section 30.4.

12.2.3. Creating Windows

Objects of the CDocument class are responsible for instantiating and managing a common set of windows. Group your application windows based on functionality or other similarities and use one CDocument object per group to create and manage the windows. The BuildWindow method of CDocument is where you put the code that instantiates a window. The BuildWindow mechanism can be called from DoOpen or DoNew.

12.2.3.1. Creating a Task Window

Inheriting from XVT-Power++'s basic window class, CWindow, is a variant child class: CTaskWin.

CTaskWin is used on platforms that require a task window to enclose all other windows in the application, such as MS-Windows, Windows NT, Windows 95, and OS/2.

12.2.3.2. Creating Modal Windows

Use the DoModal method of CWindow to make a window modal. Modal windows are useful when your application needs an item of information or a commitment from the user before it can continue. When a modal window is opened, it takes over the screen and disables all other windows so that nothing else can happen while it is open.

12.2.3.3. Creating Dialog Windows

XVT-Power++ has a class for creating dialog boxes: CDialog. In XVT-Power++, dialog windows are different than regular windows. Dialogs are defined in URL resource format and thus do not inherit the properties of CWindow, such as the ability to nest other objects.

Dialogs can contain only objects that are defined as controls in a URL resource file Like windows, dialogs can be made modal with a call to CDialog:DoModal() so that they take over the screen when they are invoked, disabling all other objects and operations until the user clicks on the necessary button or responds to the dialog in some other way that is indicated. In contrast, a regular, modeless dialog does not take over the screen and can go into the background when another window or dialog is brought to the front.

See Also: For more information on dialogs, see the "Dialogs" chapter in the *XVT Portability Toolkit Guide*.
For more information on using resources, see the "Resources and URL" chapter in the *XVT Portability Toolkit Guide*.

12.3. View Level

The view hierarchy is the most extensive branch of XVT-Power++. It comprises all of the classes that display some type of object on the screen when they are instantiated. The parent of all these classes is CView, an abstract class.

12.3.1. Displaying Data

Any class that inherits from CView can display itself.This includes CWindow, which constitutes the link between views and documents. CWindow is responsible for displaying data. The window is the topmost view in the nesting of views.

CWindow is an abstract class. but each CWindow object corresponds to an actual window on the screen. A window object can be of any XVT Portability Toolkit type and receives all window events. Most of the window management, such as moving and sizing, is done by the window manager or the XVT-Power++ desktop. (CDesktop is a class in charge of managing the placement and stacking order of windows on the screen.)

See Also: For information on the XVT window types, see CWindow in the online *XVT-Power++ Reference*; also see the "Windows" chapter in the *XVT Portability Toolkit Guide*.

12.3.2. Supplying Native Controls

CNativeView is an abstract class from which several different types of controls have been derived for you, among them buttons, icons, check boxes, scrollbars, list edits, list buttons, list boxes, and radio buttons. When native view classes are instantiated, they adopt the look-and-feel of the native window system in which the application is running.

Native views are the means of communication between the application and the user who is operating the mouse. The user performs such operations as:

- Clicking on a button to start an event
- Scrolling by clicking on a scrollbar
- Making selections by clicking on check boxes
- Making different choices by clicking on radio buttons
- Selecting an item from a list box

While native views can be placed inside views, they cannot contain any views themselves.

12.3.3. Nesting One View Within Another

CSubview and its subclasses can nest views recursively within other views. For example, you can display your own bitmap drawings and other GUI objects inside subviews. This is the basic property that distinguishes subviews from other classes in the CView hierarchy. CSubview classes can propagate events to all of their nested subviews. Subviews can have a *selected view* or *selected key focus*, which is the first (and possibly only) subview to receive a certain kind of event.

12.3.4. Drawing Basic Shapes

The shape hierarchy, probably the most self-explanatory of the XVT-Power++ view hierarchies, includes squares, circles, arcs, polygons, rectangles, and other basic shapes that a user can draw. Some of the objects can be rotated. Application designers can use the shapes to decorate windows and other objects or to communicate with the user pictorially.

Like all *objects* in the frame hierarchy, the XVT-Power++ shapes can receive and communicate events. Like all *views*, shapes can generate automatic DoCommands when the user clicks the mouse on them. Thus, they are useful not only for purposes of decoration but

also as building blocks to create other objects, such as new buttons. Several of the shapes are very easy to create. For example, you can easily create the following:

- A polygon by giving a number of sides
- An arc by giving starting and ending positions
- A circle by giving a center and a radius

Suppose that you want a stop sign shape that can act as a button. You would create an octagon using the regular polygon class, nest within this shape a piece of text that says "STOP," and then give it a command number indicating that you want it to act as a Stop button. The result is a new button object that allows the user to interact with your application.

12.3.5. Creating Grids of Cells

A grid object is a grouping of cells that are arranged into rows and columns. XVT-Power++ offers three grid classes: the abstract class CGrid, and the classes CFixedGrid and CVariableGrid that allow you to create grids with either fixed or variable-sized cells, respectively. For a color palette window in which you want all squares representing color selections to be the same size, you would use the fixed grid. In a spreadsheet, however, you would use the variable grid so that the columns can have different widths.

A grid has these characteristics:

- It can be either visible or invisible
- It is placed inside a view or window
- It enforces sizing and clipping of items

Grids are useful in CAD-type applications or other applications where you want to set the granularity of placement within the drawing area or designing board to a set of grid cells rather than to pixel locations. An item inserted into a grid snaps to a certain corner of its cell and can be centered within the cell; objects snap as they are moved or sized within a cell.

See Also: For information about two classes that provide other approaches for displaying data in neatly aligned rows and columns, see the descriptions of CTreeView and CTable in the online *XVT-Power++ Reference*.

12.3.6. Displaying Lists of Selectable Items

Use the CListBox class to display a scrollable box containing a list of selectable text items. The list box is used for choosing from a listing of directories and for navigating among directories.

12.3.7. Providing Text Editing Facilities

The abstract class CNativeTextEdit and its subclasses provide: 1) a one-line text area, 2) a variable-sized text editing area, and 3) a scrolling text area.

CNativeTextEdit objects can be nested inside a subview, clipped, or hidden. This class has easy-to-use methods for getting and setting part or all of the regular and selected text. It supports quick selection (e.g., selection of all text upon any event, such as clicking once, and backspacing). CNativeTextEdit provides text validation hooks upon key events.

Below CNativeTextEdit, the text editing tree branches out into these three variations of text editing objects:

NLineText
: Commonly used for one-line text entries, such as a place for users to enter their everyday name or computer username.

NTextEdit
: A variable-size text editing area in which you can set several attributes, to scroll or not to scroll, for example.

NScrollText
: Provides scrollbars and automatic scrolling.

Note: All of these classes can handle multibyte characters (in a multibyte-enabled XVT application).

12.3.7.1. CText versus CNativeTextEdit

CText is a static text drawing class that supports control characters such as tabs, carriage returns, and line feeds. CText also supports justification and multibyte (wide) characters, and can be customized with specific fonts and colors. It is useful for one-line instructions, titles, and button names.

You can always use the more flexible NTextEdit class in read-only mode to display single lines of text inside a window or subview. However, while NTextEdit is more flexible than CText, it also carries more baggage with it than you may want for one-line displays of read-only text.

12.3.8. Designating an Area of the Screen as a Sketching Area

The CSketchPad class is provided for interaction with the user who wants to dynamically draw or create new objects inside a window. CSketchPad can be a basis for CASE or drawing programs.

Like most drawing programs, CSketchPad has an area—called a *sketchpad* in XVT-Power++—in which the user can drag the mouse to sketch out or draw such shapes as rectangles, circles, lines, and so on. The user can drag out a region to select multiple objects within that region.

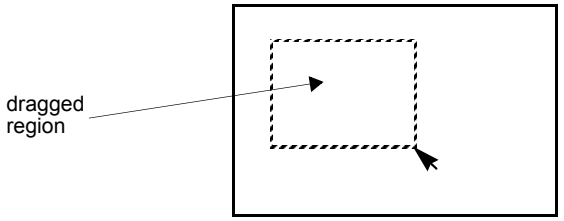


Figure 12.1. Defining a sketchpad on the screen

12.3.9. Creating a Rubberband Frame

To get a rubberband frame that surrounds a CView object, enabling it to be dragged or sized with the mouse, use CWireFrame. This CView class acts as a helper to all other CView classes. Typically, you do not need to be aware of CWireFrame since it is used internally. However, you can override this class and modify the way the wire frame draws or alter the way the rubberbanding is implemented.

12.3.10. Representing an Area on the Screen with a Virtual Size Larger Than its Display Area

CVirtualFrame is an abstract class that has two regions associated with it—a real, visible region that is located inside a window or some other view, and a virtual region. Its subclass, CScroller, represents a virtual frame with scrollbars attached to the viewing area.

Virtual frames are created for cases when the screen is not large enough to display all of the viewable information at once. The virtual frame allows you to create subviews that are of any virtual size you want (e.g., 3,000 pixels by 5,000 pixels). The information is actually displayed on the screen inside the real frame, which can be much smaller than the virtual frame. Only a certain area of the

viewable information can be viewed through the real frame at one time.

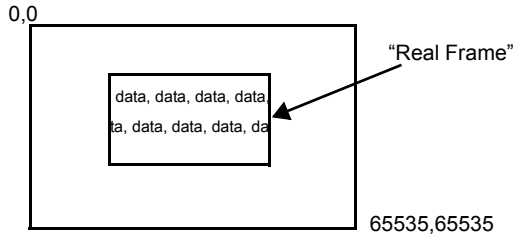


Figure 12.2. Virtual frames display small portions of large datasets

The user views the different areas of the virtual frame by scrolling the real frame up and down. CVirtualFrame is used in conjunction with the CScroller class, which attaches scrollbars, either vertical or horizontal or both, to the real frame area. Thus, the user can scroll the virtual frame around to view whatever displayable text resides inside the virtual area.

12.3.11. Attaching Scrollbars to a Window or View

XVT-Power++ contains two classes that pertain to scrollbars. NScrollBar provides a horizontal and/or vertical scrollbar that has the look-and-feel of scrollbars in the native window manager. These scrollbars can be created anywhere inside a CSubview. Several views automatically create NScrollBar objects to allow users to scroll though the view's contents—this includes view classes like CScroller, CListBox, NScrollText, and CTable.

The second scrollbar class is NWinScrollBar. NWinScrollBar provides scrollbars that are instantiated internally when windows with scrollbars are created. This is done by using the window creation flags WSF_HSCROLL and/or WSF_VSCROLL as attributes during window construction. These scrollbars are part of the window itself, whereas NScrollBarS are not attached to the window and can be resized or moved around inside the window.

See Also: For more information about window creation flags, such as WSF_HSCROLL and WSF_VSCROLL, see the "GUI Elements" chapter in the *XVT Portability Toolkit Guide*.

12.3.12. Resizing and Moving Views with Glue

The CGlue class gives stickiness properties to objects. If an object has an associated CGlue object, then its stickiness properties enable it to stay fixed by a constant distance from the borders of its enclosure. There are glue types for sticking an object to the bottom, top, left, right, bottom-right, top-left, top-right, or bottom-left of a view—or over all of the view.

13

CODING CONVENTIONS AND STYLE GUIDELINES

This chapter presents the coding conventions and language/style guidelines that XVT-Power++ follows. Awareness of these guidelines will help you to use XVT-Power++ more efficiently.

13.1. File Structure

With few exceptions, XVT-Power++ consists of a set of C++ classes. The most basic rule is that there must be one class per file. The name of a file matches the name of the class to which it pertains. Each class has a pair of files, a **.h** (header) and a **.cxx** (source) file. Thus, a class named CApplication would be stored in two files: **CApplication.cxx** and **CApplication.h**. The **.h** file contains the definition of the class and any other information that the class needs in order to be accessed by users of the class. The **.cxx** file is the source file; it contains the actual functions and methods for the class.

Occasionally, a header file will contain more than one class. In such a case, the classes must be very closely related. One is a helper class for the other.

Implementation Note: **.cxx** is the convention used by UNIX platforms. Substitute **.cp** on the Macintosh and **.cpp** on DOS. This manual uses the **.cxx** convention.

13.1.1. Including Files for Usage

When you write a piece of code that uses a certain class, you must include that class's definition. The name of the file containing the definition of the class may vary from platform to platform. For example, on some platforms CRegularPoly is stored in a file named **CRegularPoly.h**. However, for applications running on MS-Windows, this is an invalid filename because DOS restricts filenames to only eight characters with a three-character extension.

XVT-Power++ provides a special structure for including files that allows you to name your files as you desire, without worrying about platform restrictions. This structure is illustrated as follows:

```
#include "PwrDef.h"

#include CRegularPoly_i
#include CCircle_i
#include CScroller_i
#include CRect_i
```

This code uses the CRegularPoly, CCircle, CScroller, and CRect classes and thus needs the definitions of those classes. Each class name with the _i appended to it is a macro that finds the appropriate file containing the class definition. Before you can include the classes, you must include the **XVTPwr.h** file, which defines the macros.

13.2. Naming Conventions

This section describes XVT-Power++ naming conventions. It is not exhaustive, but it does cover the most common cases. Mangling is also discussed.

13.2.1. Classes

Class names begin with a prefix letter, a C for most classes. XVT-Power++ requires capital letters rather than underscores as separators. That is, the prefix and the first letter of each word in the class name are capitalized. For example:

```
CApplication

CRadioGroup
```

Native classes use the prefix N. For example:

```
NButton

NScrollBar
```

The classes that are in the Rogue Wave toolkit use the prefix RW. For example:

RWCString

RWOrdered

The classes that XVT-Power++ derives from other classes in the Rogue Wave toolkit use the suffix RW, while classes that are also collectable, derived from RWCollectable, use the suffix RWC. For example:

CStringRW

CStringRWC

13.2.2. Data Members

Class data members use a lowercase prefix of its, it or is. The latter two typically apply to BOOLEAN data members. For example:

itsData

itIsSelected

isCreateAll = TRUE

13.2.3. Methods

The initial letter of each word in a method name is capitalized, as follows:

Draw

DoDraw

Event methods are handled by the view object itself, while DoEvent methods are both handled and passed down to the rest of the subviews, which, in turn, pass them on down to any subviews they may contain. For example:

Draw
: Draw this view object.

DoDraw
: Draw this view object *and* inform all of its subviews to draw.

Initializer methods use the class name but replace the class prefix with an I, as follows:

IScroller

IIcon

The methods defined in the Rogue Wave libraries begin with a lowercase letter, but the subsequent words in the method name are capitalized. For example:

clearAndDestroy

getString

13.2.4. Class Statics

Static class methods or data members take as a prefix the name of the class to which they belong. Also, constants appear in capitalized letters (all-caps), as shown here:

```
CWireFrame::WIRESIZE // constant static
CWindow::itsNumberOfControls // not constant
CWindow::GetNumberOfControls() // a static method
```

13.2.5. Constants and Defines

The first word of a constant appears in all-caps. Any other words must appear in either all-caps or initial caps. For example:

NULL

NULLIcon

MAXSize

13.2.6. Functions

Functions are not treated differently from methods. That is, the first letter of each word in the function name is capitalized:

Foo

PrintMessage

Within the signature of a method or a function, the parameters have a prefix of the to distinguish them from local variables:

theNewRegion

theData

13.2.7. Variables

Local variables have a prefix of a or an to distinguish them from parameters for methods and functions:

aRect

anEnvironment

XVT Portability Toolkit-related names contain the word XVT, as follows:

GetXVTWindow

itsXVTControl

Table 13.1 presents an overview of the naming conventions used by XVT-Power++:

Type	Rule	Example
XVT-Power++ CLASS NAME:	[PREFIX+[Word]]	CRect
ROGUE WAVE CLASS NAME:	[PREFIX(RW)+[Word]]	RWOrdered
ROGUE WAVE-DERIVED CLASS NAME:	[PREFIX+[Word]+SUFFIX]	CStringRW or CStringRWC
DATA MEMBERS:	[its+[Word]]	itsPoint
STATIC MEMBER:	[class::its[Word]]	CWindow::itsPlatform
METHODS:	[[Word]]	DoSomething
INITIALIZER METHODS	[PREFIX+[Word]]	IScroller
ROGUE WAVE METHODS:	[word+[Word]]	clearAndDestroy
VARIABLE:	[word+[Word]]	charCounter
OBJECT:	[a+[Word]]	aPoint
XVT OBJECT:	[XVT+[Word]]	XVTWindow
PARAMETER:	[the+[Word]]	theItem
#define:	[WORD]	COMPILER
const:	[k+[Word]]	kPi
C++ FILE:	[*PREFIX*+[*Word*]].[h|cxx]	CRect.cxx
COMMANDS:	[WORD]+cmd	UPDATEcmd
ids:	[WORD]+id	UPDATEid

Table 13.1. Naming conventions used by XVT-Power++

13.3. Mangling

Mangling is a useful utility that ensures that the names of XVT-Power++ classes will not clash with the names of any other classes that you define. Assume, for example, that you are also using some other class library that, by coincidence, also contains a class named CStringRW. The compiler and linker must be able to tell whether the term "CStringRW" refers to the XVT-Power++ class of that name or to another party's CStringRW class.

By default, XVT-Power++ mangles the names of all of its classes by giving them a prefix of PWR_. At compile time, XVT-Power++ converts its class named CStringRW to PWR_CStringRW. The XVT-Power++ library does not have a definition for CStringRW; rather, it has a definition for PWR_CStringRW. Thus, you do not usually have to worry about name clashes.

Suppose you are in a file, **foo.cxx**, in which you want to use both the XVT-Power++ CStringRW class and another party's CStringRW class. Immediately following the inclusion of the string header file, you would put in the following line:

```
#undef CStringRW
```

In the rest of the file, you must explicitly put the mangling prefix on the XVT-Power++ CStringRW class. CStringRW will now refer to the other CStringRW class, and PWR_CStringRW will refer to XVT-Power++'s string class.

Example: The following code shows how to avoid classname collisions:

```
// Include PwrFiles:
#include ...
#include CStringRW_i
#include ...

#undef CStringRW

// Include Non PwrFiles
#include ...
#include "SomeOtherCStringRW.h"
#include ...

// Code:

void foo(void)
{
        CStringRW s1 = "This is a non-Pwrstring";
        PWR_CStringRW s2 = "This is a Pwrstring";
}

// Now redefine CStringRW to be a Pwrstring:

#define CStringRW PWR_CStringRW

void goo(void)

{
        CStringRW s1 = "This is a Pwrstring";
        CStringRW s2 = "This too is a Pwrstring";
}
```

13.4. C++ Style Guidelines

This section presents C++ style guidelines that XVT-Power++ users follow. Understanding them will enable you to use XVT-Power++ more efficiently; following them will make your code more compatible with XVT-Power++.

13.4.1. const and enum

Use `const` and `enum` rather than `#define` whenever possible, allowing your programs to take full advantage of C++'s type safety, like this:

```
const float pi = 3.14159;

typedef enum {TRUE = 1, FALSE = 0} BOOLEAN;
```

Make full use of constant methods whenever applicable, as shown here:

```
CRect CWindow::GetFrame(void) const;
```

A caller to GetFrame is assured that this call will not change the state of the object through which it was called.

13.4.2. Inlines

Separate inlines from the actual class definition to avoid cluttering the interface. Most inline code is placed in a separate **include** file so that implementation is not revealed when the interface is the only concern. For example:

File 1:

```
// File CRect.h

class CRect
{
        public:
                ...
                CRect Inflate(int theInflation);
                ...
};

#include "CRectInline.h"
```

File 2:

```
// File CRectInline.h

inline CRect CRect::Inflate(int theInflation)
{
        // ... code ...
}
```

13.4.3. Overloaded Methods

XVT-Power++ strives to overload methods only while preserving semantics. For example, the following two methods take different parameters but have the same outcome:

```
CCircle::Size(int theNewRadius,
        const CPoint& theNewCenter);

CCircle::Size(const CRect& theNewBoundingRegion);
```

13.4.4. Internal Structure of Classes

Classes are always organized as follows:

```
Class CApple
{
        public:
                // Data:
                // Methods:

        protected:
                // Data:
                // Methods:

        private:
                // Data:
                // Methods:
};
```

When you organize your classes as shown here, users of the class who are interested simply in the interface to the class need only look at the top of the file. They do not have to wander through the entire class definition looking at protected and private information that they cannot call.

13.4.5. Function Parameters

Many times programmers calling methods or functions wonder what they can validly pass into them, whether they need to delete what they pass in later, whether they need to worry about their object being modified or not modified, or whether the object they are passing in will actually be copied into another object.

Because all of these questions arise, XVT-Power++ simplifies the number of case situations that can happen for parameters when a function is called. Thus, you should look at the type of the function parameter, and, depending on the type, you can make some assumptions about the semantics of the function call.

XVT-Power++ function parameters can have only *one* of the following four combinations; these four flavors represent a progression in the number of things that can happen, from least to most. Of the four function parameter combinations, the non-constant pointer is the case where the most things can happen, and XVT recommends that you closely read the documentation on any method taking such parameters in the online *XVT-Power++ Reference*.

13.4.5.1. Pass by Value

```
void SetId(int theNewId);
```

Normal pass by value semantics can be assumed. Objects of user-defined types are seldom passed in this way. Instead, the syntax of constant references (discussed below) is used. However, note that from the caller's point of view, the semantics of pass by value and XVT-Power++'s use of constant reference are identical.

13.4.5.2. Constant References

```
void SetSize(const CRect& theNewSize);
```

Whenever a function takes a constant reference, the user can assume pass by value semantics. The reference is used only for efficiency. Thus, after the call, for example, the new size can be deleted without side effects. In addition, calling SetSize twice with two independent yet identical CRect objects has the same result each time. The identity of the object pointed to is irrelevant.

13.4.5.3. Constant Pointers

```
int GetId(const CView* theView);
```

The address of the object is needed, but the object itself will not be modified. The identity of the object pointed to is important to the method. Furthermore, the function guarantees *not* to store the address of the object for future use. After a call to the method, the object might be destroyed or mutated.

13.4.5.4. Non-constant Pointers

```
void SetGlue(CGlue* theNewGlue);
```

The method requires the address of the object. Depending on the method, the object pointed to may be modified or destroyed, or its address might be stored for later usage. It is therefore very important to read the documentation on such a method so that you can use it correctly.

13.4.6. Return Values

The issues here are similar to those for parameters, except that now we are concerned with what is returned. XVT-Power++ returns only the following four values: temporary values, references, constant pointers, and non-constant pointers.

13.4.6.1. Temporary Values

```
int GetId(void);
CRect GetFrame();
```

On the safest side are the temporary values. These are usually used on the spot or are copied into local variables. Note that returning temporary copies to large objects can be inefficient. XVT-Power++ sometimes avoids this by using reference counting. For example, the following function retrieves a paragraph of text and returns a temporary CStringRW object:

```
CStringRW NTextEdit::GetParagraph()
```

Since CStringRW is reference counted, making a copy of the entire paragraph into a local object has very little overhead.

13.4.6.2. References

```
CRect& operator=(const CRect& theRect);
```

The return value can be used as an lvalue. XVT-Power++ guarantees to return references only to the object through which the method is invoked. References to newly allocated objects are never returned. The example shown here makes possible the following assignment:

```
a = b = c;
```

13.4.6.3. Constant Pointers

```
const CEnvironment* GetEnvironment(void);
```

The pointer returned points to an object that must not be modified or deleted. The compiler enforces the fact that the object pointed to cannot be changed. Take care not to cast away the constants. The pointer's "life span" varies, depending on the function.The pointer's life span is the length of time the pointer remains valid. Will the pointer be valid ten function calls from now? That is, will it still be pointing to the same object? To find out how long a pointer will be useful to you, read the documentation on the method in the online *XVT-Power++ Reference*.

13.4.6.4. Non-constant Pointers

In this fourth case, simply a pointer is returned. Read the documentation on the method in the online *XVT-Power++ Reference* to see what you are and are not allowed to do. Usually, you are allowed to modify what the pointer points to, but be careful about deleting the pointer. XVT-Power++ does not return non-constant pointers if it can avoid doing so.

13.4.7. Inherited Methods

You can assume an is-a relationship when one XVT-Power++ class is derived from another. Thus, only public inheritance is used unless otherwise noted in the documentation. Following are two examples of inherited methods:

```
CView::Size(const CRect& theNewRect);
CScrollBar::Size(const CRect& theNewRect);
```

XVT-Power++'s main goal when overriding a method, which should be the goal of all XVT-Power++ users, is to maintain the semantics of a method. The Size method of a view has its own semantics, meaning that the view now has a new size, and that is all. If the documentation says that this method simply sizes and resizes the internal structure of the class and does not redraw it, then the scrollbar Size method should only size the internal structures and not redraw. Different derivations might change the implementation and take care of some extra things the class has to do in order to deal with the method, but they do not change the semantics or do anything that is not expected of the inherited method.

13.4.8. Basic Class Utility Methods

There are some utility methods present in every XVT-Power++ class. Every class is guaranteed to have a constructor and a destructor. Also, an assignment operator is defined for every class. Many classes allocate some memory and have pointers to other objects, so it is important to override the equal operator to ensure that it is safe. The same is true of the copy constructor, which is overridden everywhere for safety purposes.

Many times, usage of the copy constructor is not recommended; sometimes its usage is disabled. Nonetheless, you are guaranteed that your program will not crash because you have used an equal operator or a copy constructor.

An additional zero-argument constructor is provided for as many classes as possible. This constructor allows users to do such things as create an array of class objects. However, many XVT-Power++ classes currently do not have a way to be constructed without any parameters, and thus lack such a constructor.

13.4.9. Templates

XVT-Power++ does not use templates in its current implementation, and it will not use them until they are more portably supported across all XVT-supported platforms. Of course you are free to use templates in your own code provided they do not present portability problems among the platforms you need to support.

14

THE APPL–DOC–VIEW HIERARCHY

This chapter describes:

- the tasks performed at the application level in XVT-Power++'s application framework and also describes the startup and shutdown mechanisms for applications
- the different tasks performed at the document level: accessing data, building windows, managing data, and managing windows
- the possible relationships between the various view objects in the XVT-Power++ application framework

The hierarchy created by the application and its documents and views is an important concept that once understood, will help you use XVT-Power++ effectively.

14.1. Introduction to CApplication

CApplication is an abstract XVT-Power++ class that you must override for each application that you write; it creates and manages the application object. The application object resides at the top-level of the XVT-Power++ application framework, performing several application management functions. It controls the program from start to finish—its direction, its modes, different documents that are open at various times, and so on.

The application object initiates any object the program needs when it starts running and cleans up after the program upon shutdown. It also sets up the global objects and global data that are provided to all objects through CObjectRWC. Moreover, the application class is responsible for creating and managing the application's documents.

14.1.1. Application Startup and Shutdown

The application object for each program is created in the main function, which is located in the **StartUp** source file. The main function creates an application object, giving it the information that it needs upon creation, and then invokes a Go method. The following code shows how to start an XVT-Power++ application:

```
void main(int argc, char* argv[])
{
      CMyApp theApplication;
      theApplication.Go(argc, argv, MY_MENU_BAR_RID,
                  ABOUTBOX, "BaseName", "Application Name",
                  "Task Title");
}
```

The definition of Go is as follows:

```
void CApplication::Go(int argc, char *argv[],
                        short theMenuBarId,
                        short theAboutBoxId,
                        char *theBaseName,
                        char *theApplicationName,
                        char *theTaskTitle)
```

argc and argv are passed literally as they are supplied to main. theMenuBarId is the resource ID number of the application's menubar, theAboutBoxId is the resource ID number of the About box, and theBaseName is the name of the application stored on disk. The string passed to theApplicationName is used as the title of the windows belonging to the application. Finally, the string passed to theTaskTitle is used as the title of the application's task window, if there is a task window.

Once the Go method is called, the XVT-Power++/XVT Portability Toolkit system takes over. Execution is never returned to the main function. When the application starts executing, the XVT-Power++ library hooks up to the main event loop of the XVT Portability Toolkit system.

When both XVT-Power++ and the XVT Portability Toolkit have finished initializing, the application's StartUp method is called. As its name indicates, StartUp handles startup activities for the application. It is a virtual method that you can override, but keep in mind that it must be called by any derived classes if it is overridden. That is, CApplication::StartUp is called before anything else inside of StartUp. After that, you specify what you want your application to do when it starts, such as create certain windows, open specified documents, connect to a database, and so on.

As the application runs, the user opens documents by selecting menu items. At some point, the user will perform an action indicating that he or she is ready to quit the program, perhaps by selecting "Quit" from the File menu. The CApplication::ShutDown method is called to perform application-dependent shutdown tasks, such as closing any files and disconnecting from a database. Normally, ShutDown does not have to perform any actions related to XVT-Power++ objects. For example, if windows are up, they are closed and deleted automatically through XVT-Power++. If you override ShutDown, keep in mind that the first thing it should do is call the inherited ShutDown method. You should perform any necessary clean up of your application within your CApplication-derived class's shutdown.

14.1.2. Tasks Handled at the Application Level

When an XVT-Power++ application starts, it initializes various program defaults, such as enabling or disabling some menu items on the menubar and bringing up a default window such as a splash screen or a dialog box.

One of the responsibilities of the CApplication class is to manage global objects and global data for the application. The objects managed by the application include the following:

- The global environment that is used by any view object, unless it has a specific environment.
- The global objects inside the global class library, CGlobalClassLib, that contain information about the state of the application.
- The desktop, which manages window layout on the screen.
- Some event handler objects that channel all events coming to the application to the appropriate objects, such as windows or views. Some of these event handler objects can be accessed through the application, but most of them can be accessed through the CGlobalClassLib object. Every object that inherits from CObjectRWC (i.e., any object in the application framework), has access to those global objects through CGlobalClassLib pointers.

CApplication contains a number of "Do" methods that are called automatically when different menu actions are selected. For example, when a user selects "Open," "New," "Save," and so on, from the menubar, the application object automatically takes care of these operations, channeling them through the DoOpen,

DoSave, DoNew methods and so on. You can override these "Do" methods to perform such actions as you choose. DoOpen typically creates a document and sends it a DoOpen message. DoNew creates a new document and sends it a DoNew message. DoSave sends a DoSave message to all of the application's documents.

DoClose, another mechanism automatically provided by the application, closes all open documents. Similarly, if at any point the user elects to cancel a certain operation, DoClose ceases this operation. In addition, the application includes a number of document management functions, such as finding open documents, closing all documents, and adding and removing documents from the application.

See Also: For more information, see the description of CApplication in the online *XVT-Power++ Reference*.

14.2. Introduction to CDocument

The *document* is the link between the application level and the view level of the XVT-Power++ application framework. The CApplication object instantiates and manages CDocument objects, which in turn instantiate and manage CView objects for displaying data (see Figure 14.1).

14.2.1. Sharing Data at the Document Level

Each application normally needs to manage data in some form, whether it be stored as a file or as a record in a database. The data can be accessed in different ways, perhaps through the network from a server process. The XVT-Power++ class that is in charge of accessing and managing data is CDocument, an abstract class from which you must derive and instantiate your own specific document classes.

Consider how a document displays data in multiple windows. Suppose there is an application containing a graph and a spreadsheet that share the same data but display it in very different formats. The graph merely displays the information. The spreadsheet not only displays it but also provides the means to edit it directly. In XVT-Power++, the spreadsheet and the graph will have both been created by a CDocument object that they share. This document object manages the data and stores the information for both windows; the document manages the data that is used by the windows.

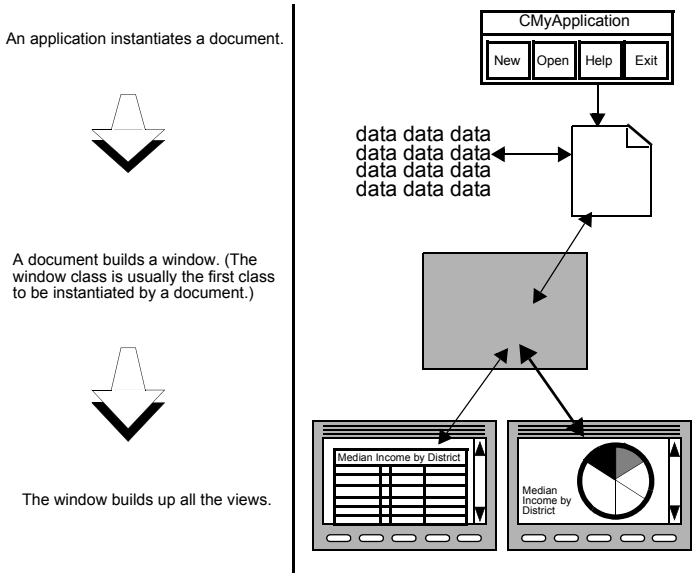


Figure 14.1. Order in which objects are created within the application

Another use of a document is communication. When you make a change inside the spreadsheet window and then click on a button to commit the changes, you also want to notify the graph window to refresh its graph. In this case there are two windows that need to communicate with each other. The need to stay "in sync" becomes even more critical when several windows sharing the same data need to communicate with each other.

Using the CDocument object to communicate changes works well, especially when more than two windows are sharing the same data. They are sharing not only the place where the data is accessed (the document) but also an object that is orchestrating the communication of data changes—all in one place.

14.2.2. Data Propagation

CDocument can create totally different type of views for viewing the same set of data in different display formats. When a document has constructed multiple types of views to display the same set of data, it is in charge of coordinating changes in the data and updating the views to reflect the changes. To accomplish this, you can use either Automatic Data Propagation (ADP) or Transparent Data Integration (TDI).

14.2.2.1. TDI Compared to ADP

TDI is a very powerful feature in XVT-Power++. Using CNotifierS and CTdiValueS, TDI automatically propagates a change of data from TDI-enabled objects to other TDI-enabled objects.

ADP provides a slightly different approach to building complex models where associated views are updated automatically. For example, the spreadsheet and the graph introduced in section 14.2.1 display the same information in two different ways. When the information is modified in one window through the ADP model, that change will automatically be reflected in the other window through a central CModel.

See Also: For more information on ADP, see Chapter 29.
For more information on TDI, see Chapter 30.

14.2.2.2. Sharing of Data

One way to update the graph would be for the spreadsheet window to communicate directly with the graph window through their common CDocument object. Yet another way is to use transparent data integration (TDI) to establish direct channels of communication between various views in the application. There are benefits to both approaches.

In the first case, the spreadsheet informs the document object of the change, which it has to do anyway since the commit button has been pressed and the document object is managing the data. The document then communicates the change to the graph and to any other window that needs to know about the change because it is sharing the data. In the second case, the edit fields in the spreadsheet use TDI messages to send this information, bypassing the central CDocument. In response, the graph window updates its graphical representation to reflect the new data.

14.2.3. DoCommand Chain of Message Propagation

Another way that XVT-Power++ propagates data is through its DoCommand mechanism, known more descriptively as the DoCommand *chain*. A DoCommand naturally flows upward through the event structure of XVT-Power++, starting from a view, perhaps moving through a series of enclosures to the window containing that view, and finally arriving at the document that owns the window.

Typically, if the data inside a view changes, the view generates a DoCommand to the document. In response, the document sets its "needs saving" state to TRUE and may update the data display in several of its windows to reflect the change.

Similarly, if a window associated with a particular document needs to send messages to other windows of that document, it sends a DoCommand to the document and lets the document take care of propagating that message to all of the other windows. Thus, in a sense, the document acts as a message server to all of its client windows. The windows can request the document to do something or pass on messages to other windows by simply generating a DoCommand.

See Also: For more information on DoCommand, refer to *Propagating Messages* on page 12-4.

14.2.4. Tasks Handled at the Document Level

The following sections describe in detail the data access and data management features provided in XVT-Power++'s CDocument class.

14.2.4.1. Accessing Data

XVT-Power++ provides different ways of creating new documents. Typically, a document object is instantiated when the user selects "New" or "Open" from the File menu. Some applications may create several documents and open them by default when the application comes up. In XVT-Power++, to *open* a document means to instantiate a CDocument object, and to *close* a document means to *delete* or *destroy* the CDocument object.

When a document is opened, its first task is to get access to data. The way a document does this partly depends on whether it is opened with or without data. A new document, of course, has no data associated with it at first. By default, to create a new document, you call the CDocument::DoNew method. Part of DoNew's task might be

to create a new document that contains no data but is simply on the screen, ready to display whatever data the user specifies.

It is important that you define DoNew and DoOpen because in many applications documents are created through the "Open" or "New" File menu items. It is also important that you define the appropriate actions to take inside those methods.

For example, the user might create a new spreadsheet that does not yet have any data to manage. In this case, the CDocument object would instantiate a window with an empty spreadsheet. On the other hand, if the CDocument object instantiates a window that displays some initial data, call the CDocument::DoOpen instead of DoNew. You can override DoOpen so that it retrieves data from a place that you specify and then sets up the views to display the data.

When you define the document interface for your application by deriving new document classes from CDocument, you can add parameters to the constructor that tell the document what data it should access, perhaps a pointer to a filename or an indication of which database record to get. Next, initialize the data in the CDocument::DoNew and DoOpen methods. If you call the inherited methods as well to take advantage of their default behavior, another method, named BuildWindow, must also be called.

14.2.4.2. Building Windows

Once you have accessed some data, probably through the CDocument constructor, and have opened it (i.e., have created an empty document), you are ready to build a window to display the data. Every document must define how to build a window. The CDocument::BuildWindow method handles this task. This method is declared as a pure virtual at the CDocument level. BuildWindow notifies the document to build the window to display its data.

Building a window involves instantiating some window objects, initializing them, perhaps creating other objects to go inside the window objects, and so on.

14.2.4.3. Managing Data

The main CDocument management tasks pertain to opening and closing documents, updating/saving their data, and printing their data.

Each document keeps track of the state of its data, that is, whether the data needs to be saved. The *save* state of a document is typically reflected in the File menu, and is a visual cue that reminds the user

to save (or not save) a document's data. Not all documents need this feature. Some documents contain data that is read-only or that never changes.

CDocument has several methods for finding out whether a document's data needs to be saved and for setting the "needs saving" state of the data to TRUE or FALSE. When you are defining your own document classes, you must also define whether the data contained in a certain type of document needs to be saved and, if so, how the data can be saved, either to a file or a database.

14.2.4.4. Default Data Management Mechanisms

XVT-Power++ provides the following default data management mechanisms that you can either choose to use or override in your particular documents:

- When you create a new document by calling DoNew, the default DoNew mechanism simply generates a BuildWindow message to that same document.
- When you open a document by calling DoOpen, the default DoOpen mechanism first brings up a dialog window that prompts the user for the name of a file to open. DoOpen then fills in the specified document's file pointer and calls BuildWindow. The user has the option of cancelling the open operation rather than entering a filename, in which case BuildWindow is not called. If you override the DoOpen method, perhaps to do some extra things, but still use these default mechanisms to find out filenames, then the overridden method should call the inherited CDocument::DoOpen method.
- When you close a document, the default DoClose mechanism goes through all of the document's open windows and notifies them to close. When notified to close, each window checks its document's data to see if the data needs to be saved. If it does, a dialog box appears that prompts the user to save the data, close without saving, or cancel the close operation completely. If the user chooses to save the data, the appropriate save message is sent to the document. If the user chooses to discard the data, the window closes. If the user opts to cancel, then the window notifies the document that it did not close. That is, the window returns a value of FALSE. When this happens, the document stops the closing operation for all of its windows.

- When you save a document by calling DoSave, the default mechanism is for the document to verify whether it has a defined filename to which it should save the data. Each document has an XVT file pointer for storing the filename; it is a protected data member that derived documents can set. If you are not dealing with files in your system, you will need to override this default behavior. If there is a filename for the document, then DoSave simply returns a value of TRUE. If no filename has been defined for the document, then the document calls DoSaveAs. It is up to you to override DoSave and put in the actual code for saving the data and then calling the inherited CDocument::DoSave to find out whether the filename has been defined.
- The default DoSaveAs mechanism is to display a dialog box that prompts the user for a filename to which it should save the data. Then it sets the specified filename and calls DoSave to do the actual saving.
- When you call DoPrint to notify a document that you want to print it, the document goes through all of its views and sends them print messages. In response, the views redraw themselves for the printer rather than for the screen. When the user selects "Print" from the File menu, these actions are performed automatically. You do not have to tie the fact that someone selects "Print" to having a document print. The same is true for page setup. If you are working on a platform that allows you to set up a page before you send it to print, then DoPageSetUp invokes the appropriate application-specific dialog box for setting up the printing page.

Note: Keep in mind that the default mechanisms of CDocument work in conjunction with those of CApplication, discussed in section 14.1.2, which are called automatically when different menu actions are selected. For example, when the user selects "Open," "New," "Save," and so on, from the menubar, the application object automatically takes care of these operations, channeling them through the DoOpen, DoSave, DoNew methods and so on, to CDocument.

14.2.4.5. Managing Windows

As already stated, BuildWindow must be defined by each derived document class in order to specify what should be done to build a window. As more windows are created for the same document, you will want to take advantage of CDocument's window management functions. If you need to find a particular window, CDocument has a FindWindow method that takes the ID number of a window and returns a pointer to that window.

Another method, GetNumWindows, returns the number of windows associated with a given CDocument object. The CloseAll method closes all of a document's windows, regardless of whether their data has been saved—in contrast to the DoClose method, which halts if a window returns FALSE to the closing operation because the user cancelled the operation, data has not been saved, or there is some other impediment.

14.2.4.6. Printing Data

In addition to defining how data can be saved, you must define a mechanism for setting up printing of the data. For each document type, you must define how to set up the printing page.

See Also: For details on how to set up printing, refer to *Application Programming with XVT-Power++* on page 12-1.
Also see the description of CPrintMgr in the online *XVT-Power++ Reference*.

14.3. Introduction to CView

The *view* resides at the third level of the XVT-Power++ application framework, serving as the layer that permits the programmer to display information on the screen. Some views are built up out of other views, as discussed in *Subviews* on page 16-15.

CView and its derived classes compose by far the largest branch of the XVT-Power++ class hierarchy and are the basis for much of the power and usefulness of XVT-Power++.

14.3.1. Views Provide a Graphical User Interface

Together, the various view classes give you access to a model for visual display that is functionally complex, yet easy to use, once you understand the model. Views display textual and graphical data, allow the user to interact with the application, and reflect the state of the application.

Views display a representation of different kinds of data in an application. When a user reads in a file, the application opens a new document that represents the file. A view within a window associated with the document—say, an NScrollText object—displays the contents of the file on the screen.

Views not only display information, such as a message or a drawing, but may allow the user to interact with the application. For example, a user not only sees a button but also can interact with it by clicking the mouse on it. Other views can interact with the user through the keyboard, displaying typed text. Still other views allow the user to make selections from a list of choices.

As the user interacts with views, their differing states may be represented visually. For example, a set of check boxes draws differently once the user has selected one of them.

14.3.2. Tasks Handled at the View Level

All of the graphical objects in a view (or any of its enclosed views) can send and receive events in response to actions that occur within the system. Whenever you move the mouse, click on a button, or release the mouse over a button, a view receives the message and may respond to it. Some views may ignore a certain message while others react to it.

In addition to mouse clicks, a view may respond to a keyboard event, a sizing event, or a move event. As an application runs, views may draw and redraw at different points. For example, when the user clicks on the border of a window to bring it to the front of the window stack, the part of the window that was covered and all of the views it contains (such as a scroller, buttons, and a text box) must redraw themselves. When the user clicks on the scroller, it responds to the scroll event by scrolling upwards/downwards or left/right.

If the user selects a print option, sending a print message to the window and all of its views, they may respond by drawing themselves somewhat differently so they can be drawn on a printer rather than on the screen.

14.3.3. General Characteristics of Views

The primary characteristic shared by all views is that they draw themselves on the screen. Each view must supply its own drawing mechanism, which is done through a method called Draw. This method takes a constant CRect reference that indicates the clipping region for the drawing and performs all the operations necessary for drawing. XVT-Power++ takes care of the clipping automatically, so you usually ignore the clipping region.

The following code shows how the drawing and clipping is handled when a DSC++ application draws a line:

```
void CLine::Draw(const CRect& theClippingRegion)
{
      CShape::Draw(theClippingRegion);
      xvt_dwin_draw_set_pos(GetCWindow()->GetXVTWindow(),
            *itsStartPoint + itsOrigin);
      xvt_dwin_draw_aline(GetCWindow()->GetXVTWindow(),
            *itsEndPoint + itsOrigin,
            itHasStartArrow, itHasEndArrow);
}
```

14.3.3.1. Drawing

If you examine the Draw method for different types of views, you will notice the native XVT Portability Toolkit calls to XVT's functions for drawing such objects as icons, arcs, lines, and so on, as shown in the following example:

```
xvt_dwin_draw_rect(GetCWindow()->GetXVTWindow(),&rct);
...
xvt_dwin_draw_oval(GetCWindow()->GetXVTWindow(),&rct);
...
xvt_dwin_draw_icon(GetCWindow()->GetXVTWindow(),
            HPhysical(itsFrame.Left() + itsOrigin.H()) + itsCenter,
            VPhysical(itsFrame.Top() + itsOrigin.V()),
            itsCurrentRID);
```

A view can actually contain several other views, as is the case with a list box, which is a composite of a scroller, a grid, and several text objects. The outermost view (the list box object itself) may do very little of the drawing and allow the inner views to finish everything else. More specifically, the list box object draws the border area of the list box but does not draw the text items contained within it. There is no DrawText function called inside of the CListBox Draw method. Instead, the text views inside the list box draw the text items.

To summarize, the drawing can either be done by the enclosing view itself, or part of it can be done by the view and the other part by any other view inside it.

14.3.3.2. Showing and Hiding

Related to a view's capacity to draw itself is its ability to show and hide itself. There may be times when you want to tell a view not to draw itself any more by sending it a Hide message as shown here:

```
aView->Hide();
```

When you send a view a Hide message, you are notifying it not to respond to update events from that point on. You are *not* notifying it to change the way the screen looks by immediately becoming invisible. You are just changing its behavior. You will at least have to send a Draw method to the enclosure of the view you are hiding and give it a clipping region big enough to cover the view like this:

```
aView->Hide();
aView->GetEnclosure()->DoDraw(aView->GetFrame());
```

You may wonder why, when you tell a view to hide, you don't just redraw the region so that this view is no longer visible on the screen. The reason is that sometimes you do not want to hide just one view; you may want to hide three or four or one hundred views at a time—without the flashing that would occur if every single view that received a Hide message also got an update event to redraw its part of the screen.

Instead, you notify all views involved to hide themselves (or perhaps change their state in some other way by sending a different message). Once you are done, you send one update event that takes care of redrawing an entire region that is now in a new state. The same applies when you tell a view to show itself when it was previously hidden. You are notifying the view to respond to update events and draw itself *from now on*. (The change does not take effect until the next update event.)

14.3.3.3. Activating and Deactivating

In addition to changing the visibility of views through Show, Hide, and Draw messages, you can notify views to be active or inactive. For example, a window may contain a spreadsheet with a grid full of text fields that can be activated one at a time so that a user can type something into each field. An active view is currently receiving keyboard events (it has focus). This does not mean that other views are disabled, just that they are not active. You can activate another text field simply by clicking on it. Now when you type, all of those events will go to that particular view.

Any view can receive an Activate or Deactivate message. Some views have customized definitions of what it means to be active or inactive. It is up to you to override the Activate message for a particular view that you are designing and specify its behavior when active. You can set a view so that it becomes active when a user selects it for dragging. You may want to deactivate it later.

Windows, like other views, can be active or inactive. A window that is at the front of the window stack and is receiving events is the active window. Any window behind it is not active at the moment.

See Also: For more information about navigating (traversing) through views using solely the keyboard, refer to *Keyboard Navigation* on page 16-18.

14.3.3.4. Enabling and Disabling

Enabling or disabling views is different from making views active or inactive. Disabling a view notifies it not to respond to any events from the user (i.e., keyboard or mouse events). If a user clicks on a disabled view, it will not respond. However, the view will still respond to certain other events. For example, an update event telling a view to redraw itself is not deterred by the fact that the view is disabled. If you want to enable the view again, you must send it an Enable message like this:

```
aView->Enable();
```

A disabled view will not accept an event from the mouse or keyboard, and in addition, it may look different when it is disabled.

Implementation Note: On many platforms, the look of a view when it is disabled is fuzzy or grayed out. Specifically, this is often the case with icons.

14.3.3.5. Dragging and Sizing

Every view has a certain size and location, which can be changed when the user drags it. You can set the dragging or sizing properties of a view to be on or off. Some views act as enclosures for other views and can scroll their contents when the user manipulates a scrollbar. Views automatically take care of any scrolling that has to be done on the screen.

You can also set a view to be automatically selected when the user clicks on it so that the user can drag the view to move it or size it, as is done in many drawing programs. This behavior is achieved through the CWireFrame class, which is a friend class that can respond to sizing and dragging mouse events.

See Also: For more details about scrolling in views, see Chapter 24.
For more details about wire frames, see Chapter 21.

14.3.3.6. Setting the Environment

Another characteristic of views is that they each have access to a helper environment object (CEnvironment) that specifies the colors pen, brush pattern, and fonts to be used to draw that view.

If the environment changes, it signals to the view to change the way it draws itself. A view may have to change its size if the font has changed or it may change the width or height of its borders, depending on the width of the pen.

```
void CText::SetFont(const CFont &theNewFont,
                                    BOOLEAN isUpdate)
{
      CView::SetFont(theFont, isUpdate);
      RecalculateSize();
}
```

See Also: For more information about CEnvironment, refer to *Setting the Environment* on page 15-8.

15

APPLICATION FRAMEWORK

This chapter explains the three levels of XVT-Power++'s application framework in terms of the basic tasks performed at each level, and then it explains how messages and keyboard events are propagated throughout this structure. The chapter also discusses how to define the look-and-feel of your application by setting display properties such as colors and fonts and drawing modes. The chapter concludes by discussing: 1) the role of factories , and 2) how to print from a DSC++ application.

15.1. Levels of the Framework

The purpose of an application framework is to provide a well-defined structure, and thus a common design, for every application developed around it. An application framework allows the developer to reuse a program design. Here, "reuse" goes beyond code reuse in which the developer reuses classes or extends them by deriving from them. If an application framework is generic enough to meet the needs of very different programs, then its entire design and structure is reusable. The XVT-Power++ application framework is designed to be reusable in such a way. This framework is built to accommodate the tasks that most applications with a graphical user interface typically perform. It is structured to deal with these tasks on three different levels: flow of control, accessing and managing data, and displaying data.

15.1.1. Flow of Control

There must be a logical order to the events that occur within the system, whatever they may be. The XVT-Power++ application framework assigns responsibility for the flow of control to its top level, where the CApplication class resides. For each application developed on XVT-Power++, this role is performed by a user-derived object of type CApplication, as discussed in section 14.1. This object takes care of starting the application, initializing it, shutting it down, and cleaning up—in short, the overall top-most logic for execution.

15.1.2. Accessing and Managing Data

The second level of the XVT-Power++ application framework, the CDocument level, is in charge of any data that an application will be using, regardless of its type—whether it is graphical or textual data, and whether it is statically stored or dynamically created during execution and discarded when the program terminates.

15.1.3. Displaying Data

The third and final level of XVT-Power++'s application framework, the CView level, is responsible for the generic task of displaying data. The term "data" is a bit misleading because it refers to all viewable features of an application. Perhaps you do not normally think of a button in a window as having data associated with it. Rather, it has an associated action, as when you click on a button to cancel your selection or Quit from the File menu. XVT-Power++ uses views either to display data, as with a text file, or to interact with the user and then give information to the application, as with the button.

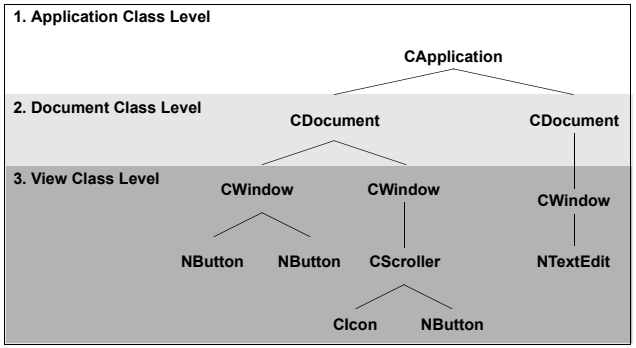


Figure 15.1. Application framework for a typical application

15.2. Propagating Messages

The core of XVT-Power++'s application framework is the ability of its different levels to communicate with each other and to delegate tasks to each other. For example, the user interacts with the application through the view level and sees the application in terms of the views that have been created on the screen. The user can send information to the application by interacting with those views through the mouse or keyboard.

Some of the user input or some of what the application communicates to the user may be information that should be handled at a level other than the view level. There must be a defined communication scheme, event propagation scheme, or delegation scheme to make possible the communication between the user and the appropriate level of the application framework.

XVT-Power++ has three main channels for message passing.

15.2.1. Bidirectional Chaining

In bidirectional chaining, messages (typically XVT Portability Toolkit event messages) go from an event handler to some window to some particular subview. Once it is at that subview, the message may turn around and start propagating back up. Some events go to a selected object and some keep going until they find a particular point or region.

15.2.2. Upward Chaining

In upward chaining, messages start at some subview and then chain upward, stopping at any point. These are the DoCommands. It is important to remember the path the DoCommand travels is based on a supervisor relationship. It follows the rule of looking for its enclosure (the object that is in charge of it).

15.2.3. Downward Chaining

In downward chaining, messages start at some object and then spread downward to all the subviews inside that object. The messages can start anywhere in the system. Examples include DoDraw messages and DoSetEnvironment.

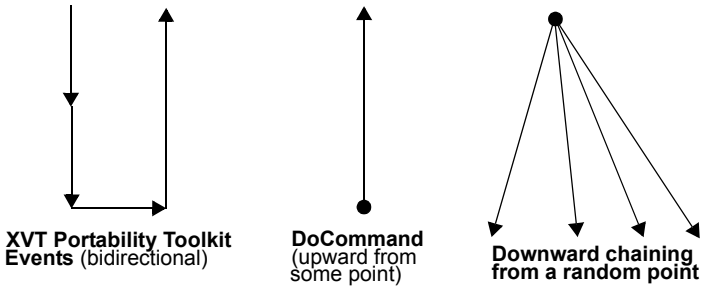


Figure 15.2. Channels of message propagation

15.2.4. The Role of CBoss and CObjectRWC

Tying the entire XVT-Power++ application framework together is a class called CBoss. More specifically, almost every class in the application framework derives from CBoss, which defines the basic messages that any object in the framework is capable of receiving. CObjectRWC provides access to global, shared objects and global, shared data. It defines some messages that can be passed along or delegated through the levels of the application framework.

15.2.4.1. DoCommand Messages

The most often used type of message is the DoCommand. Any object in the XVT-Power++ application framework can receive a DoCommand message. The DoCommand allows you to send any generic message to any object. Two items of information can be passed along with a DoCommand message:

```
virtual void DoCommand(long theCommand, void*
    theData=NULL);
```

The first is a command ID number. This parameter is of type long and can take any number. You are responsible for managing the ID numbers that you give different commands.

XVT-Power++ reserves some predefined, internal XVT-Power++ command ID numbers. As is noted in the section on Global in the online *XVT-Power++ Reference*, the XVT-Power++ ID number base is 20,000. All user-defined ID numbers should be lower than this base. In addition to the ID number, you have the option of passing a pointer to any object you desire along with the DoCommand.

By default, DoCommands are used for mouse events. Views trap the logic of MouseDown, MouseUp, MouseMove, and MouseDouble events to determine whether a mouse click has occurred inside them. If a click

has occurred, a view generates a DoCommand message that is sent upwards through the application framework.

15.2.4.2. ChangeFont Messages

Any object can receive a ChangeFont event when a user selects a change of font from the menubar. This font change can be handled on any level of XVT-Power++'s application framework: 1) at the view level as a particular view changes its font, 2) at the document level as all of the views associated with a document change their font, or 3) at the application level as every view of every document in the application changes its font.

When a user selects a change of font, a ChangeFont message is sent to the window from which the menu selection was made. The window either sends the message to the selected view, if there is one, changes its own font, or propagates the message up to its document. The document may either change its own font or propagate the message up to the application. By default, the messages are propagated upwards until a non-inherited environment is encountered. For example, if a document is not inheriting an environment, it changes its own font. You can, of course,
override ChangeFont to define different logic.

15.2.4.3. DoMenuCommand Messages

Similarly, DoMenuCommand messages can be propagated and delegated from one object to another, from a window to its document and on up to the application if no view object handles it. This type of message is sent when a user selects a generic item from the menubar, perhaps a user-defined item.

15.2.4.4. Unit Messages

You can specify the units of measure that are used by any object in your application— such as inches, centimeters, characters, or a user-defined unit—as an alternative to the XVT-Power++ default pixel units. You can propagate "update unit" messages to notify objects that the units have changed and that they must recalculate measurements based on the units.

15.3. Handling Keyboard Events

By default, keyboard events are handled the same way as menu events. When a keyboard event comes into a window, it goes through several layers of processing, as shown in Figure 15.3. CSwitchboard uses the information in the parameters of the E_CHAR event to construct a CKey object. Then the CKey object finds it way from the switchboard to the view that has keyboard focus.

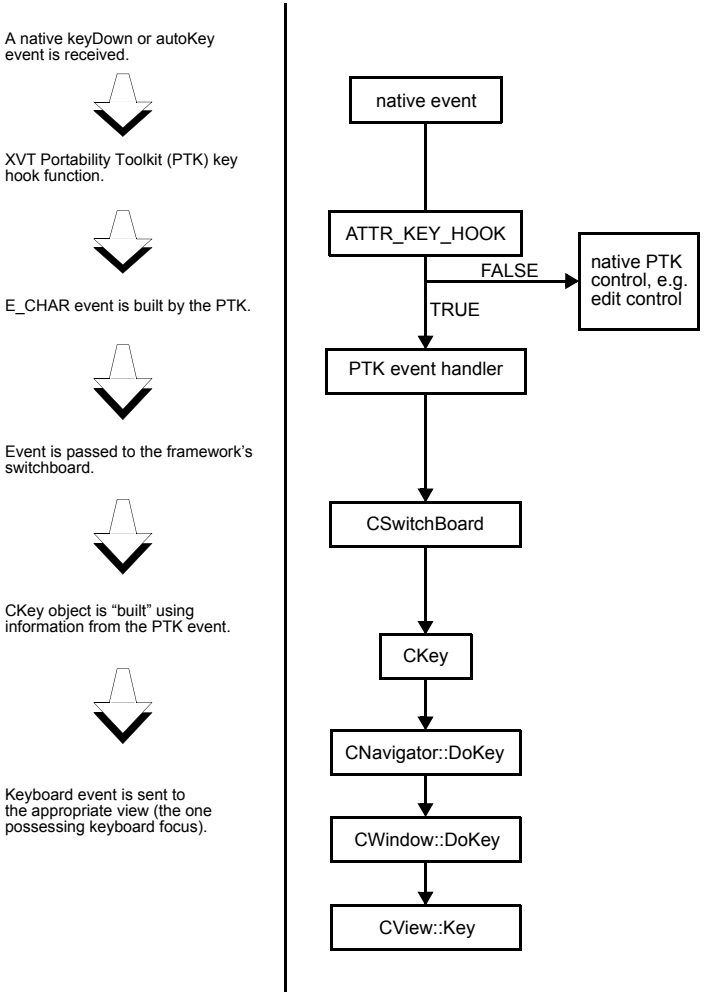


Figure 15.3. Order in which keyboard events are processed within a DSC++ application

The event is passed to the view possessing focus using the DoKey method. If that view elects not to consume the event, it is propagated on up to the document level and perhaps to the application—whichever level can take care of it. A keyboard event goes directly to the application level if no windows are open.

CSubview has a method called SetKeyFocus for setting the view that is to receive the keyboard event. This view then receives the keyboard input, regardless of which view is "on top." Setting the key focus is necessary because, unlike mouse input which uses the mouse cursor to point to a specific screen coordinate, keyboard input does not clearly point to the view to which it is directed.

15.3.1. Keyboard Navigation in Windows

Keyboard navigation is the use of keyboard input, in lieu of mouse pointing and clicking, to interact with GUI objects. Generally, native look-and-feel for keyboard navigation includes using the Tab key and Shift-Tab key (back-tab) to traverse through a list of controls. Groups of controls (such as radio buttons) may be traversed with the arrow keys.

The CNavigatorManager class manages the navigation list of windows for the entire application; it allows you to specify the navigation order for your application's windows. The CNavigator class manages the navigation for a specific window, including all its nested enclosures.

By default, XVT-Power++ uses the navigation classes to provide each window with default support. When a window is created, it automatically creates a keyboard navigator that navigates across all the top-level views in the window. The navigator is created in the virtual CWindow::ConstructNavigator() method. You can override this method if you want to provide a different type of navigator for that window and its views. An alternate approach would be to specify a different type of default navigator by defining a new CWindowFactory object. See*Factories* on page 15-11.

Navigators created internally in CWindow are automatically registered with the global navigator manager. As keyboard events arrive at the application's switchboard, the following actions are taken:

- The navigator for the window is located in the navigator manager.
- The navigator's DoKey() method is invoked to handle the event.

- If the navigator knows how to handle the view, it does so. This normally means that the navigator sets the focus to the next view in the navigation sequence. You can plug in your own navigators to handle keys in a completely different manner.
- If the navigator does not know how to handle the key, the event is sent to the window's DoKey() method for normal processing.

15.4. Setting the Environment

The "look-and-feel" is another aspect of a graphical application that is determined on all three levels of XVT-Power++'s application framework, through XVT-Power++'s CEnvironment class. CEnvironment allows you to give the various windows and views in your interface a consistent look-and-feel while reserving the option to make the look of a view or set of views as distinctive as you like.

Using CEnvironment, you set such display properties of objects as color, font type, pen attributes, brush attributes, drawing mode, and anything else that pertains to displaying an object that can have different attributes on the screen.

15.4.1. Global Environment Object

By default, there is a global environment object that is shared by every displayable object in an XVT-Power++ application. An environment object can be attached at any level in XVT-Power++, all the way from the application object to the deepest subview. An environment propagates downward, so many objects can share the same environment. If you change a property of an environment, then every object that is sharing it is affected. Each object that is using the environment for its own drawing gets an update message notifying it of the change. For example, if the font changes, a text view must resize itself to accommodate the new font.

You must create environment objects when you want specific environments for different documents. One document might have an environment with a blue background while another has a yellow background. If there is a document environment, the document's descendant windows use the document environment as their own. Each window can have its own environment object, with its own colors and fonts. If it does, it uses the environment and passes it down. If it does not, it uses the document environment. If the

document does not have an environment, then it uses the global application environment stored in CApplication (see Figure 15.4).

You can attach a different environment object to each of the windows—or to the windows that you want to have specific environments. You can go still further, allowing some of the views in a window to have their own environment objects. The subviews of those views can also have their own environment objects (see Figure 15.4).

Note: Typically, you do not want to give environment objects to each view because many of the views can share the same environment, eliminating a lot of storage and overhead. Attaching an environment at a higher point allows every object to share it from there on down.

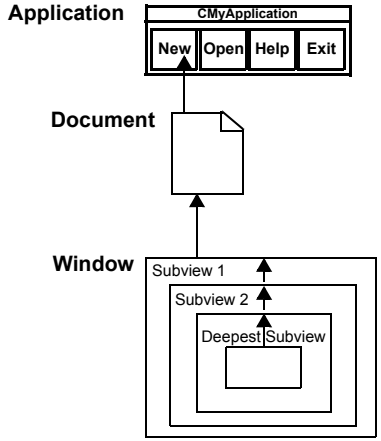


Figure 15.4. The use of environment objects in the XVT-Power++ object hierarchy

15.4.2. Customizing Colors and Fonts in Native Views

XVT-Power++ lets you assign colors and fonts to the native views, or controls, of an application. The interface for doing this is the same as the interface for setting the colors or fonts of any CView:

```
void SetEnvironment( const CEnvironment& theEnv,
        BOOLEAN is Update )
void SetFont( const CFont& theFont, BOOLEAN isUpdate )
```

These CView methods are overridden by CNativeView to supply the specific implementation of font and colors within different types of controls. Each native platform supports different control component colors, but there is much overlap.

See Also: XVT *portably* supports the most significant component colors, even though some component colors are not supported natively on all platforms. For details, refer to Figure 8.5 (the multi-page figure) in the "Controls" chapter of the *XVT Portability Toolkit Guide*.

CEnvironment contains several color attributes specific to native views. These include:

XVT_COLOR_FOREGROUND	Control text and the arrows on scrollbars
XVT_COLOR_BACKGROUND	Fill color of rectangular region occupied by control
XVT_COLOR_BLEND	Secondary background for some controls so they blend into their container window's background without visual indication of a border
XVT_COLOR_HIGHLIGHT	Visual indication that a control has keyboard focus
XVT_COLOR_BORDER	Outside edge of control (rectangular)
XVT_COLOR_TROUGH	Slider area behind scrollbar thumb
XVT_COLOR_SELECT	Indication that a control has been selected
XVT_COLOR_NULL	Value indicating last element of XVT_COLOR_COMPONENT array

You may set as many or as few of these component values as are needed for your application.

Implementation Note: A CEnvironment object can also be initialized to the default runtime color settings for controls in each platform. This is done using the INativeViewColors() method.

15.5. Factories

As part of an object-oriented application framework, many XVT-Power++ classes must instantiate other classes as part of their implementation. There are many reasons why objects are instantiated internally. For example, CView instantiates CGlue objects to delegate its geometry management; CView also instantiates CWireFrame objects to delegate its moving and sizing implementation. Elsewhere in the framework, the CButton class instantiates CPicture and CText objects to display inside the button's border. Examples like this proliferate inside the XVT-Power++ framework.

15.5.1. Abstract Factories

You may occasionally want to customize the behavior of a class in the framework by changing the types of objects it creates. For example, you may want to create your own type of CGlue object that provides specialized geometry management. One way of making this change is to derive your own set of view classes and override the code which instantiates CGlue. Such an approach is not always convenient. It would be better to simply reuse the existing framework classes and have them somehow be aware that there is a new CGlue class that they should use.

The XVT-Power++ framework gives you this extra flexibility by utilizing a design pattern known as the *abstract factory* pattern. Abstract factories provide an interface for creating families of related or dependent objects without specifying their concrete class. Basically, the framework classes have been set up to delegate the instantiation of other classes. The job is delegated to a set of factory classes which can be overridden to create your own custom objects instead.

15.5.2. Framework Factory Manager

A special factory manager object is created automatically by the framework. This manager acts as a central place where all other abstract factories are registered. The CApplication::InstallFactories() method automatically installs all the default factories used by the framework. You can install your own factories or override the default factories used by the framework by providing your own InstallFactories() method inside your application derived class.

Example: For example, the framework contains an abstract factory named CViewFactory. This factory is used by CView classes to instantiate any objects they use. The CViewFactory instantiates classes like CGlue.

t To override the CGlue generation used by all CView objects:

1. Define a concrete subclass of CViewFactory and override the virtual methods that create the objects that you have customized:

```
class CMyViewFactory : public CViewFactory
        {
        public:
                virtual CGlue* ConstructCGlue( CView*
                                        theView );
                { return new CMyGlue( theView ); }
                CMyViewFactory() : CViewFactory()
        {}
};
```

2. Install an instance of this factory into the factory manager as soon as the manager is created within CApplication::InstallFactories():

```
void CMyApplication::InstallFactories( CFactoryMgr*
                                mgr )
        {
                CApplication::InstallFactories( mgr );
                itsMyViewFactory = new CMyViewFactory;
                        mgr->AddFactory( kViewFactoryTypeId,
                        itsMyViewFactory );
        }
```

3. Delete the factory after it is no longer used, because the factory manager will not delete it for you:

```
void CMyApplication::ShutDown()
        {
                CApplication::ShutDown();
                delete itsMyViewFactory;
        }
```

15.5.3. Framework Factories

XVT-Power++ defines several factories that are automatically registered into the application's factory manager. These factories are used internally to create a variety of objects within the framework.

This section lists the factories and what objects they create. If you would like to change the type of certain objects created within the framework, all you need to do is install a derived version of the appropriate factory.

CApplicationFactory, CApplicationFactoryDefault

Global CTaskDoc object

Global CTaskWin object

Global CPrintMgr object

Global CDesktop object

Global CResourceMgr object

Global CControllerMgr object

CMenuFactory, CMenuFactoryDefault

CSubmenu objects created by CWindow (or code in your application)

CMenuItem objects created by code your application

CMenuBar objects created by code in your application

CPlatformFactory, CPlatformFactoryDefault, CWin32PlatformFactory, CMacPlatformFactory

CControlDelegate objects created for movable and sizable native views

CAttachmentWindow objects created by floating CAttachment palettes

CResourceFactory, CResourceFactoryDefault

NButton objects created from URL resources
CRadioGroup objects created from URL resources
NCheckBox objects created from URL resources
NScrollBar objects created from URL resources
NEditControl objects created from URL resources
NText objects created from URL resources
NListBox objects created from URL resources
NIcon objects created from URL resources
NListButton objects created from URL resources
NListEdit objects created from URL resources
NGroupBox objects created from URL resources
NScrollText objects created from URL resources
NRadioButton objects created from URL resources
CResourceWindow objects created from URL resources
CResourceMenu objects created from URL resources

CValidatorFactory, CValidatorFactoryDefault

CValidator objects created by users of the framework

CViewFactory, CViewFactoryDefault

CText objects created by views in the framework
CPicture objects created by views in the framework
CFixedGrid objects created by views in the framework
CGlue objects created by views in the framework
CWireFrame objects created by views in the framework

CWindowFactory, CWindowFactoryDefault

CMouseManager objects created by CWindow
CNavigator objects created by CWindow
NWinScrollBar objects created by CWindow

15.6. Printing

XVT-Power++'s interface to the XVT Portability Toolkit's printing facilities is called CPrintMgr. This class is in charge of queuing up data and printing it. At any point, you can notify any object in XVT-Power++'s application framework to print and it will place itself into the print queue and print to the designated printer. For example, when you want to print a view, you call DoPrint inside the view.

The actual implementation of printing is handled inside CPrintMgr. Upon receiving a print command, the following occurs:

- A window prints all views inside of it
- A document sends every window associated with it to the printer, which prints each window on a separate page
- The application notifies all of its documents to print all of their windows, each on a separate page

By default, whatever is drawn on the screen is drawn on the printer paper. However, you may want a given object—say, a window—that draws a certain way on the screen to draw differently on the printer, perhaps using a different font. In this case, you can override the CView::PrintDraw method for a view to define how you want it to print.

PrintDraw is a method that is in charge of doing any drawing that should be sent to the printer. This drawing is just like the drawing that is done on the screen using the XVT Portability Toolkit xvt_dwin_draw_* functions.

If you were to write your own printer Draw method, it would look just as if you were printing from the screen. You would still print using the view's XVT Portability Toolkit window, which can be obtained through the CWindow::GetXVTWindow method. GetXVTWindow can return either a regular screen window or a printed window, depending on whether printing is being done. PrintDraw just calls the regular Draw method, and in most cases this is adequate.

16

MANIPULATING VIEWS AND SUBVIEWS

Views display a representation of different kinds of data in an application. Views display textual and graphical data, allow the user to interact with the application, and reflect the state of the application. This chapter considers the possible relationships between the various view objects in the XVT-Power++ system. The chapter also discusses coordinate systems, subviews, and keyboard navigation.

16.1. Enclosures and Nested Views

As discussed in section 14.3, many views are built up by putting several views together and inserting them into a larger view that is capable of containing different kinds of smaller views. This composition is made possible by a relationship between an *enclosure* and a *nested* view:

Enclosure
Contains one or more views.

Nested view
Views inside of the enclosure.

Note that every view must have an enclosure. A text view may be nested inside a list box, which in turn is nested inside a window. In other words, the window is the enclosure of the list box, and the list box is the enclosure of the text view.

Note: Windows are a special case because a window cannot be enclosed inside another view. However, they still have a logical enclosure, which is the screen, or, on some platforms such as MS-Windows and OS/2, a “task window.” The enclosure of a window is something you can control only at the platform-specific (PTK) level.

16.1.1. Similarity Between Enclosures and Owners

The concept of view ownership is discussed in section 16.2 on page 16-6. It is worth noting that while the distinction between enclosures and owners is real and useful, there is an important similarity between them that contributes greatly to XVT-Power++'s ease of use.

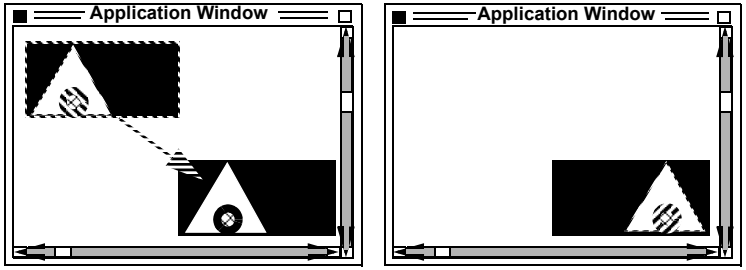
The relationship between an enclosure and the views nested within it resembles an owner/helper relationship. When an owner view is destroyed and deleted, so are all of its helper objects. Similarly, when an enclosure is destroyed and deleted, so are any views nested inside it.

Moreover, if the views contained within the window also contain nested views of their own, these views will also conveniently close. This means that you have to take care of very little memory management. You can simply create views as desired, knowing that if at a certain point you close a window, the window will take care of closing anything enclosed inside it.

16.1.2. Clipping

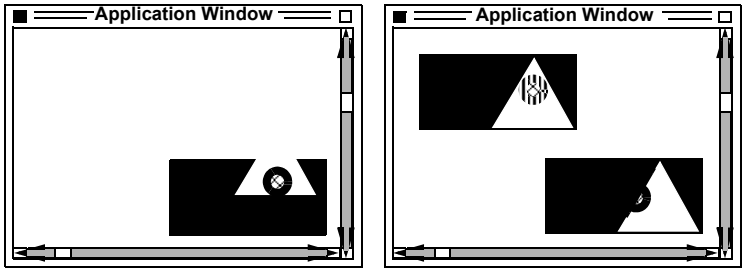
Enclosures ensure that every view contained inside them is clipped to them. *Clipping* means that a view contained inside an enclosure cannot draw itself beyond the boundaries of its enclosure. Thus, a nested view may be only partially visible.

For example, a text object or a picture contained within a window clips to the window's border so that any text or part of the picture extending beyond the window's border is not visible and will need to be scrolled or otherwise moved in order to become visible within the window's borders. Similarly, when you set different enclosures inside a window, whatever is nested is also clipped to the border of its enclosure. The nesting behavior of views, especially with regard to clipping, is shown in Figure 16.1.



The rectangle contains a triangle that in turn contains a circle. When the rectangle is moved, all of the objects nested within it move with it.

When the user clicks on the triangle to move it, the triangle can move freely inside the rectangle object.



Each shape object is clipped to its enclosure so that if a part of an enclosed object extends beyond the clipping region, that part is obscured—as the top of the triangle is here.

The circle can move freely inside the enclosing triangle and becomes partially obscured as it extends beyond its clipping region.

Figure 16.1. Nesting behavior of views; objects inside nested views may be clipped

16.1.3. Defining a View's Enclosure

When a view other than a window is created, XVT-Power++ needs to know what enclosure to give that view. Thus, the constructors of most views have a parameter that passes a pointer to a view that is acting as the enclosure, as shown here:

```
CScroller* aScroller = new CScroller(someWindow,
      CRect(0,0,100,100));
CIcon* anIcon = new CIcon(aScroller, CRect(10,10,
      42,42));
```

All views nested inside an enclosure are drawn relative to the origin of that enclosure. In effect, each enclosure sets up its own coordinate system, and the views it contains draw within that coordinate system. If the entire enclosure is moved to a new location, the views nested in the enclosure are oblivious to the move, continuing to draw in coordinates that are relative to enclosure. In other words, the enclosure defines space for a nested view, and the larger context of the screen is irrelevant.

16.1.4. Limitations on the Hierarchy of Enclosures

While some types of nested views can act as enclosures, it is not true that *any* view can enclose another view. The top-level view class is CView, which defines all the properties of views that have been discussed so far. From CView a class called CSubview has been derived. Only views that inherit from CSubview can act as enclosures; views that inherit directly from CView *cannot* act as enclosures. In Figure 14.2, the views inheriting directly from CView are shown in the gray area.

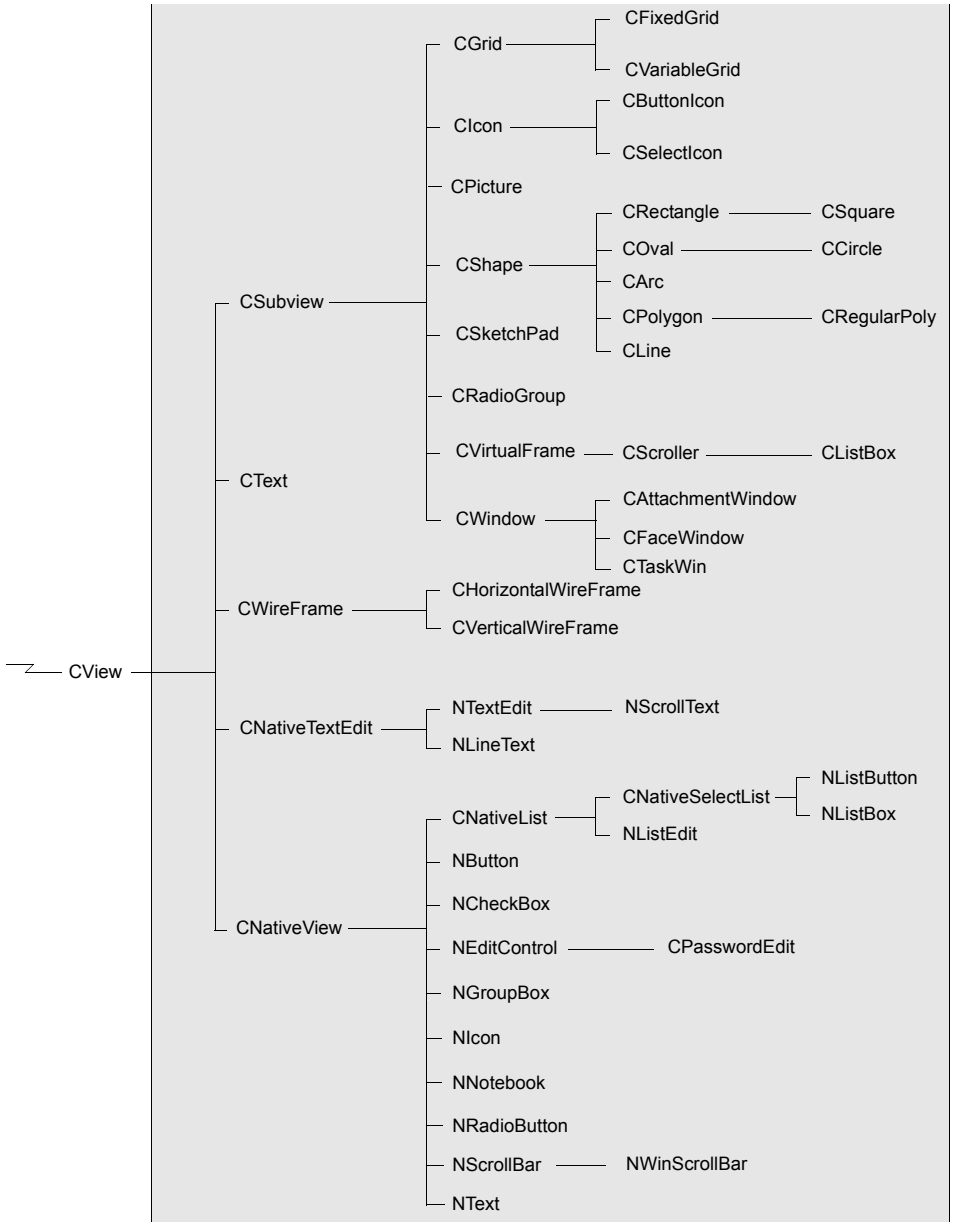


Figure 16.2. XVT-Power++ view hierarchy

As Figure 16.2 shows, a CText object is a type of object that does not inherit from CSubview. CText is a view that allows you to display static text on the screen, for example, the phrase “Enter Password:” on a login window. Obviously, this text object does not need to function as the enclosure of something else.

Another glance at the XVT-Power++ view hierarchy tree reveals other CView classes that cannot act as enclosures, such as CNativeView. Native views are objects that are drawn by the native windowing system, not by XVT-Power++ and not by the user. Since XVT-Power++ does not draw them, XVT-Power++ cannot draw anything else inside them, either. While they can be displayed inside other views, nothing else can be displayed inside them.

On the other hand, CGrid is a type of view that inherits through CSubview and therefore can act as an enclosure. You can nest many different kinds of views inside of a grid, inserting the items in its rows and columns.

16.2. Owners and Helpers

Another relationship between view objects that needs to be considered is *ownership*. Different views can be owners of other views that act as helper or auxiliary objects and provide a service to the owner view. If a view object that owns a helper object is closed, the owned object is also closed. If a view is destroyed, any objects that it owns are also destroyed.

16.2.1. CGlue

One example of a helper object is a CGlue object, which provides the “stickiness properties” of its owner. *Stickiness* refers to the behavior of a view when its enclosing view is sized. Depending on its type of stickiness, a nested view will stretch with its enclosure or stay fixed by a constant distance from the borders of its enclosure.

Suppose you want a rectangle to be “stuck” to all four sides of its enclosure, which is a window. You would give it a glue type of ALLSTICKY, as follows:

```
aRectangle->SetGlue(ALLSTICKY);
```

When the window stretches during resizing, the rectangle also stretches. When the window shrinks, the rectangle does, too. If you had specified a “bottom right” type of stickiness, only the bottom and right sides of the rectangle would be stuck to the window. In this

case, if the window were resized, the rectangle would move along with the right bottom corner of its enclosing window.

Instead of embedding all the code inside CView that tells it how to act as a "sticky" object, XVT-Power++ uses a separate CGlue object to take care of the logic of stickiness. When you specify the stickiness of a view, a CGlue object is automatically created, and it knows that its owner is the particular view that you have made sticky. Thus, there is an "owner" relationship in which the view owns the glue.

See Also: For a list of all the types of stickiness that can be set for a view, refer to CGlue in the online *XVT-Power++ Reference*.
To see an example showing how to override the CGlue generation used by all CView objects, refer to section 15.5.2 on page 15-11.

16.2.2. CEnvironment

Another helper class is CEnvironment, which provides an object that keeps track of its owner's colors, pen, brush, fonts, drawing mode, and so on. You set the environment of a view as shown in the following example:

```
CEnvironment anEnv(
        COLOR_WHITE,    // Background
        COLOR_BLACK,    // Foregound
        COLOR_WHITE,    // Brush color
        PAT_SOLID,      // Brush pattern
        COLOR_BLACK,    // Pen color
        PAT_SOLID,      // Pen pattern
        1,              // Pen width
        STDFont,        // Font
        M_COPY,         // Drawing mode
        P_SOLID );      // Pen style

aView->SetEnvironment(anEnv);
```

When you set the environment of a view, that view creates its own environment object to store this data. When the owner view is destroyed, its internal environment object is also destroyed.

An environment object can be shared by many views. For the sake of a consistent look-and-feel, an application typically has several windows and other views that use the same environment. Thus, borders are drawn in the same color, the brush patterns and colors in the view interiors are the same, the fonts used in textual views are consistent, and so on.

Although one particular view acts as the owner of an environment object that is destroyed when it is destroyed, other view objects may also be using that environment while it exists. This is possible because views can only share an environment with an object that

is above them in the object hierarchy; that is, with an enclosure. The types of sharing relationships that can be established between owners of environment objects is discussed in section 15.4 on page 15-8.

16.2.3. CWireFrame

CWireFrame objects are helper objects that can be owned by other views. A wire frame is an object that enables a view to be moved and sized. On the screen, a wire frame appears as a rubberband (flexible, dynamically changing) frame surrounding an object that is being sized or dragged. You set the sizing and dragging of an object as follows:

```
aView->SetDragging(TRUE);
aView->SetSizing(TRUE);
```

When you thus specify that a view object is to be sizeable and draggable, that view creates its own wire frame object. At the appropriate times, the wire frame can take care of different events that it may receive, such as mouse down events and mouse dragging events. The wire frame also can notify its owner of the new location to which it is being moved or of the new size to which it is being stretched.

16.2.4. CPoint and CRect

Each view has its own CPoint and CRect objects that enable it to keep track of where it is on the screen; that is, of the region where it is located, the origin from which it is drawing, and so on. When the owning view is destroyed, its CPoint or CRect object is destroyed as well. These two heavily used helper classes are discussed in detail in section 16.3.

16.2.5. CDrawingContext

CDrawingContext supports queue invalidation for high performance drawing.

By deferring updates during complex rearrangements of the user interface, such as geometry changes, you allow the native system the chance to combine updates into a single, more efficient, update event. QueueInvalidate() adds a rectangle to the update queue, maintained by itsInvalidateQueue. FlushInvalidate() sends the queue invalidated rectangles to the native system for updating.

CDrawingContext flushes queued updates using one of two algorithms:

Region updates
This algorithm combines all the the invalidated rectangles into one large enclosing rectangle and passes this region to the native system for updating.

Pass-through updates
This algorithm simply passes each invalidated rectangle to the native system for updating.

Implementation Note: The update method that works best depends on the native operating system.

16.3. The Coordinate System

Coordinate systems are crucial to the use of views because the screen location of a view must be specified before it can be drawn. Understanding the coordinate system will help you use XVT-Power++. You must learn about three types of coordinates:

- screen-relative coordinates
- global (window-relative) coordinates
- local (view-relative) coordinates

You also need to understand when these coordinates are applied in order to use them correctly and manage them efficiently. The basis for managing the coordinate system is provided by XVT-Power++'s CRect and CPoint classes.

16.3.1. CRect

CRect is simply a class that provides a data structure for storing information about a rectangular region of the computer screen. CRect manages four different values: left, top, right, and bottom. Together, these four values form a rectangle that is located somewhere in space. Each of the four values represents a value in the coordinate system. The CRect object itself has no information about what coordinate system it is mapping. It merely contains the mapping.

CRect's mapping allows you not only to set different rectangular regions but also to perform many useful operations, such as computing the union of some rectangular regions.

A union operation adds one region to another, which has the effect of resizing the region. Suppose you want to update the entire region that encompasses two overlapping views. You can take the union of

one view's rectangular region with the other view's rectangular region; the result is a much larger region that contains both of the regions. The union of two views in shown in Figure 16.3.

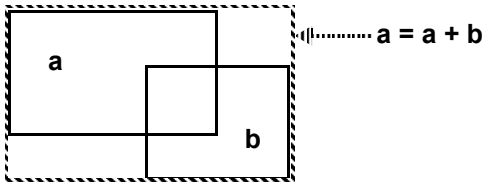


Figure 16.3. The union of two views

Similarly, you can compute the intersection of two views. For example, if a view is partially covered by another view or is being clipped by an enclosure, you can take the intersection of those two views through the CRect to find the shared rectangular region. The intersection of two views in shown in Figure 16.4.

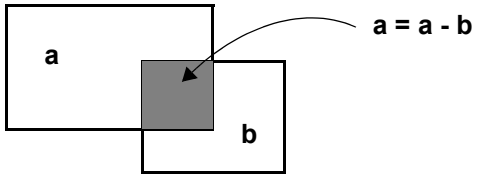


Figure 16.4. The intersection of two views

In addition to union and intersection, you can perform several other operations on a region. You can expand it or shrink it, and you can translate it from one position on the screen to another. All of these operations for managing regions on the screen are available through the CRect class.

See Also: For more details, see the description of CRect in the online *XVT-Power++ Reference*.

16.3.2. CPoint

CPoint is similar to CRect in that it also manages positions within XVT-Power++'s coordinate system. However, CPoint manages a value for a single point or single unit. Instead of an entire rectangular region, CPoint consists of a horizontal and a vertical value for one position in space. It is irrelevant to CPoint where the coordinate system is (that is, whether it originates at the top-left of the screen) or what the sizes of the units are (whether in pixels, characters, inches, or some other measure).

CPoint allows you to move from one point to another, set either the horizontal or vertical coordinate of a point, add points together, subtract one point from another, or make two points equal. Moreover, it includes some methods for coordinate system conversion so that you can translate the coordinates of a point, convert a CPoint's coordinates from view-relative to window-relative coordinates, and vice versa.

See Also: For details, see CPoint in the online *XVT-Power++ Reference*.

16.3.3. The Point of Origin

Central to any coordinate is the point of origin from which the coordinate is calculated. That is, before a coordinate such as *3,9* makes sense, you need to know the context for the numbers — whether the point is screen-relative, window-relative, or view-relative. In XVT-Power++, the need to know the context of a coordinate is complicated by the fact that each enclosure defines its own coordinate system. Therefore, the context is crucial.

Each XVT-Power++ view uses a point of origin to calculate its position on the screen. This origin is the *0,0* point that normally represents the position of the enclosure's top-left corner. Every view knows where its enclosure's top-left corner is because it has a CPoint object it uses called itsOrigin. The origin is automatically managed and updated in XVT-Power++. If the enclosure changes from one position to another, the origin of any nested views is updated automatically and internally.

Note: XVT-Power++ has some methods for manipulating a view's origin, but these are reserved for advanced XVT-Power++ programming. Normally, there is no reason for you to concern yourself with a view's origin.

Suppose your application contains a scroller in which a rectangle is nested. Typically, the rectangle draws at a certain position relative

to the scroller's top-left corner. However, when a user clicks on the scrollbar, the rectangle draws at a different position because the entire contents of the scroller have been scrolled to the left or right, up or down, depending on the scrollbar's orientation and the direction of the click. Now the origin for the rectangle becomes the top-left corner of the scroller, plus or minus some scrolling coefficient. It is usually unnecessary for you to know these specifics of how the point of origin changes with a scroller. It is mentioned here to illustrate the importance of origins and how they are used to keep track of where views should draw relative to an enclosure, especially if the enclosure happens to be in a particular state, such as a scrolled state.

16.3.3.1. Screen-relative Coordinates

Windows are the outermost enclosures. All windows are placed relative to the screen's top-left corner and are the only screen-relative views in XVT-Power++.

16.3.3.2. Window-relative (Global) Coordinates

In many places in XVT-Power++, you will encounter the term "global coordinates." This term is synonymous with "window-relative coordinates"—"global coordinates," as it is used here, refers to the window, not to the entire screen.

Several types of methods take window-relative coordinates. For example, the "Do-" methods, such as DoMouse and DoDraw, all take global (window-relative) coordinates, as do all the drawing methods for any region these methods receive. More precisely, they take a region or a certain point where the mouse is clicked, which is relative to the entire window. Other methods, such as the so-called narrow mouse methods (the ones without the "Do-" prefix), take view-relative coordinates. Thus, when a view gets a MouseDown message and the mouse is at point *5,5,* this point is relative to the view's top-left corner. If you are calling the narrow methods directly, it is imperative to look at the method and see what kind of coordinates it takes.

Suppose that you want to insert a view, such as a rectangle, into a window. You place the top-left corner of the rectangle at coordinate *10,10* and the bottom right corner at *35,40*. These coordinates are relative to the window's top-left corner because the window, as an enclosure, sets up its own coordinate system, and anything that is inserted directly inside of it is positioned relative to its top-left corner.

16.3.3.3. View-relative (Local) Coordinates

Suppose you decide to place a view inside the rectangle, say, another rectangle, as shown in Figure 16.5. The rectangle is inserted at position *5,15* and extends to *15,25*. These coordinates, as you may have guessed, are relative to its enclosures's top-left corner and are thus view-relative coordinates.

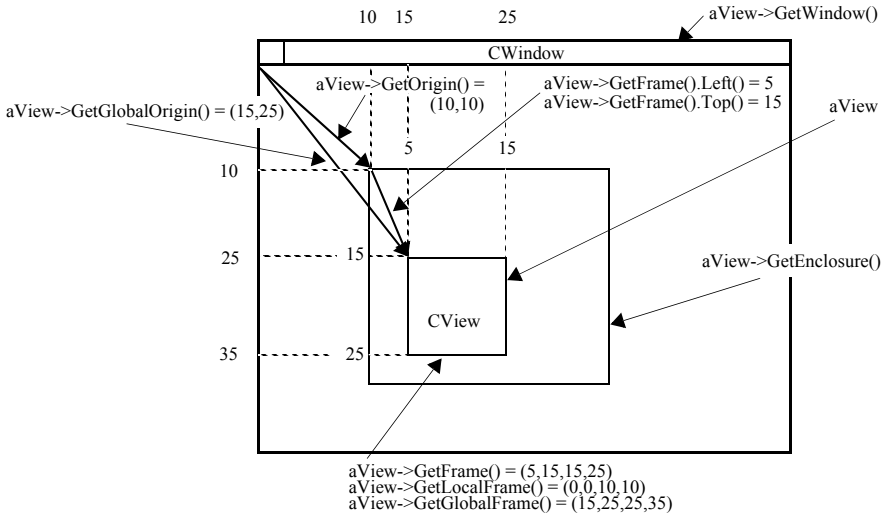


Figure 16.5. View-relative coordinates for nested views

16.3.4. Units of Measure

Another critical issue for coordinate systems besides the point of origin is the size of the units—whether they are pixels, inches, centimeters, characters, or a user-defined unit. If the units are pixels, each unit maps one-to-one with pixels on the screen. If the units are pixels and a CLine object, for example, extends from point 0 to point 10, then ten pixels on the screen are turned a certain color to represent the line.

By default, XVT-Power++ uses pixel coordinates. However, to maximize the portability of an application, as well as the ease of programming a graphical user interface, you often may not want to use pixels because they provide device-dependent coordinates. An application that looks beautiful running on one platform may not look right any more when it runs on a different machine with a smaller screen or completely different resolution. When you

combine text and graphics, you typically position the graphics and shape/scale them in terms of the text. If the application runs on a different machine where the font size is totally different, suddenly the graphics will be "off."

The solution to this problem is to use logical coordinates rather than physical or device-dependent coordinates when you program a graphical interface. Through the CUnits class, XVT-Power++ allows you to use a variety of logical coordinate systems.

CUnits allows you to program in coordinates that map to inches, centimeters, a user-defined unit that maps logical coordinates to some physical representation on the screen, or character units in a font that you choose (normally the system font). You can specify, for example, that you want a two-inch by three-inch rectangle or an object that is ten characters long and three characters wide.

16.3.5. Translating Coordinates

XVT-Power++ provides facilities for translating from one coordinate system to another. Suppose you have defined a view that inherits from CSubview and therefore can act as an enclosure. This view contains several nested views. Moreover, it is defined to trap and receive any mouse events that occur within its region, regardless of whether an event occurs within the region of one of its nested views. Now suppose that this enclosure gets a MouseDown message, which, in XVT-Power++, has a CPoint coordinate that is relative to the view receiving the message. This point indicates where the mouse happened to be, in this case, at point *5,5*. The view sends the message down to another view that is nested within it at this location by calling the nested view's MouseDown method, which takes as its first parameter a CPoint coordinate that is relative to the nested view's coordinate system.

The point that the enclosure received is in coordinates relative to the enclosure itself. Thus, the enclosure must translate the point into coordinates that the enclosed view can use. This is a case where we want to translate a point from one view's coordinate system (the enclosure) to another view's coordinate system (the nested view). The CRect and CPoint classes provide several easy-to-use utility methods that do the translation for you. There are methods for localizing and globalizing different points. In the case considered here, you simply call the CPoint::Translate method, which takes two parameters: a view from which to translate the coordinate and a view to which to translate the coordinate.

Call Translate and pass it the enclosure and the nested view, as follows:

```
aPoint.Translate(theEnclsure, theNestedView);
```

16.4. Subviews

All XVT-Power++ views can, and in fact must, be nested inside an enclosing view, with the exception of the window, whose logical enclosure is the screen or the task window. However, not all views can act as enclosures; the subset of views that can act as enclosures is clearly defined by a class called CSubview. This section discusses in detail all of the views that can act as enclosures and the implications of this property within the larger context of XVT-Power++.

16.4.1. Nesting Behavior

"Nesting" means that an enclosure can contain numerous views that are nested at the same level, perhaps overlapping, as well as views that act as enclosures for nested views of their own. Figure 16.1 on page 16-3 shows the behavior of nested views.

16.4.1.1. Overlapping Views

Suppose there is a window in which you have inserted a circle object. You then place a square directly on top of the circle. In this case, the square is not nested inside the circle. Both shapes are nested inside of their common enclosure, a window, but they are overlapping—or, more precisely, stacked. In a stack of views, the last view to be created and inserted into the window is the top view at that location. In this case, the square is on top of the circle. If you click the mouse on the area shared by both the square and the circle, the square will receive the event because it is covering the circle, and it is as if the circle is not even there.

XVT-Power++ provides several ways to control which view is on top of a stack of views. One way is through the order in which the views are created since the last view created is the one on top. Also, CSubview has two methods, PlaceTopSubview and PlaceBottomSubview, for placing a certain view at the top or bottom of a stack. In the example discussed here, the square lays directly on top of the circle and is the top subview of the window, while the circle is the bottom subview. If you decide that you want to reverse this order by placing the square beneath the circle, call the window's PlaceBottomSubview method, which it inherits from CSubview, and give it the square as a

parameter. The square then becomes the bottom subview in the window. PlaceTopSubview, as you can imagine, works very much the same way.

16.4.1.2. Obtaining Information About Nested Views

The interface of CSubview allows you to find out several different things about a given enclosure's nested views. You can find a view that is nested inside the enclosure, either by using a view ID or by specifying a location in coordinates relative to the enclosure and getting the top-level view that contains this CPoint. You can also get a list of every view that shares this certain point. These operations are made possible by the CSubview FindSubview and FindSubviews methods.

16.4.2. Routing Events to a Specific Subview

There are times when it is desirable to circumvent FindEventTarget and always send the event to a particular view. This is especially true when you are dragging or sizing a view but want it to receive all mouse events. In this case, you call the CSubview::SetSelectedView method on a given enclosure so that the enclosure will send all events to the specified view. If later you decide to take away the view's status as the selected view, you can call SetSelectedView again and give it a value of NULL. Then the enclosure will revert back to the default behavior of searching for a nested view via FindEventTarget when it receives a mouse event.

16.4.3. Propagating Messages from Enclosures to Nested Views

XVT-Power++ has many different types of messages that are propagated from an enclosure to all of its nested views, and then from each of these nested views (which may also be enclosures) to their respective nested views, and so on in a recursive fashion all the way down to the deepest views.

Suppose you have inserted into a window a rectangle that contains many different kinds of objects. Several of the objects enclosed inside the rectangle contain other objects inside of them. Now you want to send an update message that will reach all of these objects, from the rectangle enclosure to the tiniest and deepest enclosed view. To do this, you send a DoDraw message which propagates recursively as described here.

Typically, the messages sent to CSubview objects—such as drawing, showing/hiding, activating/deactivating, enabling/disabling, dragging, and sizing—consist of two methods: the base method and the "Do-" version of that method (i.e, Draw/DoDraw, Show/DoShow, or Activate/DoActivate).

The "Do-" methods are in charge of the propagation scheme. Thus, when a method such as DoDraw is called on an enclosure, the enclosure uses its own Draw method to draw itself and then propagates the message by calling all of the nested views' DoDraw methods. This means that each of the nested views will use its own Draw method to draw itself and then call the DoDraw methods of all of its nested views. In this way, the message propagates recursively until all views have received it.

16.4.4. CView and CSubview—Interface Similarities

This schema (base method/"Do-" method) works quite well for all views that can act as enclosures, that is, for objects derived from CSubview. It is unnecessary for views that derive directly from CView. A CView object, has no need to propagate messages because it cannot contain nested views. Thus, when a view from a class deriving directly from CView—for example, CNativeView—receives a Draw event, it simply draws itself.

It is clear that only views inheriting from CSubview, that is, views that can act as enclosures, need and use the "Do-" methods. For example, you might expect DoShow to appear only at the CSubview level and not at the CView level in the XVT-Power++ class hierarchy. However, if you examine CView, you will notice that it contains the "Do-" methods, just like CSubview does. For any CView, you can call DoShow, DoDraw, DoHide, and so on.

XVT-Power++ is structured this way so that the CView and the CSubview classes can have a very compatible interface. In other words, when you get access to a view object, you do not have to worry about whether it has the specific properties of a CView or CSubview object. You can treat these objects in the same way.

16.4.4.1. Wide Interface

At the CView level, a "wide interface" is in place to make CView objects almost identical in use to CSubview objects. CView's "Do-" methods simply call the view's basic method and do not attempt to do looping through enclosed views because there are not any enclosed views. For example, DoDraw simply calls Draw.

While this "wide interface" may seem redundant, it is intended as a convenient arrangement for you, the application developer. XVT refers to this "wide interface" at several points in the *XVT-Power++ Reference*.

16.4.4.2. Narrow Interface

In general, avoid direct calls to the narrow interface; always call the wide interface. The wide interface (DoDraw, DoActivate, DoKey, ...) is for your use, while the narrow interface (Draw, Activate, Key, ...) is used by the XVT-Power++ framework and also by experienced programmers who are extending the framework. C++ does not provide "package" type access control so these narrow methods must be public, even though they must be used with care.

Caution: Unless you're extending the framework, avoid using the narrow interface.

16.5. Keyboard Navigation

The XVT-Power++ framework contains a set of classes that work together to provide keyboard navigation across multiple views of a window. The architecture of the keyboard navigation system is very flexible and allows you to customize it to the needs of even the most complex interfaces.

16.5.1. Navigation Terminology

The CNavigator class provides many ways of customizing the navigation across views. The following components are used to define a navigation sequence:

Tabbing sequence

A navigator owns an ordered list of CTabStops. A navigator can have any number of tab stops. Usually, each tab stop represents a single view which is activated when reached by a navigation sequence. However, a tab stop can also represent a nested navigator which manages key movements for views in a nested enclosure. For example, consider a window containing an edit

field and a radio group with two buttons. This window's navigator contains two tab stops—one for the edit field and one for the radio group. The second tab stop, however, represents a nested navigator which manages navigation among the buttons within the radio group.

Movements
A set of movements that the application will recognize and advance focus when they occur. You can establish a dynamic mapping of keystrokes to movements for your end user. For example, you might want to map the right-arrow key (K_RIGHT) to a right movement (M_Right). Usually, movements are defined for the Tab and Back-Tab keys. Movements are defined with the assistance of the CCourse class.

EndJump
A set of decisions for tabbing past the last (or first) tab stop in the list. Options are:

EJ_Circular
: Causes navigation to start over when the last (or first) tab stop is reached.

EJ_None
: Stop the navigation when the last (or first) tab stop is reached.

EJ_Out
: Used by those navigators that behave as if there is seamless navigation between one navigator and a sibling navigator.

JumpInto
Defines the tab stop the application starts at when jumping down into a nested navigator. Options include:

JI_Selected
: Move down to selected tab stop.

JI_End
: Move down to home or end.

Hot keys
CNavigator allows the mapping of hot keys to specific tab stops.

16.5.2. Automatic Default Navigation

CWindow automatically creates a default navigator to manage top-level views (i.e., views nested one level deep). The type of navigator created is of the class CWindowNavigator. By default, this navigator performs the following:

- Defines movements for Tab and Back-Tab

- Defines a circular EndJump navigation
- Appends tab stops for all top-level views
- Initializes window focus to the first tab stop
- Registers itself with the global CNavigatorManager

The tab stops for the navigator are added using the CNavigator::AppendSubviews() interface. This interface adds the views to the navigation as long as they support keyboard focus. For example, CText does not support keyboard focus, but NEditControl does. In addition, AppendSubviews() creates special nested navigators for any radio groups in the window.

The views are added to the navigator in their order of creation, or stacking order. Views at the back of the stacking order are created first.

Special Instructions for XVT-Architect Users

In XVT-Architect, you can set the default tabbing order of views managed by a CNavigator object by following these steps:

1. Open the Drafting Board.
2. Select Palettes|Alignment.
3. Select the first view in the tabbing sequence.
4. Click on the "Send to Front" button in the Alignment palette.
5. Repeat steps 3 and 4 for the remaining views in the order that they should be tabbed.

In other words, if the views in your application are created by XVT-Architect, you can change their order of creation by using the “Send to Front” and “Send to Back” buttons in the window’s Drafting Board alignment palette.

See Also: The **...samples/arch/keynav** creates a window containing all the XVT-Architect views that support navigation; the sample shows how automatic navigation is provided in an application with a minimal amount of code.

16.5.3. Keyboard Navigation Classes

The following classes provide keyboard navigation:

CNavigatorManager
: Global manager of keyboard navigators. It maintains the

navigator associated with each window so that events can be delegated to the navigator when they are sent to its window.

CNavigator
: Delegate class which handles keyboard navigation for a window or other enclosure.

CKey
: Object representing information sent with a keyboard input event.

CTabStop
: Helper class that manages information relevant to each stop in a navigation sequence.

CCourse
: Helper class that manages information relevant to the direction taken when navigating from one tab stop to another.

By default, XVT-Power++ uses the navigation classes to give each window default support. When a window is created, the window automatically creates a keyboard navigator that navigates across all the top-level views in the window. The navigator is created in the virtual CWindow::ConstructNavigator() method, which you can override if you want to provide a different type of navigator for that window and its views. In addition, you can specify a different type of default navigator by defining a new CWindowFactory object; for more details, see section 15.5.3 on page 15-13.

16.5.4. Handling Keyboard Events

Navigators created internally in CWindow are automatically registered with the global navigator manager. As keyboard events arrive at the application's switchboard, the following actions are taken:

- The navigator for the window is located in the navigator manager.
- The navigator's DoKey() method is invoked to handle the event.
- If the navigator knows how to handle the keyboard event for that view, it does so. This normally means that the navigator sets the focus to the next view in the navigation sequence. Alternatively, you can plug in your own navigators to handle keyboard events in a completely different manner.
- If the navigator does not know how to handle the key, the event is sent to the window's DoKey() method for normal processing.

- For more details on how keyboard events are handled, refer to section 15.3 on page 15-6.

16.5.5. Customized Navigation

There are times when the default automatic navigation is not suitable to the interface of an application. Navigation can be customized using the following approach:

1. After a window's views are created, obtain a pointer to the window's navigator:

```
CNavigator* aWindowNav = aWindow->GetNavigator();
```

2. Add tab stops as necessary. For example, the following code adds tab stops for a window with nested groups of views:

```
// Clear old tab stops:
    aWindowNav->ClearTabStops();

// Create a nested navigator for views
// inside itsTopViews enclosure.
    CNavigator* aTopNav = new CNavigator();
    aTopNav->AppendSubviews( itsData.itsTopViews );
    aWindowNav->AppendSubNavigator( aTopNav );
```

3. Add more tab stops for a view in itsBottomViews enclosure.

```
aWindowNav->AppendSubviews( itsData.itsBottomViews );
```

   In actual practice, you don't have to create a nested navigator just to add nested views. This was done above just to demonstrate that by using nested navigators, you can add specialized navigation to a subset of nested views.

4. Define hot keys if necessary. The following code maps the "A" key to the selection of the "A" view.

```
aWindowNav->AddHotKeyToView( CKey('A'), itsData.itsA );
```

5. Add movements. The following code adds movements for the Tab and Back-Tab keys:

```
aTopNav->DefineMovement(
    CKey( '\t' ),
    CCourse( CCourse::kRight, CCourse::kNear ) );

aTopNav->DefineMovement(
    CKey( CKey::VirtualKey, K_BTAB ),
    CCourse( CCourse::kLeft, CCourse::kNear ) );
```

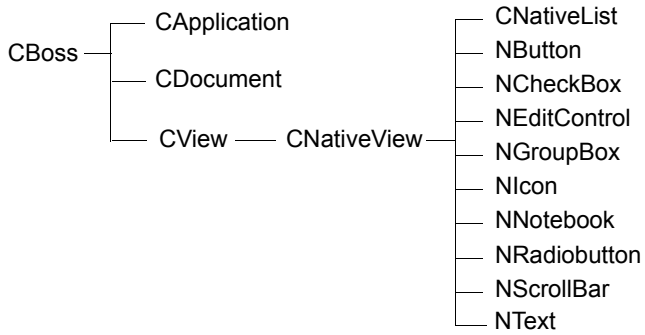
6. Set any other CNavigator attributes as needed.

See Also: For more details about how to customize a navigator, refer to the **...samples/arch/keynav** example, from which the code snippets in this section were excerpted.

17

NATIVE VIEWS

17.1. Introduction



Native views are views that have the look-and-feel of graphical objects provided by the native window manager. “Native views” is XVT-Power++’s term for controls, such as scrollbars, buttons, list boxes, radio buttons, check boxes, and pop-down menus, all of which provide some means for the user to interact with the application. They are standard items on almost any GUI but look a little different from platform to platform.

XVT-Power++’s native views “fit in” visually and functionally with the analogous graphical items on your platform, whether you are working on Motif, on MS-Windows, or on a Macintosh. They are thus not implemented by XVT-Power++ but, at the lower level, are implemented by the native toolkits. For this reason, you have less flexibility in what you can do with them. For example, the native classes are derived directly from CView and thus cannot act as enclosures for other views. You cannot draw anything inside them and would not expect to on most platforms.

However, when you instantiate one of XVT-Power++'s native view classes, you do get a lot of extra features that you might not expect to get out of controls. Since native views derive from the CView class, they have all of the capabilities of other objects at the view level. That is, they automatically propagate events and adhere to their environment, plus they can be enabled/disabled, shown/hidden, activated/deactivated, moved, and sized as discussed in section 14.3.3.

All of XVT-Power++'s native views have a DoHit method that functions as the interface for the actual events the native views can receive. Every native view class handles these events automatically, and you do not have to do anything about the DoHit. However, if you want to interact with the events at the lower level and create your own native view class, you may need to override DoHit.

17.1.1. CNativeView

At the top of the native view hierarchy is CNativeView, which handles much of the work that must be done to manipulate the native views that inherit from it—all the way from moving and sizing to creation/destruction, enabling/disabling, and so on. CNativeView is an abstract class that cannot be instantiated because XVT does not know exactly which native view you want to create. XVT-Power++ provides several native view classes, and this chapter surveys them all.

See Also: For details on each class discussed in this chapter, see its respective section in the *XVT-Power++ Reference*.

17.2. Types of Native Views

17.2.1. NButton

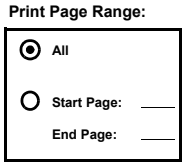
The NButton class allows you to create a button that you can "press" using the mouse, to generate a DoCommand. You must set the number of the command for the button to generate. Upon receiving a DoHit message, a button generates a command that may be handled at different levels in the XVT-Power++ application framework. When you instantiate a button, you give it a CRect to specify its size and give it a title. The title is clipped inside of the button.

17.2.2. NCheckBox

The NCheckBox class provides an object that becomes selected (generating a select command) when the user clicks on it and deselected (generating a deselect command) when the user clicks on it again. Check boxes are commonly used for menu items or items on dialog boxes so that the user can know whether an option is currently turned on or off.

You can set the ID numbers for the select and deselect commands of a check box. In addition, a check box can have a title beside it, just as a button can have a title *inside* it. You give the check box a top-left point for the title, and the entire check box is sized to fit the title. Unlike the title of a button, then, the title is not clipped to the check box.

17.2.3. NRadioButton and CRadioGroup



A radio button is similar to the check box. Because the radio button object by definition must be used in groups, it is provided by two classes, NRadioButton and CRadioGroup.

While check boxes allow you to select multiple options by checking more than one box at a time, only one of the radio buttons in a group can be selected at a time. Selecting one radio button means to deselect another. The helper class CRadioGroup serves as a grouper.

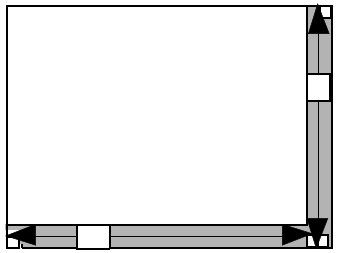
To instantiate radio buttons, you must first create a CRadioGroup object and then add the buttons to that group, either one-by-one or as a set. If you add them as a set, the buttons are automatically placed inside the group, vertically or horizontally, so you do not have to calculate the locations. When the radio group is instantiated, it is initially empty. You give it a point in the top-left corner to use as a reference for placing the buttons, and as radio buttons are created, the group's size increases.

Each radio button is assigned a URL resource ID number. When a single button is added to a radio group, the AddButton method returns an integer, which is the ID number of the radio button. When multiple buttons are added to a group, the AddButtons method returns the ID number of the button corresponding to the first resource ID. The IDs of the remaining buttons follow sequentially.

If you indicate that you want the radio group to be drawn, it draws a box around the buttons inside it. You can have several radio button groups on the screen at one time that will act independently.

When you select a radio button, it generates a DoCommand. As with check boxes, the titles of radio buttons are located beside the button and are not clipped—unless the radio group itself is clipped, in which case the titles will clip to that enclosure. The radio group, of course, acts as the enclosure for the radio buttons it contains, and you can nest other objects besides radio buttons within it if you desire. For example, you might want to add a text field (such as an NLineText object) beside a radio button or associate a small picture with it.

17.2.4. NScrollBar



Another native view is XVT-Power++'s NScrollBar, which provides a horizontal and/or a vertical scrollbar for a view. Many NScrollBar objects are created automatically in XVT-Power++.
For example, CListBox, CScroller, and NScrollText use NScrollBar to create scrollbars automatically. NScrollBar is a convenient class for creating your own scrollbar by instantiating one of these objects, specifying whether it is to be horizontal or vertical, and giving it a starting and ending position and a range. You can find out an NScrollBar object's native height and width using its NativeWidth and NativeHeight methods.

The scrollbars are updated automatically. For a vertical scrollbar, the topmost position is the *minimum* position and the bottom is the *maximum* position. For a horizontal scrollbar, the left is the minimum and the right is the maximum position. Through its DoHit

method, an NScrollBar automatically captures the events it can get, such as mouse clicks or drags on its thumb or clicks for a page or line scroll. The scrollbar automatically sends these events up to its enclosure after translating them into HScroll and VScroll messages, which contain information about the type of event (e.g., line up, line down, page up, or page down) and the thumb position.

A class derived from NScrollBar called NWinScrollBar uses special types of scrollbars that are attached to the window. These scrollbars are special because they can have a special look-and-feel when they are attached to the window. For example, on the Macintosh, if a scrollbar is attached to a window, it automatically becomes disabled when the window moves to the background. On MS-Windows, a scrollbar attached to a window disappears when there is nothing left to scroll and reappears when there is something to scroll. When you size the Program Manager, for instance, and objects are clipped, scrollbars appear automatically.

An NWinScrollBar object is instantiated automatically when you create a window and specify scrollbars as one of its properties. This is one of the XVT Portability Toolkit properties that you can give to a CWindow object. NWinScrollBar objects can also appear on list boxes, scrollers, and other views if they are given a special parameter that notifies them to use the window-attached scrollbar.

See Also: For information on native list controls, see NListEdit, NListButton, and NListBox in the *XVT-Power++ Reference*.

17.2.5. NNotebook

A Notebook is comprised of three objects: The shell, the page windows and face windows of the notebook. The notebook shell is responsible for drawing the border of the notebook and the Tabs. Each Tab has a border, and possibly text and an image. The notebook shell is implemented by the NNotebook class.

The face window is where information is presented to the user (i.e. static text, listboxes or other controls). The face window is represented by the CFaceWindow class. The face is physically contained within the NNotebook.

At the C++ layer, NNotebook is not physically set as the enclosure for the face window, yet the NNotebook object contains it. This arrangement might seem awkward at first, since CWindows (from which CFaceWindow is derived) may only have other CWindows as enclosures. To clarify the issue, we must look at the PTK layer.

In the underlying PTK, the shell is a window and is the parent of all Face windows, each Face being associated with a particular Page. A Page is a logical construct that contains the number of Faces associated with each Tab and tracks the currently displayed Face. The notebook shell contains the Page information and uses it internally. Therefore, the NNotebook class is a special control, and Faces are special windows which may have a notebook control as their enclosure.

Navigation between Pages is user-defined, and objects can be implemented for moving from one Page to another.

NNotebook provides methods for managing Tabs and Pages. Tabs and Pages are created and destroyed through NNotebook, as are the methods for displaying Faces and implementing Tab navigation. Faces (CFaceWindow objects) inherit all of the capabilities for managing other views (child windows, controls, etc.) with fully configurable navigation.

17.2.5.1. Creating and Destroying a Notebook

There are five steps to creating a Notebook as follows:

1.) Create an NNotebook.

2.) Create a CFaceWindow.

3.) Create a notebook tab.

4.) Create a notebook page.

5.) Set the tab and page face.

Note: you cannot create subviews within the CFaceWindow until after the call to SetFace to set the CFaceWindow to the NNotebook object.

Example:

```
NNotebook* aNotebook = new NNotebook(theParentWindow, theCRectSize);
CFaceWindow* aFace = new CFaceWindow(theDocument, aNotebook);

aNotebook->AddTab (theTabNum, theTitle);
aNotebook->AddPage (theTabNum, thePageNum, theTitle);
aNotebook->SetFace (aFace, theTabNum, thePageNum);
```

To remove a tab or a page, use NNotebook->RemoveTab or NNotebook->RemovePage before deleting the CFaceWindow. The CFaceWindow may be reused after the corresponding Remove method.

Note: Do not call Close on a CFaceWindow. The corresponding RemovePage will call the CFaceWindow's Close.

17.2.5.2. Interface Objects

After calling NNotebook->SetFace, objects may be added to or removed from the CFaceWindow. Any desired Environment may also be set at this time.

Example:

```
aNotebook->SetFace (aFace, theTabNum, thePageNum);
NButton* aButton = new NButton (aFace, aRect, aCStringRW);
```

17.2.5.3. Navigation

There are three navigators of concern when discussing notebook navigation. The three navigators involved are the notebook's enclosure, the NNotebook, and the CFaceWindow.

The notebook's enclosure provides navigation to the NNotebook. This is accomplished by adding the NNotebook's navigator to the enclosure's navigator.

The NNotebook's navigator interprets navigation from both the enclosure and the CFaceWindow for seamless navigation. The NNotebook's navigator also provides Tab to Tab and Tab Hot Key navigation. The zero'th CTabStop in the NNotebook's navigator is a subnavigator pointing to the CFaceWindow for the current Tab and Page of the notebook. When forward navigation comes from the enclosure, the NNotebook passes navigation to the CFaceWindow found in the zero'th CTabStop of the NNotebook's navigator.

When forward navigation comes from the last object in the CFaceWindow, the NNotebook passes navigation back to the notebook's enclosure's navigator. If the NNotebook's navigator receives a backward navigation from the notebook's enclosure or from the first object in the CFaceWindow, the current Tab is selected for Tab to Tab navigation. The right and left arrow keys are used to navigate the notebook tabs.

Finally, the CFaceWindow's navigator provides navigation of its views. The CFaceWindow's navigator is fully definable. However, to change

or modify the behavior of the enclosure's and/or face's navigators, The following conditions must be met.

1.) Navigation from the notebook enclosure to the CFaceWindow must use a CKey of TAB and CCourse(kLeft, kFar).

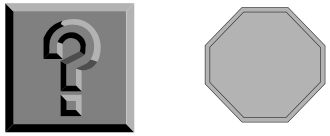
2.) Navigation from the CFaceWindow to the notebook enclosure must use a CKey of TAB and CCourse(kRight, kNear) from the last object in the CFaceWindow.

3.) Navigation from the notebook enclosure to an NNotebook Tab must use a Ckey of BTAB and CCourse(kLeft, kFar).

4.) Navigation from the CFaceWindow to an NNotebook Tab must use a CKey of BTAB and CCourse(kRight, kNear) from the first object in the CFaceWindow.

Note: To use tab hot keys with Motif, the ATTR ATTR_X_PROPAGATE_ECHAR must be set to TRUE.

17.2.6. Icons



An icon resource is a bitmap picture that can be drawn on different platforms. XVT-Power++ contains a basic icon class called NIcon. NIcon derives from CNativeview, so icons have all the properties of native views.

Each platform imposes its own restrictions on how the resources are handled. For example, on some platforms, icons can appear in only two colors, while on other platforms they can be drawn in a wide range of colors. Some platforms limit the size of icons to 32-by-32 pixels, while others impose no limitations on size.

17.2.6.1. Icon Portability Issues

When you want the most portability from platform to platform, you can safely assume that icons must be drawn in only two colors and are limited in size to 32-by-32 pixels. If you are willing to sacrifice portability to obtain more flexibility, you can choose to work with each platform to its fullest extent when you draw your own resources.

Tip: There are several programs on the market and in the public domain that allow you to convert icon resources from one platform type to another.

17.2.6.2. Environment Settings for Icons

On some platforms, the icon drawing appears in the foreground color, as does its title. Open spaces in the drawing appear in the background color. You can set the background and foreground colors. On other platforms, the icon's color is fixed as defined.

Tip: To ensure portability, set the colors appropriately even if the information is not used.

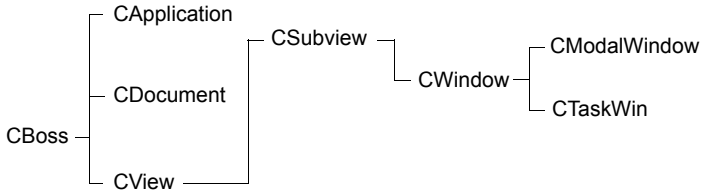
17.2.7. Icon Resources

The inclusion of resources for GUI components such as icons (and cursors) varies from platform to platform. Under X/Motif, icons are created somewhat differently than they are for other platforms, where they can simply be incorporated through a URL definition.

See Also: For step-by-step information on how to build an icon or cursor resource, see the platform-specific book, *XVT Platform-Specific Book for Motif*.

18

WINDOWS



Windows are a special type of view in XVT-Power++ because although they inherit from CSubview, they have a predefined enclosure. This enclosure is either the screen itself or, on platforms such as Windows or OS/2 PM, the task window. Objects of type CWindow are the only views derived from CSubview that cannot be nested inside other views. In fact, windows are the topmost enclosure for any other type of XVT-Power++ view, and their layout on the screen is managed by a class named CDesktop.

This chapter discusses the attributes a window can have, the possible types of windows, how they are constructed, and how you can derive your own classes from CWindow. It briefly considers the task window, an example of a class of windows that is derived from CWindow. The chapter concludes with a look at CDesktop.

18.1. Window Attributes

When you instantiate an object of type CWindow, you can set several different attributes that are available through the XVT Portability Toolkit in order to tailor the window to the very specific needs of your application: deciding whether it has scrollbars or a menubar of its own and whether it can be moved, sized, or iconized. The possible window attributes are listed in Table 18.1. XVT supplies other, platform-specific flags.

WSF_NONE	No flags set
WSF_SIZE	Is user-sizeable
WSF_CLOSE	Is user-closeable
WSF_HSCROLL	Has horizontal scrollbar outside of the client area
WSF_VSCROLL	Has vertical scrollbar outside of the client area
WSF_DECORATED	A convenient combination of WSF_SIZE, WSF_CLOSE, WSF_HSCROLL, and WSF_VSCROLL
WSF_INVISIBLE	Is initially invisible
WSF_DISABLED	Is initially disabled
WSF_ICONIZABLE	Is iconizable (XVT/Win16, XVT/PM, and XVT/XM only)
WSF_ICONIZED	Is initially iconized
WSF_FLOATING	Is a floating window (XVT/Mac only)
WSF_SIZEONLY	Lacks border rectangles (XVT/Mac only)
WSF_NO_MENUBAR	Has no menubar of its own (see Note)
WSF_MAXIMIZED	Is initially maximized
WSF_DEFER_MODAL	Modal status deferred (not processed by xvt_win_create*)
WSF_PLACE_EXACT	Modal window is placed exactly where specified

Table 18.1. XVT Portability Toolkit window-creation flags

Note: WSF_NO_MENUBAR implies that the window has no menubar. You can use this only with top-level windows; child windows never have menubars.

If the window has a scrollbar, you can choose between horizontal or vertical scrolling; if it has a menubar, you must decide how it will react to the menubar's events when the user selects it. Of course, you will want to ensure that the window can respond to events that come through the system—such as mouse clicks, drawing events, moving to the front of the window stack, and so on—unless you intentionally disable it.

See Also: For more information on these window flags, see the *XVT Portability Toolkit Guide* and the online *XVT Portability Toolkit Reference*.

18.2. Interaction With the Document

A CWindow object can be of any XVT type; it receives all window events. Most of the window management, such as moving and sizing, is done by the window manager or the XVT-Power++ desktop. Possible XVT window types are shown in Table 18.2.

W_DOC	Document window
W_PLAIN	Single-bordered window
W_DBL	Double-bordered window
W_NO_BORDER	No border
W_MODAL	Modal document window

Table 18.2. XVT Portability Toolkit window-type flags

Windows are normally created inside CDocument::BuildWindow method and are in charge of viewing the data that the document is managing. Each window has access to its own document object through a pointer, and interacts with it by default at several different points in the life of the application. For example, when the window is going to close, it checks to see whether the data in the document needs to be saved. If it does, the window invokes a dialog box giving the user the option to either to save the data, close without saving, or cancel the close operation.

Windows also interact with their documents when they delegate tasks to them. For example, if a window receives a DoCommand and elects not to trap it, it delegates the DoCommand up the chain of the application framework to its document. Similarly, this is true of the menu events described in section 15.2 on page 15-3 and the keyboard events described in section 15.3 on page 15-6.

18.3. Window Construction

t To construct a window:

1. Derive your own application-specific window class.
2. Add some objects that will be nested inside the window, creating them in the window's constructor.
3. Give the window itself as the enclosure of the objects that will be nested inside.

For example, if you want to create a window that has three buttons and a text field object, you would instantiate three buttons and a text field in the CWindow constructor and give the constructor the this pointer as the enclosure of those objects, as follows:

```
MyWindowClass::MyWindowClass(CDocument* theDocument,
      const CRect& theRegion)

      : CWindow(theDocument, theRegion, "", NULL,
                  W_DOC, NULL)
{
      NButton* aButton = new NButton(this,
                  CRect(10,10,60,50), "Button1");
      ...
      aButton = new NButton(this, CRect(60,10,100,60),
                  "Button2");
      ...
      aButton = new NButton(this, CRect(110,10,150,60),
                  "Button3");
      ...
      NLineText* aField = new NLineText(this, CPoint(10,80),
                  140);
}
```

Once this specific class is defined and you want to create a window with three buttons and a text field, you simply instantiate your new window class, which is derived from CWindow.

To destroy a window, call the Close method on it. In response, the window disappears and the memory that it occupies is freed. Closing a window is semantically equivalent to deleting the window. If you have a pointer to a window object, window object -> close is equivalent to delete window object, except that you are not allowed to say "delete window object" because the constructor of CWindow is protected.

18.4. The Task Window

The task window, CTaskWin, is a class that XVT-Power++ uses internally. It is a private class that can be instantiated only by XVT-Power++. This class is created for use on platforms that require a task window to enclose all other windows in the. CTaskWin maps directly to the XVT task window representation, TASK_WIN.

A task window has several of the properties of other windows, but it is much more limited in what it can do, especially since it is confined to certain platforms. Even within those platforms, the limitations on the task window depend upon whether the window is drawable, a property that can be toggled through XVT's xvt_vobj_set_attr function.

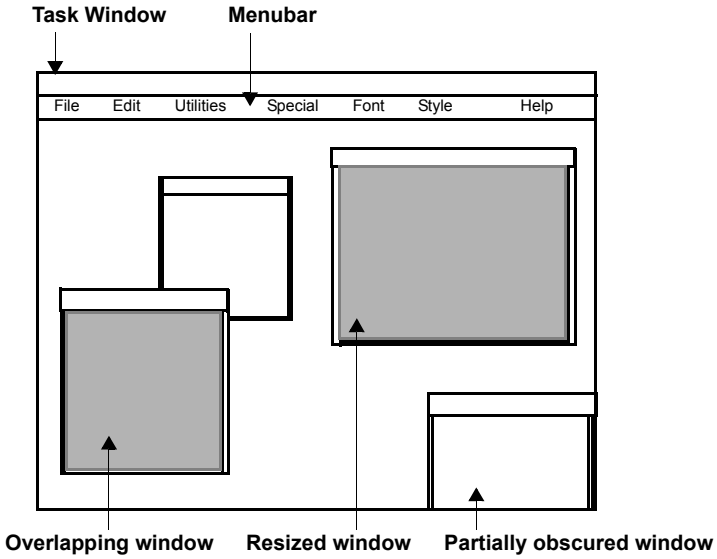


Figure 18.1. Sample task window

18.5. The Desktop

No discussion of CWindow is complete without a consideration of CDesktop, which is responsible for managing the layout of the different windows on the screen. At any time while it is running, an XVT-Power++ application will have a number of windows open on the screen, without regard to their associated documents. The desktop keeps track of all these windows and their active/inactive states.

The desktop is notified each time a different window is brought to the front of the window stack. It has methods for setting and getting the front window and a method for placing a window on the screen, staggering it with windows that are already present. Also, you can use CDesktop::FindWindow to find out whether a certain window is in the desktop and GetNumWindows to find out how many windows are currently in the desktop. Finally, CDesktop has protected methods for adding and removing a window from the desktop. These methods can be accessed only by CDocument objects.

19

MOUSE EVENTS AND MOUSE HANDLERS

One of the key features of graphical user interface applications is that they are mouse-driven. The mouse is a very popular way for the user to interact with the application. This chapter considers the kinds of interactions that occur as views receive mouse events.

19.1. Basic Mouse Events (Methods)

These are the four different kinds of mouse events that can occur in XVT-Power++:

MouseMove
: Generated as the mouse cursor moves around on the screen, regardless of whether a mouse button is being pressed.

MouseDown
: Generated when the user depresses a mouse button. Since a mouse can have from one to three buttons, each button can have a different meaning when it is pressed in an application. Thus, different values are assigned to the MouseDown event, depending on which button is pressed.

MouseUp
: Generated when the user releases a mouse button and depends upon the value that has been assigned to that particular button.

MouseDouble
: Generated when the user double-clicks a mouse button; a MouseDouble event has a different meaning than just clicking the button once.

19.1.1. Clicking the Mouse

Typically, mouse events occur in a sequence. By default, all CView objects generate a click command whenever the mouse is clicked on them. A mouse “click” is a sequence of a MouseDown followed by

some possible mouse moves and concluded with a MouseUp. In other words, to "click" the mouse, the user must at the very minimum press and release the mouse button, generating a MouseDown, MouseUp sequence. If the user presses down a mouse button over a view and then drags the mouse outside the view's region without letting go, this is not a click. Such behavior enables a user to prevent an unintended click by simply dragging the mouse away from a view before releasing the mouse button.

See Also: For more information about implementing drag-and-drop behavior in an application, refer to section 19.3 on page 19-9.

19.1.2. Mouse Event Parameters

Each of the four basic mouse methods has the following three parameters:

- A parameter that contains a point, called theLocation, that indicates exactly where the mouse was when the event occurred. This location is a point representing, in logical units, the coordinate relative to the view containing the point.
- A parameter indicating which mouse button, if any, is associated with the event. This parameter takes a value of 0, 1, or 2 (with 0 representing the left-most button), depending on which button was pressed. For a one-button mouse, the button value will always be 0.
- A parameter that indicates whether the Shift or Control key was pressed in conjunction with the mouse button.

19.1.3. The "Do-" Mouse Methods

The mouse events described in section 19.1 are not the only ones included in the interface. Actually, there are four additional mouse events:

- DoMouseDown
- DoMouseMove
- DoMouseDouble
- DoMouseUp

The "Do-" mouse methods are the methods that trigger the mouse method calls. They are decision-making methods in that they determine which view should receive a given method. Normally, a window receives a "Do-" mouse event and a "Do-" mouse method

is called for the view (associated with that window) that lies under the mouse pointer.

The "Do-" mouse methods carry the same kind of information as the basic mouse methods—a CPoint location, a button value, and an indication of whether the Shift or Control key was also pressed. However, the CPoint that indicates where the mouse event occurred is in *window-relative* coordinates because it has not yet been decided which view should receive the event. The only thing known at this point is the particular window that should be in charge of this method.

To summarize, inside every "Do-" mouse method, the following three steps occur:

1. It determines which view should receive the event.
2. It localizes the location to the coordinate system of the view that is to receive the event.
3. It calls the receiving view's basic mouse method theLocation with the localized information.

19.1.4. Propagating Mouse Events Through Views

Most view classes are programmed to respond with very specific behaviors to mouse events. For example, all moving and dragging is handled by the views themselves. A button is an example of a view that is programmed to respond to MouseDown and MouseUp events in a certain way. When a user presses a mouse button over a button view, the appearance of the button view changes to indicate that it has been activated. Upon a release of the mouse button, the button view's appearance changes again, and the MouseUp event generates a DoCommand message.

The mechanism in XVT-Power++ for deciding which view should receive a mouse event is the concept of the "deepest subview." When a mouse event occurs at a certain location, that event first goes to the window. The window then searches for the deepest view that contains this location. "Deepest" means that several views can be nested around a given point and that the target of the mouse event is the view that is nested most deeply.

Finding the deepest subview is accomplished using the CView::FindEventTarget method. For classes derived from CView, you can override this method, perhaps changing the logic used to find the deepest subview so that it returns a different target for the mouse

event. FindEventTarget is commonly overridden for views that handle an event themselves instead of passing it on to another view.

19.1.5. Using the Mouse to Resize a View

If the view that is to receive the event is movable and/or sizeable, this state is treated as a special case. For example, if a button receives a mouse event, it typically changes its appearance when pressed and generates a DoCommand when it is released. However, if the button is movable and sizeable, the normal behavior does not occur when the user clicks on it. Instead, a rubberband frame appears around the button so that it can be dragged to another location or sized. CWireFrame is the class that implements all the moving and sizing in XVT-Power++.

XVT-Power++ must therefore be able to find out whether the view that is to receive the event is movable or sizeable; this is achieved using a method called FindHitView. Once FindEventTarget returns a view as the target of the event, then FindHitView is called on that view. FindHitView either returns the view itself or the view's wire frame. When a view is movable or sizeable, it channels the mouse events it receives to its wire frame helper.

The following is a summary of the steps that occur in sending a mouse event to a view:

1. A user clicks the mouse over a certain point on the screen.
2. A CSubview "Do-" method receives this event and interprets the event's location in window-relative coordinates; it is clear which window should be in charge of the event.
3. The "Do-" method calls the CView FindEventTarget method to find the deepest subview containing the event's location.
4. FindEventTarget calls CView::FindHitView, which returns either the target view or its wire frame.
5. If FindEventTarget returns the target view, the "Do-" method localizes the event location to the coordinate system of the target view.
6. The target view's basic mouse method is called with the localized information.

19.2. Mouse Event Processing

The XVT-Power++ framework provides different ways of dealing with mouse events in an application. Events originate in the system and are sent to the application's CSwitchBoard object. As described in section 19.1, CSwitchBoard may receive four kinds of mouse events: down, up, double, and move. Figure 19.1 illustrates the flow of these events into an application and the different objects that are involved in handling these events:

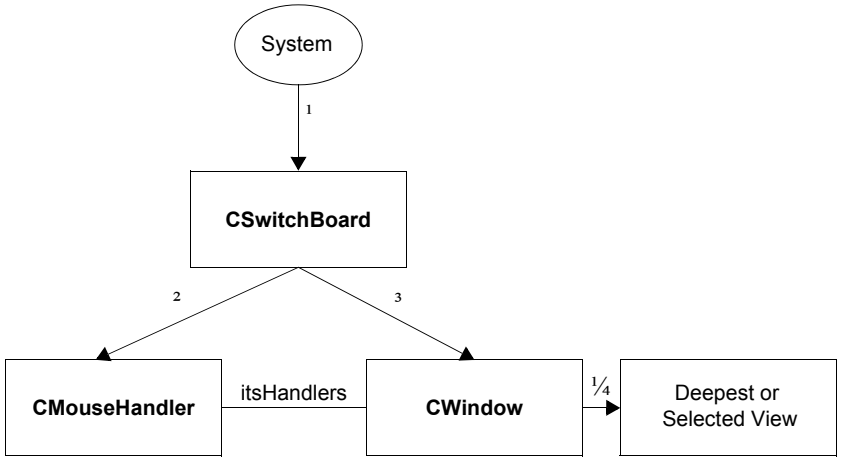


Figure 19.1. Mouse event processing in an XVT-Power++ application

XVT-Power++ view classes "expect" to receive four types of mouse events; with each event, the system provides the following information:

- The window that is the target of the event
- A point representing the mouse location of the cursor, relative to the target window's coordinates
- Which mouse button has been pressed if any (e.g., left, middle, or right)
- Is the Shift key down?
- Is the Control key down?

In the second phase of the event handling, (represented in Figure 19.1 with ²), the switchboard locates the CWindow object that is

responsible for the event. When it is determined that a particular view is the target of a mouse event, one of these four methods is called and the view receives a message. However, before sending the event to the window, the switchboard sends the events to the window's mouse handler objects. A window may have zero or more such mouse handlers.

If the window has no mouse handlers, or if the mouse handlers do not consume the mouse event, then the switchboard sends the event directly to the window by a call to one of its virtual DoMouse*() event methods. From this point, events are further delegated to the window's deepest or selected view through a call to the one the view's virtual Mouse*() methods.

19.2.1. Mouse Handlers

Mouse handlers are ideal when mouse behavior needs to be changed for a window whenever a special kind of view is inserted into the window. For example, suppose you define a special view that requires that the cursor bitmap change whenever the mouse enters or leaves the view. The view can accomplish this by registering a mouse handler with its window. The alternative (old approach) would have required that you override the DoMouse*() methods of each window that contained the new kind of view to ensure it changed cursors depending on the location of the mouse.

Tip: Mouse handlers also make it easy to combine several mouse behaviors—simply plug in several handlers at a time.

19.2.1.1. Why Use Mouse Handlers?

CMouseHandlerS are potential timesavers because they allow you to define a mouse behavior once and reuse it with any window by simply plugging an instance into the window. For example, XVT-Power++ provides a predefined CMouseHandler derived class that defines drag-and-drop mouse handling. By plugging one of these handlers into a window, you can easily add drag-and-drop mouse behavior to your application. This handler is described in detail in section 19.3 on page 19-9.

Another handler provided as a sample with DSC++ provides "grid snapping" behavior to a window with movable and sizable objects—it modifies mouse events before a window gets them so that objects in the window will snap to an imaginary grid as they are dragged or resized by the user.

See Also: For more information about a mouse handler that implements grid snapping, refer to the **...samples/pwr/drag** sample; this sample program demonstrates how a mouse handler like this is used in a DSC++ application.

19.2.1.2. Registering a Mouse Handler

Each XVT-Power++ window may have zero or more mouse handler objects that deal with the mouse events received by that window. Mouse handlers are registered with a window through the use of CWindow's CMouseManager object. The mouse manager is simply an object that manages the collection of various mouse handlers. It determines the order in which the handlers receive events, and it can be customized to redefine when and how events are received by each handler.

The following code shows how a mouse handler is registered with a window:

```
CWindow * aWindow = ...;
CMouseHandler * aHandler = ...;
aWindow->GetMouseManager()->RegisterMouseHandler(
      aHandler );
```

CMouseHandler is an abstract class, but you can define (or use) derived classes that provide overridden versions of virtual mouse handling methods. These methods fulfill several different roles. They can modify the mouse event by altering the parameters associated with the event (e.g., mouse location, button pressed, etc.). The methods can also decide to consume the mouse event. When an event is consumed, it is not passed on to the window.

19.2.2. Virtual Mouse Event Methods

When an event is not consumed by a window's mouse handler, it is passed on to the window though one of the following virtual mouse methods:

DoMouseDown

DoMouseMove

DoMouseDouble

DoMouseUp

The default behavior of these methods is to locate the window's view that should receive the event and pass the event on by calling one of the following virtual mouse methods:

MouseDown
MouseMove
MouseDouble
MouseUp

If a window has a selected view, that view will receive the event. Otherwise, CWindow uses FindEventTarget() to find the view for the mouse event. Unless overridden, FindEventTarget() returns the deepest subview under the mouse location.

See Also: For more information about virtual mouse events, refer to section 19.1.3 on page 19-2.

19.2.2.1. Overriding DoMouse*() Methods

You may occasionally decide to override a window's DoMouse*() methods to provide window-specific mouse handling behavior, but remember that this is less versatile than using mouse handlers. First, it makes re-use more difficult. Mouse handlers are designed to be reusable because they define general mouse behavior which can be reused whenever necessary by simply plugging the handler into a window instance, no matter its type. On the other hand, to provide mouse behavior through the overriding of mouse methods, you must either override the methods for each window class that requires the behavior, or derive each of these classes from a common base that defines the desired mouse behavior.

19.3. Drag Sources and Drag Sinks

Drag-and-drop is a special behavior which allows a user to depress the mouse over a specific area or object in a window and drag that object or some form of data to a new location in the window or to another window.

19.3.1. CDragSource and CDragSink

XVT-Power++ provides this type of behavior through the implementation of a special mouse handler class named CDragSource. Figure 19.2 illustrates the CDragSource class and its relationship to other classes involved in the definition of a drag-and-drop operation.

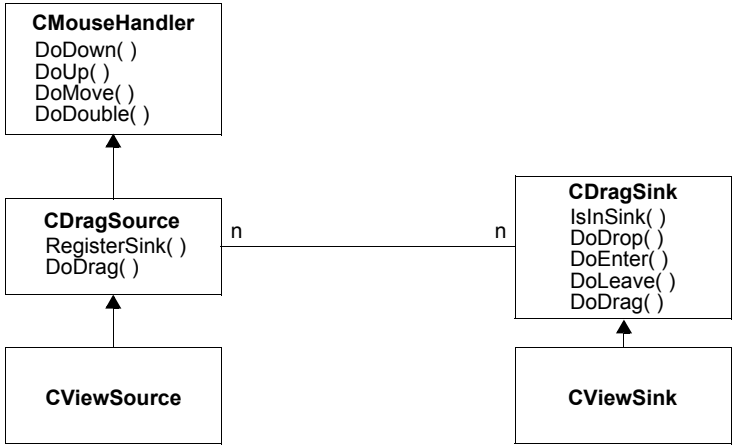


Figure 19.2. Drag-and-drop mouse behavior; relationship of drag sinks and drag sources to a mouse handler object

As shown in Figure 19.2, CDragSource is derived from CMouseHandler. CDragSource handles mouse events and converts them into drag-and-drop events propagated to zero or more CDragSinks. Drag sinks are objects that represent areas on the screen that are capable of receiving drag-and-drop events.

CDragSource and CDragSink have a many-to-many association—a source can service zero or more sinks, and a sink can be registered with zero or more sources.

For example, a drag source could be registered with a window and triggered to initiate a drag operation whenever the user clicks and drags over a certain object or area in the window. In addition, a specific drag sink associated with another window could be registered with the drag source. This arrangement allows the end user to drag-and-drop from one window into the other window.

19.3.2. CViewSource and CViewSink

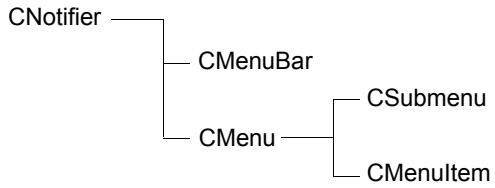
Figure 19.2 also introduces two other classes: CViewSource and CViewSink. These are specializations of the more generic CDragSource and CDragSink classes. CViewSource takes a pointer to a specific view during construction. Thereafter, this drag source will automatically initiate a drag-and-drop event whenever the mouse is clicked and dragged over that view. Similarly, CViewSink also takes a pointer to a specific view during construction. Thereafter, the drag sink will automatically send DoCommand() messages to that view whenever the mouse enters, leaves, moves over, or drops over that view.

See Also: For more information about any of the classes mentioned in section 19.3, refer to their individual descriptions in the online *XVT-Power++ Reference*.
For more information how drag-and-drop works and how to add it to your applications, refer to the **...samples/pwr/drag** sample.

20

MENUS

20.1. Introduction



XVT-Power++ menus are handled through the following classes:

CMenuBar
CMenu
CMenuItem
CSubmenu

20.2. Menubar, Menu, Menu Item, and Submenu

Menubars can be built from resources or dynamically. Note that only windows can contain menus; dialogs do not have menubars.

A top-level menubar is made up of submenus such as File, Edit, and Font. Each submenu is a collection of menu items and separators. In cascading menus, a submenu can also include other submenus.

When creating a separator, XVT-Power++ provides a convenient CMenuItem object called MENUSeparator.

Using the CMenuBar::DoUpdate method, XVT-Power++ also provides a way to delay the physical update of a menubar until all changes to the menubar have been made.

A CMenuBar object is the only object that can directly modify the physical state of a menubar. CMenu, CMenuItem, and CSubmenu are data structure classes to maintain the internal state of a menubar. XVT-Power++ handles these structures using reference counting to limit the allocated memory.

See Also: For information about reference counting, see CStringRW in the *XVT-Power++ Reference*.

20.3. Menubar Creation

Each XVT-Power++ window can use a default menubar or create its own. Dialogs do not have menubars.

Windows create CMenuBar objects, dynamically or from resources. Windows access their menubar using their GetMenuBar method. They can also change their menubar after creation using their SetMenuBar method.

The CMenuBar class provides methods to append, insert, remove, or replace submenus in the menubar hierarchy, such as:

AppendSubmenu
InsertSubmenu
RemoveSubmenu
ReplaceSubmenu

These methods do *not* take effect until the CMenuBar::DoUpdate method is called.

20.3.1. Traversal of the Menubar Hierarchy

A powerful feature of XVT-Power++ menu handling is automatic traversal of the menubar hierarchy. Automatic menu item traversal is especially important when using multi-level cascading menus.

A menu item ID is called a *tag*. Given a tag, CMenuBar can check, enable, and set the title of any menu item in the menu tree without having to traverse the tree. To do this, CMenuBar uses these methods:

SetChecked
IsChecked
SetEnabled
IsEnabled
SetTitle
GetTitle

Note that these methods *do* take effect immediately.

You may want to traverse the menu tree yourself. To do this, CMenuBar offers the GetSubmenus method. GetSubmenus returns the list of submenus for that menubar. Then, for each submenu, call CSubMenu::GetMenus method to get a list of submenus and menu items.

20.3.2. Defining Pop-up Menus

A pop-up menu is a temporary menu displayed at a specified location over a window (only windows that can receive mouse events may be specified). Pop-up menus are created by sending a DoPopup message to a CMenuBar object. Different pop-up menus can be created from the same section of code, depending on whether a keyboard modifier, such as Control or Shift, is pressed along with the mouse button.

Generally, applications should invoke a pop-up menu only in response to an E_MOUSE_DOWN event (by overriding a window's DoMouseDown() method). When the user selects an item from the pop-up menu, a normal DoMenuCommand() is sent to the window specified in the CMenuBar object.

The CMenuBar::DoPopup method can be called in a way that checks: 1) which button of the mouse was pressed, and 2) whether a keyboard modifier, such as Control or Shift, is pressed along with the mouse button. The pop-up menu can be displayed with either one of two orientations, as shown in Figure XXX.

> Figure here shows top/left orientation compared to centered vertically WRT to a particular menu item.

t To display a pop-up menu inside a window:

1. Define a CWindow with a standard or customized menubar.
2. Edit that window's CMenuBar, calling the DoPopup method in a way that forces the pop-up menu to have the desired behavior.

Example: The following DoMouseDown method creates pop-up menus for a window:

```
void CPopWin::DoMouseDown(
        CPoint theLocation,
        short theButton,
        BOOLEAN isShiftKey,
        BOOLEAN isControlKey)
{
        CMenuBar aMenu( this, POPUP_MENUS );
        if ( isShiftKey )
                aMenu.DoPopup( theLocation, M_FILE,
                                XVT_POPUP_LEFT_ALIGN );
        else
                aMenu.DoPopup( theLocation, M_NUMBERS,
                                XVT_POPUP_OVER_ITEM );
}
```

The first statement creates a temporary menubar with a pointer to the window and the ID of a menubar defined using XVT-Architect. The code then calls DoPopup using different submenus of the POPUP_MENUS menubar; the XVT_POPUP_OVER_ITEM and XVT_POPUP_LEFT_ALIGN alignment parameters control the actual location of the pop-up menu (when it is drawn) with regards to the location of the pointer. When the user selects an item from one of the pop-up menus, the window's DoMenuCommand() is called to process the event.

See Also: Refer to a sample application in **...samples/arch/popup** and its associated XVT-Architect project file for more information about using pop-up menus in your DSC++ application.
For more information about menubars, see CMenuBar in the on-line *XVT-Power++ Reference*.
For more information about the XVT_POPUP_ALIGNMENT parameter, refer to the *Data Types* section of the on-line *XVT Portability Toolkit Reference*.

20.3.3. Menubar Deletion

The CMenubar object associated with a CWindow is automatically deleted in the CWindow destructor

See Also: For more information, see CWindow in the *XVT-Power++ Reference*.

20.4. Menubar Handling

When you want the menubars to be consistent for an entire document, the CDocument object should be in charge of setting up the menubars for all the windows associated with it. When you want the menubars of all windows in an application to be consistent, the CApplication object should be in charge of setting up the menubars.

20.4.1. SetUpMenus and UpdateMenus

CApplication has the SetUpMenus method. CApplication, CDocument, and CWindow all have the UpdateMenus method. It is important to distinguish the difference between setting up a menu and updating the menu.

Setting up the menu is a task that is done once. For the task window, CTaskWin, this is accomplished with the CApplication::SetUpMenus method, which is called when the task window is created. For a window, a CWindow-derived class, you set up the menus in the constructor of that window.

On the other hand, UpdateMenus is called every time a window comes to the front of the window stack. When the window comes to the front, the menubar can be updated. Usually there is nothing to do, because the state of a window's menubar is saved. Sometimes, however, the menubar will need an update because changes have occurred while the window was in the background.

If you decide to have the document maintain menubars for all its windows, you should be aware that window may contain its own menubar or may share the task window's menubar. In these cases, you must iterate through each window owned by the document and determine the appropriate action. This is typically a simple process of applying the same menu attributes to each menubar.

20.4.2. Menu Events Handling (DoMenuCommand)

XVT-Power++ menu events are generated when a menu item is selected from a menubar.

CBoss provides a "wide interface" for menu selection handling through its DoMenuCommand method. DoMenuCommand messages can be propagated and delegated from one object to another, from a window up to its document, or on up to the application.

In CBoss, the DoMenuCommand method does not do anything, but any object that inherits from it can choose to respond to a menu selection. The XVT-Power++ default behavior is as follows:

1. CSwitchBoard finds which window had the focus at the time the menu selection occurred, and calls that window's DoMenuCommand.
2. The window can handle any menu selection which affects it, and it can pass the message up to its document.
3. The document's DoMenuCommand method calls its appropriate method in response to one of the following menu items on the File menu:

Close	Calls its DoClose method
Save	Calls its DoSave method
Save As	Calls its DoSaveAs method
Page Set Up	Calls its DoPageSetUp method
Print	Calls its DoPrint method

Any object derived from CDocument can override DoMenuCommand to modify the standard behavior already provided.

DoMenuCommand arguments are a menu tag (ranging from 1 to MAX_MENU_TAG's limit of 32,000) and two BOOLEANs indicating whether the Shift key and/or the Control key were depressed at the time the menu selection was made.

20.4.3. Handling Menu Commands

As discussed earlier (section 20.4.2), the DoMenuCommand is first called for the window from which the menu item is selected. The DoMenuCommand method has parameters for specifying which menu item was selected and whether the Shift or Control key was pressed as well, as shown here:

```
virtual void DoMenuCommand(MENU_TAG theMenuItem,
                           BOOLEAN isShiftKey,
                           BOOLEAN isControlKey);
```

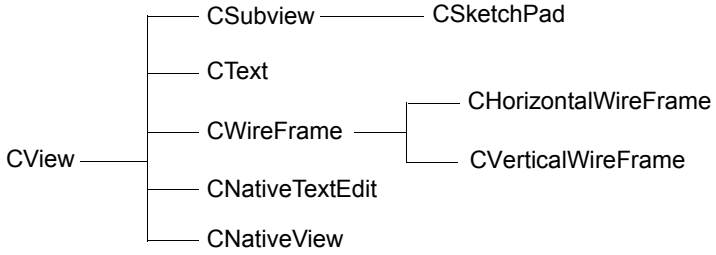
theMenuItem is the menu item number of the command, starting at one.

Setting up menus is another task that can be done at different levels in the XVT-Power++ application framework. Obviously, you can set the menus of a particular window.

See Also: For a detailed discussion of what it means to have a menubar on a window, see Chapter 8, *Object Factory*.
For a discussion of menubar creation, see section 20.3.

21

WIRE FRAMES AND SKETCHPADS



One of the features of XVT-Power++'s view classes is that you can easily make them movable and/or sizable. When you click on a view to select it, a rubberband frame surrounds the view, enabling you to drag the mouse to change the view's size or move it to another screen location. Moving and sizing are handled automatically, and the mechanisms that make these operations possible are all embedded into one class: CWireFrame.

Related to CWireFrame is CSketchPad, which uses the wire frame to sketch shapes within a drawing area on the screen. When a user drags the mouse across a sketchpad's drawing area, a rectangular wire frame appears and stretches with the mouse. When the user releases the mouse button after creating, sizing, or moving a drawing on the sketchpad, the wire frame disappears. The wire frame can also act as a selection box, allowing you to drag out a rubberband frame that selects every object inside it. This chapter discusses useful features of the CWireFrame and CSketchPad classes.

21.1. Wire Frames

The CWireFrame class acts as a friend class. When a view is set to be sizeable or draggable, it instantiates a helping CWireFrame object to enable the appropriate behavior. If an object is movable or sizeable and thus has a helper wire frame, this object is in effect disabled. The object will no longer receive any mouse events. Instead, the mouse events are sent to the wire frame that it owns. For example, a button that normally invokes a dialog box when it is pressed will not do so if it is made moveable. Instead, its wire frame receives the event so that the button can be moved to another location on its enclosing window.

XVT-Power++ takes care of most of the functionality of CWireFrame internally, and you will rarely have to deal with this class directly. However, when developing your application, you may decide to change the behavior or the appearance of a wire frame by deriving a new class from CWireFrame. After instantiating the new class, you would use CView::SetWireFrame method to set a given view's wire frame to the new one instead of the default one.

XVT-Power++ offers more than one wire frame class. CWireFrame has two child classes, CHorizontalWireFrame and CVerticalWireFrame, that allow the user to drag the mouse only in a horizontal or a vertical direction and which serve as examples of how you can override CWireFrame.

See Also: For more information on these classes, see their respective sections in the online *XVT-Power++ Reference*.

21.1.1. Selection and Multiple Selection

XVT-Power++ allows you to select several views and move them around on the screen simultaneously. Clicking on a view to select it causes any previously selected view(s) to become deselected. However, if you press and hold down the Shift key while clicking on a view that own a wire frame, you can select multiple views simultaneously. A group of nested views moves together inside the top-level enclosure, regardless of which of them you may have selected.

21.1.2. DoCommands

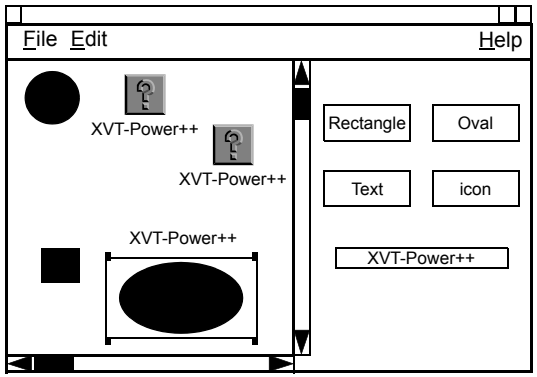
A CWireFrame object generates internal XVT-Power++ commands through the DoCommand mechanism in response to certain events. For example, when a view is selected, a CWireFrame Select command is generated, and when the view is de-selected, a CWireFrame Deselect command is generated. When a view is sized a CWireFrame Size event is generated. Along with the command, the DoCommand takes a pointer to the object that was moved, selected, sized, or so on.

21.1.3. Drawing

Two kinds of drawing occur for a wire frame. Thus, if you want to change the look-and-feel of the wire frame, you must override two CWireFrame drawing methods. First is the drawing of the wire frame itself, which occurs inside DrawWireFrame. Second, CWireFrame draws the wire frame's handles inside DrawFrameGrabbers. These are the handles that appear on the wire frame when it is selected. Dragging these handles with the mouse, you can resize the wire frame's owner.

21.2. Sketchpads

XVT-Power++'s CSketchPad class works in conjunction with CWireFrame to provide the drawing functionality that users typically expect from a graphical user interface. You can use it to draw objects on the fly and to select objects that have been drawn. On the drawing area, you can drag out either a rectangular wire frame or a line wire frame between the MouseDown point and the MouseUp point. The line sketching is done through a class derived from CWireFrame that draws lines rather than rectangles.



XVT-Power++ objects, with COval selected. Each time the user clicks on one of the buttons, a new object appears inside the scroller. The objects can be selected by a mouse click. Once selected, objects can be dragged and sized. Objects are deselected when the user either clicks on the background area or selects another object.

Figure 21.1. Window with a sketchpad embedded inside a scroller

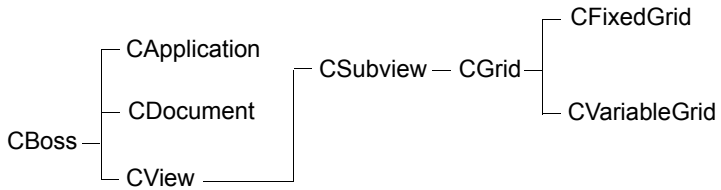
When you release the mouse button and the wire frame disappears, a Docommand is generated. Your application should react to the command by calling one of CSketchPad's sketch event methods, which are described in detail in the online *XVT-Power++ Reference*. For example, you can call GetSketchedRegion to find out the coordinates of the region that was sketched and then use this information as appropriate for your application.

You can also use SetSketchEverywhere to specify whether the sketchpad itself will receive the events or whether the objects drawn in it will receive the events. This is an important point because it determines the basic behavior of the sketchpad. If the sketchpad receives the events, the user can draw overlapping objects and even sketch one object directly on top of another. However, if

the objects themselves are receiving the events, drawing can occur only over the empty space within the sketchpad.

22

GRIDS



As a type of CSubview, XVT-Power++'s grid classes act as enclosures that divide a portion of the screen into rows and columns. The widths and heights of these rows and columns can be set in different ways, depending on whether the grid is fixed or variable. Each intersecting row and column of a grid forms a certain grid cell.

Grids are frequently used in graphical user interfaces: in list boxes, color charts, and panels where it is important that a number of textual or graphical items be precisely placed and aligned. You frequently encounter them in spreadsheets and drawing programs. As you design and develop your application, you will often find that grids are useful and even necessary, whether the end user can see them or not.

Thus, XVT-Power++'s application framework provides three classes that offer a full range of grid functionality. The base class, CGrid, contains a fairly extensive set of methods for manipulating grids and the objects they contain. Two variant classes allow you to create either a grid in which the cells are all the same size or a grid with variable-sized cells.

See Also: For details on each of these classes, consult their respective descriptions in the on-line *XVT-Power++ Reference*.

22.1. Basic Grid Functionality

XVT-Power++'s abstract grid class, CGrid, provides methods for manipulating a grid: inserting and removing objects and placing them in different ways inside their cells, sizing the grid, getting an object from the grid either by specifying a grid location or by specifying an object, and so on. You can turn the lines and columns of a grid on or off to make the grid visible or invisible.

All operations that take cell numbers or row and column numbers have a numbering system starting with zero (0) at the top-left and moving towards the bottom-right in increasing order.

22.1.1. Inserting and Removing Objects

You can insert as many views as you want into a grid cell by calling the Insert method. When you instantiate an object that you want to place within a grid, you must give it the grid object as its enclosure and then call the grid's Insert method, specifying the row and column that will contain the object. If you do not specify a row and column, Insert calculates a row and column location based on the coordinates of the object relative to the grid. You can also insert an object into a grid cell by calling the Replace method, which removes any object already present within the cell and replaces it with the new one.

If the objects nested within a grid are movable and sizable, they exhibit *snapping* behavior. As you drag an object, it snaps from cell to cell. You cannot place it between two rows or columns because it will snap into a row or column.

22.1.2. Placing an Inserted Object Within Its Cell

When you insert a view into a grid cell, the view can be clipped to the cell or the view may just overlay its cell, extending beyond the cell's borders if it is too big to fit inside it. You can define how the view is placed within the cell boundaries: top-left, bottom-right, top-right, bottom-left, or justified. You can even give it an offset from each of the sides.

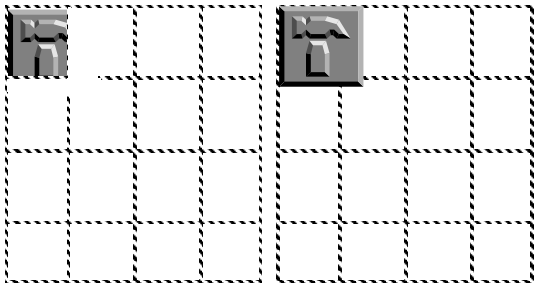


Figure 22.1. A Clipped object and an overlayed object

22.1.3. Sizing a Grid

You can get the size of any grid cell, and you can change the size of a grid in one of two ways:

- Increase or decrease the *number* of grid cells
- Increase or decrease the *size* of a cell or cells while the number or rows and columns remains constant

To determine which of these ways you will size a grid, you must set its sizing policy through CGrid::SetSizingPolicy, which takes a value of either ADJUSTCellSize or ADJUSTCellNumber, as shown here:

```
virtual void SetSizingPolicy(POLICY thePolicy);
```

Through CGrid::AdjustCells, you can either maximize or minimize the size of a grid by finding the largest or the smallest object contained in the grid and making all grid cells that size.

22.2. Fixed and Variable Grids

Within a CFixedGrid object, all of the rows are the same size and all the columns are the same size. Thus, the cells of a fixed grid all have the same dimensions. When you change the dimensions of one cell in a fixed grid, then the size of all other cells in the grid also changes to match the new size. Fixed grids are useful when you want all the items shown in a grid to have equal weight, as in a panel of icons or in a list box. For an example of a fixed grid, see Figure 22.2.

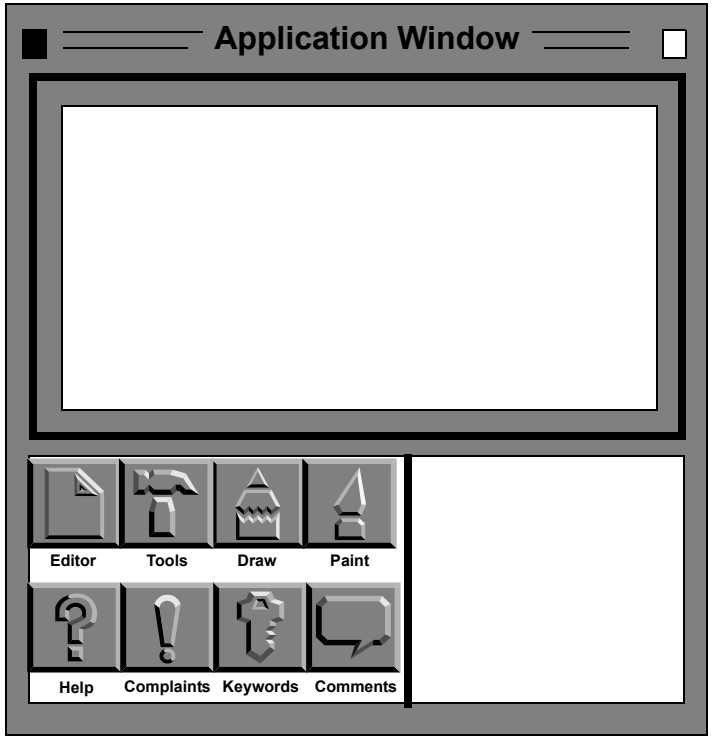


Figure 22.2. Use of a fixed grid, an icon panel

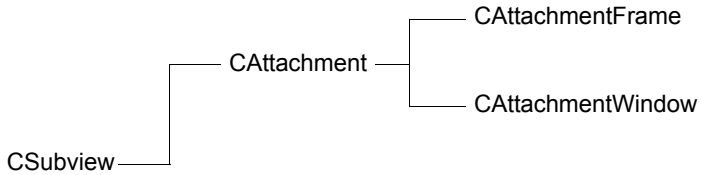
On the other hand, within a CVariableGrid object, the rows and columns have variable widths and heights, as in spreadsheets. Thus, you can set the size of a row or column individually. There is a default width and height for all of the rows and columns, and unless you set the size of a specific row or column, it takes the default size. When you resize an entire variable grid, the default size changes, but any fixed sizes that have been set for specific rows and columns do not change. Figure 22.3 shows an example of a variable grid.

Order No. Item Name	Quantity	Distribution Medium	Price	Total

Figure 22.3. Use of a variable grid, an order form

23

ATTACHMENTS AND PALETTES



23.1. Attachment Classes

XVT-Power++ applications can provide support for attachments and palettes through the use of three flexible framework classes: CAttachment, CAttachmentFrame, and CAttachmentWindow:

CAttachment
: A special class that manages a view object for attachment purposes. CAttachmentFrame works together with CAttachment serving as a special enclosure for views as they are attached to the different sides of a window or enclosure. When you construct an attachment frame, you specify the sides to which views can be attached. Since CAttachmentFrame is a type of CDragSink and CAttachment is a type of CDragSource, the classes automatically support drag-and-drop as a means of attaching views to different locations.

CAttachmentFrame
: A special type of subview that can have views attached to it. Normally you use attachment frames around a window and allow the user to attach or detach tools such as palettes and toolbars. The user can customize the look-and-feel of the application by dragging and attaching these tools to the location of choice.

CAttachmentWindow
: A helper class for CAttachment. It provides the functionality needed for an attachment to appear inside a popup or floating palette window. You should not need to use this class directly since it is created and manipulated automatically by CAttachment. However, you can create derived versions of CAttachmentWindow to customize or extend its behavior.

Figure 23.1 illustrates these three classes and their relationships.

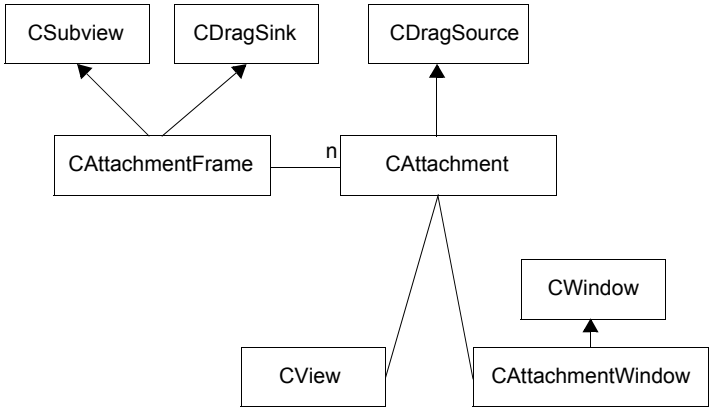


Figure 23.1. Managing attachments to XVT-Power++ views

23.2. Managing Specialized Attachments — Toolbars and Status Bars

One of the responsibilities of CAttachment is to provide a list of possible "fit" sizes for the view it manages. These sizes help the attachment classes figure out how to size and place views as they are dragged and attached to different sides of the frame. While it is not always required, you may opt to derive specialized classes from CAttachment that calculate fit sizes appropriate for different kinds of views. XVT-Power++ provides two such classes: CStatusBarAttachment and CToolBarAttachment.

Usually, you create floating and attachable palettes by defining a toolbar object in XVT-Architect and creating a CToolBarAttachment object to manage it. In addition to this approach, the framework provides one additional class for your convenience:

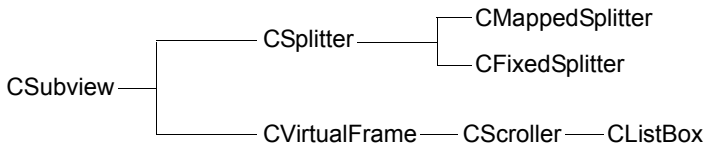
CToolPalette
: This class serves as a logical object definition that allows you to

dynamically build palettes consisting of plain image buttons as well as the added support for image buttons representing nested sub-palettes that can be popped up or torn off.

See Also: Refer to a sample application in **...samples/arch/attach** and its associated XVT-Architect project file to learn more about the behavior of attachments and palettes in a DSC++ application. For more information about palettes, see CToolPalette in the on-line *XVT-Power++ Reference*.

24

SCROLLBARS, SPLITTERS, AND VIRTUAL FRAMES



When you are programming a graphical user interface, what you want to display is often too large to fit into the display area on the screen. The object to be displayed might be a window or just a portion of view inside a window.

To allow you to bring different parts of a large object into view, XVT-Power++ provides virtual frames. A *virtual frame* consists of a large virtual area into which you can insert various types of view objects and a smaller display frame through which only a certain portion of the area is visible at a given time.

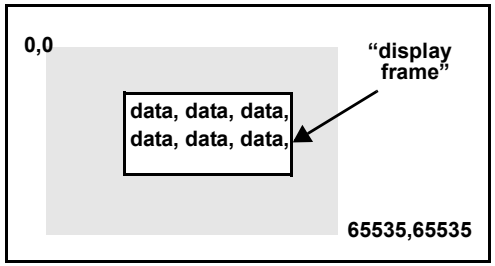


Figure 24.1. Sample virtual frame

24.1. The CVirtualFrame Class

XVT-Power++'s CVirtualFrame class is an abstract class that provides mechanisms for viewing different areas of a virtual frame. For example, you can call the ScrollViews method, which scrolls the virtual area to the right, bottom, top, and so on. However, CVirtualFrame has no mechanism that allows the user to scroll the virtual frame directly. This mechanism must be provided by a derived class such as CScroller, which attaches scrollbars to the display area of the virtual frame. A user manipulates the underlying virtual area by means of these scrollbars to bring different parts of it into view. That is, CScroller automatically calls ScrollViews whenever necessary.

You can write other classes that offer a different kind of scrolling mechanism, perhaps a virtual pane that a user can drag to move the contents of the virtual area. Or you may prefer to create navigation buttons that a user can press to move around inside the virtual area.

24.1.1. Automatic Sizing Capabilities

When you create a CVirtualFrame, you can set the size of the virtual area as well as the size of the visible area. If you do not set the size of the virtual area, then it initially has the same size as the display area.

As a user inserts objects into the virtual area, it expands, if necessary, to ensure that all objects nested within it can fit inside the virtual area even if they do not fit inside the display area. CVirtualFrame contains methods such as EnlargeToFit and ShrinkToFit that automatically size the virtual area to fit a certain number of enclosed views as objects are added or removed.

24.1.2. The Scroll Range

Tied to the automatic sizing capabilities of the virtual frame is the idea of both a vertical and a horizontal scroll range. A *scroll range* specifies the range that is allowed for scrolling a given virtual frame. If a virtual frame has a virtual area that is twice the size of its display area, then its range might start at the top of the display area and end at the bottom of the virtual area to define the entire range of scrolling that is possible. Included with this range is a maximum and minimum position as well as an origin indicating where the display area is currently located relative to the virtual area.

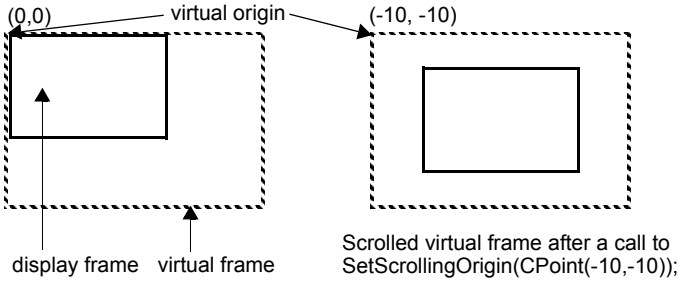
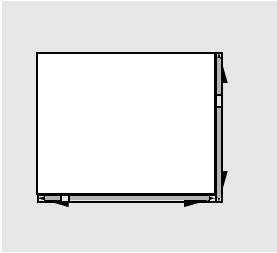


Figure 24.2. Vertical and horizontal scroll range is defined within the virtual frame

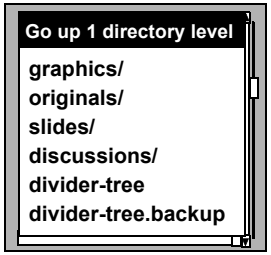
As the virtual frame sizes itself relative to the display area, the scrolling range varies. Thus, CVirtualFrame has two pure virtual methods, called SetHScrollRange and SetVScrollRange, that are called whenever the range changes. These methods must be overridden by derived classes. For example, CScroller overrides these two methods to set a scrollbar's position or thumb proportion depending on the range of scrolling that is possible inside the virtual frame.

24.1.3. The CScroller Class



The CScroller class derives from CVirtualFrame and adds scrollbars to the display area of a virtual frame; these scrollbars allow the user to manipulate the virtual area. A CScroller object can have a horizontal and/or a vertical scrollbar. When users drag or size objects inside a scroller, the contents scroll automatically as the object is dragged beyond the visible borders.

24.1.4. The CListbox Class



The CListbox class derives from CScroller, providing a scrollable box that contains a list of selectable text items. A CListBox object is a composite of the CScroller and CGrid classes. The scroller contains a grid into which the text items are inserted. Of course, CListBox has several utility methods for inserting, removing, selecting, and deselecting the text items in its grid. If you have a large amount of text, we recommend that you insert this text when you initialize the list box, giving it a list of CStringRW objects.

Giving a list box a list of text upon initialization is better than going through a loop calling InsertLine because every time a new line is inserted, the thumb positions are adjusted if necessary. Thus, if you insert a hundred items, the thumb will be adjusted a hundred times, and you may notice some flashing on the screen.

At any time, you can use GetSelectedLine to find out which line in the list box, if any, is selected. This method returns either the number of

the line that is selected or a minus one (–1) if no line is selected. In addition to XVT-Power++'s list box, there is a native list box that is constructed through the native toolkit. The advantage of using CListBox is that you can modify and extend it, perhaps arranging the text items into two or three columns or deriving a list box class that displays picture items instead of CStringRWs.

24.1.5. Use of the Environment

The borders of both CScroller and CListBox are drawn with the pen, and the interiors are painted with the brush. You can set the color, pattern, and width of the pen, and also, the brush color. In addition, keep the following points in mind:

- The text items contained in the list box are drawn in the background color, but, when selected, are drawn in the foreground color. In other words, selected text items are displayed in reverse video.
- The scrollbars inherit the environment of the scroller or listbox. The environment is applied to the scrollbars in the same way as it is applied to NScrollBar.

See Also: For more information about how each part of a control is affected by the different color attributes of CEnvironment, refer to section 15.4.2 on page 15-9.

24.2. Split Windows

The XVT-Power++ framework contains several classes that work together to provide your application with GUI components that can be split into sections with the use of objects known as "splitters." Splitters are frequently used in the user interface of commercially-available software applications.

Depending on the type of interface required by the application, a window with splitters may allow the user to split the window one or more times, adjust the size of the panes, and even merge together previously split panes. For example, commercial text editors or word processors allow you to split a text editing window into two or more sections. Doing this allows you to browse different sections of the same document within one window. Other applications that utilize splitters include file system browsers. Browser windows are commonly divided into panes, one displaying directories, and the other displaying the contents of the currently selected directory.

24.2.1. Types of Splitters

XVT-Power++ supports two types of splitter interfaces. The first is referred to as a *fixed* splitter interface, and is shown in Figure 24.3. Think of a fixed splitter as a user-driven geometry manager. This interface is used when you need to allow the user to adjust or compare adjacent panes of dissimilar information. A classic example of a fixed splitter is a directory/file browser. Another example is a class browser with three panes: one pane displays names of classes, a second pane displays the methods of the selected class, and a third pane displays the source code of the selected method. As these examples illustrate, fixed splitters are “fixed” because the number of panes is fixed by the application, and all the user may do is change their size.

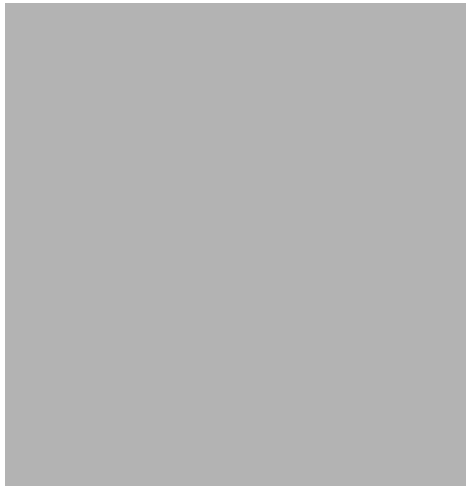


Figure 24.3. Sample fixed splitter

The second type of splitter is referred to as a *mapped* or *dynamic* splitter. These splitters are used when a single view uses a large region to display itself, and you want to give the user the option to split the view in two or more panes. Each pane always displays the same view, but may be scrolled to view different portions of that view. A mapped splitter supports synchronized scrolling of panes and dynamic creation and deletion of panes. A typical example of a mapped splitter is in a word processor where each document window can be split in half. Another example is a spreadsheet application that allows the user to split a large table in half so that they can scroll and view different portions of data at the same time.

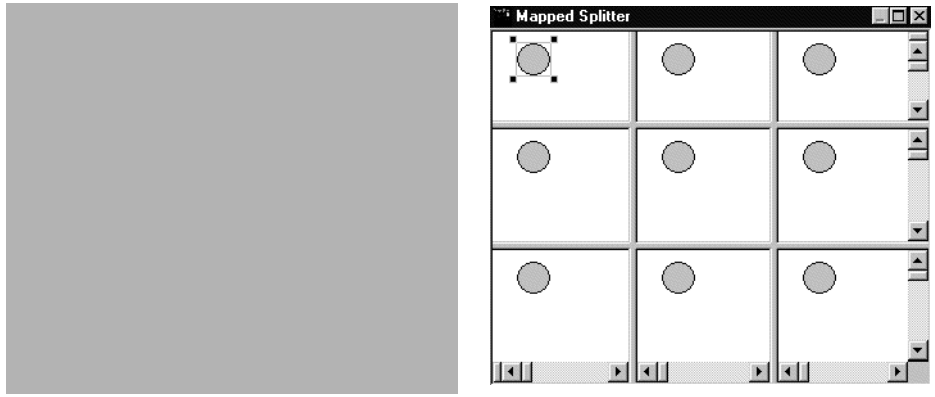


Figure 24.4. Sample mapped splitters (one with multiple panes)

24.2.2. Split Window Classes

Use one or more of the following classes if you wish to use splitters:

CSplitter
: Abstract CSubview derived class that defines the basic interface for creating splittable interfaces.

CFixedSplitter
: A type of CSplitter that supports fixed splitter interfaces.

CHorizontalFixedSplitter
: A type of CFixedSplitter adapted to split horizontally.

CVerticalFixedSplitter
: A type of CFixedSplitter adapted to split vertically.

CMappedSplitter
: A type of CSplitter that supports mapped splitter interfaces.

CSplitBar
: Delegate class that provides services for managing the splitbars used to split an interface.

CHorizontalSplitBar
: A type of CSplitBar adapted to split horizontally.

CVerticalSplitBar
: A type of CSplitBar adapted to split vertically.

CPane
: Delegate class that provides the services for enclosing split interfaces.

CSplitterMouseAgent
: A type of CMouseHandler that responds to mouse events received by splittable interfaces.

CSplitter is derived from CSubview, making it an enclosure capable of enclosing any other view in the XVT-Power++ framework. Figure 24.5 shows the relationship among the various classes.

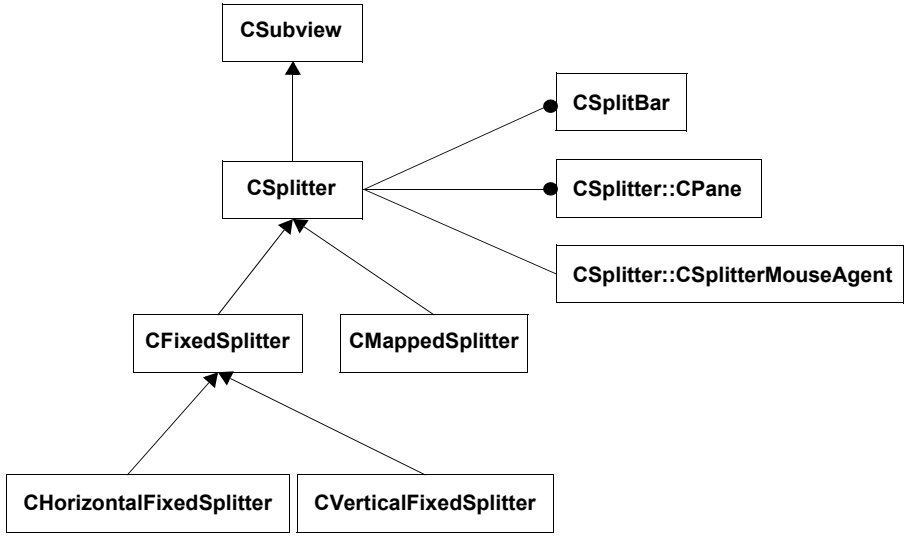


Figure 24.5. Hierarchy of classes used to implement split windows

CSplitter uses several delegate classes to provide important services:

CSplitBar
: A class that manages those little “bars” that divide a split interface into panes. CSplitBar does the work of drawing the bar and managing its movement as it is dragged by the user.

CPane
: A nested class that manages the geometry of individual panes in a split interface.

CSplitterMouseAgent
: A type of CMouseHandler that traps mouse events and sends them to the appropriate split bar, changing cursors as appropriate.

24.2.3. Instantiating a Splitter

When creating a splittable interface, you must first decide if you need a fixed or a mapped splitter. If you need a fixed splitter interface, you instantiate classes derived from CFixedSplitter such as CHorizontalFixedSplitter or CVerticalFixedSplitter. Fixed splitters can be nested inside each other. For example, you can create an interface split vertically into two panes. This is done using a single CVerticalFixedSplitter with two panes. You can then split the top pane horizontally into two panes. You do this by nesting a CHorizontalFixedPane into the top pane.

See Also: For more information about how to nest panes, refer to the **...samples/pwr/fixsplit** sample; this sample program demonstrates how this is done and shows a sample interface where multiple splitting of the interface makes sense.

On the other hand, if what you need is a mapped splitter, then instantiate the CMappedSplitter class. The mapped splitter manages decorations added to the window including: 1) scrollbars that are shared by different panes, and 2) split boxes, which are small areas where the user can click to drag-and-split an interface.

It is up to you (the programmer) to provide the actual implementation for managing the view which is split into separate panes. The mapped splitter provides the application with information about the size of each pane, and initiates events when new panes are created or old ones are deleted.

24.2.3.1. Using Fixed Splitters

In a static layout, you can only use a fixed splitter. That is, to use a fixed splitter, you must know ahead of time how many panes you need as well as their layout (arrangement) relative to each other. The fixed layouts are created by nesting splitters inside other splitters. After creating a top level splitter, you initialize the contents of each pane by calling IFixedSplitter() for each pane. This initialization happens in ascending order (left to right, top to bottom).

Each pane must be initialized with a single view. Of course, the view can be a CSubview object with multiple nested views. Furthermore, the view associated with a pane can be another splitter, which would then split the pane further.

Example: This example shows how to use fixed splitters. The example contains a browser interface with four panes—the window is split vertically once into top and bottom panes, and the top pane is then split horizontally into a left, a middle, and a right pane.

First, create the top-level vertical splitter; this splitter contains two panes:

```
CFixedSplitter* aBrowser = new
    CVerticalFixedSplitter( this,
        GetFrame().GetInflatedRect( -2.0F ), 2,
        SPLITTER_GRAB_H,
        SPLITTER_DRAG_H );
    aBrowser->SetGlue( ALLSTICKY );
    aBrowser->IFixedSplitter();
```

Next, create a horizontal splitter; this splitter contains three panes:

```
CFixedSplitter* aTopRow = new
    CHorizontalFixedSplitter( aBrowser,
        aBrowser->GetPaneCreationSize( 0 ), 3,
        SPLITTER_GRAB_V,
        SPLITTER_DRAG_V );
    aTopRow->SetGlue( ALLSTICKY );
    aTopRow->IFixedSplitter();

// ... Population of horiz splitter omitted ...
```

Now, the top pane of the browser is initialized by nesting into it the horizontal splitter with three panes:

```
    aBrowser->IFixedPane( kMethodImpPane, aTopRow );
```

In a similar way, a scrollable text object is created and nested inside the bottom pane of the browser:

```
NScrollText* aEdit = new NScrollText( aBrowser,
      aBrowser->GetPaneCreationSize( 1 ) );
      aEdit->SetAttribute( TX_BORDER, FALSE );

// turn off border

      aEdit->Append( kMethodImplementation );
      aBrowser->IFixedPane( kMethodImpPane, aEdit );
```

The resultant interface is shown in Figure 24.3 on page 24-6.

See Also: For more details, see the **...samples/pwr/fixsplit** sample program.

24.2.3.2. Using Mapped Splitters

Setting up mapped splitters is simpler than setting up fixed splitters since usually you must only create a single splitter and then let the user dynamically subdivide it into multiple vertical and horizontal panes. Note, however, that the CMappedSplitter class *does* allow you to restrict the number and orientation of panes that are created dynamically. (Good user interface sense dictates that you limit the number of panes, since splitting an interface too much leads to confusion for end users.)

While the setup is simpler, managing the panes in a mapped splitter is much more involved. Unlike the fixed splitter where each pane contains a separate view, mapped splitters must create the illusion that all panes are displaying an image of the same view.

The CMappedSplitter object delegates to you (the programmer) all management of the view being split. This management involves the tracking of a certain model, or data, which is being displayed by the splittable view. As the view is split into panes, you must create new instances of the view for the new panes. These views must all be synchronized to display and modify the same common model that they all share. Basically, you must set up a subject-observer relationship between the views (observer) and their model (subject).

Example: This example shows how to use mapped splitters. The sample contains a splittable scroller that encloses a movable and sizable oval.

The user can split the scroller horizontally or vertically an unlimited number of times. As this is done, the application creates new scrollers and ovals for each pane, and it ensures the state of all ovals is synchronized. This means that as one oval is sized or moved, all other ovals must follow suit. The same types of steps shown here for

splitting an oval can be translated to the splitting of more complex objects such as a table or a text editing object.

First, a mapped splitter is created inside the window:

```
itsMappedSplitter = new CMappedSplitter( this,
        GetFrame().GetInflatedRect( -2.0F ),
//CMappedSplitter::MB_HORIZONTAL_NONE,
//CMappedSplitter::MB_HORIZONTAL_ONE,
        CMappedSplitter::MB_HORIZONTAL_MANY,
//CMappedSplitter::MB_VERTICAL_NONE,
//CMappedSplitter::MB_VERTICAL_ONE,
        CMappedSplitter::MB_VERTICAL_MANY,
                SPLITTER_GRAB_H,
                SPLITTER_DRAG_H,
                SPLITTER_GRAB_V,
                SPLITTER_DRAG_V,
                SPLITTER_GRAB_HV,
                SPLITTER_DRAG_HV,
                TRUE,                   // horiz scrollbar
                TRUE );                 // vert scrollbar
        itsMappedSplitter->SetGlue( ALLSTICKY );
        itsMappedSplitter->IMappedSplitter();
        Invalidate();
```

Next, trap the event generated each time a new pane is created. When the event is received, the application responds by creating a new scroller and placing it inside the frame. Furthermore, a new oval is created inside the scroller, and the oval is added to the list of observers so that it can be updated if the instance of another pane is modified:

```
CSurrogateScroller* aPane = new CSurrogateScroller(
                itsMappedSplitter, thePane->GetFrame() );
        aPane->IScroller( thePane->GetHorScrollBar(),
                thePane->GetVerScrollBar() );
        aPane->SetGlue( ALLSTICKY );
        aPane->Invalidate();
        COval* anOval = new COval( aPane, itsSubject );
        itsObservers.append( anOval );
        itsMappedSplitter->IMappedPane( thePane, 1L, aPane );
```

Similarly, the application must trap the event generated each time a pane is deleted:

```
itsObservers.remove(
                thePane->GetView()->GetSubviews()->at( 0 ) );
```

Finally, the application must trap events generated when the user modifies an oval in any one of the panes. When this happens, all other ovals must be updated accordingly since they all mirror the same data:

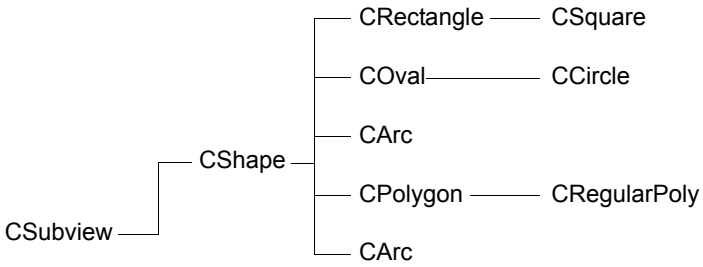
```
CDrawingContext aDC( this );
      aDC.SetLocalRegionedQueue( TRUE );
      RWOrderedIterator doTo( itsObservers );
      while ( doTo() )
      {
            CView* anObserver = (CView*) doTo.key();
            if ( anObserver==theObserver ) continue;
            CRect anOldFrame = anObserver->GetGlobalFrame();
            anObserver->DoSize( itsSubject );
            aDC.QueueInvalidate( anOldFrame );
            aDC.QueueInvalidate(
                  anObserver->GetGlobalFrame() );
      }
aDC.FlushInvalidate();
```

The resultant interface is shown in Figure 24.4 on page 24-7.

See Also: For more details, see the **...samples/pwr/mapsplit** sample program.

25

DRAWING BASIC SHAPES



XVT-Power++ includes a wide range of shape classes that allow you to draw different shapes on the screen. Since all of the shape classes derive from CShape, which in turn derives from CSubview, they have the properties of subviews.

Each shape is an object that can act as an enclosure for other views, receive events, be moved or sized, have its own clipping area, contain its own coordinate system, and so on. You can use shapes not only as decorations for windows but also as buttons or other types of objects that generate commands and allow the user to interact with the application.

This chapter surveys the different kinds of shape objects available in XVT-Power++, considers the resources for drawing them, and offers guidelines for when you can most appropriately use them rather than directly calling the XVT Portability Toolkit's drawing functions.

25.1. Use of CEnvironment for Drawing

The shape classes use CEnvironment for drawing purposes. The border of a shape is drawn with the pen, and its interior is painted with the brush. You can set the color and pattern of both the pen and the brush. Also, you can set the pen width. Several of the various options are shown in Figure 25.1.

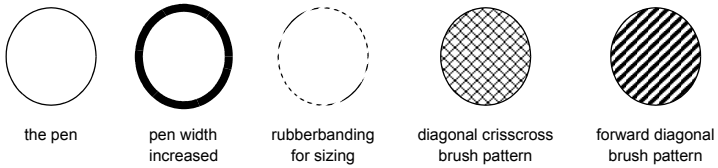


Figure 25.1. Various pen and brush patterns

See Also: For details on the available colors, brush patterns, and pen patterns, see CEnvironment in the *XVT-Power++ Reference*.

25.2. Rectangles and Squares

An interesting feature of the CRectangle class is that rectangles can optionally have rounded corners. Specified values for the width and height of the corners indicate how high and how deep the rounding should be. A sample rectangle is shown in Figure 25.2.

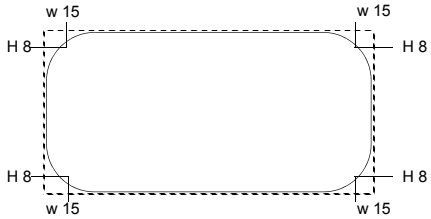


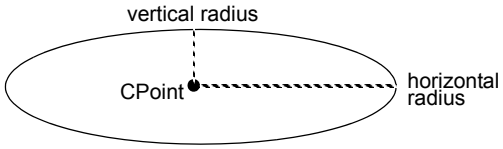
Figure 25.2. Example rectangle

The CSquare class, which derives from CRectangle, creates a square that, like its parent, can have rounded corners. If you size a square and do not specify an equal height and width, the CSquare class takes the average to calculate the square's new size.

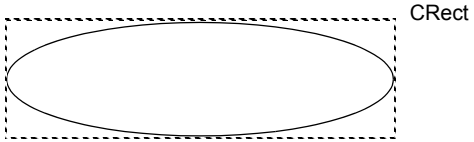
25.3. Ovals and Circles

When you instantiate an object of the COval class to create an oval shape, you can construct it in one of the following two ways:

- Give it a center point and a horizontal and vertical radius



- Specify a region (CRect) within its enclosure that is used to place the oval



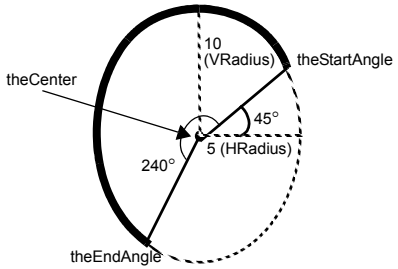
Unlike the COval class from which it derives, CCircle requires only one radius because all points are equidistant from the center.

25.4. Arcs

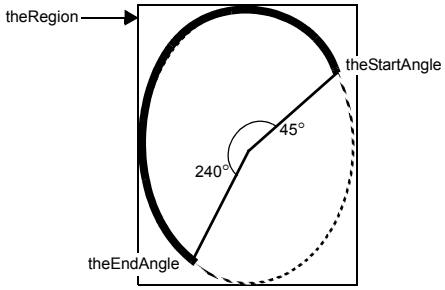
CArc is analogous to the COval class, except that you specify a starting and ending angle for drawing the arc. You can also give the arc an interior fill so that you are drawing a piece of pie rather than an arc.

Like the oval, an arc can be constructed in one of the following two ways; in fact, the arc is drawn counterclockwise along an implicit oval from one given angle to another:

- Give it a center point, horizontal and vertical radii, and starting/ending angles

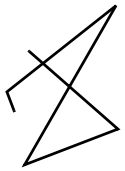


- Specify a region (CRect) within its enclosure that is used to place the arc

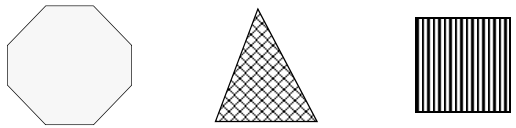


25.5. Polygons

CPolygon does not necessarily create a polygon. Basically, you give this class a set of points, and these points are connected.



Deriving from CPolygon is CRegularPoly, which draws a true polygon, calculating the positions where the points should be connected in order to construct a regular polygon—a triangle, a square, a pentagon, and so on. You provide the number of sides and a certain radius, and perhaps a rotation angle, and CRegularPoly draws the shape for you.



25.6. Lines

XVT-Power++'s CLine class draws a line inside a view enclosure. Like the other shape classes, CLine brings with it all the freight of a CSubview. A line can receive events, generate commands, have different types of stickiness properties (see CGlue), and so on. Lines can also have beginning and/or ending arrows.

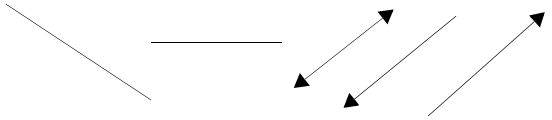


Figure 25.3. Sample lines drawn with CLine

25.7. Drawing Shapes in XVT-Power++

XVT-Power++ users may be unsure about when to use the different shape classes to draw objects on the screen versus overriding the Draw method of a view and using the XVT Portability Toolkit drawing functions to draw lines, squares, ovals, and so on. As noted earlier, XVT-Power++'s shape classes enable you to draw shape objects derived from CSubview with all the properties of CSubview. Obviously, there is more overhead to using one of the shape classes than there is if you just draw a shape directly by calling one of XVT Portability Toolkit's drawing functions, such as the function for drawing a line. In short, you need a rule of thumb to determine when it is appropriate to draw from XVT-Power++ rather than the XVT Portability Toolkit, and vice versa.

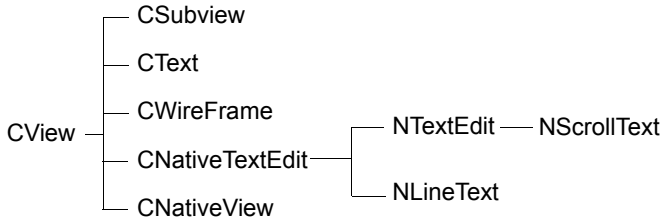
If you are creating a complex view that is derived from CView or CSubview and that contains several kinds of intricate drawings, many lines, and so on, you should use the XVT Portability Toolkit function calls. The drawing process will be much faster and involve much less overhead. If, on the other hand, you want your drawings to have certain behaviors, such as moving/sizing capabilities or the ability to generate commands like a button or nest other objects, you can readily get access to these behaviors

using inherited XVT-Power++ code. These are design decisions that you must make while developing your XVT-Power++ application.

Another design issue is the apparent awkwardness of some XVT-Power++ shapes when functioning as enclosures. For example, you may wonder what could fit inside a line. Actually, the enclosure region of a line includes an imaginary box around the line from one end of it to the other. Thus, the region of the line can easily contain a label or similar object. This is also true of CPolygon objects, where the smallest imaginary box that could enclose the entire object acts as the region defined for that enclosure.

26

TEXT AND TEXT EDITING



XVT-Power++ provides two overall text facilities. One is CText, XVT-Power++'s static text drawing class. The other is a set of native text editing classes that harness the text editing capabilities of the XVT Portability Toolkit. These classes are "native" classes because the implementation of the actual text editing is done by the XVT Portability Toolkit.

XVT-Power++ provides some extra features and encapsulates a lot of the work involved in using the text editing objects, but the native objects are implemented by the XVT Portability Toolkit rather than by XVT-Power++.

Both CText and the native text editing facilities allow you to choose from a variety of font families (Courier, Helvetica, Times, and so on), styles (italics, bold, and so on) and sizes (in points). CText has a SetFont method through which you can set the font of a CText object. The platform on which you are working determines the availability of different fonts. You can also set a font through the CEnvironment class, through the XVT Portability Toolkit xvt_dwin_set_font_* functions, or through a Font menu event.

26.1. CText

CText displays a string of read-only text that is useful for one-line instructions, button names, titles, and so on. When you instantiate a CText object, you give it a CStringRW object, which may or may not be initialized using a string resource ID. When you give a CText object a string, it automatically sizes itself at a designated point on the screen indicating the coordinate at which the line of text starts.

This code creates a textual view, as shown in Figure 26.1:

```
itsMessage = new CText(this, CPoint(100,100),
      "Hello World");
```

CPoint(100,100) is the theTopLeft parameter.It is a coordinate, relative to the CText's enclosure, where the text will be displayed.

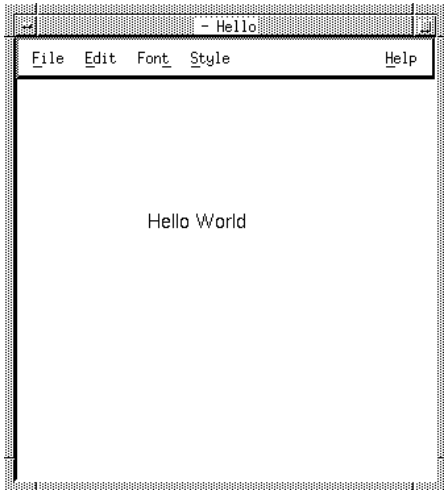


Figure 26.1. Window displaying a textual view

When you select a CText object by sending it a Select message, it inverts its colors. For example, XVT-Power++'s CListBox class provides an object that consists of a set of CText objects displayed as a list inside a grid. When the user clicks on one of the items, it receives a Select message and becomes highlighted.

See Also: For detailed information on using CStringRW resources, refer to the description of CStringRW in the on-line *XVT-Power++ Reference*.

26.2. Native Text Editing Classes

All of the native text editing classes derive from an abstract class called CNativeTextEdit, which has methods for setting/getting, selecting/deselecting, cutting, copying, and pasting text, and many other editing operations. As an abstract class, CNativeTextEdit supplies no means to organize text into lines and paragraphs or to scroll text. These concepts are embodied in its three child classes, discussed in the following section.

26.2.1. NLineText, NTextEdit, and NScrollText

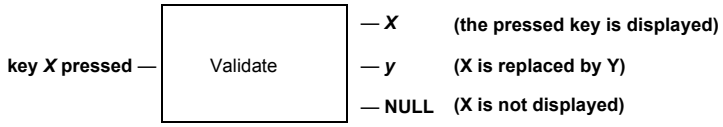
NLineText is the simplest of the native text editing classes. It allows you to create a one-line text editing field. You give it the length of the line, and the height is calculated in terms of the font's size.

NTextEdit is organized into paragraphs, lines, and characters in a line. This class has methods for setting/getting paragraphs or lines, and so on. When the font of an NLineText object (or a CText object) changes, the text box changes size to accommodate the new font. However, if the font size of an NTextEdit object changes, the object does not change size but the text inside of it changes so that more or less of it becomes visible.

NScrollText is a class that is derived from NTextEdit and thus includes all of its paragraph organization features while adding the further feature of scrollbars. All of the scrolling is done automatically, and the scrollbars are updated.

26.2.2. Text Validation

Whenever a text box receives a keyboard event, a CNativeTextEdit Validate method is called, which determines how each character is to be displayed. Validate can opt to display the character, map the character to some other character, or not display it and return NULL, as shown in the following diagram.

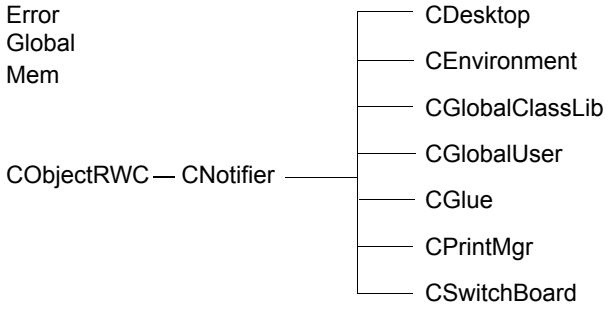


See Also: For more information about CPasswordEdit, a view that provides an easy-to-set-up interface for edit fields that are intended solely for entering passwords, refer to its description in the on-line *XVT-Power++ Reference*.

For more information about different approaches to validating text entered into text fields, refer to section 27.6 on page 27-7.

27

UTILITIES AND DATA STRUCTURES



All of the classes in XVT-Power++'s application framework derive from one common class, CObjectRWC, and they share a number of features: global data, message passing channels, data propagation, and so on. XVT-Power++ contains another set of classes that are independent of the application framework but widely used within it, the utility classes and data structures. This chapter surveys the functionality that is available to you through XVT-Power++'s utilities and data structures.

27.1. Rogue Wave Tools.h Class Library

The utility classes serve as a link between various XVT Portability Toolkit features and XVT-Power++. XVT-Power++ uses the Rogue Wave class library to implement many of its utility classes and data structures. Rogue Wave provides a rich set of collections, data structures, and utility classes that you can take advantage of while using XVT-Power++. The features of Rogue Wave that are available to you through XVT-Power++ include the following:

- Multibyte and wide character strings

- Localized string collation
- Parse and format times, dates, and currency in multiple locales
- Support for multiple time zones and daylight savings rules
- Support for localized messages
- Localized I/O streams
- A complete set of collection classes
- Templates
- Persistence (not implemented for all XVT-Power++ classes)
- B-trees
- Multi-thread safe

Caution: Although the XVT PTK libraries link with thread-safe libraries, XVT needs to "own" the primary thread. Calling XVT functions from any other thread (besides the primary one) is not allowed.

See Also: For more information on Rogue Wave, see the *Tools.h++* manual.

27.1.1. XVT-Power++ and Rogue Wave Collectables

All XVT-Power++ classes, except lightweight classes, now inherit from RWCollectable. A lightweight class is one to which it is generally inexpensive to apply copy semantics. However, for the other classes, the use of XVT-Power++ with Rogue Wave implies certain functionality that may not necessarily be implemented.

Specifically, XVT-Power++ 4.5 does not implement persistence as defined by RWCollectable, and yet XVT-Power++ does not preclude its use within the framework. However, if you attempt to use RWCollectable persistence without appropriately overriding the necessary methods, XVT-Power++ asserts and issues an error.

Also, XVT-Power++ provides run-time type identification that can conflict with the Rogue Wave usage of this feature. In general, it is better to use one mechanism consistently. XVT-Power++ therefore provides guidelines for run-time type identification usage within the framework.

27.1.1.1. Guidelines for Run-Time Type Identification Usage

These are the guidelines for using XVT-Power++ run-time type identification when only an RWCollectable pointer is available:

- Use both macros like this (note the IDs):

```
RWDEFINE_COLLECTABLE(CObjectRWC, CObjectRWCID)
PWRRegisterClass0(CObjectRWC, CObjectRWCID,
        "CObjectRWC")
```

- Use an ID greater than PWRClassIDBase; XVT-Power++ reserves the range from PWRClassIDBase to PWRClassIDMax, so start client programmer classes at PWRClassIDMax + 1
- Use the RTTIRogueWaveToPower macro like this:

```
RWCollectable *aCol = doTo().find(...);
if (RTTIRogueWaveToPower(CPointRWC, aCol))
{
        CPointRWC *aPoint = (CPointRWC*)aCol;
        anOutStream << aPoint->RTTIName;
        ...
}
```

Note: To use the RTTIRogueWaveToPower macro, the ID must be within the valid range. If this is not the case, the result is undefined.

27.2. Managing Global Information

Through CObjectRWC, all classes in XVT-Power++'s application framework have access to global XVT-Power++ information and to global user information, that is, application-specific information.

Two classes manage this global information: CGlobalClassLib and CGlobalUser. Each of these classes is instantiated once in an application. They are accessible through two functions in the CObjectRWC class: GetG() for CGlobalClassLib and GetGU() for CGlobalUser.

27.2.1. The Role of CGlobalClassLib and CGlobalUser

CGlobalClassLib provides an object that is automatically instantiated in XVT-Power++. It cannot be created or instantiated in any other way. It has pointers to the CApplication object and to the CTaskWin object. It supplies information to global flags, for example, about whether the application is terminating. Sometimes it is useful to know whether a destructor for a window is being called because the application is in an exit mode or simply because the window is being closed by a user. CGlobalClassLib also has a flag for text editing boxes, accessed by IsTextEvent() and a global ID count. A method called GetID gives you a unique ID number each time it is called and is used to provide ID numbers for different objects. Basically,

CGlobalClassLib exists to supply information to different classes in the application framework.

Similarly, any global information that you wish to provide for your particular application is made available through CGlobalUser, which is initially empty. You must derive from CGlobalUser so that you can add your own data and methods to it.

See Also: For details on how to set the appropriate pointers and initialize your global information, see CGlobalClassLib, CGlobalUser, and CObjectRWC in the online *XVT-Power++ Reference*.

27.2.2. Managing Window Layout Through the Desktop

Each time a window or a dialog is created in an XVT-Power++ application, the CDesktop object is notified. CDesktop is automatically instantiated within XVT-Power++ through CGlobalClassLib. Of course, you can derive your own desktop, create it, and set it by informing CGlobalClassLib of the user-created desktop. You can also specify a different default desktop creation using the CApplicationFactory interface.

Through CDesktop, you can define a window layout and keep track of the windows in your application. For example, you can find out how many windows are up or get a list of all open windows. You can also find out which window is at the front of the window stack or notify the desktop to put a certain window at the front.

See Also: For details on how the desktop works, see CDesktop in the online *XVT-Power++ Reference*.

27.2.3. Global Definitions

XVT-Power++ also defines the global values and objects in the file **Global.h**. You should not modify this file, but you may want to refer to it occasionally to find out how something is defined. For example, you may need to know what resources XVT-Power++ defines internally or what the XVT-Power++ ID number base is.

See Also: For more information about this file, see the description of Global in the online *XVT-Power++ Reference*.

27.3. Setting Up the Environment

One data structure class that is pervasively used by XVT-Power++'s view classes is CEnvironment. This class contains different kinds of information about the environment: colors, types of pens and brushes, patterns, fonts, drawing modes, and so on. CEnvironment allows you to work in color or monochrome mode and to specify the way colors are used in monochrome mode, such as black for the foreground and white for the background, and so on. At any point, you can set the drawing environment before you do any drawing by calling the XVT Portability Toolkit drawing functions.

See Also: For further information on setting the environment, see section 16.2.2 in this manual, or refer to the description of CEnvironment in the online *XVT-Power++ Reference*.

27.4. Handling XVT Portability Toolkit Events

A utility class that XVT-Power++ instantiates automatically is CSwitchBoard, one of the most heavily used classes. It provides an interface between XVT-Power++ and XVT Portability Toolkit events.

CSwitchBoard has event handlers for dialogs, task windows, windows, and so on. It is in charge of channeling events to the appropriate object. You do not have to initialize the switchboard, and you should not modify it. If you want to respond to a mouse event without notifying the switchboard, for example, you would override methods in CWindow or attach a CMouseHandler object rather than modify the switchboard.

See Also: For more details about how event messages are propagated through the XVT-Power++ framework, see section 15.2 on page 15-3.
For more information about XVT Portability Toolkit events, refer to the *Events* chapter in the *XVT Portability Toolkit Guide*.

27.5. Transferring Data Using the Clipboard

The CClipboard class and its associated streams can be used to put or get data from the native clipboard. This class is a wrapper for the XVT Portability Toolkit's xvt_cb functions and allows text, application (binary), and PICTUREs to be put on, or retrieved from the native clipboard.

27.5.1. Streaming Data into the Clipboard

Any persistent, streamable object (text, XVT PICTURE, or binary data) can be streamed to and from the clipboard. The classes used to handle the streams are:

CClipboardAppIStream
CClipboardAppOStream
CClipboardPictIStream
CClipboardPictOStream
CClipboardTextIStream
CClipboardTextOStream

When using the CClipboard you must use the CClipboard*Stream classes because they are designed specifically for streaming data into the CClipboardBuf. CClipboardBuf is an internal class whose purpose is to contain data that is going to the clipboard. CClipboard is managed by the CClipboardImplementation class, another internal-only class, which handles all the buffers, creates streams, and does other managerial tasks.

Data streamed into a CClipboard gets copied into the CClipboardBuf. When the data is flushed from CClipboardBuf, the data is read (written) from (to) the clipboard. Data is flushed either when a CClipboard of type CB_WRITE is deleted, a CClipboard of type CB_READ is created, or the Flush() method is called.

Note: "Memory" of what application was used to create the data is not supported by the PTK API and thus is not available in the CClipboard classes.

27.5.2. Using Multiple Clipboards

You may create more than one CClipboard object but when you do this, there are a few things to keep in mind. First, the clipboard will adopt the mode of the first opened CClipboard. Thus, if you create a CClipboard in CB_WRITE mode, then all CClipboards created after this must be created with a CB_WRITE mode until all CClipboard objects are deleted.

Second, if you are streaming data into separate (multiple) clipboards, remember that text and application data will be concatenated and PICTURE data will be replaced. It is best to allocate one CClipboard and get a stream for each data type from it.

See Also: To learn more about the CClipboard refer to the online *XVT-Power++ Reference*.

27.6. Field Formatting and Validation

XVT-Power++ supports field formatting and validation, hereafter referred to in this section by the single term, *validation*, with the CValidator class. The primary advantages gained by using CValidator are:

- eliminate character flicker
- provide cut/paste safe formatting

Validation allows your XVT-Power++ applications to be more secure, since confidential passwords and other sensitive data are not flashed to the screen, and thus inadvertently revealed. CValidator provides this additional level of security by trapping and manipulating keyboard events and placing calls to the validation expression parser available in the various Portability Toolkits.

XVT-Power++ supports validation on the following view objects:

- CNativeList and its descendants
- CNativeTextEdit and its descendants
- NEditControl

Advanced users can create their own validators and even change the validators used internally by XVT-Power++. Before attempting either one of these tasks, familiarize yourself with CValidator, CValidatorImplementation, CValidatorFactory, as well as the portable key event hooks (ATTR_KEY_HOOK) used by the PTK.

27.6.1. Validation Basics

The most straightforward way to use validation is to: 1) create a CValidator from a validation expression, and 2) attach a validator to a CView. You can usually combine both steps into a single line of code, as shown in the following example:

```
#include PwrFactoryDef_i
#include CValidatorFactory_i
...
    itsNEditControl->SetValidator( VALIDATOR_FACTORY
        ->ConstructValidator(
        "{Sun,Mon,Tue,Wed,Thu,Fri,Sat}" ) );
```

In this example, the expression consists of a list of words that can be entered using this edit field. The macro VALIDATOR_FACTORY accesses the validator factory associated with the application (for more details, see section 15.5.1). From this factory, the example requests a validator built from the expression "{Sun,Mon,Tue,Wed,Thu,Fri,Sat}" which it passes to the text object—in this case, an NEditControl.

Auto-completion

SetValidator() has a second argument, AutoComplete, that, if omitted, defaults to TRUE. With AutoComplete set to TRUE, if the user types a partial word, such as 'S', the field automatically completes the entry to the first word in the list that matches, 'Sun'. If the user then types an 'a', the match would change to 'Sat'. You can disable auto-completion by setting the argument to FALSE:

```
itsNEditControl->SetValidator( VALIDATOR_FACTORY
    ->ConstructValidator(
    "{Sun,Mon,Tue,Wed,Thu,Fri,Sat}", FALSE ) );
```

When auto-completion is disabled, the end user must type the complete string before a match is made.

See Also: For more information on the syntax that you can use in validation expressions, refer to the description for the function xvt_format_create in the online *XVT Portability Toolkit Reference*.

27.6.2. Writing Your Own Validators

CValidator is a proxy class that contains a pointer to another class, CValidatorImplementation, that does the actual validation "work." The TestMatch() method takes a string and returns TRUE only if the string is acceptable to your validation mechanism. The FormatString() method takes a string and a selection range and is called each time the user modifies the entry field.

27.6.2.1. Customizing a Validator

To implement your own validation routines, subclass CValidatorImplementation and override the TestMatch() and FormatString() methods. For example, your overridden version could modify the input string and return the modified string (instead of returning TRUE), or change the selection range.

To use your customized validator, you must create the validator implementation, add a validator (CValidator), and pass the validator to an edit field, as shown in the following example:

```
#include PwrFactoryDef_i
#include CValidatorFactory_i
      itsNEditControl->SetValidator( CValidator(
            new MyValidatorImplementation(
            *VALIDATOR_FACTORY) ) );
```

Note: Note that your validator operates independently from the validator factory; the factory reference shown in the code above is simply a placeholder.

27.6.2.2. Substituting Your Own Validators

The validator factory enables you to easily replace the factory that XVT-Power++ uses to create validators. Though validation is not currently supported directly by XVT-Architect, the factory approach could be useful in the future if you are using a form of automatic code generation, e.g., from XVT-Architect, that uses the validator factory to issue validators.

If you wish to use your own validator in these situations, subclass CValidatorFactory, override the ConstructValidator() method, subclass CApplication, and install your factory in the InstallFactories() method of the application.

See Also: The file **...include/CValidatorFactory.h** contains more details.

27.6.3. Other Approaches to Validation

Two other approaches allow you to supplement (or circumvent) your use of CValidator for validation:

NEditControl

If your view is a native view, such as NEditControl, you can set a validator at the PTK level by calling the function xvt_vobj_set_formatter. The first argument (in the call to xvt_vobj_set_formatter) is an XVT control whose window ID you get from the native view by calling GetXVTWindow() on that view.

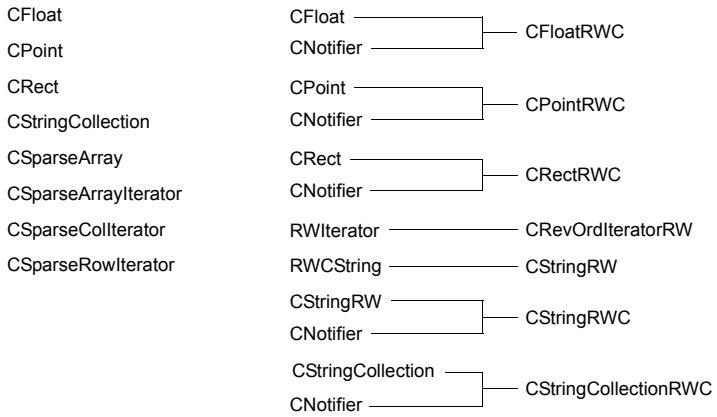
NEditControl includes a subclass, CPasswordEdit, which is a view that provides an easy-to-set-up interface for edit fields that are intended solely for entering passwords.

NLineText

NLineText accepts the validation expression directly as an argument to the method SetValidation(). However, NLineText does not provide native look-and-feel, and when using complex validation syntax, you may see flashing on very slow equipment. This method only works with NLineText, which is *not* a native view type.

See Also: For more details about formatters for native views, see the description of the function xvt_vobj_set_formatter in the *XVT Portability Toolkit Reference*.
For more details about entering passwords, see the description of CPasswordEdit in the online *XVT-Power++ Reference*.

27.7. Data Structures



XVT-Power++ uses the Rogue Wave class library, which provides a rich set of collections, data structures and utility classes. This section summarizes some information about Rogue Wave and how it is used in XVT-Power++.

See Also: For more information on the classes discussed in this section, see the online *XVT-Power++ Reference*.
For more information on Rogue Wave classes, see the online *Tools.h++* manual.

27.7.1. Collectables

Rogue Wave assumes that objects referenced in its collections all inherit from RWCollectable. RWCollectable defines a virtual interface that the collections and utility classes use to determine identity, ordering, equality and for persistence. The root of the XVT-Power++ tree, CObjectRWC, inherits from RWCollectable and uses the default methods for all attributes except persistence, which it disallows (persistence will be added in a future XVT-Power++ release).

See Also: For more information on using Rogue Wave collectables with XVT-Power++, see section 27.1.1.

27.7.1.1. Temporary Collectables

For some utility classes that are frequently used or that are used as temporary instances, direct inheritance from RWCollectable is

undesirable because of the construction expense involved in creating the base classes and initializing the vtable. In the XVT-Power++ hierarchy, CPoint, CRect, and CStringRW are examples of these classes. To make these classes acceptable for use in Rogue Wave collections, it is necessary to provide a collectable version.
Rogue Wave does this by creating a composite class via multiple inheritance from the utility class and RWCollectable. XVT-Power++ uses the utility class and a class derived from CObjectRWC, typically CNotifier.

The pointer-based implementation of the collections and the requirement that a contained object must be derived from a particular class means that you must be careful when creating an object to add to a collection.

Caution: In particular, automatic data (i.e., objects created on the stack without a "new" operator) cannot safely be added to collections that have a lifetime longer than the created object. This means that if a collection is passed it, anything added to it must be created with the new operator.

27.7.1.2. Dictionary Collections for Collectables

Dictionary collections allow arbitrary associations between any two collectable objects. Since it is often desirable to associate a simple data type, like an integer, with a more complex type, Rogue Wave provides collectable versions of strings (RWCollectableString) and integers (RWCollectableInt). XVT-Power++ provides collectable versions of rectangles (CRectRWC), points (CPointRWC), and floats (CFloatRWC).

See Also: If you need to add collectable semantics to a simple type, you can look at the XVT-Power++ implementations for example code.

27.7.2. Collections

RWOrdered is the collection class that XVT-Power++ uses internally for most purposes, and it is returned and accepted by the XVT-Power++ API. RWOrdered was selected because it is the fastest implementation for traversal (i.e., examining or operating upon each contained object in sequence), which is the most common operation performed by XVT-Power++. It also offers reasonable performance for insertion and ordering semantics—objects are accessed in the same order they are added.

RWOrdered collections start with a default capacity. Addition of objects beyond this capacity cause a relatively expensive memory

allocation operation to be performed. To avoid this problem, use the RWOrdered::RWOrdered(size_t) flavor of constructor to give an approximate size when this is known.

27.7.2.1. Converting RWOrdered into a Sorted Collection

If you have a RWOrdered, or any other collection type, and need a sorted collection, you can generate a RWBinaryTree collection by using the RWCollection::asSortedCollection method. Since only shallow copies are performed, this is relatively inexpensive. However, you should only do this on collections whose contents have meaningful sequencing (e.g., CStringRWC does, CSubview does not). In general, objects that inherit from CObjectRWC do not have meaningful sequence semantics.

27.7.2.2. Iterators

Iterators are helper classes that automate the traversal of collections. Some complex collections require iterators while others can be traversed without them.

Caution: It is dangerous to modify a collection while iterating on that collection. This comes up in the common task of traversing a collection and removing objects that meet a particular test. One solution to this problem is to make a helper collection to which positive objects can be added. That is, at the end of the iteration, loop over the helper and remove its objects from the original and then call ClearAndDelete on the helper collection.

27.7.3. Strings

XVT-Power++'s CStringRW provides string functionality. It is derived from RWCString, which provides a rich set of string manipulation operations and is assisted by the RWTokenizer and RWSubstring helper classes. CStringRW adds the ability to construct a string object from an XVT Portability Toolkit resource, and CStringRWC is the collectable variant.

CStringCollectionRWC is derived from CStringCollection, which maintains a list of CStringRWC objects. CStringCollectionRWC provides constructors to initialize an ordered list of CStringRWC objects from a resource ID and conversion operators to generate an XVT Portability Toolkit SLIST from such an object.

Rogue Wave strings use “copy on write” semantics, so they can be copied and assigned very efficiently.

See Also: For more information on RWCString and RWOrdered, see the *Tools.h++* Rogue Wave manual.
For more information on CStringCollection and CStringCollectionRWC, see their descriptions in the online *XVT-Power++ Reference*.

27.7.4. The Coordinate System: CPoint, CRect, and CUnits

XVT-Power++'s CPoint class allows you to manage a coordinate system point in different possible types of units that you choose through the CUnits class. Similarly, CRect allows you to manage a rectangular set of coordinates in different possible types of units.

See Also: For a full discussion of these classes, see section 16.2.4.

27.8. Checking For Errors

XVT-Power++'s **Error.h** file defines a macro called PwrAssert that is used throughout XVT-Power++. It allows you to assert that certain things are true or false and to get an XVT Portability Toolkit error if something is wrong.

See Also: For a description of PwrAssert, see Error in the online *XVT-Power++ Reference*.

28

RESOURCES AND URL

This chapter explains how XVT-Power++ supports XVT Portability Toolkit's Universal Resource Language (URL). You can use these resources to create XVT-Power++ objects and classes. In addition, XVT-Power++ supplies helper classes for the efficient loading of monolithic resources.

Note: XVT-Architect, the visual application builder that is part of DSC++, uses URL resources for only a few things. More frequently, XVT-Power++ programs use these graphical and textual resources.

28.1. Why Use Resources?

URL resources are specifications for menus, dialogs, windows, strings, images, and fonts that are kept in a small, read-only database located outside your application's runtime address space. Resources do such things as:

- Set object attributes, such as those that determine the size, position, and alignment of windows, dialogs, and controls
- Establish an object's default appearance, such as initializing its label or title, and also controlling whether it is initially enabled or disabled
- Configure the menubars and menus for application windows

When your application needs a resource, the application requests the resource by an ID number. XVT or the native window system brings the resource into memory so it can be accessed. This saves space at runtime and makes it possible to access multiple resource files without recompiling.

For example, externalized strings and graphics allow your application to be run in more than one locale using localized resources, if that is a requirement for your organization. XVT

provides pre-translated resources for five languages: Japanese, Italian, French, German, and English.

Most programmers find the Universal Resource Language (URL) easy to learn. Since the URL code is portable, you only need to define your resources once. However, teaching you to use URL is beyond the scope of this chapter.

See Also: For more information on XVT Portability Toolkit resources and URL, see the "Resources and URL" chapter of the *XVT Portability Toolkit Guide*.

28.1.1. Resources in XVT-Power++

XVT-Power++ supports URL resources as defined by the XVT Portability Toolkit. URL resources are all read-only with respect to the Toolkit's API calls. However, writing to an alternate source is supported by some objects, such as images. XVT-Power++ supports all of the XVT Portability Toolkit resources, including:

- controls
- dialogs
- fonts
- icons
- images
- menu accelerators
- menus
- strings
- windows

Although XVT-Power++ does not provide abstractions for all of the XVT Portability Toolkit resource API calls, it does not preclude their use. For example, you can use window resource data (e.g., xvt_res_get_win_data).

28.1.2. X Window System Resources

For applications running on X platforms, the resources must be coded separately in CMyResourceFile.cxx. Use CMyResourceFile.cxx to indicate what resources your X-based application will use.

See Also: For step-by-step information on how to build an icon or cursor resource under X, see the platform-specific book, *XVT Platform-Specific Book for Motif*.

28.2. Creating Objects from Resources

XVT-Power++ creates objects from resources using a class constructor of the following format:

```
CResourceClass *aResource = new CResourceClass(itsEnc,
    RES_ID);
```

28.2.1. Using XVT-Power++ Classes

The following list contains XVT-Power++ classes that may be used to construct objects from resources:

`CStringRW::CStringRW(RESOURCE_ID);`
: Creates a resource string.

`CStringCollectionRWC::CStringCollectionRWC(RES_START, RES_END);`
: Creates a collection of strings from a resource.

`CWindow::CWindow(itsDoc, WIN_101, TRUE);`
: Creates a window and all its native views. Passing FALSE for the last parameter creates only the window. You can make the created window modal by a calling CModalWindow::DoModal.

`CDialog::CDialog(DLG_101);`
: Creates a dialog and all its controls. You can make the created dialog modal by a calling CDialog::DoModal.

`CFont::CFont(RES_FONT_1);`
: Creates a font object from a resource. This constructor is useful when you want to maintain a set of portable fonts.

`CImage::CImage(RES_IMAGE_2);`
: Creates an image from a resource.

`CMenuBar::CMenuBar(itsCWindow, MENU_BAR_RID);`
: Populates a CMenuBar from resources.

Note: CDialog and CModalDialog are supported only for the purpose of backward compatibility. You should use the CWindow equivalents in their place. The view hierarchy is not supported by dialogs.

28.3. Creating CNativeView-derived Classes

You can create the following CNativeView-derived classes from resources:

```
NButton::NButton(itsEncl, WIN_101,
    WIN_101_BUTTON_1);

NIcon::NIcon(itsEncl, WIN_101, WIN_101_ICON_1);

NScrollBar::NScrollBar(itsEncl, WIN_101,
    WIN_101_SBAR_1);

NEditControl::NEditControl(itsEncl, WIN_101,
    WIN_101_EDIT_1);

NText::NText(itsEncl, WIN_101, WIN_101_TEXT_1);

NListBox::NListBox(itsEncl, WIN_101,
    WIN_101_LBOX_1);

NListEdit::NListEdit(itsEncl, WIN_101,
    WIN_101_LEDIT_1);

NListButton::NListButton(itsEncl, WIN_101,
    WIN_101_LBUTTON_1);

NCheckBox::NCheckBox(itsEncl, WIN_101,
    WIN_101_CHECKBOX_1);

NRadioButttton::NRadioButton(itsEncl, WIN_101,
    WIN_101_RADIOBUTTON_1);

NScrollText::NScrollText(itsEncl, WIN_101,
    WIN_101_SCROLLTEXT_1);
```

Caution: It is important that you use these classes correctly. For details on the correct usage of these classes, read the following section on loading resources.

28.4. Optimizing the Loading of Resources

CResource, CResourceMgr, CResourceWindow, and CResourceMenu are helper classes used by some resources for the efficient loading of monolithic resources. When reading windows and menubars, the XVT Portability Toolkit API allows the reading of one large structure, which must then be parsed for contained objects (e.g., menu items and controls). Given that it is expensive to load, parse, and free these types of resources, XVT-Power++ supplies both the CResource pure abstract class and the CResourceMgr class to maximize this process.

28.4.1. Window Resources

CResource supplies an interface for the holding and releasing of resources. When the resource CWindow constructor is called, this is done. In this case, CResource creates the resource and holds it. It keeps the monolithic structure in memory, in whatever form is conducive to the parsing of those resources. The resource remains held until all resource items (sub-objects) are parsed. For the CWindow example, these items are controls. Once all the controls are created, the resource is released, and the memory freed.

Example: When a contained object is created from resources its container is held (if it is not already held), and the object's attributes are parsed and used to initialize the object. When reading many objects from the same container, it is more efficient to hold the container first, read all the contained objects, and then release the container, like this:

```
// First hold resource:
CResourceWindow *theRes = G->GetResourceMgr()->Find(
      theContainerId);
assert(theRes != NULL);
theRes->Hold();

// Create views:
new NButton(theScr, WIN_101, kButton101_1);
new NButton(theScr, WIN_101, kOkButton);
new NListEdit(theScr, WIN_101, kEdit101_1);

// Release held resource:
theRes->Release();
```

28.4.2. Using CResourceItems

XVT-Power++ provides a convenience class, CResourceItems, to hide the hold/release protocol through object construction and destruction, like this:

```
{
CResourceItems theHeldRes(theContainerId);
new NButton(theScr, WIN_101, kButton101_1);
new NButton(theScr, WIN_101, kOkButton);
new NListEdit(theScr, WIN_101, kEdit101_1);
}
```

28.4.3. Iterating Held Resources

CResource also provides a mechanism to iterate over a held resource. Note that the iteration supported by CResource does *not* allow for multiple clients to iterate or for stacked iterations.

```
void CreateNButtons(CWindow *theWin, long theContainerId)
{
    CResourceItems theRes(theContainerId);

    // Iterate through resources
    //creating only the push buttons:
    long anId = theRes.First();  // First item is window

    anId = theRes.Next();
    while (anId)
    {
        // Create resource button:
        if (theRes.GetType(anId)==WC_PUSHBUTTON)
            new NButton(theWin, theContainerId, anId);

        // Iterate to next item:
        anId = theRes.Next();
    }
}
```

See Also: For more details on these helper classes, see their individual descriptions in the online *XVT-Power++ Reference*.

28.5. Resources for Internationalized Applications

When writing an internationalized/localized XVT application, resources become an integral aspect of the application design and your software development process. In addition to other attributes, your localized application must notify the PTK that it is multibyte aware. The application does this by setting the value of the attribute ATTR_MULTIBYTE_AWARE to TRUE.

Select the resource file to be used by setting the value of the attribute ATTR_RESOURCE_FILENAME. The application name and the task window title are localized by setting the attributes ATTR_APPL_NAME_RID and ATTR_TASKWIN_TITLE_RID.

The application name and task window title in the XVT_CONFIG structure passed to xvt_app_create are overridden by localized strings obtained from resources. ATTR_APPL_NAME_RID and ATTR_TASKWIN_TITLE_RID set the resources IDs from which to obtain these strings.

If you have created a locale-specific error file (**ERRCODES.TXT**), use the ATTR_ERRMSG_FILENAME attribute (as you would the ATTR_RESOURCE_FILENAME attribute), to override the default filename before xvt_app_create is called.

See Also: For more information on binding resource files to your application, refer to the *Resources and URL* chapter of the *XVT Portability Toolkit Guide*.

29

DATA PROPAGATION

Automatic Data Propagation (ADP) is a powerful feature of XVT-Power++. Using the Model-View-Controller approach, ADP automatically propagates a change of data from objects to other objects.

ADP lets you build complex models and have the associated views updated automatically. For example, a spreadsheet and a graph can display the same information in two different ways. When the information is modified in one window through the ADP model, that change will automatically be reflected in the other window.

ADP is very flexible in its design. Models can depend on other models, and do not have to be GUI models. For example, a string could depend on other strings.

29.1. How to Use ADP

XVT-Power++ uses these four basic classes to provide Automatic Data Propagation:

CModel

CControllerMgr (CController manager)

CController

CNotifier

There are two types of notifiers, *providers* and *dependents*. Providers have the ability to request a change to the model data. Dependents are notified of changes in the model data. All objects are derived from CNotifier, so any object can be a provider and/or a dependent.

To use ADP, first create a model representing the data. Register that model with a CController. Then register the CController with the CControllerMgr. (See Figure 29.1.)

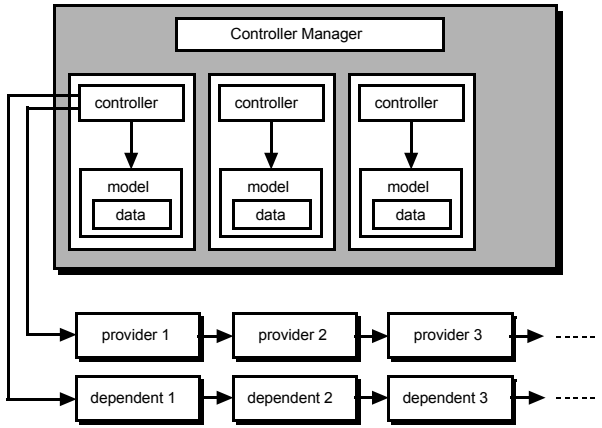


Figure 29.1. Using ADP

When an object is interested in the data associated with a model, it adds itself as a dependent of the CController where that model is registered. When the object requests a model change (for instance, if the end user modifies the representation of the data), it notifies the CController. If the CController allows the model to change, it notifies all its dependents of the change. This is the basic principle behind ADP.

29.2. ADP Classes

Let's review in more detail each class that contributes to the ADP mechanism.

29.2.1. CModel Class

CModel is an abstract class. When you are using ADP, override this class and specify your own model. This is a place to specify which data set will be subject to ADP.

To specify the ways the model can change, you override the pure virtual Change(command, model) method. A model might change in several ways. For instance, only part of the model might be altered by the end user. Each type of model change is specified under a certain protocol, which is called a *command*. A command is defined as a long.

From each command:

- The model knows which part of it has been modified
- The model extrapolates new information from the model passed to the Change method and updates its data accordingly
- A dependent extracts the model information and modifies its parts accordingly (upon a DoUpdateModel call)

The command mechanism is a way to allow partial change of the model without having to update the entire model. For instance, if a window with edit controls is a dependent of a certain model and only one edit control changes, the window updates only the edit control content.

A model can be made of several models. The model passed to the Change method does not have to be of the same type as the entire model. The model granularity is under the user's control and allows flexibility for partial changes of the model data.

29.2.2. CControllerMgr Class

CControllerMgr manages CControllers, and assigns CController IDs.

CControllerMgr is automatically instantiated in CGlobalClassLib and can be accessed through the global G pointer (in CObjectRWC) G->GetControllerMgr(). XVT-Power++ allows only one instance of this class. Its copy constructor is protected; it does not allow a CControllerMgr to be assigned or copied.

The CControllerMgr uses its Insert method to automatically register a new CController in its list of CControllers. The CControllerMgr can also remove a CController from its list using its Remove method. Its Find method returns a CController pointer based on a CController ID.

29.2.3. CController Class

CController is responsible for managing a list of dependents and a list of providers, and provides access to its model. There is one CController per model. However, that model can be made of several models.

When an update is passed to CController, the method CController::DoChange(provider, command, new model) first checks its list of providers, because not every object has the ability to change the model. CController then calls the method CModel::Change(command, new model) to alter the model data. Finally, if the Change method returns True (which means the model has indeed changed), CController goes through its list of dependents and calls their respective DoUpdateModel methods.

29.2.4. CNotifier Class

CNotifier is a base class that provides the "wide interface" to request model changes and to update dependents of a model change. Every dependent and every provider is derived from CNotifier.

Its DoChangeModel(CController id, provider, command, new model) method allows a provider to request a model change. DoChangeModel first finds the CController using the CControllerMgr::Find method. Once identified, the CController verifies that the provider passed as an argument belongs to its list of providers. If so, it allows the model to change and then updates the dependents through the "wide interface" call to the dependent's DoUpdateModel method.

DoUpdateModel method is called by a CController for each of its dependents when a physical model changes. It is up to each dependent to override this method and update itself correctly when a model change occurs.

CNotifier maintains these two RWOrdered pointers:

- isProviderControllers is a list of CControllers of which the CNotifier is a provider
- isDependentControllers is a list of CControllers of which the CNotifier is a dependent

When a CNotifier or a CController is removed, XVT-Power++ automatically takes care of updating these lists to reflect the change.

29.3. Example

Let's look at a simple example of how ADP is used.

While you are reading this example, keep in mind that there are many ways ADP can be implemented. XVT-Power++ does not force any specific rules on how to use ADP. The instantiation of ADP objects can be done at several locations in your code. In our example we have the document instantiate the CController, but the CController could also be instantiated at other places in the framework. The example provided shows how to use ADP to display the same information in different views, but ADP does not require views or any specific XVT-Power++ objects.

Suppose we have two windows, a spreadsheet and a graph, which share the same information. In XVT-Power++, they share the same *document*.

Note: The following example is similar to the one provided on the distribution media (Model example).

29.3.1. Setting up the Document for ADP

Following the usual XVT-Power++ paradigm, we first instantiate a document object. In this example, the document instantiates a model representing some piece of data. The data is passed to the model constructor in order to fill the model with the initial information. To do this, in the document constructor we instantiate an object derived from CModel. We then provide a protocol through a series of commands to determine which parts of the model might change. Each command defines a specific change to the model. For example, if our model is a spreadsheet, the command mechanism might allow us to pass just the modified cells instead of the whole spreadsheet. We now have a working model.

Next we instantiate a CController object. The CController object passes the model as the first argument to the CController constructor which registers the model with that CController. Calling the CController constructor automatically adds that CController to the CControllerMgr's list of CControllers.

So far, we have built the data that will be automatically updated using ADP, and we have created and registered our CController.

29.3.2. Setting up the Views for ADP

Now we need to create the spreadsheet and the graph windows.

29.3.2.1. Setting up a Provider View

We instantiate the spreadsheet window. The window constructor then builds the spreadsheet. In order to display the model data, the spreadsheet will need to query the model. The CController queries the model using its GetModel method. To get to the CController, we use:

```
G->GetControllerMgr()->Find(CController id)
```

where G represents the CGlobalClassLib pointer.

Then we call the "wide interface" spreadsheet's DoUpdateModel method, which is inherited from CNotifier, and pass it the queried model. One of the commands passed to DoUpdateModel might specify that this is the first time we are querying the model; therefore, it should look at all the information provided in the model parameter. This way we can use the same method for the initial model query and subsequent model changes.

Once the spreadsheet queries the model data and sets the information in its data members, it needs to register itself as a

dependent to the model's CController so that it can be notified of any changes in the model. We do this by calling the following:

```
G->GetControllerMgr()->Find(CController id)->
    AddDependent(spreadsheet object)
```

Now we have a spreadsheet that acts as a dependent of the model data.

For this example, suppose we want the spreadsheet to be able to request model changes. That is, the spreadsheet will be a provider of the CController. We do this by calling the following:

```
G->GetControllerMgr()->Find(CController id)->
    AddProvider(spreadsheet object)
```

29.3.2.2. Setting up a Dependent View

Now let's create the graph window. The window constructor queries the model and passes it to the DoUpdateModel(command, model) method, just like the spreadsheet. The window constructor then draws the window content based on the queried model. The command passed to DoUpdateModel would specify that all data is new, so the entire model is passed.

We then register the graph window as a dependent of the CController, just like the spreadsheet. In this example, we do not want the graph to be able to request model change so we do not register it as a provider to the CController.

In summary, the spreadsheet can request a model change and is notified of any model data changes. It is both a provider and a dependent. The graph window does not have the ability to request a model change, but it is notified of model data changes. It is only a dependent.

29.3.2.3. How ADP Looks to the End User

Now let's see what happens at runtime.

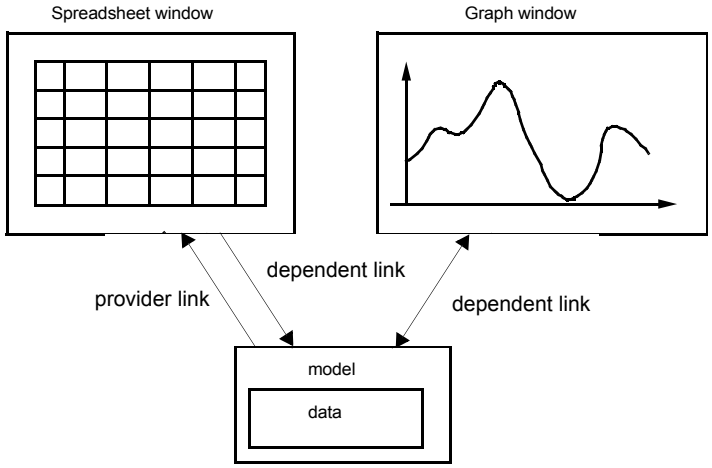


Figure 29.2. ADP at runtime

If a spreadsheet cell is modified by the end user at runtime, the spreadsheet requests a model change. It calls its DoChangeModel method passing a CController ID, a provider (itself), a command (which specifies the protocol under which the model has been changed), and the new model (the new data). When the information in its window changes, the spreadsheet creates a new model (which does not have to be of the type of the CController model). DoChangeModel calls the CController::DoChange method, which in turn checks if the provider is in the list of its providers. If this succeeds, it calls the model's Change(command, new model) method, passing the command and a model that contains information about the new data. Based on this command, the model extracts information from the model passed to the Change method and updates its data accordingly. Finally, the CController iterates through the list of dependents to update them with the new model data.

In this example, the graph window is on the list of dependents, so the graph window's DoUpdateModel method is called. DoUpdateModel will redraw the graph window contents based on the new model information.

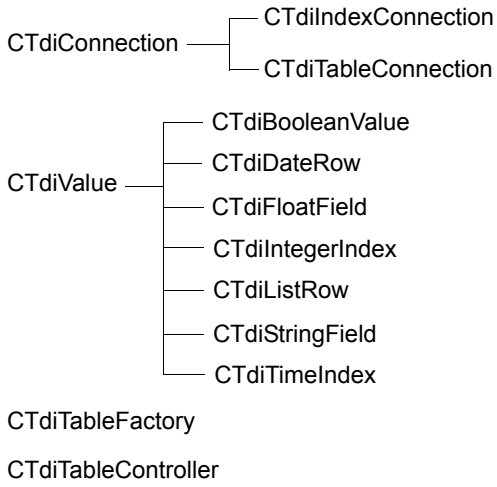
29.4. Automatic Data Propagation Key Points

Keep in mind the following key points about ADP:

- The model data is completely encapsulated in the CModel object. Only a CModel-derived object can physically modify the data set. Providers provide a new model to the model CController, which invokes the model's Change method.
- Every object is derived from CNotifier. Therefore, every object has the ability to be a dependent and/or a provider, including any CView-derived objects. Consequently, ADP allows data to be dependent upon other data.
- CController is also derived from CNotifier. This allows controllers to depend on other controllers, and makes it possible to derive complex controllers. For example, you might need to list some information that must be verified before it can be used. A CController could be derived from CController to add auxiliary lists. To implement a group of privileges, you can combine providers in a group and then designate the group as a provider. The CControllerMgr can register any object derived from a CController.
- Objects can create several different CControllers. There is no restriction on the number of CControllers that a single object can create.
- A CNotifier object can be a dependent of one or more CControllers. In the spreadsheet example, every cell could be associated with a separate CController and have a list of dependents.
- Although each CController controls only one model, ADP allows for nesting of CModels. This is why CModel is distinct from CController.

30

TRANSPARENT DATA INTEGRATION



XVT-Power++ provides an easy-to-use mechanism for creating associations between objects that share data—*transparent data integration* (TDI). The associations, known as TDI connections, establish a communication channel among objects in which messages about object state and data are exchanged and processed automatically.

Common uses of TDI include:

- Synchronizing the state of different objects within an application
- Connecting views with (arbitrary) data sources and back-ends
- Creating well-defined communication links with third-party objects

Section 30.2 discusses individually each of these uses for TDI.

30.1. Synchronizing Your User Interface with TDI

TDI addresses the needs that most XVT-Power++ developers face. Consider a typical C++ application that links a user with a database. The interface of the application contains objects that display particular pieces of information obtained from the database. While the information is displayed by the interface objects, it is commonly managed and stored by a separate group of objects that model the business aspects of the application and communicate with a persistent back-end. In addition to displaying information, the views must also be able to change and update the data as the user interacts with them.

Developers faced with implementing such an application normally must themselves create several classes and infrastructures to accomplish these needs. The developer(s) would have to create objects that model and manage the data, define communication channels between the data and views, and derive interface objects that can receive and display data needed by the application.

Traditionally, XVT-Power++ has provided several mechanisms that facilitate this, such as: 1) the Application-Document-View model that cleanly separates application responsibilities, and 2) the ADP mechanism that automates the definition of data dependencies. More recently, XVT-Power++ has provided TDI plus set(s) of TDI adapter classes.

30.1.1. Advantages of TDI

TDI helps DSC++ application developers by automating most of the details described in section 30.1. Using TDI, creating a typical application is reduced to:

- Creating data models and connections
- Laying out a user interface (using XVT-Architect, for example)
- Creating TDI connections between each view and the data

30.1.2. Flexibility of TDI

Much of the flexibility of TDI comes from the use of prototypes. Each dependent managed by a TDI controller can be coupled with a special CTdiValue object known as a prototype. Prototypes translate from messages understood by one type of object to messages understood by another type of object, as discussed in section 30.3.2 on page 30-9.

The second aspect of TDI's flexibility is provided by adapters. Sometimes messages sent by a provider need to be translated into a set of actions or operations before being sent on to the dependent. The logic for such operations can be coded in a TDI adapter. The TDI adapter is simply an extra CNotifier object placed in between the provider and dependent, as discussed in section 30.3.2.2 on page 30-10.

30.1.3. Scope of TDI

The following XVT-Power++ classes participate in the definition and implementation of TDI:

CNotifier
: Any CNotifier-derived object may be a TDI provider or dependent. In addition, the CNotifier class defines a set of methods that support TDI connections.

CTdiConnection
: Establishing and managing TDI connections is made easy by this special interface class, because by default, it handles the most common prototype combinations. The class encapsulates (hides) most details of the TDI architecture, making TDI programming much simpler.

CTdiValue
: Each TDI message may contain data that is bundled into a CTdiValue object. (The CTdiValue class derives from CModel.) This

class is actually an abstract interface to a number of useful ready-to-use derived classes.

CView
: Most XVT-Power++ CView classes have been made "TDI aware." This means that they contain code that automatically handles common TDI messages that they may receive.

30.2. Common Uses for TDI

TDI addresses a problem that most C++ application developers must solve. Three common ways to use TDI are shown in Figure 30.1.

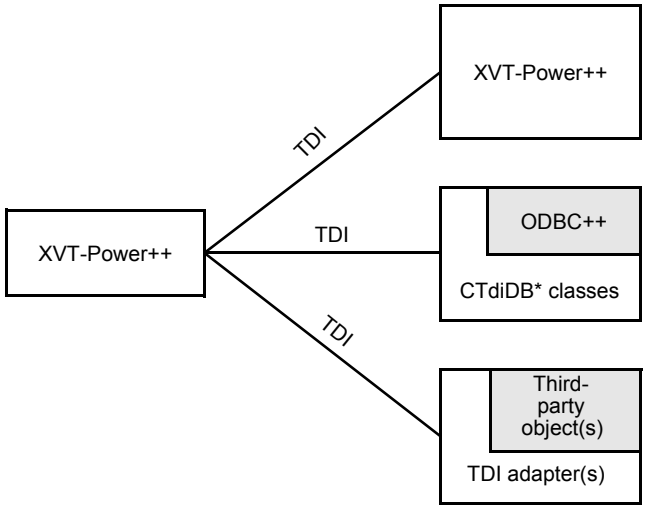


Figure 30.1. Typical uses of TDI (adapter classes not needed when two TDI-aware XVT-Power++ objects communicate directly with one another)

30.2.1. Synchronizing the State of Different Objects

TDI can be used to connect XVT-Power++ views together so that their behavior and state is synchronized. Imagine the interface for a typical “File Open” dialog—this dialog contains a list box displaying known filenames that can be selected as well as a text edit field allowing users to type in a filename. A TDI connection between these two views synchronizes them so that as the user types in a name, the file is selected in the list box, and as the user selects filenames in the list box, the edit field displays that name.

30.2.2. Creating Data Models and Connections

You can use TDI to connect views to back-end objects such as those supplied with the ODBC++ product. The back-end database objects will likely contain data returned by SQL queries. If you connect each view in the user interface to the appropriate database query object that specifies the fields of data it is interested in, TDI takes over from there.

As the data changes, the view is notified and it automatically redraws with the new information. In addition, when the user changes the data by interacting with the view, the back-end is automatically notified with information about the change. All this is done with an average of a single line of code per view.

30.2.3. Communication Links with Third Party Objects

TDI can be used to connect any independently developed (third party) object with XVT-Power++ framework objects. For example, XVT-Power++’s CTdiDB* classes provide TDI extensions that connect XVT-Power++ objects with database objects defined in the DB* classes of ODBC++, a separate product available from XVT. This is just one example of how TDI can serve as a bridge between objects developed independently in separate products/projects.

30.3. Structure and Implementation

Connections are always made between CNotifier-derived objects. In each connection, one CNotifier acts as a provider, while the other acts as a dependent. Figure 30.2 illustrates the basic structure of TDI connections (depicted conceptually).

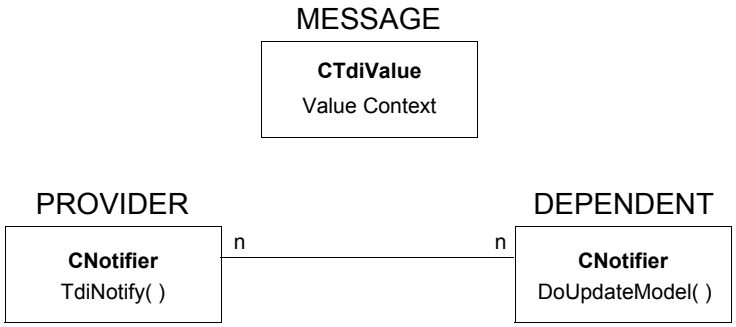


Figure 30.2. Basic structure of TDI communication

Each provider may have zero or more dependents, while each dependent may have zero or more providers. It is also common to set up two TDI connections making each CNotifier both a dependent and a provider for the other.

30.3.1. Communication Between Dependents and Providers

TDI providers normally send a message to their dependents via a call to one of several TdiNotify methods defined in CNotifier. In essence, the TdiNotify function packages the message into a CTdiValue object, and sends the message to each dependent by calling its DoUpdateModel method.

Most XVT-Power++ CViews are "TDI aware," meaning that they automatically define DoUpdateModel() methods that process different TDI messages.

CTdiValue is a class designed specifically to carry data during TDI messages. Because CTdiValue is actually derived from CModel, and because the TDI message is initiated by a dependent's DoUpdateModel, the handling of TDI messages is very similar to the handling of ADP messages.

See Also: For a more thorough comparison of ADP and TDI, see section 30.4 on page 30-11.

30.3.1.1. Important Components of TDI Messages

Each TDI message conveys the following information:

- A *context* describing the nature of the message
- A *type* for the information sent with the message
- A *value* or data object sent with the message
- An *originator* that represents the provider of the TDI message

Each of these terms is explained in section 30.3.1.2, next.

30.3.1.2. TDI Message Terminology

This section introduces the terminology that is used later in this chapter and elsewhere in this *Guide* to describe TDI functionality/

Context

The *context* is represented by the command sent in a TDI message call to DoUpdateModel. DoUpdateModel offers a dozen predefined commands, such as replace, append, clear, select, next, previous, and so forth, that tells views how to respond to TDI messages that they receive. The context is also stored within the CTdiValue object sent with the message.

Type

The *type* is represented by the run-time type of the CTdiValue object received in a TDI message. All CTdiValues support run-time type identification (RTTI), enabling you to verify their types when receiving a message. In addition, CTdiValue provides a separate interface for identifying a common set of data types (integer, floating point, string, and BOOLEAN) without using RTTI.

Value

The *value* is stored by the CTdiValue, and varies according to the implementation of each CTdiValue-derived class.

Originator

The *originator* is embedded within CTdiValue and represents the provider that initiated the message. Although recipients of TDI messages may access the originator, this approach is not encouraged since it can lead to a tighter coupling between an application's front and back ends, resulting in less flexible and extensible code.

30.3.1.3. Internal Configuration of a TDI Connection

Each CNotifier provider actually uses a helper CTdiController object to manage its list of dependents. In turn, each dependent managed by the controller may be coupled with a special CTdiValue object known as a prototype. Figure 30.3 shows the actual internal details of a TDI connection, including the role of the prototype.

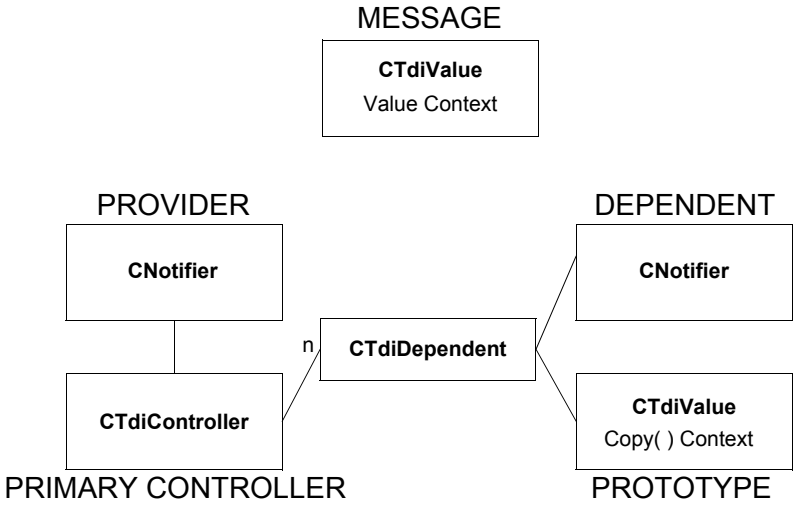


Figure 30.3. Specialization with prototypes; detailed internal structure of TDI communication with a CTdiConnection

CTdiController is an internal class (used by the framework) that routes TDI messages from providers to dependents. CTdiConnection, the public class, uses CTdiController to register the connection. The difference between the two classes is that CTdiController implements the message passing during the connection, and CTdiConnection only sets up (or removes) a TDI connection.

Note: In most TDI situations, you use the CTdiConnection class, and then the details shown in Figure 30.3 are completely hidden "under the covers" of encapsulation. However, knowing about these internals can help you understand how TDI works and how to use it effectively.

30.3.2. Using Prototypes with a TDI Connection

To understand the use of prototypes, you must first understand how messages are sent when a prototype is *not* registered with a dependent. As described in section 30.3.1, in a normal TDI message, a CNotifier sends a TDI command and a CTdiValue object to a dependent. In turn, dependent objects interpret the TDI command and value and act accordingly.

This process works well for many connections between related objects. However, there are times when a provider may be sending context commands or CTdiValue objects that are not readily interpreted by a dependent's DoUpdateModel() method. After all, TDI encourages the connection of independently developed components, providing a common means for communication. It is very likely that two CNotifiers developed independently may generate and accept completely different types of TDI messages.

Because of these potential differences, TDI allows specialized prototype values to be installed in a connection to handle the job of translating messages sent by a provider to a form understandable by a dependent. In other words, a prototype is simply a CTdiValue which a dependent is able to successfully interpret.

When a TDI connection has a prototype installed, the CTdiController sends this prototype value to the dependent instead of sending the original CTdiValue created by the provider. If the prototype has its own context command, that command is used in the message to the dependent. In addition, the prototype's Copy() method is invoked to copy the data stored in the provider's original CTdiValue. CTdiValue::Copy() is a pure virtual method declared in CTdiValue. It is up to the Copy() of each derived CTdiValue class to do the "translation" from provider data to dependent data.

30.3.2.1. Specializing Connections with Prototype Values

Prototypes add a lot of flexibility to TDI. Furthermore, the CTdiConnection class usually knows automatically when it needs to use a prototype. In general, you only need to explicitly add a prototype to a connection in the following cases:

- The type of TDI data sent by the provider is different than the type of TDI data acceptable to the dependent; it is the job of the prototype to make the conversions
- The TDI command sent by the provider needs to be changed into some different command expected by a dependent
- The dependent is only interested in a specific portion of the provider's data

Example 1: Consider an interface with an edit field that allows users to enter dates. The edit field may send the data (a date) as a CTdiStringValue to its dependents. A back-end dependent may require the date as a CTdiDateValue, so a prototype value is inserted into the TDI connection to do the necessary conversions from string to date.

Example 2: Consider TDI connections made between a database client and user interface objects. The database client may send CTdiValueS comprising entire rows of data to its dependents. A dependent such as an edit field or list box may only be interested in one field of the row. The Copy() method of the prototype for such a connection must be able to pick out the correct field of data.

30.3.2.2. Specializing TDI Connections with Adapters

TDI provides a second way to customize connections. Sometimes messages sent by a provider need to be translated into a set of actions or operations on the dependent. The logic for such operations can be coded in a TDI adapter which is simply an extra CNotifier object placed in between the provider and dependent, as shown in Figure 30.4.

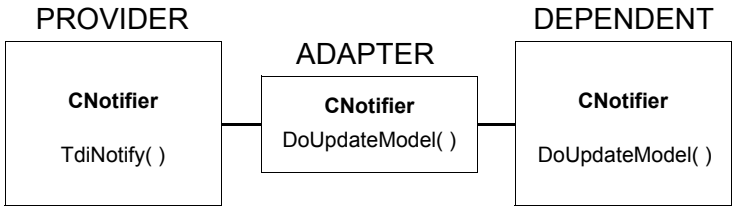


Figure 30.4. Specializing a TDI connection with an adapter

The messages sent by the provider are intercepted by the adapter, which acts as a proxy dependent. The proxy may either propagate the TDI message to the real dependent, or may simply operate upon the dependent as necessary.

For example, consider a connection in which the visibility state of a set of views, such as a radio group, is controlled by the selection state of a check box. A TDI connection could be set up between these two objects so that the radio group is notified whenever the check box is enabled or disabled by the user.

However, the current CRadioGroup class in XVT-Power++ does not have code which shows or hides itself because of a TDI message (i.e., it is not TDI-aware). You could easily derive a special radio group that is TDI-aware. But a more flexible and reusable solution is to create an adapter class, named something like CVisibilityAdapter, that calls its dependent's Show() and Hide() methods as a result of a TDI message. Now you can easily reuse the CVisibilityAdapter class whenever a similar visibility relationship needs to be established.

30.4. TDI and ADP Compared

TDI (transparent data integration) is similar in some ways to ADP (Automatic Data Propagation). Both mechanisms contain objects that act as dependents and/or providers. Both mechanisms rely on the use of CModel objects and DoUpdateModel() messages as the means by which they communicate. Internally, both mechanisms rely on CController objects to manage the delivery of information.

It is important, however, to recognize how these paradigms differ. An ADP architecture is built around a central CModel object that represents the entire state of a system or subsystem. Dependents and providers of data are then registered with a controller that manages this central CModel. When the model is modified, each dependent is notified by a call to DoUpdateModel(). In this paradigm, the providers and dependents do not communicate directly, and the typical granularity of data propagation is coarse—it is often defined at the CDocument and CWindow level.

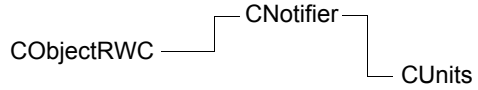
On the other hand, a TDI architecture contains no central CModel object. Instead, connections are made directly between providers and dependents. Each provider sends messages reflecting changes in its own state. Providers and dependents share a direct channel of communication, and the granularity of such communication is much finer—each CView in the interface can be directly connected to multiple specific data sources.

In many ways, TDI provides all the benefits of ADP and adds benefits of its own. The main benefit is a simpler-to-use interface. In TDI, the programmer rarely uses CControllers, defines central CModel classes, or manages connections. Using TDI, these details can be omitted because of the encapsulation provided by the CTdiConnection classes. Because TDI makes data management simple, you, the programmer, write fewer lines of code. Indeed, the amount of data management code you write may be reduced to one-third of the amount you would have had using ADP by itself.

Note: It is also logical to view TDI as a specialization of ADP, since ADP remains the mechanism through which TDI is implemented.

31

LOGICAL UNITS



The CUnits class and the classes derived from it provide objects that allow you to code viewable locations or regions using logical coordinates. These logical coordinates can be of different types: inches, centimeters, pixels, characters, or user-defined coordinates. When objects are drawn, these logical coordinates must be converted to a physical output device. The physical coordinates are always either dots on a printer or pixels on the screen that will be turned on or off according to the unit provided.

The most general class of the units classes is CUnits, which provides a generic unit that can be set to any desired type of logical to physical mapping. Each of the following class derived from CUnits provides a specific type of mapping:

CCharacterUnits
: Maps logical units to physical units according to a base font while maintaining a proportion of character width and height in that font.

CInchUnits
: Maps units that represent inches on the screen to numbers of pixels.

CCentimeterUnits
: Maps units that represent centimeters on the screen to numbers of pixels.

31.1. Setting the Units of Measure

XVT-Power++'s CUnits class enables you to set the size of the units of measure used in your application, with the options being pixels (the default), inches, centimeters, characters, or a user-defined unit.

Units of measure are treated in exactly the same way as environment properties. That is, by default there is a global application CUnits object that propagates through all levels of the XVT-Power++ application framework to the deepest subview. However, you can set the units of measure at different levels: for different documents, windows, and view enclosures. When you change the units of a view, all objects sharing its CUnits object are affected and must accommodate the change.

When a CUnits object is created, the constructor requires you to pass in information about the horizontal and vertical screen mappings as well as the horizontal and vertical printer mapping:

```
CUnits(float theHScreenMapping = 1.0,
       float theVScreenMapping = 1.0,
       float theHPrinterMapping= 1.0,
       float theVPrinterMapping= 1.0,
       OutputDevice theDevice = SCREEN);
```

If the horizontal screen mapping is 2, this means that there is a 2-to-1 ratio; that is, each horizontal unit will be translated into 2 pixels. The basic formula for calculating the physical coordinates is to take the logical coordinate and multiply it by the mapping like this:

physical = logical x mapping

If there is one logical unit and a mapping of two, then the physical mapping is:

1 x 2 = 2

In addition to the screen and printer mappings, the CUnits constructor has a parameter for the output device to which you are currently translating, either the screen or a printer. At any time, you can change the output device that is being used for a particular CUnits object. Methods on the CUnits class allow you to set and get the output device, as well as the different screen and printer mappings. In addition, CUnits provides methods for converting *logical units to physical units* for both horizontal and vertical mappings. You can convert a CPoint or a CRect specified in units to an XVT Portability Toolkit PNT or RCT. Similarly, CUnits contains methods for converting *physical units to logical units*.

Note: These conversions are handled automatically inside XVT-Power++. Thus, while it is important to understand the logic of how units work, you will almost never need to call any of the CUnits mapping functions. All you must do is create a CUnits object and assign it to an object in the object hierarchy of the application framework. Everything else takes care of itself.

31.2. Dynamic Mapping

When a program is ported from one platform to another, it may suffer in appearance because different machines have different screen widths, heights, and resolutions. CUnits has a dynamic mapping capability that is designed to take care of this problem. An application is assumed to be developed on one platform and then ported to other platforms.

CUnits::SetDevelopmentMetrics is called to specify the development platform, with the metrics of the platform that was used for development: the device width, the device height, and the horizontal and vertical resolution. Once you have defined the development metrics, you can turn on dynamic mapping via SetDynamicMapping. When the application executes, it compares the development metrics to the execution metrics. Thus, at runtime, the application can compute the resolution, width, and height of the display device it is using. If the execution metrics are identical to the development metrics, then the program is running on a machine that is identical to the development platform, and no further computations are necessary. However, if the execution metrics differ from the development metrics, CUnits calculates the change of proportions in the resolution, width, and height; then it adjusts the horizontal and vertical mapping so that the application looks right on the new machine with its new metrics.

To make use of the dynamic mapping capability of CUnits, all you must do is call SetDevelopmentMetrics with the hard-coded values of the metrics of the machine you used for development and then turn on SetDynamicMapping. Everything else is automatic.

31.3. Owners of Units

Each CUnits object has a pointer to a boss object, and the pointer is the unit's *owner*. This owner is always the highest object in the object hierarchy that has been assigned the particular CUnits object. For example, when a CUnits object is assigned to a document, then every window and every view inside that document shares

this CUnits object. The boss of the CUnits object is always the document. Although every window and every view is using the CUnits object, the document is the object that actually created it and therefore owns it. Whenever the units change, the CUnits object notifies its owner, and a message about the change, called UpdateUnits, trickles down to all objects that are sharing the units so they can accommodate the change. For example, if the units change for a window, it may need to resize itself, and every view inside it will also have to redraw itself according to new logical units. These updates occur automatically any time the units change. All updates occur through the UpdateUnits method. Whenever the units change, a set of UpdateUnits methods is called for all objects sharing the units so that they can update themselves.

In addition, when units are created (normally, on the heap) and assigned to an object by calling SetUnits, the ownership of the units belongs to the object to which you passed the units. The owner is responsible for deleting its CUnits object when it is itself deleted and for notifying any object sharing the CUnits object that the units are deleted. All of this is done internally, and you need not be concerned about it.

CUnits objects can be shared only in the manner described through the application framework. In other words, a subview can share the units of its enclosure, its window, its document, or its application. However, two different documents *cannot* share the same CUnits object as owners. If you want two documents to have the same kind of units, you must create two separate CUnits objects and set them for each of the documents. Documents can share the same CUnits object only through the CApplication object; that is, they can both use the CUnits owned by the application. In short, sharing can only occur downwards through the parent or child and cannot occur between peers.

31.4. Incorporating Units into XVT-Power++ Applications

By themselves, CUnits objects are simply data structures that do mappings between logical and physical coordinates. Their usefulness lies in the way that they are tied in to the rest of the application framework. Any object derived from CBoss can have its own units. The units hierarchy works in a very similar way to the environment hierarchy. For example, you can set the application's units using a method belonging to all objects derived from CBoss named SetUnits. You set the units object like this:

```
application -> SetUnits units object
```

Then the application owns those units, and any object in the application that does not have its own units set uses these units.

Suppose a view is nested three levels down inside a window. Before this view draws itself, it will check to see whether it has a units object of its own and whether it must do any units conversions. Normally, it does have its own units object and uses it. If it does not have its own units object, it checks to see whether its enclosure has a units object, and if it does, the view uses the units object of its enclosure. If the enclosure does not have a units object, then the view checks to see whether its enclosure's enclosure has one and so on all the way up to the window. From the window, the search continues to the document associated with the window, and finally, if the document does not own a units object, it checks the application object. The application may also not have its units set, in which case the whole application is simply using physical coordinates, and no units are set.

Here we have described simply the abstraction of how units work. In reality, units are directly available to any object, and there is no run-time search up the hierarchy because it would be too time-consuming. However, the semantics of how it all works are as described here.

32

DISPLAYING LIST AND COLUMNAR DATA

CTable CTreeView

XVT-Power++ provides many ways to display columnar (including list) data, each of which offers an optimized solution to a particular problem. This chapter will help you choose the best method for your application.

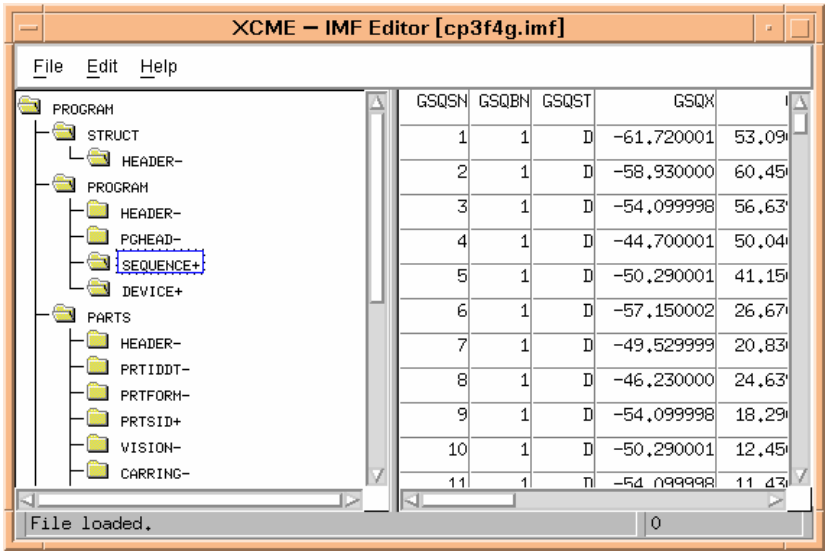


Figure 32.1. Sample view

32.1. Choosing the Method for Displaying Data

Use the summaries given in Table 32.1 on page 32-3 to help you decide which view to use for your application. Keep in mind that CTable or CTreeView may be the best solution (rather than CListBox, NListBox) for displaying some lists and that CTreeView may by preferable to CTable for displaying small tables because it's a lighter weight object.

In general, heavyweight objects have more features than lightweight objects, however, lightweight objects give better performance than heavyweight objects. For example, if you have thousands of rows of data, CTable is the best object to use, even if you have only a single column (a list), because CTable only fetches the data it needs to display the current visible cell range. If you were to use a CListBox for this application, you would need to supply all of the data up front before the view could draw itself, which would be much slower.

32.1.1. Table Data

If you need to display table data, choose between CTreeView and CTable based on the size of data set and the amount of control you need over appearance. For example, can you supply the entire data set in less than a second? CTable handles very large data sets and provides cell level control of appearance, but CTreeView is easier to use and quicker with small data sets. For example, you would probably use CTable to display a list of 10,000 employees, while you would use CTreeView for a list of pending mail messages.

32.1.2. Tree-style Data

A tree view is typically used to display hierarchical data; for example, directories and subdirectories, or rows of financial data (as in a detailed budget). CTreeView provides everything normally needed for this type of view, including keyboard navigation, embedding pictures, and tab-justified fields. You can provide all of the tree data up front (a *static tree*), or supply data "on the fly" as each node is opened by the user (a *dynamic tree*).

	CTable	CTreeView	NListBox	CListBox
Data Needed	Only for visible cells	All data for an open node up front (a list would be one open node)	All data up front. May have native size limitations.	All data up front
Object Overhead	Heavyweight view object (expensive to create)	Lightweight object (fast to create and manipulate)	Lightweight object (fast to create and manipulate)	Heavyweight view object (impractical for > 50 rows)
Native Look-and-Feel	No	No (Windows 95 on all platforms)	Yes	No
Attribute Control	Cell-level, including borders	Row-level, no cell borders	View-level, no cell borders	Row-level
Embedded View Objects	Some allowed: CPicture, NCheckBox, NListButton	Only CPicture	No	Any other view
Justification	Yes, including multiline text	Yes	No	Possible
Column Placement	User-adjustable	User-adjustable; tab-aligned fields in each row	No tab-aligned fields	No tab-aligned fields
Titles	Optional	Optional	No	No

Table 32.1. View trade-offs for displaying columnar data

32.1.3. Long Lists of Data

Choosing which view to use for a list is more complicated, but two factors may make the decision for you. If you must have a native list, use NListBox. If you must be able to embed arbitrary objects (such as other lists), use CListBox. If you can relax these two requirements, choose between CTreeView (one node) and CTable (a single column) as you would with a table data set.

Notice that using multiple fields in your list, which you can do with either CTreeView or CTable, is often required by the user interface design. For example, you can display a small picture next to each item of the list to indicate the nature of that item. There is little memory associated with these pictures because both CTable and CTreeView implement sharing of such resources.

32.2. CTable

CTable readily displays textual data, as well as embedded pictures, check boxes, and button menus. Its flexible event-driven data-feeding mechanism makes the CTable useful for large lists and tables that would be time-consuming to completely populate up front. For small tables, you might want to consider using the lighter weight CTreeView.

CTable's features include:

- Row and column labels with user adjustable sizes (optional)
- Vertical and horizontal scrolling
- Cell, row, column, and table level attribute control (colors, font, justification, data representation, ...)
- Fast coordinate system for flexible row/column sizing while maintaining performance with large tables
- Fast display and scrolling of textual data
- Optimized for row access
- Sharing of picture data and attributes to minimize memory footprint
- Keyboard or mouse navigation
- User-configurable general-purpose data cache
- Selection policies (row select, single cell select, ...)
- In-cell editing
- Event notification of selection, focus, row/column resize or delete, key stroke, and scrolling
- All table events can be trapped and handled in a manner that you define for your application

32.2.1. How to Use Tables in Your Application

Including a table in your application requires the following steps:

- create a table
- initialize the table
- install a data source

Creating a table can be as simple as creating any other view, but generally you will want to do some additional initialization such as setting the cell bounds and column widths.

A callback scheme is used to supply data to the table. The table view only requests the data that it needs to display the currently visible range of cells (possibly plus some margin of cells to increase scrolling performance). This dynamic request/supply means that your data source must be able to supply cell data in a random fashion. While this may seem complex, writing table code is not difficult and XVT-Power++ provides a ready-to-go cache to help you optimize performance.

See Also: For more information about table creation and initialization, see section 32.2.2, next. In addition, section 32.2.3.1 on page 32-9 shows you how to create table sources and give a couple of simple examples for both read-only and read-write tables.

32.2.2. Creating Tables

32.2.2.1. Creating a Table View

Like most views, CTable has a single constructor that takes an enclosure and a rectangular region as the only required arguments.

```
CTable(
    CSubview* theEnclosure,
    const CRect& theRegion,
    const CFont& theTableFont          = STDFont,
    UNITS theDefaultRowHeight          = kNoHeight,
    UNITS theDefaultColumnWidth        = kNoWidth,
    long theChunkRows                  = 128,
    long theChunkColumns               = 32);
```

Choosing Row Height and Column Width

CTable uses the first three arguments to determine the optimum row height and column width based on font metrics. The font specified for theTableFont becomes the table's default font. If theDefaultRowHeight is set to kNoHeight, then the table uses the font height to calculate the default row height. The default height is high enough to allow in-cell editing. You may wish to use a lower height, such as the font height plus four, for read-only tables. Likewise, if theDefaultColumnWidth is set to kNoWidth, then the table uses the font width to calculate the default column width (12 "0"s plus border pixels).

Note: If you want to use a different default height or width, you must supply that value to the table's constructor. You should not change the defaults after the view has been built.

You would probably want to change the default row height if the table displays data other than text. For example, if you are

embedding 64-by-64 images in the first column of each row, set the default row height to 64 plus 1 or 2, depending on the border width you want to use, for the best image quality. Likewise, if you are using a control, such as an NListButton, you may want to set the default row height to the native height of the control. To get the native control height of a list button call:

```
xvt_vobj_get_attr(NULL_WIN, ATTR_CTL_EDIT_TEXT_HEIGHT)
```

Internal Chunk Size

CTable organizes its internal data in square chunks of cells that it allocates when those cells become visible. The final two arguments of the CTable constructor determine the chunk size that the table uses internally. This scheme allows the table to work well with both small and very large data sets and even data sets of unknown size.

If you know that your table will always have a fixed number of columns or rows, you can optimize the table performances by providing these values as the chunk dimensions. If these dimensions exceed about 128 (application dependent), you may want to use the default values anyway (to avoid erratic performance) as each large chunk is managed during operations such as scrolling.

32.2.2.2. Setting the Initial Attributes

After creating a table, you should call ITable( ) to initialize it before calling any other table methods or allowing the table to be drawn on the screen.

Example: The following code shows the call to ITable( ):

```
BOOLEAN ITable(
    SelectionPolicy theSelectionPolicy = MultipleRow,
    BOOLEAN hasHorizontalScrollBar = TRUE,
    BOOLEAN hasVerticalScrollBar = TRUE,
    BOOLEAN hasColumnLabels = TRUE,
    BOOLEAN hasRowLabels = TRUE,
    BOOLEAN isThumbTracking = FALSE,
    BOOLEAN isVisible = TRUE,
    GLUETYPE theGlue = NULLSTICKY);
```

The first argument, theSelectionPolicy, is unique to CTable and determines the type of cell/row/column selection operations available to the user. The ITable( ) method can be called multiple times after the view is created.

Selection Policies

Selection policies are grouped into three categories: row, column, and cell. A row selection policy allows only the selection of rows and a column selection policy allows only the selection of columns. Cell selection policies are more general and can allow row and column selection in addition to individual cell selection.

Within each of the three groups, there are three selection policy subcategories: single, mandatory, and multiple.

Single
: The single selection policy means that at most one selection can exist at a time. For example, CTable::SingleCell, means that at most one cell can be selected at a time.

Mandatory
: The mandatory selection policies are slightly different in that, once the initial selection has been made, they ensure a single selection always exists.

Multiple
: The multiple selection policies allow multiple, including non-contiguous, selections.

Available Policies

Table 32.2 shows the policies available for CTable.

Policy	Comments
CTable::None	No selections allowed
CTable::SingleRow	Zero or one row
CTable::MandatoryRow	After an initial selection, always one row
CTable::MultipleRow	Multiple row selections
CTable::SingleColumn	Single column selection
CTable::MandatoryColumn	After an initial selection, always one column
CTable::MultipleColumn	Multiple column selections
CTable::SingleCell	Single cell selection
CTable::MandatoryCell	After an initial selection, always one cell
CTable::MultipleCell	Multiple cell selections

Table 32.2. Available policies

Focus Compared to Selection

The concept of selected cells/rows/columns differs from the concept of focus. There is at most a single cell that has focus at any one time. In general, it is the cell that receives keystrokes and is indicated by a dotted border. Selected cells, in contrast, are indicated by a change in background color.

See Also: For complete information on focus see the description of CView in the online *XVT-Power++ Reference*.

32.2.2.3. Setting the Table Size

XVT-Power++ tables can be as big as you want to make them—only limited by the size of a "long". CTable manipulates the table structures as infinitely large sparse arrays. You can dynamically set the bounds (number of column and rows) of a table by calling SetCellBounds( ) which only affects the scrollbars and certain optimizations.

Example: To set a table bounds to 8 columns and 64 rows, call SetCellBounds using the following parameters:

```
myTable->SetCellBounds( CBounds(0, 0, 64, 8) );
```

32.2.3. Supplying Data to Tables

32.2.3.1. Table Data Sources

Tables get their data using callbacks. When a cell becomes visible on the screen, CTable requests the data for that cell from its data source. The data source must then provide a CTdiValue that represents the contents of this cell.

Creating a Data Source

Create your data source by subclassing CTableSource abstract class and override its three methods: Prime( ), GetTableData( ), and PutTableData( ).

See Also: An example of overriding CTableSource's three methods is given in *Read-only Table Sources* on page 32-10.

Adding to an Existing Class

If you have an existing object you can mix it in. *Mixing in* refers to the multiple inheritance technique of adding additional methods to an existing class. This approach makes it easy to convert existing classes into table data sources with little modification.

Example: If you already have a table data structure called CMyTable and you wish to use it to supply data to the table view, you would create a new class, as shown in this example.

```
class CMyTableSource : public CMyTable, public CTableSource
{
...
public
        virtual public CTdiValue* GetTableData(const CCell& theCell);
        virtual public BOOLEAN PutTableData(
                const CCell& theCell,
                const CTdiValue* theData);
        virtual public void Prime(const CBounds& theBounds);
...
};
```

Read-only Table Sources

For read-only tables you need only implement the GetTableData method.

Note: GetTableData( ) does not pass ownership of the CTdiValue to the table.

Example: This shows how to subclass CTableSource and simply provide the cell coordinates as its data. Consequently, PutTableData( ) and Prime( ) have empty implementations.

```
class CMyTableSource : public CTableSource
{
        ...
        public
                virtual const CTdiValue* GetTableData(const CCell& theCell)
                        {
                                char s[64];

                                xvt_str_sprintf( s, "(%d, %d)", theCell.V(), theCell.H() );
                                itsTempValue.FromString( CStringRW(s) );

                                return &itsTempValue;
                        }

                virtual BOOLEAN PutTableData(const CCell& theCell,
                                                        const CTdiValue* theData)
                        {
                                return FALSE;
                        }

                virtual void Prime(const CBounds& theBounds)
                        { }

        protected :
                CTdiStringValue itsTempValue;
};
```

This case uses a CTdiStringValue to represent the data, but you can easily use another TDI prototype, such as CTdiDateValue, if that represents your data better than a simple string.

See Also: For more information on TDI values and TDI prototypes see Chapter 30, *Transparent Data Integration*.

Read-write Table Sources

Read-write table sources require that you implement the PutTableData( ) method of your table source. Like GetTableSource( ), the table is not passing ownership of the CTdiValue. Remember this pointer will be invalid upon exiting the PutTableData( ) method. You should extract the information from theData and store it or clone the CTdiValue and store the clone.

To clone a CTdiValue, use the following code:

```
CTdiValue* aClone = (CTdiValue*)theData->newSpecies();
aClone->Copy( *theData );
```

The table calls the PutTableData( ) method you have implemented anytime it suspects the user has changed cell data. With some control types, the table may not always know if the user has changed a cell. So the table takes a conservative approach that may produce PutTableData( ) calls when, in fact, the cell was not really modified. However, if the cell was modified you are guaranteed to get a PutTableData( ) call.

The table calls your Prime( ) method as a "heads up" when it is ready to fetch data for a range of cells, such as when a new page is drawn. This allows you to bring in a block of cell data into a cache to be retrieved later with strategically timed calls to GetTableData( ). This is strictly a performance enhancing method and is never required.

Once you have created a table data source, pass the object to the SetSource( ) method on table.

Caution: You are responsible for creating and deleting the data source. Do not delete the data source until after you have destroyed the table.

32.2.3.2. Using CTableTdiSource as a Data Cache

CTableTdiSource is a flexible class that can both provide and consume table data. In other words, you can easily place it between your table data source and the table view.

Example: The following code shows how to insert a CTableTdiSource between the table data source and the table view:

```
CTableTdiSource aCache( aTable );
CMyRealTableSource aSource;
aCache.SetSource( &aSource );
```

Using CTableTdiSource as a cache minimizes the calls to your table data source by:

- caching rows
- avoids PutTableData( ) calls with a read-write table when the new data matches the cache data (a condition indicating that the user really didn't change the cell data).

You can set the maximum number of rows that you want the CTableTdiSource to cache by calling the SetCacheSize( ) method with the number of rows to cache.

Because of the ownership chain, the order of destruction must be aTable, then aCache, and finally aSource.

See Also: For more information on ownership between views, see section 16.2 on page 16-6.

32.2.3.3. Using TDI to Supply Data to a Table

To make a TDI connection to a table, create a CTableTdiSource and connect the TDI client to this object rather than the table. CTable does not accept TDI connections directly, but uses CTableTdiSource as an adapter.

Example: The following code connects a CTable to an ODBC++ data source:

```
// Set up a TDI table adapter using CTableTdiSource
itsCTableTdiSource = new CTableTdiSource(
        itsData.itsCTableUserView);

// Connect the new table to the CTable in the user view
CTdiTableConnection aRowConnection(
        itsTdiTable,
        itsCTableTdiSource,
        CTdiDBQuery::CURRENT_INDEX,
        CTdiDBQuery::CURRENT_INDEX,NULLcmd,
        TRUE,       // autoConnect
        new CTdiDBTableFactory()
);
```

See Also: For more information, see the example samples/arch/tditable.

32.2.4. Controlling Rows and Columns

There are many methods available for controlling various aspects of the table. Some of these methods take the row or column number as a parameter. These are numbered sequentially starting with 0.

See Also: For information on labels for rows and columns, see section 32.2.6 on page 32-20.

32.2.4.1. Setting Column Width and Row Height

Call the SetColumn( ) method of CTable with a numerical argument to set the width of an individual column and call the SetRow( ) method of CTable to set individual row heights. These two methods have the following signatures:

```
void        SetRow(long theRow, UNITS theHeight);
void        SetColumn(long theColumn, UNITS theWidth);
```

Note: The row and column indices are zero-based.

32.2.4.2. Deleting and Inserting Rows and Columns

CTable has methods to insert and delete rows and columns from a table view. CTable automatically adjusts any rows and columns beyond the insertion or deletion.

The delete and insert methods are:

```
void        InsertColumns(long theColumn, long theCount = 1);
void        DeleteColumns(long theColumn, long theCount = 1);
void        InsertRows(long theRow, long theCount = 1);
void        DeleteRows(long theRow, long theCount = 1);
```

When inserting a row at row index 2, for example, the new row is now the third row and accessed through a w index of 2 (zero-based).

32.2.5. Setting Attributes

You can set attributes at the cell, row, column or table level. Each cell inherits unset attributes from its row, then its column, and finally the table.

t To set attributes at the cell, row, column, or table level:

1. Create a CTableAttributes object.
2. Set the attributes you want to change.
3. Apply the attribute set to the table, or to a row, column, or cell.

The list of attributes you can set includes:

- fonts
- foreground and background color
- text justification
- borders (top, left, bottom, right)
- whether the cell is read-only.

The attribute set also includes advanced attributes for setting the data interpreter (page 32-16) and the validator (page 32-19).

Each attribute has three associated methods in the CTableAttributes class: GetXXX, SetXXX, and UnsetXXX.

32.2.5.1. Colors, Fonts, and Justification

Colors, fonts, and justification are the simplest attributes to understand.

The font attribute has these three methods:

```
// Set methods
void Font(const CFont& theFont);
// Unset methods
void UnFont();
// Get methods
const CFont& Font() const;
```

Example: The following code sets the colors on a diagonal line of cells:

```
// Setting some attributes on a diagonal line of cells
for( i = 0; i < 100; i += 1 )
{
      CTableAttributes    anAttribute;
      anAttribute.Foreground(COLOR_GREEN);
      anAttribute.Background(COLOR_RED);
      aTable->SetCell(CCell(i, i), anAttribute);
}
```

Justification Values

CTable supports compass justification of multi-line text within each cell. The possible values of justification are:

```
CTableAttributes::JustifyTop
CTableAttributes::JustifyTopRight
CTableAttributes::JustifyRight
CTableAttributes::JustifyBottomRight
CTableAttributes::JustifyBottom
CTableAttributes::JustifyBottomLeft
CTableAttributes::JustifyLeft
CTableAttributes::JustifyTopLeft
CTableAttributes::JustifyCenter
```

The default justification is JustifyLeft, which means flush left and centered top to bottom.

Example: The following code sets the justification of the third column to centered. Note the zero-based column index.

```
// Column justification
{
    CTableAttributes    anAttribute;
    anAttribute.Justification(CTableAttributes::JustifyCenter);
    aTable->SetColumn(2, anAttribute);
}
```

32.2.5.2. Borders

Cell borders lie within the bounds of each cell and are specified independently for the four cell sides.

Border Styles

The possible border styles include:

```
CTableAttributes::BorderNone
CTableAttributes::BorderSingle
CTableAttributes::BorderDouble
CTableAttributes::BorderDashed
CTableAttributes::BorderSunken
CTableAttributes::BorderRaised
```

By default, the table uses BorderSingle for the top and left sides, and BorderNone for the right and bottom sides. This configuration gives a single line border between all cells. BorderSunken and BorderRaised have special behavior in that setting one of these on any cell side implies that the entire cell has a sunken or raised 3D appearance on all sides.

You can also set the color of each cell border, as shown in the following example.

Example: This example takes a block of cells, specified by theBounds, sets a red border around the entire block and dashed blue borders within.

```
void SetBorders(CTable* theTable, const CBounds& theBounds);
{
	CTableAttributes::Border anInteriorBorder;
	CTableAttributes::Border anExteriorBorder;

	// Set up border types
	anInteriorBorder.itsColor = COLOR_BLUE;
	anInteriorBorder.itsStyle = CTableAttributes::BorderDashed;
	anExteriorBorder.itsColor = COLOR_RED;
	anExteriorBorder.itsStyle = CTableAttributes::BorderSingle;

	for( int h = theBounds.Left(); h < theBounds.Right(); h++ )
		for( int v = theBounds.Top(); v < theBounds.Bottom(); v++ )
		{
			CTableAttributes anAttributeSet;

			// Top
			anAttributeSet.TopBorder( (v == theBounds.Top()) ?
				anExteriorBorder : anInteriorBorder
			);

			// Bottom
			anAttributeSet.BottomBorder( (v == theBounds.Bottom() - 1) ?
				anExteriorBorder : anInteriorBorder
			);

			// Left
			anAttributeSet.LeftBorder( (h == theBounds.Left()) ?
				anExteriorBorder : anInteriorBorder
			);

			// Right
			anAttributeSet.RightBorder( (h == theBounds.Right() - 1) ?
				anExteriorBorder : anInteriorBorder
			);

			theTable->SetCell( CCell(h, v), anAttributeSet );
		}
}
```

See Also: More features of CTableAttributes are described in the online *XVT-Power++ Reference*.

32.2.5.3. Data Interpreters for Other Types of Data

CTable provides an extensible framework for representing data other than text, such as pictures. Currently, CTable supports (in addition to text) native check boxes, list buttons, and pictures.

Objects that handle data display and user interaction are called table interpreters and inherit from CTableInterpreter. The interpreter is an attribute, just like color, and can be set at the cell, row, column, or table level.

By default, the table-level interpreter is CTableTextInterpreter, which handles data displayed as text. Usually each column of a table displays data in the same way so you will generally set the interpreter attribute put on columns, not rows.

Example: The following code fragment sets the interpreter on column 0 (the first column) to display pictures:

```
CTableAttributes anAttribute;
CTableInterpreter* anInterpreter = new CTablePictureInterpreter(aTable);
...
anAttribute. Interpreter(anInterpreter);
aTable->SetColumn(0, anAttribute);
```

The table is responsible for deleting all interpreters for you automatically.

Displaying Pictures

If a cell's data equals a key string, the picture interpreter displays the image associated with that key in the cell. When the interpreter cannot find a key string matching the cell data, it displays the default image, if defined. The key string comparisons are case sensitive.

You are responsible for creating and deleting images that you pass to the picture interpreter; however, you can delete the images immediately as the interpreter only uses the images temporarily (long enough to create pictures).

Example: To associate each image that the interpreter displays with a key string, we can add to the example in the previous section. The following code sets up the picture interpreter for the first column of a table:

```
CTableAttributes                anAttribute;
CTablePictureInterpreter*       anInterpreter = new CTablePictureInterpreter(aTable);
CImage*                         anUserPicture = new CImage("user.bmp");
CImage*                         anAdminPicture = new CImage("admin.bmp");
CImage*                         aVisitorPicture = new CImage("visitor.bmp");
CImage*                         anUnknownPicture = new CImage("unknown.bmp");

anInterpreter->AddImage( *anUserPicture, "user" );
anInterpreter->AddImage( *anAdminPicture, "admin" );
anInterpreter->AddImage( *aVisitorPicture, "visitor" );
anInterpreter->AddDefaultImage( *anUnknownPicture );

delete anUserPicture;
delete anAdminPicture;
delete aVisitorPicture;
delete anUnknownPicture;

anAttribute. Interpreter( anInterpreter );
aTable->SetColumn( 0, anAttribute );
```

Displaying Check Boxes

The check box interpreter draws a native check box in the cell. You associate a string with each check box state—*on* or *off*. If the cell data matches the on string, the check box appears checked. If it matches the off string, it appears unchecked. If the data matches neither, the state of the check box is unchanged. Likewise, if the user clicks on the check box (changing its state) the data associated with the cell is either the on string or the off string depending on the current state of the check box.

Example: The following code sets up the check box interpreter for the first column of a table:

```
CTableAttributes                anAttribute;
CTableCheckBoxInterpreter*      anInterpreter = new CTableCheckBoxInterpreter(
                                                    aTable,
                                                    "Is Registered"
                                                    );

anInterpreter->OnString( "Yes" );
anInterpreter->OffString( "No" );

anAttribute.Interpreter( anInterpreter );
aTable->SetColumn( 0, anAttribute );
```

Note: You need a separate interpreter for each column. For example, if you want to use a series of check boxes with the title **Is Registered** in column 1 and another series of check boxes with the title **C++ Literate** in column 2, you need to create two interpreters—one to handle each case.

Displaying List Buttons

To use list buttons in your table cells, create a CTableListButtonInterpreter and use the AddString( ) method to populate the list. The cell data corresponds to the selected item in the list, or is an empty string if no item is selected. You must supply the height of the list box to the CTableListButtonInterpreter. This is the total height of the control when the list is popped up (just like creating a NListButton, see page 32-5). This height is not the height of the cell containing the list button.

Example: The following code sets up the list button interpreter for the first column of a table:

```
CTableAttributes                anAttribute;
int                             aFontHeight =
                                        aTable->GetEnvironment()->GetFont().GetHeight();
CTableListButtonInterpreter*            anInterpreter = new CTableListButtonInterpreter(
                                                aTable,
                                                (UNITS)(4 * aFontHeight + 12)
                                        );

anInterpreter->AddString( "red" );
anInterpreter->AddString( "green" );
anInterpreter->AddString( "blue" );

anAttribute.Interpreter( anInterpreter );
aTable->SetColumn(0, anAttribute);
```

Note: For example, if you want to use a list in column 1 containing a list of colors and another list in column 2 containing shapes, you need to create two interpreters—one to handle each list.

32.2.5.4. Field Validation

You can add a validator to cells using text edit field or list button interpreters. Adding validators to other types of cells does not cause an error—they simply are not used. The code that attaches a validator to a cell looks similar to the code that adds an interpreter as an attribute except that validators are passed by value so there is no need to new or delete them.

Example: The following code demonstrates field validation:

```
CTableAttributes        anAttribute;
anAttribute. Validator(
        VALIDATOR_FACTORY->ConstructValidator(
                "{Sun,Mon,Tue,Wed,Thur,Fri,Sat}"
        )
);
aTable->SetColumn(0, anAttribute);
```

See Also: For complete information on validators see the description of `CValidator` in the online *XVT-Power++ Reference*.

32.2.6. Adding Row and Column Labels

Row and column labels are specialized tables that are clients to the main interior table. Consequently, you can set attributes such as font, color, and border style on labels much as you would with regular table cells.

Two overloaded methods, SetRow( ) and SetColumn( ), take special constant parameters for the row or column number and manipulate the column label row or the row label column. These constants are TITLE_ROW for SetRow( ) and TITLE_COLUMN for SetColumn( ). If you do not want your table to use labels, pass FALSE for hasColumnLabels or hasRowLabels in CTable::ITable( ).

32.2.6.1. Setting Label Text

The most likely thing you will change with labels is their text. You can easily change the text of labels by calling the following table methods:

```
SetRow(theRowNumber, "A Label")
SetColumn(aColumnNumber, "A Label")
```

Note: Row and column numbers are zero-based.

32.2.6.2. Setting Label Width and Height

You can also use the TITLE_ROW and TITLE_COLUMN constants with the SetRow( ) and SetColumn( ) table methods to change label height and width.

Example: The following code sets the height of the column labels to 40 units with the table method:

```
SetRow(TITLE_COLUMN, (UNITS)40)
```

Example: The following code sets the width of the row labels for the string "Hello" (plus a few pixels on either side):

```
CFont aFont = aTable->GetEnvironment()->GetFont();
if( !aFont.IsMapped() )
      aFont.Map(aWindow)
int aWidth = aFont.GetTextWidth("Hello");
aTable->SetColumn(TITLE_ROW, (UNITS)(aWidth + 6));
```

To change the width of a column label, change the width of the column using:

```
aTable->SetColumn(aColumnIndex, aWidth)
```

Likewise, to change the height of a row label, you should change the height of the row.

32.2.6.3. Setting Label Attributes

Any of the attributes that you can set on a regular cell can also be applied to a label.

Setting attributes on labels is similar to setting the width and height. Use the constant TITLE_ROW in SetRow( ) to indicate the row containing the column labels. Use the constant TITLE_COLUMN in SetColumn( ) to indicate the column containing the row labels.

Example: The following code sets the column label to red:

```
CTableAttributes anAttribute;
anAttribute.Foreground(COLOR_RED);
aTable->SetRow(TITLE_ROW, anAttribute);
```

32.2.7. Tracking Selection Areas in the Table

CTable maintains a complex region, CRegion, that represents each selected cell, row, or column. You can get a reference to this region using the GetSelectedRegion() method of CTable. All of the CRegion iterators work with this region, thus allowing you to visit selected cells, rows, or columns as your application requires.

CTable also provides a number of convenience methods for manipulating the selected region without extracting the CRegion object. These methods include IsRowSelected( ), IsColumnSelected( ), and IsCellSelected( ) for querying the selection, along with SelectRow( ), SelectColumn( ), SelectCell( ), SelectCells( ), and DeselectAll( ) to change the selection.

Tip: If your application needs to make complex changes to the CRegion object, you can set it as the current selection region by calling the SetSelectedRegion( ) method on your table object.

See Also: For complete information on how complex regions are maintained see the description of CRegion in the online *XVT-Power++ Reference*.

32.2.8. Processing Events in Tables

CTable uses two classes of events: permission and notification.

Permission
Permission events ask your application if it's "OK" to do something.

Notification
Notification events simply tell you that something happened.

In both cases, CTable delivers the events to your chain of command, which you may intercept in the DoCommand( ) method of your window, document, or application as you find appropriate for your application.

For example, if the user resizes a column, the table sends a permission event stating that the user has requested a resize operation. If your application denies this event (by calling Deny() method of the event object), the table will not perform the resize.

In the previous situation, if your application did not deny the resize request, the table would resize the column and then send another event up the chain of command indicating that the resize operation had occurred. You can set the table event style to CTable::EventPermission, CTable::EventNotify, or CTable::EventBoth using the SetEventStyle( ) method of CTable. The default event style is CTable::EventNotify.

Example: This example demonstrates the recommended way of handling and responding to table events. The example shows how to trap the FocusOut permission event to handle some sort of validation. (The table must have had its event style set to CTable::EventPermission or CTable::EventBoth for this to work).

```
void CMyDocument::DoTableEvent(CTableEvent * theEvent )
{
	switch (theEvent->GetType())
	{
		case CTableEvent::FocusOut:			// The user changed keyboard focus
			// Only handle permission events, not notification, here
			if( theEvent->IsPermissionEvent() ) // Validate this cell's data
			{
				CCell theCellToValidate = *theEvent->CellEvent.theCell;
				BOOLEAN isGood = FALSE:

				// Do some stuff here to see if the cell contents are valid
				// and set isGood accordingly

				if( !isGood )
					theEvent->Deny();// Don't let focus out unless data is good
			}
		break;

	case CTableEvent::Select:// The user changed the region of selected cells
		// Do some stuff
		break;
	}
}

void CMyDocument::DoCommand( long theCommand, void* theData )
{
	switch ( theCommand )
	{
		case TableEventCmd:
			{
				CTableEvent* anEvent = PtrCast(CTableEvent, theData);
				PwrAssert( anEvent != NULL, "Yikes!");
				DoTableEvent(anEvent);
				}
				break;

		default:
				CDocument::DoCommand(theCommand, theData);
	}
}
```

IsPermissionEvent() is used to determine whether the event is requesting permission and notifying your handler of some change:

You detect a table event in your DoCommand( ) when theCommand is equal to TableEventCmd. After a simple validation, DoCommand( ) calls DoTableEvent( ), a method you have added to your document (or window, or application) class to handle table events specifically. DoTableEvent( ) has its own switch group to handle the various table events.

32.2.8.1. Table Events

Delete and Insert Events

```
CTableEvent::DeleteRow          // Row delete
CTableEvent::DeleteColumn       // Column delete
CTableEvent::InsertRow          // Insert row
CTableEvent::InsertColumn       // Insert column
```

- To access the parameters of a delete/insert event:

```
theEvent->DeleteInsertEvent.itsFirst
                                // First row or column
theEvent->DeleteInsertEvent.itsCount
                                // Number of rows or columns
```

Size Events

```
CTableEvent::SizeRow            // Change row height
CTableEvent::SizeColumn         // Change column width
```

- To access the parameters of a size event:

```
theEvent->SizeEvent.itsRowColumn
                                // Row or column changed
theEvent-> SizeEvent.itsSize
                                // New or proposed size
```

Focus Events

```
CTableEvent::FocusIn            // Cell focus in
CTableEvent::FocusOut           // Cell focus out
```

- To access the parameters of a focus event:

```
theEvent->CellEvent.itsCell
                                 // Pointer (CCell*) to
                                 // affected cell
```

Select Events

```
CTableEvent::Select              // Selection (the region of
                                 // selected cells) changed
```

- To access the parameters of a select event:

```
theEvent->SelectEvent.itsProposedRegion
                                 // CRegion representing the
                                 // selection change
theEvent->SelectEvent.itsSelect
                                 // TRUE if itsProposedRegion
                                 // will be added to the current
                                 // selected region, FALSE
                                 // if it will be removed from
                                 // the current selected region
```

Key Events

```
CTableEvent::Key          // Key event to focused cell
```

t To access the parameters of a key event:

```
theEvent->KeyEvent.itsCell
                          // Pointer (CCell*) to target of
                          // the key event
theEvent->KeyEvent.itsKey
                          // Pointer (CKey*) to the
                          // keystroke data
```

Origin Events

```
CTableEvent::Origin       // Origin changed (table scrolled)
```

t To access the parameters of a focus event:

```
theEvent->CellEvent.itsCell
                          // Pointer (CCell*) to new origin
```

32.3. CTreeView

CTreeView displays tree type data with tab-justified text and pictures. The flexible tabbing scheme and lightweight construction of the tree view makes it useful for displaying lists and small tables that do not require cell level attributes or in-cell editing.

The main features of CTreeView are:

- Title bar with user adjustable tab stops (optional)
- Vertical and horizontal scrolling
- Line level attribute control (colors, font, tab stops, pictures)
- Embedded pictures at the beginning of a line or at any tab stop
- Sharing of picture data and attributes to minimize memory footprint
- Keyboard or mouse navigation
- Static or dynamic “feeding” of tree data
- Event notification of selection, node expansion or collapse, and mouse clicks

Building a static tree starts with populating the root node. Dynamic trees work on a callback basis.

See Also: For information on building a static tree, see section 32.3.1 on page 32-27. For information on building a dynamic tree, see section 32.3.3 on page 32-33.

A CTreeView consists of many instances of CTreeItem, each item representing a line in the view. Some of the tree items are nodes (like the symbol that represents a folder) that the user can expand and collapse and other items are terminal (like the symbol that represents a file).

Associated with each item is a set of attributes, a string to display, and a picture (optional) that the view displays to the left of the string. Unless you specify a unique set of attributes for a tree item, each item inherits the attributes of its parent node.

One of the most important attributes of CTreeView is CTabSet, a utility class that defines the position of the tab stops and whether each tab stop represents a picture or text.

See Also: For information on setting CTreeView attributes, see section 32.3.4 on page 32-35.

32.3.1. Creating Static Trees

32.3.1.1. Creating a CTreeView

Like XVT-Power++'s simplest views, CTreeView has a single constructor that takes an enclosure and a rectangular region as arguments. All other attributes are set in the initialization method ITreeView( ).

The arguments in ITreeView default to reasonable values, so calling ITreeView( ) with no arguments usually suffices; however, you must call ITreeView( ) before calling any other methods of CTreeView and before CTreeView displays itself.

32.3.1.2. Initializing the Root Node

When you create a tree view, the root node is automatically created for you. After calling ITreeView( ), you need to set the attributes of the root node. Subsequent population of the tree, either statically or dynamically, adds additional tree items that inherit the characteristics of the root node.

Most applications explicitly set the images associated with expanded nodes, collapsed node, and terminal nodes. These images are created and destroyed by your own code, not by the CTreeView. Be sure your application destroys these images after it has destroyed the tree view.

Example: The following code shows a typical initialization sequence:

```
itsOpenDirectoryImage = new CImage("open.bmp");
itsClosedDirectoryImage = new CImage("closed.bmp");
itsFileImage = new CImage("file.bmp");
itsCheckImage = new CImage("check.bmp");
// Set title string
itsTree->SetTitleString(
        "Name" + CTabSet::TabString() + "Size" + CTabSet::TabString() + "checked"
);
itsTree->ITreeView();
itsTree->GetRoot()->SetString(".."); // .. seems fairly portable for 'go up one'
itsTree->GetRoot()->SetImage(itsFileImage);
itsTree->GetRoot()->SetCollapsedImage(itsClosedDirectoryImage);
itsTree->GetRoot()->SetExpandedImage(itsOpenDirectoryImage);
itsTree->GetRoot()->SetHeight(CTreeItem::kBestHeight);
```

Note: When you pass CTreeItem::kBestHeight to the SetHeight( ) method, the tree view sets the height of the items to the best height based on the size of the images and the font size. When the application changed the images, it called SetHeight( ) to recalculate the optimum height based on the new information.

32.3.1.3. Populating the Tree

Building a static tree starts with populating the root node, which is automatically created for every tree. To populate a static tree view you must recursively add children to each node of your tree.

The most general method to add a child is:

```
AddChild(
        const CTreeItemInfo& the Info,
        const CStringRW& theString = NULLString
    (;
```

You must first completely fill out a CTreeItemInfo structure (see the example, page 32-28). This object has variables representing all of the parameters for building the new tree item including color, font, height, indentation, string, and images. (You can call Info( ) on the parent node to get a copy of the parent's info structure.) Then call AddChild( ) on the parent node to create and add a new child based on the supplied CTreeItemInfo structure and the optional item string.

XVT-Power++ supplies two shortcut methods for added children: 1) AddNodeChild( ), and 2) AddTerminalChild( ). In both cases, the new child inherits all of its attributes from its parent, with the exception of the item string supplied as the only argument to the methods:

```
virtual void            AddChild(CTreeItem* the Item);
virtual CTreeItem*      AddChild(
                            const CTreeItemInfo& the Info,
                            const CStringRW& theString = NULLString
                            );

virtual CTreeItem*      AddNodeChild(
                            const CStringRW& theString = NULLString
                            );
virtual CTreeItem*      AddTerminalChild(
                            const CStringRW& theString = NULLString
                            );
```

Example: This example shows how to propagate attributes from parent to child. BuildTreeFromString( ) builds a static tree (itsTree) by parsing a string. To simplify the tokenizing, this code requires that braces and semi-colons be delimited by spaces or tabs. The call might look like:

```
BuildTreeFromString(
    "colors { red ; blue ; green ; } fonts { serif
                                        { Times ; Century ; } sans serif
                                        { Helvetica ; }}"
);
```

which would build a tree like the one shown below:

colors
- **red**
- **blue**
- **green**

fonts
- **serif**
  - **times**
  - **century**
- **sans serif**
  - **Helvetica**

```
void CTreeWin::AddItem(CTreeNodeItem* aNode,
                                        const CStringRW& theString)
{
    aNode->AddTerminalChild(theString);
}

CTreeNodeItem* CTreeWin::AddNode(CTreeNodeItem* aNode,
                                        const CStringRW& theString)
{
    return aNode->AddNodeChild(theString);
}

void CTreeWin::BuildTreeFromString(
                                        const CStringRW& theString)
{
    RWCTokenizer        aTokenizer( theString );
    CStringRW           aString;
    CStringRW           anItem;
    CTreeNodeItem*      aNode = itsTree->GetRoot();

    while( !(aString = aTokenizer(" \n")).isNull() )
    {
        aString.strip(RWCString::both);

        // Add item as a node
        if( aString == "{" )
        {
            aNode = AddNode(aNode, anItem);
            anItem = NULLString;
        }
```

```
                // Done with this node, go back to parent
                else if( aString == "}" )
                {
                        if( !anItem.strip(RWCString::both).isNull() )
                                AddItem(aNode, anItem);
                        anItem = NULLString;
                        aNode = aNode->GetParent();
                }

                // Finished an item
                else if( aString == ";" )
                {
                        AddItem(aNode, anItem);
                        anItem = NULLString;
                }

                // Continue with item
                else
                        anItem += " " + aString;
        }
}
```

32.3.1.4. Traversing a Tree Programmatically

You can visit all of the items of a tree by starting at the root item and recursively getting the child item of each node. The GetRoot() method of CTreeView returns a CTreeNodeItem representing the root node. Calling the GetNChildren() method of a node item gives the number of children belonging to this node, which you can then retrieve with the GetChild() method.

Example: The following code performs an unspecified action to each terminal (leaf) item of the tree:

```
void DoSomething(CTreeItem&  theItem)
{
      // Do whatever to the item here
}

void DoSomethingToNode(CTreeNodeItem& theNode)
{
      for( int i = theNode.GetNChildren() - 1; i >= 0; i--  )
      {
            CTreeItem* anItem = theNode.GetChild( i );
            if( anItem.IsTerminal() )
                  DoSomething( *anItem );
            else
                  DoSomethingToNode( *(CTreeNodeItem)anItem );
      }
}

void DoSomethingToTree(CTreeView& theTreeView)
{
      DoSomethingToNode( theTreeView.GetRoot() );
}
```

32.3.2. Attaching User Data to Tree Items

CTreeView supports user data attached to each tree item. The user data must be a pointer to a subclass of RWCollectable. To attach user data to a tree item, call myItem->SetUserData(myData). CTreeItem::GetUserData will retrieve the stored data.

You can either manage the destruction of the user data yourself, the default, or let the tree view delete the user data when deleting the associated tree item. To change this behavior, call CTreeView::SetDeleteUserData(theDelete) on your tree view. Pass TRUE to let the tree view manage the user data or FALSE if you wish to manage the data yourself. You can also call CTreeView::GetDeleteUserData to query the current behavior of the tree view.

```
void                          SetDeleteUserData(BOOLEAN);
BOOLEAN                       GetDeleteUserData() const;
const RWCollectable*      GetUserData() const;
RWCollectable*                GetUserData();
void                          SetUserData(RWCollectable);
```

32.3.3. Creating Dynamic Trees

Dynamic trees work on a callback basis. When a node is expanded, by the user or programmatically, the tree view requests the data for that node from its data source. The data source provides descriptions of all the tree items, including additional nodes.

To create a data source, "mix in" the CTreeSource abstract class and override its only method GetTreeData( ). *Mixing in* refers to the multiple inheritance technique of adding additional methods to an existing class. This approach makes it easy to convert existing classes into tree data sources with little modification.

Example: If you already have a tree data structure called CMyTree and you wish to use it to supply data to the tree view, you would create a new class as shown in this example.

```
class CMyTreeSource : public CMyTree, public CTreeSource
{
...
public
virtual public RWGVector(CTreeItemInfo)* GetTreeData(
                                    CTreeItem* theParent) const;
...
};
```

Once you have created a tree data source, pass the object to the SetSource( ) method on the root node.

Caution: You are responsible for creating and deleting the data source. Do not delete the data source until after you have destroyed the tree view.

Now you only need to write the GetTreeData( ) method to supply the tree data. Earlier, with a static tree, we populated the tree by taking the parent node information, modifying it to suit each child item, and adding the child to the parent. In GetTreeData( ) you do much the same, except the tree view takes care of actually adding the children. You only need to supply a vector of information objects describing each child.

Example: The following code shows an implementation of GetTreeData( ) to provide file directory information to a tree view:

```
RWGVector(CTreeItemInfo)* CTreeFileSource::GetTreeData(CTreeItem* theParent) const
{
    CTreeItem                   *anItem;
    CStringRW                   aPath;
    size_t                      fileCount = 0;
    RWGVector(CTreeItemInfo)*   anInfoList = new RWGVector(CTreeItemInfo);
```

```
        // Build path name from this parent up to the root node
        anItem = theParent;
        do
        {
                // Note that xvt_fsys_build_pathname cannot be done in place
                char new_path[MAXPATHLEN + 1];
                xvt_fsys_build_pathname(
                        new_path, NULL, CTabSet::GetField(anItem->GetString(), 0), aPath,
                                                                        NULL, NULL
                );
                aPath = new_path;
        } while( (anItem = anItem->GetParent()) != NULL );

        // Open as directory
        DIRECTORY aDirectory;

        // This call is not const correct, so we need to cast around
        if( xvt_fsys_convert_str_to_dir((char*)(const char*)aPath, &aDirectory) )
        {
                // Get a list of files and directories for this path
                xvt_fsys_save_dir();
                xvt_fsys_set_dir(&aDirectory);
                RWOrdered aFileList(
                        CStringCollection(xvt_fsys_list_files("", NULL,
                                                                TRUE)).asOrderedCollection()
                 );
                RWOrderedIterator aFile( aFileList );
                CStringRWC* aFileName;

                // For each file in the list, add an info object to the return vector
                while( (aFileName = (CStringRWC*)aFile()) != NULL )
                {
                        if( *aFileName != "." && *aFileName != ".." )
                        {
                                FILE_SPEC aFileSpec;

                                // Extend the info vector in chucks of 32
                                while( (*anInfoList).length() <= fileCount )
                                (*anInfoList).reshape( (*anInfoList).length() + 32 );

                                // Get information about this file
                                aFileSpec.dir = aDirectory;
                                xvt_str_copy(aFileSpec.name, (const char*)*aFileName);
                                xvt_str_copy(aFileSpec.type, "");
                                xvt_str_copy(aFileSpec.creator, "");
                                BOOLEAN isTerminal = !xvt_fsys_get_file_attr(
                                                                &aFileSpec, XVT_FILE_ATTR_DIRECTORY
                                                        );
                                // Copy the info object from parent and add child specifics
                                (*anInfoList)[fileCount] = theParent->Info();
                                (*anInfoList)[fileCount].itsString = *aFileName ";
                                (*anInfoList)[fileCount].itsHeight = CTreeItem::kBestHeight;
                                (*anInfoList)[fileCount].itsIsTerminal = isTerminal;
                                fileCount++;
                        }
                }
                // Reshape list to final size
                (*anInfoList).reshape( fileCount );
                xvt_fsys_restore_dir();
        }
        return anInfoList;
}
```

32.3.4. Changing Attributes of Items in a Tree View

CTreeView provides you with many choices in setting individual attributes on each line item of the view. You can do this when populating the tree by modifying the information objects, as in the previous examples, or you can change the attributes after creating the view by calling a method, such as SetFont( ), on the appropriate CTreeItem object.

Note: The inheritance of attributes applies when creating tree items, but not when attributes are changed after creation. If you were to change the color of the root node, it would not change the color of the root node's children that have already been created.

32.3.4.1. Instance Variables

The data members of the information object that you can set while populating the tree are:

itsColor	The foreground color of the item
itsFont	The font used for text
itsHeight	Item height. Generally you should use CTreeItem::kBestHeight
itsIndent	The amount children of the node should be indented to the right
itsString	The string that appears in this item
itsIsTerminal	FALSE if this is a node item, TRUE if this is a terminal item
itsImage	The image displayed if this node is a terminal item
itsCollapsedImage	The image displayed if this is a collapsed (closed) node item
itsExpandedImage	The image displayed if this is an expanded (open) node item
itsTabSet	A tab set used for tab justification

Given a CTreeItem object, you can change its attributes post creation. For each attribute listed above, there are corresponding "Set" and "Get" methods on CTreeItem: SetFont( ), SetTabSet( ), GetFont( ), etc. If possible, you should set the correct attributes using the information objects at creation to avoid view flashing.

See Also: You may wish to refer to the section on tree traversal, section 32.3.1.4 on page 32-31, for an example of finding specific tree items within the tree.

32.3.4.2. Setting Tab Stops

A CTabSet object contains an ordered list of justified tab stops represented by CTextTab objects.

To use tabs, build a CTabSet by adding CTextTabs and then associate it with a CTreeItem. The easiest approach is to build the tab set and associate it with the root node before populating the tree; this ensures that all items share the same tabs. However, you can also set the tab set of an item individually by setting the itsTabSet data member of the information object when you populate the tree or by calling the SetTabSet( ) method of CTreeItem.

To create a new CTextTab, pass the tab type (e.g., centered, left, right, and decimal) and the position to the constructor. The types are enumerations within CTextTab. Add each tab to the tab set with the AddTab( ) method of CTabSet.

Note: The position is relative to the left of the tree, not the left of the item, so that all fields line up regardless of the depth of an item within the tree.

Example: The following code builds a tab set for a tree view:

```
// Set up tabs
itsTabSet = new CTabSet;
itsTabSet->AddTab( CTextTab(CTextTab::Left, (UNITS)180) );
itsTabSet->AddTab( CTextTab(CTextTab::Decimal, (UNITS)250) );
```

Note: Like images and data sources, your application is responsible for creating and deleting tab sets and should not delete a tab set until after deleting the tree view that uses the tabs. Valid values for text justification types are CTextTab::Left, CTextTab::Right, CTextTab::Center, and CTextTab::Decimal.

32.3.4.3. Embedding Images

Image tab types in a CTextTab object tell the tab set to use the text in the field as a key string to find an image to display rather than draw the text. Thus, an image tab set maintains a list of images and the key string associated with each image.

To add an image to the tab set, use the AddImage( ) method of CTabSet. This method takes an image pointer and a key string as arguments.

Example: The following code builds on the previous example and adds an image tab:

```
itsCheckImage = new CImage("check.bmp");

// Set up tabs
itsTabSet = new CTabSet;
itsTabSet->AddTab( CTextTab(CTextTab::Left, (UNITS)180) );
itsTabSet->AddTab( CTextTab(CTextTab::Decimal, (UNITS)250) );
itsTabSet->AddTab( CTextTab(CTextTab::LeftImage, (UNITS)325) );
itsTabSet->AddImage(itsCheckImage, "check");
```

You can create toggle fields by adding two images associated with the keys on and off, for example, and then toggling that field of a tree item in response to mouse clicks.

See Also: For more information about the event handling for toggle fields see section 32.3.5 on page 32-39.

32.3.4.4. Manipulating Fields of a String

The previous sections described how to set both text and image tabs of a tree view. This section shows how to divide a string into tab fields, analogous to embedded tab characters in the text.

CTabSet provides a number of utility functions for getting and setting tab fields of a string so that you do not have to manually parse the string yourself. When you provide a string for a tree item, you can build up a complex string of multiple tab fields by appending each field separated by a tab object, CTabSet::TabString().

Example: This example builds on the file directory example introduced in section 32.3.3 . It appends: 1) a field displaying the file size, and 2) an image field containing a check mark. If the item is a directory, the file size field contains the string "...".

```
char fileSize[32];
if( isTerminal )
        xvt_str_sprintf( fileSize, "%ld",
                                xvt_fsys_get_file_attr(&aFileSpec,
                                XVT_FILE_ATTR_SIZE) );

 (*anInfoList)[fileCount] = theParent->Info();
if( isTerminal )
        (*anInfoList)[fileCount].itsString =
                *aFileName + CTabSet::TabString() +
                fileSize + CTabSet::TabString() +
                "check";
else
        (*anInfoList)[fileCount].itsString =
                *aFileName + CTabSet::TabString() +
                        "...";
```

Field Manipulation Methods of CTabSet

The field manipulation methods of CTabSet include:

```
CStringRW CTabSet::ToggleField(
                const CStringRW& theString,
                size_t theTab,                          // one based
                const CStringRW& theState1,             // also default
                const CStringRW& theState2
        );

CStringRW CTabSet::GetField(
                const CStringRW& theString,
                size_t theTab                           // one based
        );

CStringRW CTabSet::SetField(
                const CStringRW& theString,
                size_t theTab,                          // one based
                const CStringRW& theValue
        );
```

Notice that the tab field indices are one-based, i.e., they use 1 for the first field. These methods are static class methods so you do not need a tab set object to call them.

32.3.5. Processing Events in a Tree View

Tree events (CTreeEvent objects) are received by the tree view whenever the user performs any of the following operations:

- Focus changes (clicking on an item)
- Selections changes (changing the set of selected items)
- Node expansion
- Node collapse
- Mouse click in a field
- Mouse click in the title bar

The events are received through the DoCommand( ) methods; they have a command ID of TreeEventCmd and a data pointer that points to a CTreeEvent object.

Example: The following code shows how to trap tree events in your view, window, document, or application DoCommand( ) method:

```
void CTreeWin::DoCommand(long theCommand, void* theData)
{
	if( theCommand == TreeEventCmd )
	{
		CTreeEvent* anEvent = (CTreeEvent*)theData;
		CTreeItem* anItem = anEvent->GetTreeItem();

		if( anItem != NULL )
			switch( anEvent->GetType() )
			{
				case CTreeEvent::MouseClick:
					...
					break;
				case CTreeEvent::FocusIn:
					...
					break;
			}
	} else CWindow::DoCommand(theCommand,theData);
}
```

Example: The following code shows how to process events in a tree view:

```
case CTreeEvent::MouseClick:

	// If a second tab is detected, toggle the check mark
	if( anEvent->GetTabHit() == kCheckTab )
	{
		anItem->SetString(
			CTabSet::ToggleField(
				anItem->GetString(), kCheckTab, "check", "" )
		);
		anEvent->GetTree()->InvalidateItem( anItem );
	}
	break;
```

CTreeEvent provides several methods for querying the nature of the event. One important method is GetType(), which returns the specific type of tree event. Two other important methods, GetTree() and GetTreeItem(), return the IDs of the tree view sending the event and the tree item that the event describes, respectively.

Mouse-clicks in the title bar of a tree view generate events with the tree item set to NULL, so be sure that your application checks for a NULL tree item as shown in the previous code example.

Mouse-click events carry additional information such as the state of the modifier keys and the tab field hit by the mouse click, if any.

CTreeEvent Methods for Mouse Clicks

The following methods of CTreeEvent only apply to MouseClick and TitleMouseClick events.

```
CPoint          GetPoint() const;
short           GetButton() const;
BOOLEAN         GetIsShift() const;
BOOLEAN         GetIsControl() const;
size_t          GetTabHit() const;
```

The first four return the position of the mouse and key states as passed to the mouse handler. The last one, GetTabHit(), returns the tab field hit by the mouse click or zero if the mouse was clicked outside of any tab field. You can use this method to determine which part of a tree item the user clicked and possibly take some action such as toggling a picture associated with that field.

See Also: For more information on manipulating fields of a string, see the code example in section 32.3.4.4 on page 32-37.

32.3.6. Expansion Policies

Tree view expansion policies determine what the tree view does when: 1) the user expands a node, or 2) when you expand a node programmatically.

ExpandOne
: CTreeView::ExpandOne is the default behavior, which means only expand the node and none of its children.

ExpandAll
: CTreeView::ExpandAll says to expand the node and all of its children.

ExpandRestore
: CTreeView::ExpandRestore says to expand the node and the children that were expanded before this node was collapsed.

You can change the expansion policy of a tree view (after it has been created) by calling its SetExpansionPolicy( ) method.

32.3.7. Tree Styles

The tree view style controls how the tree view draws the tree "skeleton."

StyleOrthogonal
: CTreeView::StyleOrthogonal is the default, which displays the skeleton as vertical lines joining children with horizontal stubs.

StyleSlant
: CTreeView::StyleSlant substitutes angled stubs for the horizontal stubs.

StyleNone
: CTreeView::StyleNone suppresses skeleton drawing.

DrawRoot
: A related attribute of the tree view, DrawRoot, controls whether to draw the root node. You can change this with a call to the method SetDrawRoot( ) of CTreeView and passing TRUE or FALSE.

Example: The following code shows how to use tree view as a list:

```
// Don't draw connecting lines
itsTree->SetStyle( CTreeView::StyleNone );
// and don't draw the root item
itsTree->SetDrawRoot( FALSE );
```

32.3.8. Selection Policies

Tree view selection policies determine what the tree view does when the user clicks on a tree item. (Clicking is equivalent to "navigation" by keyboard.)

SelectMany
: CTreeView::SelectMany is the default, which means the user can select an item by clicking on it and select additional items using shift-click.

SelectOne
: CTreeView::SelectOne allows the user to select only one item at a time.

SelectNone
: CTreeView::SelectNone prevents any user selection. This policy does not, however, prevent the user from setting focus to an item.

Focus differs from selection and only indicates the last item that the user clicked on. The tree view indicates selected items with a dark solid background and the focus item (there can be only one) by a dotted rectangle around the item.

You can get and set the current focus item with the SetFocusItem( ) and GetFocusItem( ) methods of CTreeView.

32.3.9. Sorting and Re-sorting Tree Items

Addition of sorting options can enhance certain applications. The tree view normally displays items in the order that you have provided them from the data source. However, you can sort tree items within a node and recursively from a node by calling the CTreeItem method SortChildren( ).

To sort an entire tree, call SortChildren( ) on the root and pass TRUE for the last parameter, which indicates that you want to sort recursively down the tree. SortChildren( ) takes two arguments, a CTreeSorter object and a Boolean flag to indicate a single level (FALSE) or recursive (TRUE) sort.

The Compare( ) method takes two tree items as arguments and returns a value indicating how the two items compare. The return values are CTreeSorter::GreaterThan, CTreeSorter::LessThan, and CTreeSorter::EqualTo.

Note: Like other objects passed to the tree view, your code is responsible for creating and destroying the sort object and you must not destroy the sort object until you have destroyed the tree view.

XVT-Power++ provides a simple tree sort object, CTreeStringSorter, that sorts items alphabetically based on the contents of any single field.

Example: The following code sorts the entire tree based on the contents of the second field:

```
CTreeSorter* mySorter = new CTreeStringSorter( 2 );
itsTree->GetRoot()->SortChildren( mySorter, TRUE );
```

For more complex sorting, subclass CTreeSorter and override the Compare( ) method.

Example: This code snippet is taken from the CTreeStringSorter class, and demonstrates how to override the CTreeSorter::Compare( ) method.

```
CTreeSorter::Comparison                 CTreeStringSorter::Compare(
                                                const CTreeItem* theFirstItem,
                                                const CTreeItem* theSecondItem
                                        ) const
{
        int     result;

        if( itsSortTab > 0 )
                result =
                        CTabSet::GetField( theFirstItem->GetString(), itsSortTab ).compareTo(
                                CTabSet::GetField( theSecondItem->GetString(), itsSortTab )
                        );
        else
                result =
                        theFirstItem->GetString().compareTo(
                                theSecondItem->GetString()
                        );

        if( result < 0 )
                return LessThan;
        else if( result == 0 )
                return EqualTo;
        else
                return GreaterThan;
}
```

32.3.10. Changing Mouse Behavior

If you are an experienced XVT-Power++ programmer you can change the mouse behavior of a tree view by installing your own mouse handler. To install a mouse handler, subclass CTreeMouseHandler and override the methods, such as DoDown( ), that you want to change by using the source code of CTreeMouseHandler as a guide. After you create the new mouse handler, subclass CTreeView and override the CreateMouseHandler() method to create your own mouse handler rather than using the default CTreeMouseHandler. You will, of course, need to provide constructors for your new class.

Example: The following code shows how to install a customized mouse handler:

```
class MyTreeMouseHandler : public CTreeMouseHandler
{
        ...
};

class MyTreeView : public CTreeView


{
        public:
                MyTreeView( CSubview *theEnclosure, const CRect& theRegion) :
                        CTreeView( theEnclosure, theRegion ) { }

                virtual CTreeMouseHandler* CreateMouseHandler()
                        {
                                return new MyTreeMouseHandler;
                        }
};
```

See Also: For more information about changing mouse behavior with mouse handlers, see section 19.2.1 on page 19-6.

33

INTERNATIONALIZATION AND LOCALIZATION

This chapter provides an overview of concepts you need to understand to write a DSC++ application that is easily internationalized and localized. Cross-references point to other sections of this *Guide* (or other XVT documentation) where you can obtain specific information about particular subjects related to internationalization (I18N).

This chapter also describes an overall methodology for writing XVT applications that support locales and international languages, including specific steps you can follow to implement the methodology. The chapter also lists compile constants you can use to take advantage of XVT's pre-translated resources.

See Also: For the most detailed information on internationalization and localization of XVT applications, refer to the "Multibyte Character Sets and Localization" chapter of the *XVT Portability Toolkit Guide*.

33.1. Multibyte Character Set and Localization Support

XVT-Power++ (along with its underlying PTK libraries) has recently added support for application development for multiple locales and international languages. All XVT classes, methods, and functions, including text edit object functions, now handle multibyte strings. New string processing API functions portably process multibyte strings. XVT-Architect has a new user interface for defining locales and the corresponding codesets used by that locale. However, existing XVT applications that do not need to be internationalized are unaffected.

Your XVT application can receive and process keyboard input that contains international (multibyte) characters. Input Method Editors

(IMEs), provided by the native window systems or operating systems, can be used to enter composed characters.

Three multibyte codesets are explicitly supported: ASCII, Shift-JIS, and EUC. Character sets that can be supported must, at least, provide the invariant character set as a subset.

Note: XVT does not *directly* support the Unicode character set. An application can always use this character set by converting to the proper multibyte codeset when calling the XVT API.

33.1.1. Externalized Resource Files

XVT applications can allow the user to select the language/locale of the user interface at application startup time. The user selects the resource file used by the application before invoking the application.

All resources are separated from the executable code and can be selected at application startup time. This mainly affects the PC and Macintosh platforms, since the Motif platform has always provided separate resource files. Of course, running any localized application requires that the appropriate operating system, window system, and fonts are installed and set up correctly for the selected language and locale.

XVT has already localized its resources in English, Japanese, French, German, and Italian. For these languages, XVT provides localized files containing all of the standard resources used by the Portability Toolkits. Localized versions of the XVT standard help topics for each platform are also provided in these languages.

XVT's URL compiler, **curl**, handles quoted strings containing multibyte characters, including strings used for the following:

- menu and menu item titles
- window, dialog, and control titles
- edit control and text edit object initial text
- font family names
- font mapper native descriptors
- string resources
- user data

See Also: XVT can support any left-to-right language. To see a complete list of supported languages, refer to Appendix A, *Appendix A: Languages and Codesets*.
For more details about how to generate XVT-Architect applications

that use a particular layer at runtime, see section 9.4 on page 9-5. To see a list of predefined filenames recognized in the XVT Portability Toolkits, refer to section 33.5 on page 33-10.

33.1.2. How to Adapt an Application

Remember to debug your application prior to undertaking the localization effort. Resolving localization issues is much easier if your application is working well to begin with. Although this seems to add extra steps to your development process, it actually reduces the total amount of effort by cleanly separating coding problems. As you gain experience in adapting applications, you may begin to find it easier to write localized XVT applications from scratch.

See Also: Applications can be localized to some languages without using wide (multibyte) characters. However, localizing to other languages, such as Japanese, will require "special characters." For more information, refer to the "Multibyte Character Sets and Localization" chapter of the *XVT Portability Toolkit Guide*.

33.1.3. More Support for Internationalized Applications

The help compiler, **helpc**, handles help text containing multibyte characters. The help viewer, **helpview**, displays help text containing multibyte characters.

Furthermore, file and pathnames may contain multibyte characters. All PTK functions and data types that accept file or pathname strings are multibyte capable.

The error processor, **errscan**, produces the error message file **ERRCODES.TXT**—you can localize this file for any language. Furthermore, attributes are provided that allow you to explicitly set the path to **ERRCODES.TXT**.

The utility compiler, **maptabc**, reads a text file that defines a mapping of a codeset into the Unicode codeset and creates a binary file for that codeset mapping. The binary file is in turn used by the XVT function xvt_str_create_codeset_map.

33.2. Internationalization

Internationalization requires disassociating any locale-sensitive information from your application and encapsulating it in external files such as resource files. Any locale-sensitive processing operations also must be encapsulated and handled in a general manner.

33.2.1. Considerations for Internationalization

Some of the factors you must consider when internationalizing include the following:

- String literals
- Special strings and data that require locale-specific formatting or parsing (i.e., sprintf, textual representation of numbers, date/time formats, proper word- and line-wrap)
- Dialog and window layout
- Graphics (icons, bitmaps)
- Colors
- Font references
- Keyboard modifiers, mnemonics, and accelerators
- Help source files

If you are using character codesets that use wide character or multibyte encoding schemes, your application code for manipulation of strings must be modified to handle these character codesets. The following string operations are candidates for modification or replacement:

- Collation
- Parsing
- Incrementing or decrementing character pointers
- Character or string comparison
- Handling upper and lower case (some languages are indifferent)
- Conversion between character codesets

Appropriate text and graphic object positions and dimension data should also be removed from the application and be placed instead in external resource files.

33.2.2. Specific Instructions for XVT-Architect Users

If you are using XVT-Architect, the basic steps for internationalizing a DSC++ application are as follows:

1. Create the basic application.
2. Define the global locales (and their corresponding codesets) that XVT-Architect needs to support.
3. Create a layer for each locale that must be supported by the application.
4. Localize the objects of each layer (translate strings, change colors, replace icons, etc.).
5. Generate XVT-Architect factories with the needed layers.
6. Add code to the application to select the default layer based on the locale.

Runtime Considerations

In an application generated by XVT-Architect, you can ensure that the application uses a particular layer at runtime by setting the default factory creation in your application class constructor like this:

```
factory.SetDefaultLayer(CFooFactory::FRENCH);
```

Note: For this to work, your project must have a layer named “FRENCH” defined (obviously, for a French-speaking locale).

See Also: For more details about localizing menus, see section 7.1.2.3 on page 7-9.
For more details about localizing resources, see section 28.5 on page 28-7.
For more details about localizing icons, see section 17.2.7 on page 17-9.

33.3. Localization

Localization is quite straightforward once your application has been internationalized. The biggest part of localization is placing string literals in an external file that can be modified as required by specific locales. If you are using XVT-Architect, this is done for you automatically by separating the strings into different layers.

33.3.1. Considerations for Localization

Your application must be localized for each unique environment in which it will operate. In addition to the steps involved when using XVT-Architect, you should be aware of other steps which vary slightly depending on the application and the selected locales. Generally speaking, plan on the following steps:

1. Decide which character codeset to use for translation depending on: 1) which languages you need to support, and 2) which operating systems your application must work with. Different codesets used on the various platforms that XVT supports are listed in section A.2 in Appendix A
2. Translate string literals to the target language.
3. Set up special strings such as dates and times for formatting.
4. Select the appropriate keyboard modifiers, mnemonics, and accelerators.
5. Select fonts appropriate to the character codeset.
6. Provide locale-specific icons and colors.
7. Adjust text and graphic object sizes and positions.
8. Compile locale-specific resource and help files.
9. Establish the proper operating/window system locale-specific environment (set up environment variables, code pages, etc.).
10. Set the application locale environment information (locale information can be bound at application build time or application startup time.

See Also: To see hundreds of examples of international symbols used in various fields of endeavor, refer to *Symbol Sourcebook: An Authoritative Guide to International Graphic Symbols*, by Henry Dreyfuss, published by Van Nostrand Reinhold, New York, N.Y., 1984.

33.3.2. Compile-time Considerations

In addition to localizing the resources and code of your application, you will need to make sure that the application is compiled to take advantage of localization support available from within XVT's libraries and resources.

Building the locale-specific executable requires the setting of one or more specific #defines. XVT source code files are "localized" when XVT_LOCALIZABLE is defined, and switch to a specific language based upon other #defines, as well. To build the locale-specific executable, follow these additional steps:

1. Modify your makefile or makefile templates to build localized versions of your resources. If you wish to build, for example, a German version, you would also define LANG_GER_W52. The various compile constants you can use are listed in Table 33.2 on page 33-11.

   Refer to the example at the end of this section for an example of how to modify a UNIX makefile. Different programmers or organizations have their own personal preferences and different platforms will require slightly different syntax.

   On some platforms, you may need to run curl manually from the command line, as shown in the following XVT/Win16 curl compile statement:

   ```
   curl -r rcwin -i..\..\ include -dLANG_GER_W52
                 -dLIBDIR=.\..\..\lib app.url
   ```

   Although the command line shown above is printed on two lines, you should enter a command line as a single line.

   You now have a resource file—if you view it, you will see, in this case, that all strings are now in German.

2. If your makefile did not completely finish the build, you should now complete any unfinished steps in your build process.

Example: This example shows a UNIX makefile that builds a German version of an XVT application:

```
# Define localized options.
# Start a German build.

LOCALIZE_OPTS     = -dLANG_GER_W52
CC_OPTS           = -c $(INC_PATH)
CURL =            $(XVT_DSP_DIR)/bin/curl
...
#
# Include the defines in all source code compilations
.c.o
      $(CC) $(CC_OPTS) $(LOCALIZE_OPTS) $<
# Also pass them to curl
app.uil: app.url
      $(CURL) $(CURL_OPTS) $ (LOCALIZE_OPTS) app.url
...
```

See Also: For more information about specifying resources with URL, see the "Resources and URL" chapter in the *XVT Portability Toolkit Guide*. For more information about using **curl**, including a list of **curl** options, see the online *XVT Portability Toolkit Reference*.

33.4. Localized PTK Resources

For your convenience, XVT provides compatible localizations of standard PTK and XVT-Power++ resources and help text; the various codesets used to provide these resources are listed in Table 33.1.

Language:	XVT/Win16, XVT/Win32:	XVT/PM:	XVT/Mac:	XVT/XM:
US English	**ANSI** (Windows 1252)	**ASCII** (Code page 850)	**Mac-Roman**	**ASCII** (ISO 646)
French German Italian	**ANSI** (Windows 1252)	**ISO Latin-1** (Code page 850)	**Mac-Roman**	**ISO-Latin-1** (ISO 8859-1)
Japanese	**Shift-JIS** (Codeset 932)	**Shift-JIS** (Code page 932)	**Shift-JIS** (Mac-Japanese)	**Shift-JIS, AJEC** (Japanese EUC)

Table 33.1. Localized versions of standard PTK resources and help text predefined for five languages

See Also: Filenames and filenaming conventions for the files listed in Table 33.1 are discussed in section 33.5.

Note: You are not limited to these localizations, but you may want to use these as a basis for localizing your own applications.

The XVT PTK data is externalized in one of three file types for localization by your application:

- Standard XVT resource strings (URL)
- Standard XVT help strings
- Standard XVT error messages

For convenience, a set of XVT constants is provided to allow the standard XVT resource strings and help text files to be easily included in your applications; these constants are listed in Table 33.2 on page 33-11.

33.5. PTK Filenaming Conventions

XVT's PTK uses a set of *conventions* for defining relevant constants and filenames using three character abbreviations for language and three or four character abbreviations for character codeset (see Appendix A for a complete list of these abbreviations):

- Language constant

  LANG_<3 character language>_<3-4 character codeset>

  Default: U.S. English ASCII does not require a language constant

- PTK-level URL standard resource strings file

  u<3 character language><3-4 character codeset>**.h**

  Default: **uengasc.h** (U.S. English ASCII)

- XVT-Power++ URL standard resource strings file

  p<3 character language><3-4 character codeset>**.h**

  Default: **pengasc.h** (U.S. English ASCII)

- XVT standard help text file (included by **xvt_help.csh** to provide help topic text on reserved help topic symbols)

  h<3 character language><3-4 character codeset>**.csh**

  Default: **hengasc.csh** (U.S. English ASCII)

- XVT error code strings file:

  e<3 character language><3-4 character codeset>**.txt**

  Default: **ERRCODES.TXT** (US. English ASCII, the default filename does not adhere to this convention)

Table 33.2 lists the language and character codeset constants and filenames recognized in the XVT Portability Toolkit. XVT resource and help compilers recognize these constants for automatic inclusion of appropriate filenames:

Language:	Compile Constant:	PTK URL Strings Filename:	XVT-Power++ URL Strings Filename:	Help Text Filename:
XVT/XM:				
U.S. English	(Default)	**uengasc.h Đ**	**pengasc.h Đ**	**hengasc.csh Đ**
French	LANG_FRE_IS1	**ufreis1.h Đ**	**pfreis1.h Đ**	**hfreis1.csh Đ**
German	LANG_GER_IS1	**ugeris1.h Đ**	**pgeris1.h Đ**	**hgeris1.csh Đ**
Italian	LANG_ITA_IS1	**uitais1.h Đ**	**pitais1.h Đ**	**hitais1.csh Đ**
Japanese (SJIS)	LANG_JPN_SJIS	**ujpnsjis.h Đ**	**pjpnsjis.h Đ**	**hjpnsjis.csh Đ**
Japanese (EUC)	LANG_JPN_UJA	**ujpnuja.h Đ**	**pjpnuja.h Đ**	**hjpnuja.csh Đ**
Norwegian	LANG_NOR_IS1	**unoris1.h**	**pnoris1.h**	**hnoris1.csh**
Russian	LANG_RUS_IS1	**urusis1.h**	**prusis1.h**	**hrusis1.csh**
Spanish	LANG_SPA_IS1	**uspais1.h**	**pspais1.h**	**hspais1.csh**
Swedish	LANG_SWE_IS1	**usweis1.h**	**psweis1.h**	**hsweis1.csh**
XVT/Win16, XVT/Win32:				
U.S. English	(Default)	**uengasc.h Đ**	**pengasc.h Đ**	**hengasc.csh Đ**
French	LANG_FRE_W52	**ufrew52.h Đ**	**pfrew52.h Đ**	**hfrew52.csh Đ**
German	LANG_GER_W52	**ugerw52.h Đ**	**pgerw52.h Đ**	**hgerw52.csh Đ**
Italian	LANG_ITA_W52	**uitaw52.h Đ**	**pitaw52.h Đ**	**hitaw52.csh Đ**
Japanese	LANG_JPN_SJIS	**ujpnsjis.h Đ**	**pjpnsjis.h Đ**	**hjpnsjis.csh Đ**
Norwegian	LANG_NOR_W52	**unorw52.h**	**pnorw52.h**	**hnorw52.csh**
Russian	LANG_RUS_W51	**urusw51.h**	**prusw51.h**	**hrusw51.csh**
Spanish	LANG_SPA_W52	**uspaw52.h**	**pspaw52.h**	**hspaw52.csh**
Swedish	LANG_SWE_W52	**uswew52.h**	**pswew52.h**	**hswew52.csh**

Note: XVT provides only those localized files denoted by **Đ**.

Table 33.2. Language and character codeset constants and filenames recognized in XVT-Power++ and the XVT Portability Toolkit (part 1 of 2)

Language:	Compile Constant:	URL Strings Filename:	XVT-Power++ URL Strings Filename:	Help Text Filename:
XVT/PM:				
U.S. English	(Default)	**uengasc.h Ð**	**pengasc.h Ð**	**hengasc.csh Ð**
French	LANG_FRE_D850	**ufred850.h Ð**	**pfred850.h Ð**	**hfred850.csh Ð**
German	LANG_GER_D850	**ugerd850.h Ð**	**pgerd850.h Ð**	**hgerd850.csh Ð**
Italian	LANG_ITA_D850	**uitad850.h Ð**	**pitad850.h Ð**	**hitad850.csh Ð**
Japanese	LANG_JPN_SJIS	**ujpnsjis.h Ð**	**pjpnsjis.h Ð**	**hjpnsjis.csh Ð**
Norwegian	LANG_NOR_D865	**unord865.h**	**pnord865.h**	**hnord865.csh**
Russian	LANG_RUS_D866	**urusd866.h**	**prusd866.h**	**hrusd866.csh**
Spanish	LANG_SPA_D850	**uspad850.h**	**pspad850.h**	**hspad850.csh**
Swedish	LANG_SWE_D850	**uswed850.h**	**pswed850.h**	**hswed850.csh**
XVT/Mac:				
U.S. English	(Default)	**uengasc.h Ð**	**pengasc.h Ð**	**hengasc.csh Ð**
French	LANG_FRE_MRMN	**ufremrmn.h Ð**	**pfremrmn.h Ð**	**hfremrmn.csh Ð**
German	LANG_GER_MRMN	**ugermrmn.h Ð**	**pgermrmn.h Ð**	**hgermrmn.csh Ð**
Italian	LANG_ITA_MRMN	**uitamrmn.h Ð**	**pitamrmn.h Ð**	**hitamrmn.csh Ð**
Japanese	LANG_JPN_SJIS	**ujpnsjis.h Ð**	**pjpnsjis.h Ð**	**hjpnsjis.csh Ð**
Norwegian	LANG_NOR_MRMN	**unormrmn.h**	**pnormrmn.h**	**hnormrmn.csh**
Russian	LANG_RUS_MCYR	**urusmcyr.h**	**prusmcyr.h**	**hrusmcyr.csh**
Spanish	LANG_SPA_MRMN	**uspamrmn.h**	**pspamrmn.h**	**hspamrmn.csh**
Swedish	LANG_SWE_MRMN	**uswemrmn.h**	**pswemrmn.h**	**hswemrmn.csh**

Note: XVT provides only those localized files denoted by **Ð**.

Table 33.2. Language and character codeset constants and filenames recognized in XVT-Power++ and the XVT Portability Toolkit (part 2 of 2)

The file **url.h** has a conditional compile statement for the compile constants defined in the preceding table (such as LANG_JPN_SJIS) that will include the appropriate resource strings file (for example, **ujpnsjis.h**). This constant can be defined on the **curl** compile line or in your URL file. The file **xvt_help.csh** has a conditional compile statement which will include the appropriate help strings file (like **hjpnsjis.csh**). The constant can be defined on the **helpc** compile line or in the help file. The file **xvt_help.csh** should be included in your application help source (**.csh**) file if you intend to use the XVT default help topics.

XVT supplies only the localized resources and help text noted on the previous pages (U.S. English, French, German, Italian and Japanese). Use these localizations as a basis for adapting your own application locales. You may also want to add your own language constants.

See Also: For more information on using localized resources with your XVT applications, refer to the *XVT Platform-Specific Book* for your particular platform.
For a complete list of XVT language and character codeset abbreviations, refer to Appendix A.

A

APPENDIX A: LANGUAGES AND CODESETS

This appendix lists XVT abbreviations for languages and character codesets. However, XVT does not directly support all these languages and character codesets. The five languages that are fully supported at this time are:

- Japanese
- Italian
- French
- German
- English

Because XVT string resources are now stored in a separate file, your application can be programmed in any language, but the five languages listed above are the only languages for which pre-translated resources are shipped with the XVT Portability Toolkits. If you need to support any other language, you must translate many standard resources yourself, such as the strings that are displayed in XVT's predefined dialogs.

Remember that bi-directional languages are not supported. However, for your convenience, all recognizable language abbreviations are listed below—both bi-directional and left-to-right. You need to know the language abbreviation, because you must use it as part of the filename for the file that contains your internationalized resources. (For more information on filenaming conventions in internationalized applications, see section 33.5 on page 33-10.)

All listed languages are uni-directional, left-to-right unless specified otherwise.

A.1. Language Abbreviations

XVT <3 character language code> abbreviations are as follows:

Abbrev:	Lanquage:	Direction:
afr	Afrikaans	
alb	Albanian	
amh	Amharic	
ara	Arabic	bidirect
arm	Armenian	
asm	Assamese	
aze	Azerbaijani	bidirect
bah	Bahasa Indonesia	
bal	Baluchi	
bel	Belorussian	
ben	Bengal	
bih	Bihari	
bul	Bulgarian	
bur	Burmese	
cat	Catalon	
che	Chewa	
chi	Chinese	
chu	Chuang	
cop	Coptic	
cro	Croatian	
cyr	Cyrillic	
cze	Czech	
dan	Danish	
dar	Dari Persian	bidirect
dut	Dutch	
dzo	Dzongkha	
eng	English	
est	Estonian	
ewe	Ewe	
fae	Faeoese	
far	Farsi	bidirect
fij	Fijian	
fin	Finnish	
fle	Flemish	
fre	French	
ful	Fulani	

Abbrev:	Lanquage:	Direction:
gal	Galla	
geo	Georgian	
ger	German	
gre	Greek	
grl	Greenlandic	
guj	Gujarati	
hau	Hausa	
hbr	Hebrew	bidirect
hin	Hindi	
ibo	Ibo	
ice	Icelandic	
iri	Irish Gaelic	
ita	Italian	
jpn	Japanese	(also top/bot)
jav	Javanese	
kan	Kanarese	
kas	Kashmiri	
kaz	Kazakh	
khm	Khmer	
kir	Kirghiz	
kor	Korean	(also top/bot)
kur	Kurdish	bidirect
kuy	Kuy	
lad	Ladino	
lao	Laotian	
lap	Lappish	
lat	Latin	
ltv	Latvian	
lav	Lavana	
lit	Lithuanian	
lux	Luxembourgian	
mac	Macedonian	
mad	Madurese	
mag	Magyar	
mlg	Malagasy	
mag	Malay	bidirect
mlm	Malayalam	
mld	Maldivian	
mlt	Maltese	
mao	Maori	
mar	Marathi	

Abbrev:	Lanquage:	Direction:
mol	Moldavian	
mon	Mongasque	
mng	Mongolian	top/bot
nau	Nauruan	
nep	Nepali	
nor	Norwegian	
ori	Oriyan	
pal	Pali	
pas	Pashto	bidirect
pid	Pidgin	
pol	Polish	
por	Portuguese	
pun	Punjabi	
rom	Romanian	
rmh	Romansch	
rua	Ruandan	
run	Rundi	
rus	Russian	
sam	Sami	
smn	Samoan	
san	Sango	
snk	Sanskrit	
ser	Serbian	
ses	Sesotho	
set	Setswana	
sho	Shona	
sin	Sindhi	bidirect
snh	Sinhalese	
slo	Slovak	
sln	Slovenian	
som	Somali	
spa	Spanish	
sud	Sudanese	
swa	Swahili	
swz	Swazi	
swe	Swedish	
tad	Tadzhik	
tag	Tagalog	
tak	Taki-Taki	
tam	Tamil	

Abbrev:	Lanquage:	Direction:
tel	Telugu	
tha	Thai	
tib	Tibetan	
tig	Tigre	
tgr	Tigrinya	
ton	Tongan	
tsw	Tswana	
tur	Turkish	
trk	Turkmen	
tuv	Tuvaluan	
ukr	Ukrainian	
urd	Urdu	bidirect
uzb	Uzbek	
ven	Venda	
vie	Vietnamese	
xho	Xhosa	
yid	Yiddish	bidirect
yor	Yoruba	
zul	Zulu	

A.2. Character Codeset Abbreviations

The XVT <3-4 character codeset> abbreviations are one of the following:

Abbrev:	Codeset:	Languages:
General use:		
inv	Invariant ASCII Codeset (ASCII Subset)	
asc	ASCII Codeset (7-bit)	
jis	JIS	Japanese
sjis	Shift-JIS	Japanese
XVT/XM:		
is1	ISO 8859-1 (ISO Latin-1)	Western European (Danish, Dutch, English, Faeroese, Finnish, French, German, Icelandic, Italian,

		Norwegian, Portuguese, Spanish, Swedish)
is2	ISO 8859-2 (ISO Latin-2)	Eastern European (Albanian, Czechoslovakian, English, German, Hungarian, Polish, Romanian, Serbo-Croatian, Slovak, Slovene)
is3	ISO 8859-3 (ISO Latin-3)	Southeastern Europe (Afrikaans, Catalan, Dutch, English, Esperanto, German, Italian, Maltese, Spanish, Turkish)
is4	ISO 8859-4 (ISO Latin-4)	Northern European (Danish, Estonian, English, Finnish, German, Greenlandic, Lappish, Latvian, Lithuanian, Norwegian, Swedish)

Abbrev:	**Codeset:**	**Languages:**
is5	ISO 8859-5 (ISO Cyrillic)	Bulgarian, Belorussian, English, Macedonian, Russian, Serbo-Croatian, Ukrainian
is6	ISO 8859-6 (ISO Arabic)	Arabic, English
is7	ISO 8859-7 (ISO Greek)	English, Greek
is8	ISO 8859-8 (ISO Hebrew)	English, Hebrew
is9	ISO 8859-9 (ISO Latin-5)	Western European (Danish, Dutch, English, Faeroese, Finnish, French, German, Italian, Norwegian, Portuguese, Spanish, Swedish, Turkish)
is10	ISO 8859-10 (ISO Latin-6)	Danish, English, Estonian, Faeroese, Finnish, German, Greenlandic, Icelandic, Lappish, Latvian, Lithuanian, Norwegian, Swedish

uja	EUC-JA	Japanese, English
uctw	EUC-CH_tw	Traditional Chinese, English
uccn	EUC-CH_cn	Simplified Chinese, English
uko	EUC-KO	Korean, English

XVT/PM:

d437	DOS code page 437US
d850	DOS code page 850Multilingual
d852	DOS code page 852Slavic
d855	DOS code page 855Cyrillic
d857	DOS code page 857Turkish
d860	DOS code page 860Portuguese
d861	DOS code page 861Icelandic
d863	DOS code page 863Canadian-French
d865	DOS code page 865Norwegian
d866	DOS code page 866Russian
d874	DOS code page 874Thai

Abbrev:	**Codeset:**	**Languages:**

XVT/Win16, XVT/Win32:

big5	Big-5	Traditional Chinese
gbc	GB-Code	Simplified Chinese
cns	CNS	Simplified Chinese
kcs	KCS	Korean
w50	Windows 1250WINEE	
w51	Windows 1251WINCYR	
w52	Windows 1252ANSI	
w53	Windows 1253WINGREEK	
w54	Windows 1254WINTURK	
w55	Windows 1255WINHEB	
w56	Windows 1256WINARAB	
w57	Windows 1257WINBALT	
d874	DOS code page 874WINTHAI	

XVT/Mac:

mrmn	Mac-Roman	Roman-based languages
mce	Mac-CE	Central and Eastern Europe
mcro	Mac-Croatian	Croatian
mheb	Mac-Hebrew	Hebrew, Ladino, Yiddish
mcyr	Mac-Cyrillic	Belorussian, Bulgarian,

		Kazakh, Kirghiz, Macedonian, Moldavian, Russian, Serbian, Tadzhik, Turkmen, Ukrainian, Uzbek, Azerbaijani, Mongolian
mtha	Mac-Thai	Thai, Kuy, Lavna, Sanskrit, Pali
mara	Mac-Arabic	Arabic, Baluchi, Dari Persian, Farsi, Kurdish, Pashto, Sindhi, Urdu, Azerbaijani, Kashmiri, Malay
mice	Mac-Icelandic	Icelandic
mgre	Mac-Greek	Greek, Coptic
mtu	Mac-Turkish	Turkish
big5	Big-5	Traditional Chinese
kcs	KCS	Korean

B

APPENDIX B: TDI EVENTS IN XVT-POWER++

This appendix lists the XVT-Power++ classes that are TDI-aware. A TDI-aware class is capable of responding to one or more TDI events and may even generate TDI events of its own.

B.1. TDI Events Received

Table B.1 lists the TDI events recognized by XVT-Power++ views. This implies that the classes listed provide an overridden DoUpdateModel() method that implements the event handling.

The "TDI value" column describes the type preferred by a class when handling a command. In most cases, this means that the TDI value is convertible to the desired type (see the description for CTdiValue::IsConvertible()). When different value types are accepted for the same command, you can use a call to CTdiValue::GetType() to determine if an exact type match can be made before trying to implement type conversions.

In the table, entries where a TDI value is followed by a "*" indicate that the value is traversed using the CTdiValue iteration interface to handle the case where a list of values was actually sent.

Class	TDI Event	TDI Value	Comments
CButton	TDISelectCommand	CTdiValue::BOOLEAN_TYPE	Toggles selection
CListBox	TDIReplaceCommand	CTdiValue::INTEGER_TYPE	Selects line
	TDIClearCmd	none	Clears selections
	TDIReplaceCmd	CTdiValue::STRING_TYPE	Selects line
	TDIOptionClearCmd	none	Clears contents
	TDIOptionReplaceCmd	CTdiValue::STRING_TYPE *	Replaces contents
	TDIOptionAppendCmd	CTdiValue::STRING_TYPE *	Appends lines
	TDIFirstCmd	none	Selects first item
	TDILastCmd	none	Selects last item
	TDINextCmd	none	Selects next item
	TDIPreviousCmd	none	Selects previous item
CNativeList	TDIOptionClearCmd	none	Clears contents
	TDIOptionReplaceCmd	CTdiValue::STRING_TYPE *	Replaces contents
	TDIOptionAppendCmd	CTdiValue::STRING_TYPE *	Appends lines
CNativeSelectList	TDIReplaceCmd	CTdiValue::INTEGER_TYPE *	Selects line
	TDIAppendCmd	CTdiValue::INTEGER_TYPE *	Multi-selects line
	TDIClearCmd	none	Clears selections
	TDIReplaceCmd	CTdiValue::STRING_TYPE *	Selects line
	TDIAppendCmd	CTdiValue::STRING_TYPE *	Multi-selects lines
CNativeTextEdit	TDIClearCmd	none	Clears text
	TDIAppendCmd	CTdiValue::STRING_TYPE	Appends text
	TDIReplaceCmd	CTdiValue::STRING_TYPE	Replaces contents
CRadioGroup	TDISelectCmd	CTdiValue::STRING_TYPE	Selects button
CView	TDIClearCmd	none	Clears itsTitle
	TDIAppendCmd	CTdiValue::STRING_TYPE	Appends to itsTitle
	TDIReplaceCmd	CTdiValue::STRING_TYPE	Replaces itsTitle
NCheckBox	TDISelectCmd	CTdiValue::BOOLEAN_TYPE	Toggles selection
NScrollBar	TDIReplaceCmd	CTdiValue::FLOAT_TYPE	Changes thumb position

Table B.1. TDI events received by XVT-Power++ classes

B.2. TDI Events Sent

Table B.2 lists the TDI events generated by XVT-Power++ views. In the table, entries where a TDI value is followed by a "*" indicate the actual values are bundled in a CTdiListValue object which your application should traverse using the CTdiValue iteration interface.

Class	TDI Event	TDI Value	Comments
CBoss	User defined	none	Sent with each DoCommand
CButton	TDISelectCmd	CTdiBooleanValue	Indicates selection
CListBox	TDIOptionReplaceCmd	CTdiStringValue*	Contents
	TDIReplaceCmd	CTdiStringValue	Selection text
	TDIIndexCmd	CTdiIntegerValue	Selection index
	TDIOptionClearCmd	none	No selection
CNativeList	TDIOptionClearCmd	none	No contents
	TDIOptionReplaceCmd	CTdiStringValue*	Contents
	TDIOptionAppendCmd	CTdiStringValue*	Newly appended contents
CNativeSelectList	TDIClearCmd	none	No selections
	TDIReplaceCmd	CTdiStringValue*	Selection text
	TDIIndexCmd	CTdiIntegerValue*	Selection index
CNativeText	TDIAppendCmd	CTdiStringValue	Newly appended text
	TDIClearCmd	none	No contents
	TDIReplaceCmd	CTdiStringValue	Contents
CRadioGroup	TDIReplaceCmd	CTdiStringValue	Selected button
CView	TDIReplaceCmd	CTdiStringValue	itsTitle
NCheckBox	TDIReplaceCmd	CTdiBooleanValue	Selection
NScrollBar	TDIReplaceCmd	CTdiFloatValue	Thumb position

Table B.2. TDI events sent by XVT-Power++ classes

C

APPENDIX C: FIELD FORMATTING LANGUAGE REFERENCE

xvt_pattern_create

Creates an XVT_PATTERN From a Pattern String [New 4.5 Function]

Summary

```
XVT_PATTERN xvt_pattern_create (const char *patstr)
```

`const char *patstr`

String describing a Regular Expression pattern.

Description

This function takes a pattern string which defines a Regular Expression pattern, compiles it into a pattern parse tree and returns an `XVT_PATTERN`.

Patterns can be composed of any literal character (single or multibyte), plus the following special symbols.

Character	Meaning
`?`	Match any single character
`#`	Match any digit character
`X`	Match only an alphabetic character
`A`	Match and auto-uppercase alphabetic characters

Character	Meaning
`a`	Match and auto-lowercase alphabetic characters
`*`	Match 0 or more instances of the previous expression
`+`	Match 1 or more instances of the previous expression
`[...]`	Match an optional expression
`(...)`	Match one of any single character contained in the set
`{...}`	Match one of the contained, comma-separated strings with auto-casing and optional auto-completion
`<...>`	Complex expression - treat the contained expression as a single element
`\`	Literal escape of one character (allows the above characters to be treated as literals)
`<all others>`	Literal formatting characters to be inserted automatically in the output

The pattern language grammar shows how the symbols in the above table can be combined. The vertical bar '`|`' signifies "`or`" and '`...`' signifies multiple entries. The language grammar follows:

`CHAR`	Any non-`NULL` character (international or ASCII)
`STRING`	Any series of characters except '`,`' or '`}`'
`LITERAL`	`CHAR | \CHAR`
`MATCH`	`? | # | X | A | a`
`COMPLEX`	`<EXPRESSION>`
`OPTIONAL`	`[EXPRESSION]`
`PICKONE`	`(CHAR...CHAR)`
`COMPLETE`	`{STRING,...STRING}`
`ZEROPLUS`	`EXPRESSION*`
`ONEPLUS`	`EXPRESSION+`
`EXPRESSION`	`LITERAL | MATCH | COMPLEX | OPTIONAL |PICKONE | COMPLETE | ZEROPLUS |ONEPLUS [EXPRESSION]`

Caveats and Limitations

- `()` expressions may only contain single character elements such as literals and single-character match elements such as `A` and `#`.
- `{}` expressions contain enumerations of strings, separated by commas.

- All complex expressions like `()`, `[]`, `<>`, and `{}` must be ended with the appropriate matching character.
- All complex expressions must contain at least one element.
- Expressions may be of arbitrary length and complexity, but the strings they filter and match against are limited to a maximum of 256 characters.
- Care must be taken in building expressions. For example, the expression "`?*A`" will never match anything to the "`A`" element because the "`?*`" expression will 'consume' all of the characters in the input string by itself.

Examples

Some representative patterns used to accomplish certain described tasks are listed below:

Match any string of arbitrary length:

?*

These strings will match: "", "a", "any string"

Match any non-null string:

?+

These strings will match: "a", "any string"
This string will not match: ""

Match a minimum of 3 characters and a maximum of 8 characters:

???[?][?][?][?][?]

These strings will match: "abc", "abcd", ..., "abcdefgh"
These strings will not match: "", "a", "ab", "abcdefghi"

Match an optionally signed integer (notice that the + sign has to be escaped so it's not treated as an operator):

[(\+-)]#+

These strings will match: "12", "-1", "+1", "-1234"
These strings will not match: "", "a", "+a", "0xFF"

Match negative numbers only:

-#+

These strings will match: "-1", "-1234"
These strings will not match: "", "a", "+a", "0xFF", "+1", "1234"

Match any number of instances of automatically upper-cased letters, each followed by a digit:

<A#>*

These strings will match: “”, “a1”, “b2”, “C3”, “D4”
Note that “a1” will resolve to “A1”
These strings will not match: “a”, “abcde”

Match a 10-digit telephone number with automatically added literals:

“\(###\) ###-####”

These strings will match: “(303) 443-4223”, “3034434223”, “(303)4434223”
Note that “3034434223” will resolve to “(303) 443-4223”

Match a US postal code with optional “Plus four” digits:

#####[-####]

These strings will match: “80301”, “80301-8750”, “803018750”
Note that “803018750” will resolve to “80301-8750”

Match British postal codes with automatically upper-cased letters:

“A[A]#[#] #AA”

Match an optionally signed float with optional 1-3 digit exponent:

[(\+-)]#+[.#+][(\+-)(eE)#[#][#]]

These strings will match: “12”, “-12.03”, “12.03-e10”, “+12.03+E10”

Match a full proper name with an optional middle name and automatically upper-cased where appropriate:

“AX+ A(<X*>.) AX+”

These strings will match: “john t. doe”, “john thomas doe”, “Jane t. Doe”
Note that “john t doe” will resolve to “John T. Doe”

Match the day-of-the-week abbreviations with automatic completion and casing:

“{Sun,Mon,Tue,Wed,Thu,Fri,Sat}”

These strings will match: “Su”, “M”, “Wed”
Note that “Su” will resolve (auto-complete) to “Sun” and “M” will resolve to “Mon”

Match a day of the week, with automatic completion and casing (do NOT add unnecessary spaces after commas):

"{Sunday,Monday,Tuesday,Wednesday,Thursday,Friday, Saturday}"

These strings will match: "Su", "M", "Wed", "Thursday"
Note that "Su" will resolve (auto-complete) to "Sunday", "M" will resolve to "Monday", and "Wed" will resolve to "Wednesday"

Match the time of day:

"{1,2,3,4,5,6,7,8,9,10,11,12}:(012345)# {AM,PM}"

These strings will match: "2:29 AM", "12:08 PM"

Match dates:

"{Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec} [(123)]#, ####"

These strings will match: "Feb 29, 1996", "Jan 1, 2000"
Note that the date pattern does not assure that the date is correct, only that the date matches the specified format.
For example, "Feb 31, 1996" would match this pattern.

Match dates in American short format:

"{1,2,3,4,5,6,7,8,9,10,11,12}/(0123)#/##"

These strings will match: "1/23/96", "11/30/00"

INDEX

A

B

C

D

E

F

G

J

K

L

M

N

O

Q

R

S

T

U

V

Y

Z